

Data Structures and Algorithms with Object-Oriented Design Patterns in Python

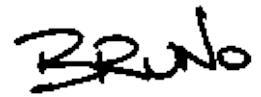
Bruno R. Preiss
B.A.Sc., M.A.Sc., Ph.D., P.Eng.

- [Colophon](#)
- [Dedication](#)
- [Preface](#)
- [Contents](#)
- [Introduction](#)
- [Algorithm Analysis](#)
- [Asymptotic Notation](#)
- [Foundational Data Structures](#)
- [Data Types and Abstraction](#)
- [Stacks, Queues, and Deques](#)
- [Ordered Lists and Sorted Lists](#)
- [Hashing, Hash Tables, and Scatter Tables](#)
- [Trees](#)
- [Search Trees](#)
- [Heaps and Priority Queues](#)
- [Sets, Multisets, and Partitions](#)
- [Garbage Collection and the Other Kind of Heap](#)
- [Algorithmic Patterns and Problem Solvers](#)
- [Sorting Algorithms and Sorters](#)
- [Graphs and Graph Algorithms](#)
- [Python and Object-Oriented Programming](#)
- [Class Hierarchy Diagrams](#)
- [Character Codes](#)
- [References](#)

- [Index](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Python Source Code

Program 2.1

Program to compute $\sum_{i=1}^n i$.

Program 2.2

Program to compute $\sum_{i=0}^n a_i x^i$ using Horner's rule.

Program 2.3

Recursive program to compute $n!$.

Program 2.4

Linear search to find $\max_{0 \leq i < n} a_i$.

Program 2.5

Program to compute γ .

Program 2.6

Program to compute $\sum_{i=0}^n x^i$.

Program 2.7

Program to compute $\sum_{i=0}^n x^i$ using Horner's rule.

Program 2.8

Program to compute x^n .

Program 2.9

Program to compute $\sum_{i=0}^n x^i$ using the closed-form expression.

Program 3.1

Program 2.2 again.

Program 3.2

Program to compute $\sum_{i=0}^j a_i$ for $0 \leq j < n$.

Program 3.3

Non-recursive program to compute Fibonacci numbers.

Program 3.4

Recursive program to compute Fibonacci numbers.

Program 3.5

Bucket sort.

Program 4.1

Array class `__init__` method.

Program 4.2

Array class `__copy__` method.

Program 4.3

 Array class `__getitem__` and `__setitem__` methods.

Program 4.4

 Array class `data` and `baseIndex` properties.

Program 4.5

 Array class `length` property.

Program 4.6

`MultiDimensionalArray` class `__init__` method.

Program 4.7

`MultiDimensionalArray` class `__getitem__` and `__setitem__` methods.

Program 4.8

`Matrix` class.

Program 4.9

`DenseMatrix` class `__init__`, `__getitem__` and `__setitem__` methods.

Program 4.10

`DenseMatrix` class `__times__` method.

Program 4.11

`LinkedList.Element` class.

Program 4.12

`LinkedList` class `__init__` method.

Program 4.13

`LinkedList` class `purge` method.

Program 4.14

`LinkedList` class properties.

Program 4.15

`LinkedList` class `first` and `last` properties.

Program 4.16

`LinkedList` class `prepend` method.

Program 4.17

`LinkedList` class `append` method.

Program 4.18

`LinkedList` class `__copy__` method.

Program 4.19

`LinkedList` class `extract` method.

Program 4.20

`LinkedList.Element` class `insertAfter` and `insertBefore` methods.

Program 5.1

`Object` class definition.

Program 5.2

abstractmethod class definition.

Program 5.3

Container class.

Program 5.4

Container methods and properties.

Program 5.5

Iterator class.

Program 5.6

Visitor interface.

Program 5.7

Container class accept method.

Program 5.8

Container class `__str__` method.

Program 5.9

SearchableContainer abstract class.

Program 5.10

Association class `__init__` method.

Program 5.11

Association properties.

Program 5.12

Association methods.

Program 6.1

Abstract Stack class.

Program 6.2

StackAsArray `__init__` and `purge` methods.

Program 6.3

StackAsArray class `push`, `pop`, and `getTop` methods.

Program 6.4

StackAsArray class accept method.

Program 6.5

StackAsArray class `__iter__` method.

Program 6.6

StackAsLinkedList `__init__` and `purge` methods.

Program 6.7

StackAsLinkedList class `push`, `pop` and `getTop` methods.

Program 6.8

StackAsLinkedList class accept method.

Program 6.9

StackAsLinkedList class `__iter__` method.

Program 6.10

Stack application-a single-digit, RPN calculator.

Program 6.11

Abstract Queue class.

Program 6.12

QueueAsArray `__init__` and `purge` methods.

Program 6.13

QueueAsArray class `enqueue`, `Dequeue` and `getHead` methods.

Program 6.14

QueueAsLinkedList class `__init__` and `purge` methods.

Program 6.15

QueueAsLinkedList class `enqueue`, `dequeue` and `getHead` methods.

Program 6.16

Queue application-breadth-first tree traversal.

Program 6.17

Abstract Deque class.

Program 6.18

DequeAsArray class `enqueueHead` method.

Program 6.19

DequeAsArray class `dequeueTail` and `getTail` methods.

Program 6.20

DequeAsLinkedList class `enqueueHead` method.

Program 6.21

DequeAsLinkedList class `dequeueTail` and `getTail` methods.

Program 7.1

Abstract `OrderedList` class.

Program 7.2

Abstract `Cursor` class.

Program 7.3

`OrderedListAsArray` class `__init__` method.

Program 7.4

`OrderedListAsArray` class `insert` method.

Program 7.5

`OrderedListAsArray` class `__contains__` and `find` methods.

Program 7.6

`OrderedListAsArray` class `withdraw` method.

Program 7.7

`OrderedListAsArray.Cursor` class `__init__` and `getDatum` methods.

Program 7.8

OrderedListAsArray class `findPosition` and `__getitem__` methods.

Program 7.9

`OrderedListAsArray.Cursor` class `insertAfter` method.

Program 7.10

`OrderedListAsArray.Cursor` class `withdraw` method.

Program 7.11

`OrderedListAsLinkedList` class `__init__` method.

Program 7.12

`OrderedListAsLinkedList` class `insert` and `__getitem__` methods.

Program 7.13

`OrderedListAsLinkedList` class `__contains__` and `find` methods.

Program 7.14

`OrderedListAsLinkedList` class `withdraw` method.

Program 7.15

`OrderedListAsLinkedList.Cursor` class `__init__` and `getDatum` methods.

Program 7.16

`OrderedListAsLinkedList` class `findPosition` method

Program 7.17

`OrderedListAsLinkedList.Cursor` class `insertAfter` method.

Program 7.18

`OrderedListAsLinkedList.Cursor` class `withdraw` method.

Program 7.19

`Polynomial.Term` class.

Program 7.20

 Abstract `Polynomial` class.

Program 7.21

`PolynomialAsOrderedList` class.

Program 7.22

`Polynomial` class `differentiate` method.

Program 7.23

 Abstract `SortedList` class.

Program 7.24

`SortedListAsArray` class.

Program 7.25

`SortedListAsArray` class `insert` method.

Program 7.26

`SortedListAsArray` class `findOffset` method.

Program 7.27

`SortedListAsArray` class `find` and `findPosition` methods.

Program 7.28

SortedListAsArrayList class withdraw method.

Program 7.29

SortedListAsLinkedList class.

Program 7.30

SortedListAsLinkedList class insert method.

Program 7.31

 Term methods.

Program 7.32

PolynomialAsSortedList class __init__ and __add__ methods.

Program 8.1

Integer class __hash__ method.

Program 8.2

Float class __hash__ method.

Program 8.3

String class __hash__ method.

Program 8.4

Container class __hash__ method.

Program 8.5

Association class __hash__ method.

Program 8.6

 Abstract **HashTable** class.

Program 8.7

 Abstract **HashTable** class f, g and h methods.

Program 8.8

ChainedHashTable class __init__, __len__ and purge methods.

Program 8.9

ChainedHashTable class insert and withdraw methods.

Program 8.10

ChainedHashTable class find method.

Program 8.11

ChainedScatterTable.Entry class __init__ method.

Program 8.12

ChainedScatterTable class __init__, __len__ and purge methods.

Program 8.13

ChainedScatterTable class insert and find methods.

Program 8.14

ChainedScatterTable class withdraw method.

Program 8.15

OpenScatterTable.Entry class `__init__` method.

Program 8.16

OpenScatterTable class `__init__`, `__len__` and `purge` methods.

Program 8.17

OpenScatterTable class `c`, `findUnoccupied` and `insert` methods.

Program 8.18

OpenScatterTable class `findMatch` and `find` methods.

Program 8.19

OpenScatterTable Class `withdraw` method.

Program 8.20

OpenScatterTablev2 class `withdraw` method.

Program 8.21

Hash/scatter table application-counting words.

Program 9.1

Abstract Tree class.

Program 9.2

Abstract Tree class `depthFirstTraversal` method.

Program 9.3

`PrePostVisitor` class.

Program 9.4

`PreOrder` class.

Program 9.5

`InOrder` class.

Program 9.6

`PostOrder` class.

Program 9.7

Abstract Tree class `breadthFirstTraversal` method.

Program 9.8

Abstract Tree class `accept` method.

Program 9.9

Abstract Tree class `__iter__` method and the `Tree.Iterator` class.

Program 9.10

`GeneralTree` class `__init__` and `purge` methods.

Program 9.11

`GeneralTree` class `getKey`, `getSubtree`, `attachSubtree` and
`detachSubtree` methods.

Program 9.12

`NaryTree` class `__init__` and `purge` methods.

Program 9.13

NaryTree class getKey, getIsEmpty, attachKey and detachKey methods.

Program 9.14

NaryTree class getSubtree, attachSubtree and detachSubtree methods.

Program 9.15

BinaryTree class __init__ and purge methods.

Program 9.16

BinaryTree left and right properties

Program 9.17

BinaryTree class DepthFirstTraversal method.

Program 9.18

BinaryTree class CompareTo method.

Program 9.19

Binary tree application-postfix to infix conversion.

Program 9.20

Binary tree application-printing infix expressions.

Program 10.1

Abstract SearchTree class min and max properties.

Program 10.2

BinarySearchTree class __init__ method.

Program 10.3

BinarySearchTree class find and getMin methods.

Program 10.4

BinarySearchTree class insert, attachKey and balance methods.

Program 10.5

BinarySearchTree class withdraw method.

Program 10.6

AVLTree class __init__ method.

Program 10.7

AVLTree class adjustHeight and getHeight methods and balanceFactor property.

Program 10.8

AVLTree class doLLRotation method.

Program 10.9

AVLTree class doLRRotation method.

Program 10.10

AVLTree class balance method.

Program 10.11

AVLTree class attachKey method.

Program 10.12

AVLTree class `detachKey` method.

Program 10.13

 MWayTree class `__init__` method and `m` property

Program 10.14

 MWayTree class `depthFirstTraversal` method.

Program 10.15

 MWayTree class `find` method (linear search).

Program 10.16

 MWayTree class `findIndex` and `find` methods (binary search).

Program 10.17

 MWayTree class `insert` method.

Program 10.18

 MWayTree class `withdraw` method.

Program 10.19

 BTree class `__init__` and `attachSubtree` methods.

Program 10.20

 BTree class `insert` method.

Program 10.21

 BTree class `insertUp` method.

Program 10.22

 Application of search trees-word translation.

Program 11.1

 Abstract `PriorityQueue` class.

Program 11.2

 Abstract `MergeablePriorityQueue` class.

Program 11.3

 BinaryHeap class `__init__` and `purge` methods.

Program 11.4

 BinaryHeap class `enqueue` method.

Program 11.5

 BinaryHeap class `getMin` method.

Program 11.6

 BinaryHeap class `dequeueMin` method.

Program 11.7

 LeftistHeap class `__init__` method.

Program 11.8

 LeftistHeap class `merge` method.

Program 11.9

 LeftistHeap class `enqueue` method.

Program 11.10

 LeftistHeap class getMin method.

Program 11.11

 LeftistHeap class dequeueMin method.

Program 11.12

 BinomialQueue.BinomialTree class __init__ method.

Program 11.13

 BinomialTree class add method.

Program 11.14

 BinomialQueue class __init__ method.

Program 11.15

 BinomialQueue class addTree and removeTree methods.

Program 11.16

 BinomialQueue class getMinTree and getMin methods.

Program 11.17

 BinomialQueue class merge method.

Program 11.18

 BinomialQueue class fullAdder method.

Program 11.19

 BinomialQueue class enqueue method.

Program 11.20

 BinomialQueue class dequeueMin method.

Program 11.21

 Simulation.Event class.

Program 11.22

 Application of priority queues-discrete event simulation.

Program 12.1

 Abstract Set class.

Program 12.2

 SetAsArray class __init__ method.

Program 12.3

 SetAsArray class insert, __contains__ and withdraw methods.

Program 12.4

 SetAsArray class __or__, __and__ and __sub__ methods.

Program 12.5

 SetAsArray class __eq__ and __lt__ methods.

Program 12.6

 SetAsBitVector class __init__ method.

Program 12.7

SetAsBitVector insert, __contains__ and withdraw methods.

Program 12.8

SetAsBitVector class __or__, __and__ and __sub__ methods.

Program 12.9

Abstract Multiset class.

Program 12.10

MultisetAsArray class __init__ method.

Program 12.11

MultisetAsArray class insert, __contains__ and withdraw methods.

Program 12.12

MultisetAsArray class __or__, __and__ and __sub__ methods.

Program 12.13

MultisetAsLinkedList class __init__ method.

Program 12.14

MultisetAsLinkedList class __or__ method.

Program 12.15

MultisetAsLinkedList class __and__ method.

Program 12.16

Abstract Partition class.

Program 12.17

PartitionAsForest.PartitionTree class __init__ method.

Program 12.18

PartitionAsForest class __init__ method.

Program 12.19

PartitionAsForest class find method.

Program 12.20

PartitionAsForest class simple join method.

Program 12.21

PartitionAsForest class collapsing find method.

Program 12.22

PartitionAsForest class union-by-size join method.

Program 12.23

PartitionAsForest class union-by-rank join method.

Program 12.24

Application of disjoint sets-finding equivalence classes.

Program 14.1

Abstract Solution class.

Program 14.2

Abstract Solver class.

Program 14.3

DepthFirstSolver class `__init__` and search methods.

Program 14.4

BreadthFirstSolver class `__init__` and search methods.

Program 14.5

DepthFirstBranchAndBoundSolver class `__init__` and search methods.

Program 14.6

Divide-and-conquer example-binary search.

Program 14.7

Divide-and-conquer Example-computing Fibonacci numbers.

Program 14.8

Divide-and-conquer example-merge sorting.

Program 14.9

Dynamic programming example-computing generalized Fibonacci numbers.

Program 14.10

Dynamic programming example-computing Binomial coefficients.

Program 14.11

Dynamic programming example-typesetting a paragraph.

Program 14.12

RandomNumberGenerator class `__init__` method, and `_seed_` and `next` properties.

Program 14.13

Abstract RandomVariable class.

Program 14.14

SimpleRV class.

Program 14.15

UniformRV class.

Program 14.16

ExponentialRV class.

Program 14.17

Monte Carlo program to compute π .

Program 15.1

Abstract Sorter class.

Program 15.2

StraightInsertionSorter class `__init__` and `_sort` methods.

Program 15.3

BinaryInsertionSorter class `__init__` and `_sort` methods.

Program 15.4

BubbleSorter class __init__ and _sort methods.

Program 15.5

Abstract QuickSorter class __init__ and selectPivot methods.

Program 15.6

Abstract QuickSorter class quicksort method.

Program 15.7

Abstract QuickSorter class _sort method.

Program 15.8

MedianOfThreeQuickSorter class __init__ and selectPivot methods.

Program 15.9

StraightSelectionSorter class __init__ and _sort methods.

Program 15.10

HeapSorter class __init__ and percolateDown methods.

Program 15.11

HeapSorter class buildHeap method.

Program 15.12

HeapSorter class _sort method.

Program 15.13

TwoWayMergeSorter __init__ method.

Program 15.14

TwoWayMergeSorter class merge method.

Program 15.15

TwoWayMergeSorter class _sort and mergesort methods.

Program 15.16

BucketSorter class __init__ method.

Program 15.17

BucketSorter class _sort method.

Program 15.18

RadixSorter class __init__ method.

Program 15.19

RadixSorter class _sort method.

Program 16.1

Abstract Vertex class.

Program 16.2

Abstract Edge class.

Program 16.3

Abstract Graph, Graph.Vertex and Graph.Edge __init__ methods.

Program 16.4

Abstract Graph class properties.

Program 16.5

Abstract Graph class accessors and mutators.

Program 16.6

Abstract Digraph class.

Program 16.7

GraphAsMatrix class `__init__` method.

Program 16.8

GraphAsLists class `__init__` method.

Program 16.9

Abstract Graph class `depthFirstTraversal` method.

Program 16.10

Abstract Graph class `breadthFirstTraversal` method.

Program 16.11

Abstract Digraph class `topologicalOrderTraversal` method.

Program 16.12

Abstract Graph class `getIsConnected` method.

Program 16.13

Abstract Digraph class `getIsConnected` method.

Program 16.14

Abstract Digraph class `getIsCyclic` method.

Program 16.15

Algorithms.Entry class `__init__` method.

Program 16.16

Dijkstra's algorithm.

Program 16.17

Floyd's algorithm.

Program 16.18

Prim's algorithm.

Program 16.19

Kruskal's algorithm.

Program 16.20

Critical path analysis-computing earliest event times.

Program 16.21

Critical path analysis-finding the critical paths.

Program A.1

Example of parameter passing.

Program A.2

Complex class `__init__` method.

Program A.3

Complex class real and imag properties.

Program A.4

Complex class r and theta properties.

Program A.5

Complex class __add__, __sub__ and __mul__ methods.

Program A.6

Complex class __main__ method.

Program A.7

Person class.

Program A.8

Parent class.

Program A.9

GraphicalObject class __init__, draw, erase and moveTo methods.

Program A.10

Point class __init__ method.

Program A.11

Circle class.

Program A.12

Rectangle class.

Program A.13

Square class.

Program A.14

Using exceptions in Python.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Provides the opus7 package.

Package Contents

abstractmethod	demo3
algorithms	demo4
application1	demo5
application11	demo6
application12	demo7
application2	demo9
application3	denseMatrix
application4	depthFirstBranchAndBoundSolver
application5	depthFirstSolver
application6	deque
application7	dequeAsArray
application8	dequeAsLinkedList
application9	digraph
array	digraphAsLists
association	digraphAsMatrix
avlTree	doubleEndedPriorityQueue
bTree	edge
binaryHeap	example
binaryInsertionSorter	exception
binarySearchTree	experiment1
binaryTree	experiment2
binomialQueue	exponentialRV
breadthFirstBranchAndBoundSolver	expressionTree
breadthFirstSolver	float
bubbleSorter	generalTree
bucketSorter	graph
chainedHashTable	graphAsLists
chainedScatterTable	graphAsMatrix
circle	graphicalObject
complex	hashTable

[container](#)
[cursor](#)
[deap](#)
[demo1](#)
[demo10](#)
[demo2](#)

[heapSorter](#)
[inOrder](#)
[integer](#)
[iterator](#)
[leftistHeap](#)
[linkedList](#)

Data

```
__all__ = ['abstractmethod', 'algorithms', 'application1', 'application11',
'application12', 'application2', 'application3', 'application4',
'application5', 'application6', 'application7', 'application8',
'application9', 'array', 'association', 'avlTree', 'binaryHeap',
'binaryInsertionSorter', 'binarySearchTree', 'binaryTree', ...]
__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '$Date: 2003/09/19 22:26:47 $'
__version__ = '$Revision: 1.13 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

FAQ

1. Where can I buy the Python version of your book?

The Python version of my book is not (yet) in print.

2. Where can I get the pdf version of your book?

There is no pdf version of this book in the public domain.

3. How do I download the web book?

Downloading the web book is not permitted. See the [copyright notice](#).

4. Where can I get the solutions manual?

I have not yet prepared a solutions manual for the Python version of the book.

5. Where can I get the complete source code to the Opus7 library?

The complete source code for the Opus7 library is available [here](#). The Python program fragments given in the book are available [here](#).

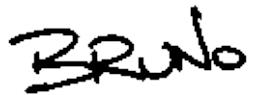
Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Errata

Nothing available yet. I haven't finished making the mistakes.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with "Bruno" written in a larger, more prominent style than the rest of the name.

Colophon

Copyright © 2003 by Bruno R. Preiss.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

This book was prepared with LaTeX and reproduced from camera-ready copy supplied by the author. The book is typeset using the Computer Modern fonts designed by Donald E. Knuth with various additional glyphs designed by the author and implemented using METAFONT.

METAFONT is a trademark of Addison Wesley Publishing Company.

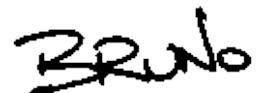
TeX is a trademark of the American Mathematical Society.

UNIX is a registered trademark of AT&T Bell Laboratories.

Microsoft is a registered trademark of Microsoft Corporation.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

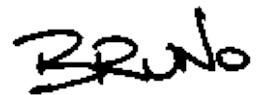


Dedication

To Patty

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Preface

This book was motivated by my experience in teaching the course *E&CE 250: Algorithms and Data Structures* in the Computer Engineering program at the University of Waterloo. I have observed that the advent of *object-oriented methods* and the emergence of object-oriented *design patterns* has lead to a profound change in the pedagogy of data structures and algorithms. The successful application of these techniques gives rise to a kind of cognitive unification: Ideas that are disparate and apparently unrelated seem to come together when the appropriate design patterns and abstractions are used.

This paradigm shift is both evolutionary and revolutionary. On the one hand, the knowledge base grows incrementally as programmers and researchers invent new algorithms and data structures. On the other hand, the proper use of object-oriented techniques requires a fundamental change in the way the programs are designed and implemented. Programmers who are well schooled in the procedural ways often find the leap to objects to be a difficult one.

-
- [Goals](#)
 - [Approach](#)
 - [Outline](#)
 - [Suggested Course Outline](#)
 - [Online Course Materials](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Goals

The primary goal of this book is to promote object-oriented design using Python and to illustrate the use of the emerging *object-oriented design patterns*.

Experienced object-oriented programmers find that certain ways of doing things work best and that these ways occur over and over again. The book shows how these patterns are used to create good software designs. In particular, the following design patterns are used throughout the text: *singleton*, *container*, *iterator*, *adapter* and *visitor*.

Virtually all of the data structures are presented in the context of a *single, unified, polymorphic class hierarchy*. This framework clearly shows the *relationships* between data structures and it illustrates how polymorphism and inheritance can be used effectively. In addition, *algorithmic abstraction* is used extensively when presenting classes of algorithms. By using algorithmic abstraction, it is possible to describe a generic algorithm without having to worry about the details of a particular concrete realization of that algorithm.

A secondary goal of the book is to present mathematical tools *just in time*. Analysis techniques and proofs are presented as needed and in the proper context. In the past when the topics in this book were taught at the graduate level, an author could rely on students having the needed background in mathematics. However, because the book is targeted for second- and third-year students, it is necessary to fill in the background as needed. To the extent possible without compromising correctness, the presentation fosters intuitive understanding of the concepts rather than mathematical rigor.

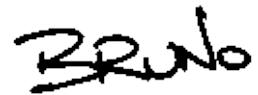
Approach

One cannot learn to program just by reading a book. It is a skill that must be developed by practice. Nevertheless, the best practitioners study the works of others and incorporate their observations into their own practice. I firmly believe that after learning the rudiments of program writing, students should be exposed to examples of complex, yet well-designed program artifacts so that they can learn about the designing good software.

Consequently, this book presents the various data structures and algorithms as complete Python program fragments. All the program fragments presented in this book have been extracted automatically from the source code files of working and tested programs. It has been my experience that by developing the proper abstractions, it is possible to present the concepts as fully functional programs without resorting to *pseudo-code* or to hand-waving.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Outline

This book presents material identified in the *Computing Curricula 1991* report of the ACM/IEEE-CS Joint Curriculum Task Force[47]. The book specifically addresses the following *knowledge units*: AL1: Basic Data structures, AL2: Abstract Data Types, AL3: Recursive Algorithms, AL4: Complexity Analysis, AL6: Sorting and Searching, and AL8: Problem-Solving Strategies. The breadth and depth of coverage is typical of what should appear in the second or third year of an undergraduate program in computer science/computer engineering.

In order to analyze a program, it is necessary to develop a model of the computer. Chapter 2 develops several models and illustrates with examples how these models predict performance. Both average-case and worst-case analyses of running time are considered. Recursive algorithms are discussed and it is shown how to solve a recurrence using repeated substitution. This chapter also reviews arithmetic and geometric series summations, Horner's rule and the properties of harmonic numbers.

Chapter 3 introduces asymptotic (big-oh) notation and shows by comparing with Chapter 2 that the results of asymptotic analysis are consistent with models of higher fidelity. In addition to $O(\cdot)$, this chapter also covers other asymptotic notations ($\Omega(\cdot)$, $\Theta(\cdot)$, and $o(\cdot)$) and develops the asymptotic properties of polynomials and logarithms.

Chapter 4 introduces the *foundational data structures*--the array and the linked list. Virtually all the data structures in the rest of the book can be implemented using either one of these foundational structures. This chapter also covers multi-dimensional arrays and matrices.

Chapter 5 deals with abstraction and data types. It presents the recurring design patterns used throughout the text as well a unifying framework for the data structures presented in the subsequent chapters. In particular, all of the data structures are viewed as *abstract containers*.

Chapter 6 discusses stacks, queues, and deques. This chapter presents implementations based on both foundational data structures (arrays and linked lists). Applications for stacks and queues are presented.

Chapter □ covers ordered lists, both sorted and unsorted. In this chapter, a list is viewed as a *searchable container*. Again several applications of lists are presented.

Chapter □ introduces hashing and the notion of a hash table. This chapter addresses the design of hashing functions for the various basic data types as well as for the abstract data types described in Chapter □. Both scatter tables and hash tables are covered in depth and analytical performance results are derived.

Chapter □ introduces trees and describes their many forms. Both depth-first and breadth-first tree traversals are presented. Completely generic traversal algorithms based on the use of the *visitor* design pattern are presented, thereby illustrating the power of *algorithmic abstraction*. This chapter also shows how trees are used to represent mathematical expressions and illustrates the relationships between traversals and the various expression notations (prefix, infix, and postfix).

Chapter □ addresses trees as *searchable containers*. Again, the power of *algorithmic abstraction* is demonstrated by showing the relationships between simple algorithms and balancing algorithms. This chapter also presents average case performance analyses and illustrates the solution of recurrences by telescoping.

Chapter □ presents several priority queue implementations, including binary heaps, leftist heaps, and binomial queues. In particular this chapter illustrates how a more complicated data structure (leftist heap) extends an existing one (tree). Discrete-event simulation is presented as an application of priority queues.

Chapter □ covers sets and multisets. Also covered are partitions and disjoint set algorithms. The latter topic illustrates again the use of algorithmic abstraction.

Garbage collection is discussed in Chapter □. This is a topic that is not found often in texts of this sort. However, because the Python language relies on garbage collection, it is important to understand how it works and how it affects the running times of programs.

Chapter □ surveys a number of algorithm design techniques. Included are brute-force and greedy algorithms, backtracking algorithms (including branch-and-bound), divide-and-conquer algorithms, and dynamic programming. An object-

oriented approach based on the notion of an *abstract solution space* and an *abstract solver* unifies much of the discussion. This chapter also covers briefly random number generators, Monte Carlo methods, and simulated annealing.

Chapter □ covers the major sorting algorithms in an object-oriented style based on the notion of an *abstract sorter*. Using the abstract sorter illustrates the relationships between the various classes of sorting algorithm and demonstrates the use of algorithmic abstractions.

Finally, Chapter □ presents an overview of graphs and graph algorithms. Both depth-first and breadth-first graph traversals are presented. Topological sort is viewed as yet another special kind of traversal. Generic traversal algorithms based on the *visitor* design pattern are presented, once more illustrating *algorithmic abstraction*. This chapter also covers various shortest path algorithms and minimum-spanning-tree algorithms.

At the end of each chapter is a set of exercises and a set of programming projects. The exercises are designed to consolidate the concepts presented in the text. The programming projects generally require the student to extend the implementation given in the text.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Suggested Course Outline

This text may be used in either a one semester or a two semester course. The course which I teach at Waterloo is a one-semester course that comprises 36 lecture hours on the following topics:

1. Review of the fundamentals of programming in Python and an overview of object-oriented programming with Python. (Appendix □). [4 lecture hours].
2. Models of the computer, algorithm analysis, and asymptotic notation (Chapters □ and □). [4 lecture hours].
3. Foundational data structures, abstraction, and abstract data types (Chapters □ and □). [4 lecture hours].
4. Stacks, queues, ordered lists, and sorted lists (Chapters □ and □). [3 lecture hours].
5. Hashing, hash tables, and scatter tables (Chapter □). [3 lecture hours].
6. Trees and search trees (Chapters □ and □). [6 lecture hours].
7. Heaps and priority queues (Chapter □). [3 lecture hours].
8. Algorithm design techniques (Chapter □). [3 lecture hours].
9. Sorting algorithms and sorters (Chapter □). [3 lecture hours].
10. Graphs and graph algorithms (Chapter □). [3 lecture hours].

Depending on the background of students, a course instructor may find it necessary to review features of the Python language. For example, students need to understand how the Python `for` statement makes use of programmer-defined *iterators*. Similarly, an understanding of the workings of Python *new-style classes*, *inheritance*, and descriptors such as `property` and `staticmethod`, is required in order to understand the unifying class hierarchy discussed in Chapter □.

Online Course Materials

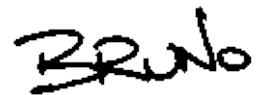
Additional material supporting this book can be found on the world-wide web at the URL:

<http://www.brpreiss.com/books/opus7>

In particular, you will find there the source code for all the program fragments in this book as well as an errata list.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

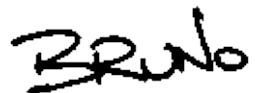


Introduction

- [What This Book Is About](#)
 - [Object-Oriented Design](#)
 - [Object Hierarchies and Design Patterns](#)
 - [The Features of Python You Need to Know](#)
 - [How This Book Is Organized](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



What This Book Is About

This book is about the fundamentals of *data structures and algorithms*--the basic elements from which large and complex software artifacts are built. To develop a solid understanding of a data structure requires three things: First, you must learn how the information is arranged in the memory of the computer. Second, you must become familiar with the algorithms for manipulating the information contained in the data structure. And third, you must understand the performance characteristics of the data structure so that when called upon to select a suitable data structure for a particular application, you are able to make an appropriate decision.

This book also illustrates object-oriented design and it promotes the use of common, object-oriented design patterns. The algorithms and data structures in the book are presented in the Python programming language. Virtually all the data structures are presented in the context of a single class hierarchy. This commitment to a single design allows the programs presented in the later chapters to build upon the programs presented in the earlier chapters.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Object-Oriented Design

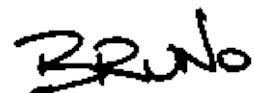
Traditional approaches to the design of software have been either *data oriented* or *process oriented*. Data-oriented methodologies emphasize the representation of information and the relationships between the parts of the whole. The actions which operate on the data are of less significance. On the other hand, process-oriented design methodologies emphasize the actions performed by a software artifact; the data are of lesser importance.

It is now commonly held that *object-oriented* methodologies are more effective for managing the complexity which arises in the design of large and complex software artifacts than either data-oriented or process-oriented methodologies. This is because data and processes are given equal importance. *Objects* are used to combine data with the procedures that operate on that data. The main advantage of using objects is that they provide both *abstraction* and *encapsulation*.

-
- [Abstraction](#)
 - [Encapsulation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Abstraction

Abstraction can be thought of as a mechanism for suppressing irrelevant details while at the same time emphasizing relevant ones. An important benefit of abstraction is that it makes it easier for the programmer to think about the problem to be solved.

For example, *procedural abstraction* lets the software designer think about the actions to be performed without worrying about how those actions are implemented. Similarly, *data abstraction* lets the software designer think about the objects in a program and the interactions between those objects without having to worry about how those objects are implemented.

There are also many different *levels of abstraction*. The lower the levels of abstraction expose more of the details of an implementation whereas the higher levels hide more of the details.

Encapsulation

Encapsulation aids the software designer by enforcing *information hiding*. Objects *encapsulate* data and the procedures for manipulating that data. In a sense, the object *hides* the details of the implementation from the user of that object.

There are two very real benefits from encapsulation--*conceptual* and *physical* independence. Conceptual independence results from hiding the implementation of an object from the user of that object. Consequently, the user is prevented from doing anything with an object that depends on the implementation of that object. This is desirable because it allows the implementation to be changed without requiring the modification of the user's code.

Physical independence arises from the fact that the behavior of an object is determined by the object itself. The behavior of an object is not determined by some external entity. As a result, when we perform an operation on an object, there are no unwanted side-effects.

Object Hierarchies and Design Patterns

There is more to object-oriented programming than simply encapsulating in an object some data and the procedures for manipulating those data. Object-oriented methods deal also with the *classification* of objects and they address the *relationships* between different classes of objects.

The primary facility for expressing relationships between classes of objects is *derivation*--new classes can be derived from existing classes. What makes derivation so useful is the notion of *inheritance*. Derived classes *inherit* the characteristics of the classes from which they are derived. In addition, inherited functionality can be overridden and additional functionality can be defined in a derived class.

A feature of this book is that virtually all the data structures are presented in the context of a single class hierarchy. In effect, the class hierarchy is a taxonomy of data structures. Different implementations of a given abstract data structure are all derived from the same abstract base class. Related base classes are in turn derived from classes that abstract and encapsulate the common features of those classes.

In addition to dealing with hierarchically related classes, experienced object-oriented designers also consider very carefully the interactions between unrelated classes. With experience, a good designer discovers the recurring patterns of interactions between objects. By learning to use these patterns, your object-oriented designs will become more flexible and reusable.

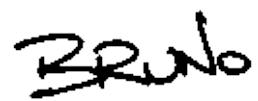
Recently, programmers have started naming the common design patterns. In addition, catalogs of the common patterns are now being compiled and published[[15](#)].

The following *object-oriented design patterns* are used throughout this text:

- [Containers](#)
 - [Iterators](#)
 - [Visitors](#)
 - [Cursors](#)
 - [Adapters](#)
 - [Singlettons](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Containers

A container is an object that holds within it other objects. A container has a capacity, it can be full or empty, and objects can be inserted and withdrawn from a container. In addition, a *searchable container* is a container that supports efficient search operations.

Iterators

An *iterator* provides a means by which the objects within a container can be accessed one-at-a-time. All iterators share a common interface, and hide the underlying implementation of the container from the user of that container.

Visitors

A visitor represents an operation to be performed on all the objects within a container. All visitors share a common interface, and thereby hide the operation to be performed from the container. At the same time, visitors are defined separately from containers. Thus, a particular visitor can be used with any container.

Cursors

A *cursor* represents the position of an object in an ordered container. It provides the user with a way to specify where an operation is to be performed without having to know how that position is represented.

Adapters

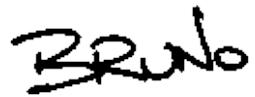
An *adapter* converts the interface of one class into the interface expected by the user of that class. This allows a given class with an incompatible interface to be used in a situation where a different interface is expected.

Singletons

A singleton is a class of which there is only one instance. The class ensures that there only one instance is created and it provides a way to access that instance.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



The Features of Python You Need to Know

This book does not teach the basics of programming. It is assumed that you have taken an introductory course in programming and that you have learned how to write a program in Python. That is, you have learned the rules of Python syntax and you have learned how to put together Python statements in order to solve rudimentary programming problems. The following paragraphs describe more fully aspects of programming in Python with which you should be familiar.

- [Objects, Values and Types](#)
 - [Naming and Binding](#)
 - [Parameter Passing](#)
 - [Classes](#)
 - [Inheritance](#)
 - [New-Style Classes](#)
 - [Other Features](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

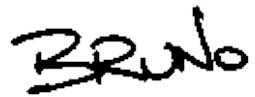


Objects, Values and Types

You must be very comfortable with the notion of an object as an abstraction for data. An object has attributes such as *identity*, *type*, and *value*.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



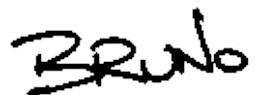
[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Naming and Binding

You must understand the difference between the name of an object and the object itself. In particular, you should understand that an assignment statement binds a new name to an object and that an assignment statement does not modify the value of any object.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



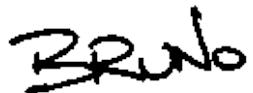
[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Parameter Passing

Parameter passing in Python is *pass-by-reference*. It is essential that you understand that when a function is called, the formal parameter names given in the function definition are bound to the objects named by the actual parameters in the function call.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



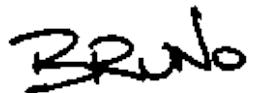
[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Classes

A Python class encapsulates a set of values and a set of operations. The values and the operations of a class are represented by the *attributes* of the class. In Python a class definition introduces a new *type*. The instances of a class type are called objects.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Inheritance

In Python one class may be derived from another. The derived class *inherits* all the attributes of the base class. In addition, inherited attributes can be overridden in the derived class and attributes can be defined. You should understand how the Python virtual machine determines the code to execute when a particular method is called.

New-Style Classes

Python provides two styles of classes -- *classic* classes and *new-style* classes. In this book we use Python *new-style* classes exclusively. New-style classes support *property* attributes and provide a method resolution order that satisfies the constraints of local precedence ordering and monotonicity◊.

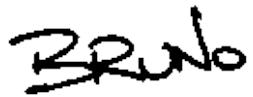
All Python all new-style classes are ultimately derived from the built-in class called `__builtin__.object`. This is how a new-style class is distinguished from a *classic* class.

Other Features

This book makes use of other Python features such as exceptions and generators. You can learn about these topics as you work your way through the book.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

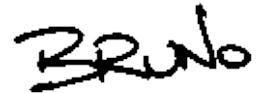


How This Book Is Organized

- [Models and Asymptotic Analysis](#)
 - [Foundational Data Structures](#)
 - [Abstract Data Types and the Class Hierarchy](#)
 - [Data Structures](#)
 - [Algorithms](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Models and Asymptotic Analysis

To analyze the performance of an algorithm, we need to have a model of the computer. Chapter □ presents a series of three models, each one less precise but easier to use than its predecessor. These models are similar, in that they require a careful accounting of the operations performed by an algorithm.

Next, Chapter □ presents *asymptotic analysis*. This is an extremely useful mathematical technique because it simplifies greatly the analysis of algorithms. Asymptotic analysis obviates the need for a detailed accounting of the operations performed by an algorithm, yet at the same time gives a very general result.

Foundational Data Structures

When implementing a data structure, we must decide first whether to use an *array* or a *linked list* as the underlying organizational technique. For this reason, the array and the linked list are called *foundational data structures*. Chapter □ also covers multi-dimensional arrays and matrices.

Abstract Data Types and the Class Hierarchy

Chapter □ introduces the notion of an *abstract data type*. All of the data structures discussed in this book are presented as instances of various abstract data types. Chapter □ also introduces the class hierarchy as well as the various related concepts such as *iterators* and *visitors*.

Data Structures

Chapter □ covers *stacks*, *queues*, and *deques*. *Ordered lists* and *sorted lists* are presented in Chapter □. The concept of hashing is introduced in Chapter □. This chapter also covers the design of hash functions for a number of different object types. Finally, *hash tables* and *scatter tables* are presented.

Trees and search trees are presented in Chapters □ and □. Trees are one of the most important non-linear data structures. Chapter □ also covers the various tree traversals, including depth-first traversal and breadth-first traversal. Chapter □ presents *priority queues* and Chapter □ covers *sets*, *multisets*, and *partitions*.

An essential element of the Python run-time system is the pool of dynamically allocated storage. Chapter □ presents a number of different approaches for implementing garbage collection, in the process illustrating the actual costs associated with dynamic storage allocation.

Algorithms

The last three chapters of the book focus on algorithms, rather than data structures. Chapter □ is an overview of various algorithmic patterns. By introducing the notion of an abstract problem solver, we show how many of the patterns are related. Chapter □ uses a similar approach to present various sorting algorithms. That is, we introduce the notion of an abstract sorter and show how the various sorting algorithms are related.

Finally, Chapter □ gives a brief overview of the subject of graphs and graph algorithms. This chapter brings together various algorithmic techniques from Chapter □ with the class hierarchy discussed in the earlier chapters.

Algorithm Analysis

What is an algorithm and why do we want to analyze one? An algorithm is ``a...step-by-step procedure for accomplishing some end." [10] An algorithm can be given in many ways. For example, it can be written down in English (or French, or any other ``natural" language). However, we are interested in algorithms which have been precisely specified using an appropriate mathematical formalism--such as a programming language.

Given such an expression of an algorithm, what can we do with it? Well, obviously we can run the program and observe its behavior. This is not likely to be very useful or informative in the general case. If we run a particular program on a particular computer with a particular set of inputs, then all know is the behavior of the program in a single instance. Such knowledge is anecdotal and we must be careful when drawing conclusions based upon anecdotal evidence.

In order to learn more about an algorithm, we can ``analyze" it. By this we mean to study the specification of the algorithm and to draw conclusions about how the implementation of that algorithm--the program--will perform in general. But what can we analyze? We can

- determine the running time of a program as a function of its inputs;
- determine the total or maximum memory space needed for program data;
- determine the total size of the program code;
- determine whether the program correctly computes the desired result;
- determine the complexity of the program--e.g., how easy is it to read, understand, and modify; and,
- determine the robustness of the program--e.g., how well does it deal with unexpected or erroneous inputs?

In this text, we are concerned primarily with the running time. We also consider the memory space needed to execute the program. There are many factors that affect the running time of a program. Among these are the algorithm itself, the input data, and the computer system used to run the program. The performance of a computer is determined by

- the hardware:
 - processor used (type and speed),
 - memory available (cache and RAM), and
 - disk available;
- the programming language in which the algorithm is specified;
- the language compiler/interpreter used; and
- the computer operating system software.

A detailed analysis of the performance of a program which takes all of these factors into account is a very difficult and time-consuming undertaking.

Furthermore, such an analysis is not likely to have lasting significance. The rapid pace of change in the underlying technologies means that results of such analyses are not likely to be applicable to the next generation of hardware and software.

In order to overcome this shortcoming, we devise a ``model'' of the behavior of a computer with the goals of simplifying the analysis while still producing meaningful results. The next section introduces the first in a series of such models.

-
- [A Detailed Model of the Computer](#)
 - [A Simplified Model of the Computer](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

A Detailed Model of the Computer

In this section we develop a detailed model of the running time performance of Python programs. The model developed is independent of the underlying hardware and system software. Rather than analyze the performance of a particular, arbitrarily chosen physical machine, we model the execution of a Python program on ``Python virtual machine'' (see Figure □).

A direct consequence of this approach is that we lose some fidelity--the resulting model cannot predict accurately the performance of all possible hardware/software systems. On the other hand, the resulting model is still rather complex and rich in detail.

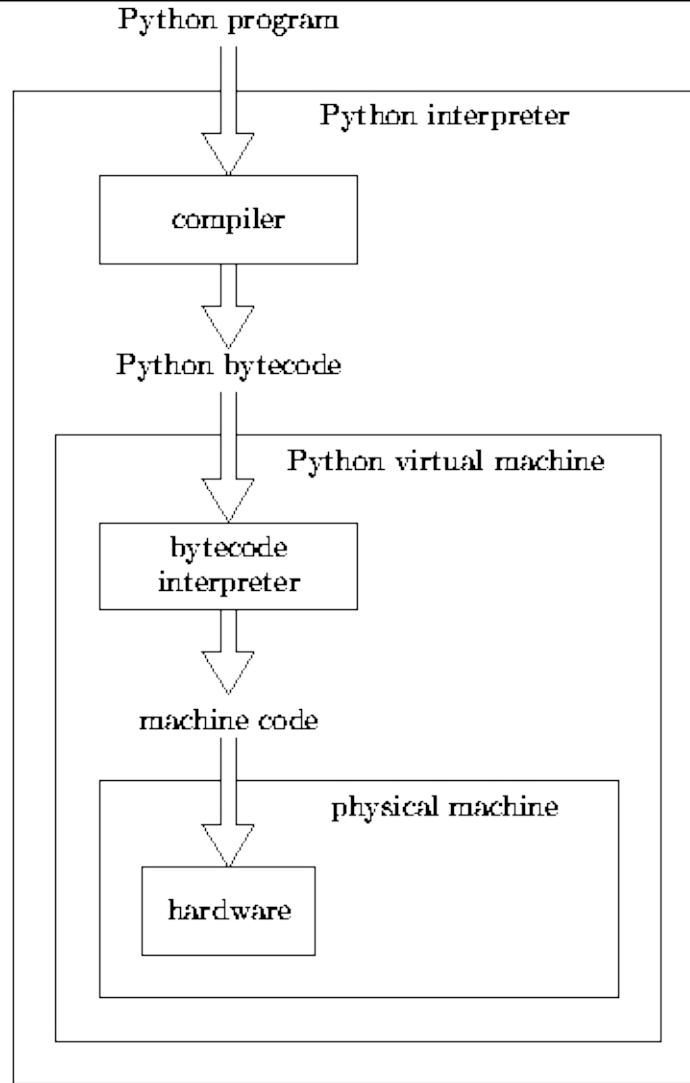


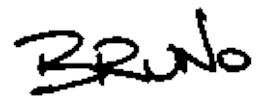
Figure: Python system overview.

- [The Basic Axioms](#)
- [A Simple example-Arithmetic Series Summation](#)
- [Array Subscripting Operations](#)
- [Another example-Horner's Rule](#)
- [Analyzing Recursive Methods](#)
- [Yet Another example-Finding the Largest Element of an Array](#)
- [Average Running Times](#)
- [About Harmonic Numbers](#)
- [Best-Case and Worst-Case Running Times](#)

- [The Last Axiom](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



The Basic Axioms

The running time performance of the common language runtime is given by a set of axioms which we shall now postulate. The first axiom addresses the running time of simple name binding operations:

Axiom The time required to fetch the identity of a named object from memory is a constant, τ_{fetch} , and the time required to bind a new name to an object and store that binding in memory is a constant, τ_{store} .

According to Axiom \square , the assignment statement

$y = x$

has running time $\tau_{\text{fetch}} + \tau_{\text{store}}$. That is, the time taken to fetch the identity of the object named x is τ_{fetch} and the time taken to bind the name y to that object and store that binding in memory is τ_{store} .

We shall apply Axiom \square to manifest constants too: The assignment

$y = 1$

also has running time $\tau_{\text{fetch}} + \tau_{\text{store}}$. To see why this should be the case, consider that the constant ``1'' names an integer object with value one. Therefore, we can expect the cost of fetching the identity of the object named 1 is the same as that of fetching the identity of any other object.

The next axiom addresses the running time of simple arithmetic operations:

Axiom The times required to perform elementary arithmetic operations, such as addition, subtraction, multiplication, division, and comparison, are all constants. These times are denoted by τ_+ , τ_- , τ_\times , τ_\div , and $\tau_<$, respectively.

According to Axiom \square , all the simple operations can be accomplished in a fixed amount of time. In order for this to be feasible, the number of bits used to represent a value must also be fixed. In Python, a *plain integer* can be represented using 32 bits. As long as the operands and the result of an operation are all plain integers, we can say that the running time of the operation is fixed. Python also supports *long integers* which can have an arbitrarily large number

of digits (subject to available memory). If long integers are used, then the basic arithmetic operations can take an arbitrarily long amount of time and Axiom \square does not apply.

By applying Axioms \square and \square , we can determine that the running time of a statement like

```
y = y + 1
```

is $2\tau_{\text{fetch}} + \tau_+ + \tau_{\text{store}}$. This is because we need to fetch the identities of the two objects named `y` and `1`; add them together giving a new object whose value is the sum; and, bind the name `y` to the result and store the new binding in memory.

Python syntax provides an alternative way to express the same computation:

```
y += 1
```

We shall assume that the alternative requires exactly the same running time as the original statement.

The third basic axiom addresses the method call/return overhead:

Axiom The time required to call a method is a constant, τ_{call} , and the time required to return from a method is a constant, τ_{return} .

When a method is called, certain housekeeping operations need to be performed. Typically this includes saving the return address so that program execution can resume at the correct place after the call, saving the state of any partially completed computations so that they may be resumed after the call, and the allocation of a new execution context (stack frame or activation record) in which the called method can be evaluated. Conversely, on the return from a method, all of this work is undone. While the method call/return overhead may be rather large, nevertheless it entails a constant amount of work.

In addition to the method call/return overhead, additional overhead is incurred when parameters are passed to the method:

Axiom The time required to pass an argument to a method is the same as the time required to bind a new name to an object and store that binding in memory, τ_{store} .

The rationale for making the overhead associated with parameter passing the same as the time to create and store a binding is that the passing of an argument is conceptually the same as assignment of the actual parameter value to the formal parameter of the method.

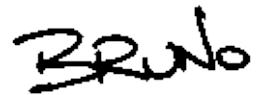
According to Axiom \square , the running time of the statement

$y = f(x)$

would be $\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + T_{f(x)}$, where $T_{f(x)}$ is the running time of method f for input x . The first of the two stores is due to the passing of the parameter x to the method f ; the second arises from the assignment to the variable y .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



A Simple example-Arithmetic Series Summation

In this section we apply Axioms 1, 2 and 3 to the analysis of the running time of a program to compute the following simple arithmetic series summation

$$\sum_{i=1}^n i.$$

The algorithm to compute this summation is given in Program 1.

```
1 def sum(n):
2     result = 0
3     i = 1
4     while i <= n:
5         result += i
6         i += 1
7     return result
```

Program: Program to compute $\sum_{i=1}^n i$.

The executable statements in Program 1 comprise lines 2-7. Table 1 gives the running times of each of these statements.

Table: Computing the running time of Program 1.

statement	time	code
2	$\tau_{\text{fetch}} + \tau_{\text{store}}$	<code>result = 0</code>
3	$\tau_{\text{fetch}} + \tau_{\text{store}}$	<code>i = 1</code>
4	$(2\tau_{\text{fetch}} + \tau_{\leq}) \times (n + 1)$	<code>i <= n</code>
5	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	<code>result += i</code>
6	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	<code>i += 1</code>
7	$\tau_{\text{fetch}} + \tau_{\text{return}}$	<code>return result</code>
TOTAL	$(6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\leq} + 2\tau_{+}) \times n$ + $(5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\leq} + \tau_{\text{return}})$	

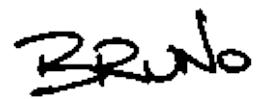
Summing the entries in Table □ we get that the running time, $T(n)$, of Program □ is

$$T(n) = t_1 + t_2 n \quad (2.1)$$

where $t_1 = 5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{\text{return}}$ and $t_2 = 6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + 2\tau_{+}$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Array Subscripting Operations

We now address the question of accessing the elements of an array of data. In general, you can think of the elements of a one-dimensional array as being stored in consecutive memory locations. Therefore, given the address of the first element of the array, a simple addition suffices to determine the address of an arbitrary element of the array:

Axiom The time required for the *address calculation* implied by an array subscripting operation, e.g., $a[i]$, is a constant, $\tau_{[.]}$. This time does not include the time to compute the subscript expression, nor does it include the time to access the array element.

By applying Axiom \square , we can determine that the running time for the statement

$y = a[i]$

is $3\tau_{\text{fetch}} + \tau_{[.]}$ + τ_{store} . Three operand fetches are required: the first to fetch the identity of the array object a ; the second to fetch the identity of the index object i ; and, the third to fetch the identity of the array element $a[i]$.

Another example-Horner's Rule

In this section we apply Axioms □, □, □ and □ to the analysis of the running time of a program which evaluates the value of a polynomial. That is, given the $n+1$ coefficients a_0, a_1, \dots, a_n , and a value x , we wish to compute the following summation

$$\sum_{i=0}^n a_i x^i.$$

The usual way to evaluate such polynomials is to use Horner's rule , which is an algorithm to compute the summation without requiring the computation of arbitrary powers of x . The algorithm to compute this summation is given in Program □. Table □ gives the running times of each of the executable statements in Program □.

```
1 def Horner(a, n, x):
2     result = a[n]
3     i = n - 1
4     while i >= 0:
5         result = result * x + a[i]
6         i -= 1
7     return result
```

Program: Program to compute $\sum_{i=0}^n a_i x^i$ using Horner's rule.

Table: Computing the running time of Program □.

statement	time
2	$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$
3	$2\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}}$
4	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n + 1)$
5	$(5\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{-} + \tau_x + \tau_{\text{store}}) \times n$

6

$$(2\tau_{\text{fetch}} + \tau_- + \tau_{\text{store}}) \times n$$

7

$$\begin{aligned} \text{TOTAL} &= \frac{\tau_{\text{fetch}} + \tau_{\text{return}}}{(9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_< + \tau_{[]} + \tau_+ + \tau_x + \tau_-) \times n} \\ &\quad + \frac{(8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[]} + \tau_- + \tau_< + \tau_{\text{return}})}{.} \end{aligned}$$

Summing the entries in Table □ we get that the running time, $T(n)$, of Program □ is

$$T(n) = t_1 + t_2 n \quad (2.2)$$

where $t_1 = 8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[]} + \tau_- + \tau_< + \tau_{\text{return}}$ and
 $t_2 = 9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_< + \tau_{[]} + \tau_+ + \tau_x + \tau_-$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Analyzing Recursive Methods

In this section we analyze the performance of a recursive algorithm which computes the factorial of a number. Recall that the factorial of a non-negative integer n , written $n!$, is defined as

$$n! = \begin{cases} 1 & n = 0, \\ \prod_{i=1}^n i & n > 0. \end{cases} \quad (2.3)$$

However, we can also define factorial *recursively* as follows

$$n! = \begin{cases} 1 & n = 0, \\ n \times (n - 1)! & n > 0. \end{cases}$$

It is this latter definition which leads to the algorithm given in Program □ to compute the factorial of n . Table □ gives the running times of each of the executable statements in Program □.

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return n * factorial(n - 1)
```

Program: Recursive program to compute $n!$.

Table: Computing the running time of
Program □.

statement	$n=0$	$n>0$
2	$2\tau_{\text{fetch}} + \tau_{\leftarrow}$	$2\tau_{\text{fetch}} + \tau_{\leftarrow}$
3	$\tau_{\text{fetch}} + \tau_{\text{return}}$	--
5	--	$3\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}} + \tau_{\times}$

$$+ \tau_{\text{call}} + \tau_{\text{return}} + T(n - 1)$$

Notice that we had to analyze the running time of the two possible outcomes of the conditional test on line 2 separately. Clearly, the running time of the program depends on the result of this test.

Furthermore, the method `factorial` calls itself recursively on line 5. Therefore, in order to write down the running time of line 5, we need to know the running time, $T(\cdot)$, of `factorial`. But this is precisely what we are trying to determine in the first place! We escape from this catch-22 by assuming that we already know what is the function $T(\cdot)$, and that we can make use of that function to determine the running time of line 5.

By summing the columns in Table □ we get that the running time of Program □ is

$$T(n) = \begin{cases} t_1 & n = 0, \\ T(n - 1) + t_2 & n > 0, \end{cases} \quad (2.4)$$

where $t_1 = 3\tau_{\text{fetch}} + \tau_{\text{c}} + \tau_{\text{return}}$ and $t_2 = 5\tau_{\text{fetch}} + \tau_{\text{c}} + \tau_{\text{--}} + \tau_{\text{store}} + \tau_{\text{x}} + \tau_{\text{call}} + \tau_{\text{return}}$.

This kind of equation is called a *recurrence relation* because the function is defined in terms of itself recursively.

-
- [Solving Recurrence Relations-Repeated Substitution](#)
-



Solving Recurrence Relations-Repeated Substitution

In this section we present a technique for solving a recurrence relation such as Equation \square called *repeated substitution*. The basic idea is this: Given that $T(n) = T(n - 1) + t_2$, then we may also write $T(n - 1) = T(n - 2) + t_2$, provided $n > 1$. Since $T(n-1)$ appears in the right-hand side of the former equation, we can substitute for it the entire right-hand side of the latter. By repeating this process we get

$$\begin{aligned} T(n) &= T(n - 1) + t_2 \\ &= (T(n - 2) + t_2) + t_2 \\ &= T(n - 2) + 2t_2 \\ &= (T(n - 3) + t_2) + 2t_2 \\ &= T(n - 3) + 3t_2 \\ &\vdots \end{aligned}$$

The next step takes a little intuition: We must try to discern the pattern which is emerging. In this case it is obvious:

$$T(n) = T(n - k) + kt_2,$$

where $1 \leq k \leq n$. Of course, if we have doubts about our intuition, we can always check our result by induction:

extbfProof (By Induction). **Base Case** Clearly the formula is correct for $k=1$, since $T(n) = T(n - k) + kt_2 = T(n - 1) + t_2$.

Inductive Hypothesis Assume that $T(n) = T(n - k) + kt_2$ for $k = 1, 2, \dots, l$. By this assumption

$$T(n) = T(n - l) + lt_2. \quad (2.5)$$

Note also that using the original recurrence relation we can write

$$T(n - l) = T(n - l - 1) + t_2 \quad (2.6)$$

for $l \leq n$. Substituting Equation \square in the right-hand side of Equation \square gives

$$\begin{aligned} T(n) &= T(n - l - 1) + t_2 + lt_2 \\ &= T(n - (l + 1)) + (l + 1)t_2 \end{aligned}$$

which makes the formula correct for $l+1$. Therefore, by induction on l , our formula is correct for all $0 \leq k \leq n$.

So, we have shown that $T(n) = T(n - k) + kt_2$, for $1 \leq k \leq n$. Now, if n was known, we would repeat the process of substitution until we got $T(0)$ on the right hand side. The fact that n is unknown should not deter us--we get $T(0)$ on the right hand side when $n-k=0$. That is, $k=n$. Letting $k=n$ we get

$$\begin{aligned} T(n) &= T(n - k) + kt_2 \\ &= T(0) + nt_2 \\ &= t_1 + nt_2 \end{aligned} \tag{2.7}$$

where $t_1 = 3\tau_{\text{fetch}} + \tau_{\leftarrow} + \tau_{\text{return}}$ and $t_2 = 5\tau_{\text{fetch}} + \tau_{\leftarrow} + \tau_{\leftarrow} + \tau_{\text{store}} + \tau_x + \tau_{\text{call}} + \tau_{\text{return}}$.

Yet Another example-Finding the Largest Element of an Array

In this section we consider the problem of finding the largest element of an array. That is, given an array of n non-negative integers, a_0, a_1, \dots, a_{n-1} , we wish to find

$$\max_{0 \leq i < n} a_i.$$

The straightforward way of solving this problem is to perform a *linear search* of the array. The linear search algorithm is given in Program □ and the running times for the various statements are given in Table □.

```
1 def findMaximum(a, n):
2     result = a[0]
3     i = 1
4     while i < n:
5         if a[i] > result:
6             result = a[i]
7         i += 1
8     return result
```

Program: Linear search to find $\max_{0 \leq i < n} a_i$.

Table: Computing the running time of Program □.

statement	time
2	$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$
3	$\tau_{\text{fetch}} + \tau_{\text{store}}$
4	$(2\tau_{\text{fetch}} + \tau_{<}) \times n$
5	$(4\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{<}) \times (n - 1)$
6	$(3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}) \times ?$
7	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times (n - 1)$
8	$\tau_{\text{fetch}} + \tau_{\text{store}}$

With the exception of line 6, the running times follow simply from Axioms \square , \square and \square . In particular, note that the body of the loop is executed $n-1$ times. This means that the conditional test on line 5 is executed $n-1$ times. However, the number of times line 6 is executed depends on the data in the array and not just n .

If we consider that in each iteration of the loop body, the variable `result` contains the largest array element seen so far, then line 6 will be executed in the i^{th} iteration of the loop only if a_i satisfies the following

$$a_i > \left(\max_{0 \leq j < i} a_j \right).$$

Thus, the running time of Program \square , $T(\cdot)$, is a function not only of the number of elements in the array, n , but also of the actual array values, a_0, a_1, \dots, a_{n-1} . Summing the entries in Table \square we get

$$T(n, a_0, a_1, \dots, a_{n-1}) = t_1 + t_2 n + \sum_{i=1}^{n-1} t_3$$

$$a_i > \left(\max_{0 \leq j < i} a_j \right)$$

where

$$\begin{aligned} t_1 &= 2\tau_{\text{store}} - \tau_{\text{fetch}} - \tau_+ - \tau_- \\ t_2 &= 8\tau_{\text{fetch}} + 2\tau_- + \tau_{[]} + \tau_+ + \tau_{\text{store}} \\ t_3 &= 3\tau_{\text{fetch}} + \tau_{[]} + \tau_{\text{store}}. \end{aligned}$$

While this result may be correct, it is not terribly useful. In order to determine the running time of the program we need to know the number of elements in the array, n , and we need to know the values of the elements in the array, a_0, a_1, \dots, a_{n-1} . Even if we know these data, it turns out that in order to compute the running time of the algorithm, $T(n, a_0, a_1, \dots, a_{n-1})$, we actually have to solve the original problem!

Average Running Times

In the previous section, we found the function, $T(n, a_0, a_1, \dots, a_{n-1})$, which gives the running time of Program \square as a function both of number of inputs, n , and of the actual input values. Suppose instead we are interested in a function $T_{\text{average}}(n)$ which gives the running time *on average* for n inputs, regardless of the values of those inputs. In other words, if we run Program \square , a large number of times on a selection of random inputs of length n , what will the average running time be?

We can write the sum of the running times given in Table \square in the following form

$$T_{\text{average}}(n) = t_1 + t_2 n + \sum_{i=1}^{n-1} p_i t_3 \quad (2.8)$$

where p_i is the probability that line 6 of the program is executed. The probability p_i is given by

$$p_i = P \left[a_i > (\max_{0 \leq j < i} a_j) \right].$$

That is, p_i is the probability that the i^{th} array entry, a_i , is larger than the maximum of all the preceding array entries, a_0, a_1, \dots, a_{i-1} .

In order to determine p_i , we need to know (or to assume) something about the distribution of input values. For example, if we know *a priori* that the array passed to the method `findMaximum` is ordered from smallest to largest, then we know that $p_i = 1$. Conversely, if we know that the array is ordered from largest to smallest, then we know that $p_i = 0$.

In the general case, we have no *a priori* knowledge of the distribution of the values in the input array. In this case, consider the i^{th} iteration of the loop. In this iteration a_i is compared with the maximum of the i values, a_0, a_1, \dots, a_{i-1} preceding it in the array. Line 6 of Program \square is only executed if a_i is the largest of the $i+1$ values a_0, a_1, \dots, a_i . All things being equal, we can say that this will

happen with probability $1/(i+1)$. Thus

$$\begin{aligned} p_i &= P \left[a_i > (\max_{0 \leq j < i} a_j) \right] \\ &= \frac{1}{i+1}. \end{aligned} \tag{2.9}$$

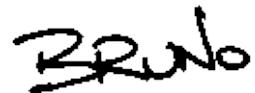
Substituting this expression for p_i in Equation □ and simplifying the result we get

$$\begin{aligned} T_{\text{average}}(n) &= t_1 + t_2 n + \sum_{i=1}^{n-1} p_i t_3 \\ &= t_1 + t_2 n + t_3 \sum_{i=1}^{n-1} \frac{1}{i+1} \\ &= t_1 + t_2 n + t_3 \left(\sum_{i=1}^n \frac{1}{i} - 1 \right) \\ &= t_1 + t_2 n + t_3 (H_n - 1) \end{aligned} \tag{2.10}$$

where $H_n = \sum_{i=1}^n \frac{1}{i}$, is the n^{th} harmonic number .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



About Harmonic Numbers

The series $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots$ is called the *harmonic series*, and the summation

$$H_n = \sum_{i=1}^n \frac{1}{i}$$

gives rise to the series of *harmonic numbers*, H_1, H_2, \dots . As it turns out, harmonic numbers often creep into the analysis of algorithms. Therefore, we should understand a little bit about how they behave.

A remarkable characteristic of harmonic numbers is that, even though as n gets large and the difference between consecutive harmonic numbers gets arbitrarily small ($\frac{H_n - H_{n-1}}{n} = \frac{1}{n^2}$), *the series does not converge!* That is, $\lim_{n \rightarrow \infty} H_n$ does not exist. In other words, the summation $\sum_{i=1}^{\infty} \frac{1}{i}$ goes off to infinity, but just barely.

Figure □ helps us to understand the behavior of harmonic numbers. The smooth curve in this figure is the function $y=1/x$. The descending staircase represents the function $y = 1/\lfloor x \rfloor$. ◇

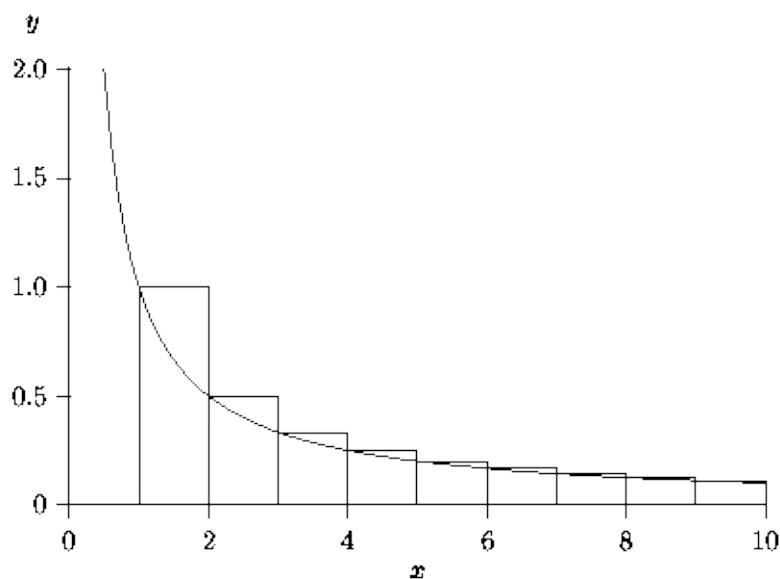


Figure: Computing harmonic numbers.

Notice that the area under the staircase between 1 and n for any integer $n > 1$ is given by

$$\begin{aligned}\int_1^n \frac{1}{[x]} dx &= \sum_{i=1}^{n-1} \frac{1}{i} \\ &= H_{n-1}.\end{aligned}$$

Thus, if we can determine the area under the descending staircase in Figure □, we can determine the values of the harmonic numbers.

As an approximation, consider the area under the smooth curve $y = 1/x$:

$$\begin{aligned}\int_1^n \frac{1}{x} dx &= \ln x \Big|_1^n \\ &= \ln(n).\end{aligned}$$

Thus, H_{n-1} is approximately $\ln n$ for $n > 1$.

If we approximate H_{n-1} by $\ln n$, the error in this approximation is equal to the area between the two curves. In fact, the area between these two curves is such an important quantity that it has its own symbol, γ , which is called *Euler's constant*. The following derivation indicates a way in which to compute Euler's constant:

$$\begin{aligned}\gamma &= \lim_{n \rightarrow \infty} (H_{n-1} - \ln n) \\ &= \sum_{i=1}^{\infty} \left(\int_i^{i+1} \left(\frac{1}{i} - \frac{1}{x} \right) dx \right) \\ &= \sum_{i=1}^{\infty} \left(\frac{1}{i} \int_i^{i+1} 1 dx - \int_i^{i+1} \frac{1}{x} dx \right) \\ &= \sum_{i=1}^{\infty} \left(\frac{1}{i} - \ln \left(\frac{i+1}{i} \right) \right) \\ &\approx 0.577215\end{aligned}$$

A program to compute Euler's constant on the basis of this derivation is given in Program □. While this is not necessarily the most accurate or most speedy way to compute Euler's constant, it does give the correct result to six significant digits.

```

1 def gamma():
2     result = 0.
3     i = 1
4     while i <= 500000:
5         result += 1.0/i - math.log((i + 1.0)/i)
6         i += 1
7     return result

```

Program: Program to compute γ .

So, with Euler's constant in hand, we can write down an expression for the $(n-1)^{th}$ harmonic number:

$$H_{n-1} = \ln n + \gamma - \epsilon_n \quad (2.11)$$

where ϵ_n is the error introduced by the fact that γ is defined as the difference between the curves on the interval $[1, +\infty)$, but we only need the difference on the interval $[1, n]$. As it turns out, it can be shown (but not here), that there exists a constant K such that for large enough values of n , $|\epsilon_n| < K/n$. \diamond

Since the error term is less than $1/n$, we can add $1/n$ to both sides of Equation □ and still have an error which goes to zero as n gets large. Thus, the usual approximation for the harmonic number is

$$H_n \approx \ln n + \gamma.$$

We now return to the question of finding the average running time of Program □, which finds the largest element of an array. We can now rewrite Equation □ to give

$$\begin{aligned} T_{\text{average}}(n) &= t_1 + t_2 n + t_3(H_n - 1) \\ &\approx t_1 + t_2 n + t_3(\ln n + \gamma - 1) \\ &\approx (t_1 + t_3(\gamma - 1)) + t_2 n + t_3 \ln n. \end{aligned}$$

Best-Case and Worst-Case Running Times

In Section □ we derived the average running time of Program □ which finds the largest element of an array. In order to do this we had to determine the probability that a certain program statement is executed. To do this, we made an assumption about the *average* input to the program.

The analysis can be significantly simplified if we simply wish to determine the *worst case* running time. For Program □, the worst-case scenario occurs when line 6 is executed in every iteration of the loop. We saw that this corresponds to the case in which the input array is ordered from smallest to largest. In terms of Equation □, this occurs when $p_i = 1$. Thus, the worst-case running time is given by

$$\begin{aligned} T_{\text{worst case}}(n) &= t_1 + t_2 n + \sum_{i=1}^{n-1} p_i t_3 \Big|_{p_i=1} \\ &= t_1 + t_2 n + t_3 \sum_{i=1}^{n-1} 1 \\ &= t_1 + t_2 n + t_3(n - 1) \\ &= (t_1 - t_3) + (t_2 + t_3) \times n. \end{aligned}$$

Similarly, the *best-case* running time occurs when line 6 is never executed. This corresponds to the case in which the input array is ordered from largest to smallest. This occurs when $p_i = 0$ and best-case running time is

$$\begin{aligned} T_{\text{best case}}(n) &= t_1 + t_2 n + \sum_{i=1}^{n-1} p_i t_3 \Big|_{p_i=0} \\ &= t_1 + t_2 n \end{aligned}$$

In summary we have the following results for the running time of Program □:

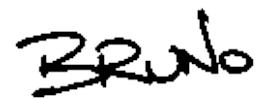
$$T(n, a_0, a_1, \dots, a_{n-1}) = t_1 + t_2 n + \sum_{\substack{i=1 \\ a_i > (\max_{0 \leq j < i} a_j)}}^{n-1} t_3$$

$$T_{\text{average}}(n) \approx (t_1 + t_3(\gamma - 1)) + t_2 n + t_3 \ln n$$

$$T_{\text{worst case}}(n) = (t_1 - t_3) + (t_2 + t_3) \times n$$

$$T_{\text{best case}}(n) = t_1 + t_2 n$$

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

The Last Axiom

In this section we state the last axiom needed for the detailed model of the Python virtual machine. This axiom addresses the time required to create a new object instance of a class:

Axiom The time required to create a new object instance of a class is a constant, τ_{new} . This time does not include any time taken to initialize the object.

By applying Axioms \square , \square , \square and \square , we can determine that the running time of the statement

```
i = IntType(0)
```

is $\tau_{\text{new}} + \tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{\text{call}} + T^{\langle \text{IntType}.\text{__init__} \rangle}$, where $T^{\langle \text{IntType}.\text{__init__} \rangle}$ is the running time of the `__init__` method of the class `IntType`.

A Simplified Model of the Computer

The detailed model of the computer given in the previous section is based on a number of different timing parameters-- τ_{fetch} , τ_{store} , τ_+ , τ_- , τ_{\times} , τ_{\div} , $\tau_{<}$, τ_{call} , τ_{return} , τ_{new} , and $\tau_{[\cdot]}$. While it is true that a model with a large number of parameters is quite flexible and therefore likely to be a good predictor of performance, keeping track of all of the parameters during the analysis is rather burdensome.

In this section, we present a simplified model which makes the performance analysis easier to do. The cost of using the simplified model is that it is likely to be a less accurate predictor of performance than the detailed model.

Consider the various timing parameters in the detailed model. In a real machine, each of these parameters is a multiple of the basic clock period of the machine. The clock frequency of a modern computer is typically between 500 MHz and 2 GHz. Therefore, the clock period is typically between 0.5 and 2 ns. Let the clock period of the machine be T . Then each of the timing parameters can be expressed as an integer multiple of the clock period. For example, $\tau_{\text{fetch}} = k_{\text{fetch}}T$, where $k_{\text{fetch}} \in \mathbb{Z}, k_{\text{fetch}} > 0$.

The simplified model eliminates all of the arbitrary timing parameters in the detailed model. This is done by making the following two simplifying assumptions:

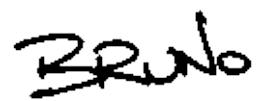
- All timing parameters are expressed in units of clock cycles. In effect, $T=1$.
- The proportionality constant, k , for all timing parameters is assumed to be the same: $k=1$.

The effect of these two assumptions is that we no longer need to keep track of the various operations separately. To determine the running time of a program, we simply count the total number of cycles taken.

- [An example-Geometric Series Summation](#)
 - [About Arithmetic Series Summation](#)
 - [Example-Geometric Series Summation Again](#)
 - [About Geometric Series Summation](#)
 - [Example-Computing Powers](#)
 - [Example-Geometric Series Summation Yet Again](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

An example-Geometric Series Summation

In this section we consider the running time of a program to compute the following *geometric series summation*. That is, given a value x and non-negative integer n , we wish to compute the summation

$$\sum_{i=0}^n x^i.$$

An algorithm to compute this summation is given in Program □.

```
1 def geometricSeriesSum(x, n):
2     sum = 0
3     i = 0
4     while i <= n:
5         prod = 1
6         j = 0
7         while j < i:
8             prod *= x
9             j += 1
10            sum += prod
11            i += 1
12 return sum
```

Program: Program to compute $\sum_{i=0}^n x^i$.

Table □ gives the running time, as predicted by the simplified model, for each of the executable statements in Program □.

Table: Computing the running time of Program □.

statement	time
2	2

3	2
4	$3(n+2)$
5	$2(n+1)$
6	$2(n+1)$
7	$2 \sum_{i=0}^n (i + 1)$
8	$4 \sum_{i=0}^n i$
9	$4 \sum_{i=0}^n i$
10	$4(n+1)$
11	$4(n+1)$
12	2
TOTAL	$\frac{11}{2}n^2 + \frac{47}{2}n + 24$

In order to calculate the total cycle counts, we need to evaluate the two series summations $\sum_{i=0}^n (i + 1)$ and $\sum_{i=0}^n i$. Both of these are *arithmetic series summations*. In the next section we show that the sum of the series $1, 2, \dots, n$ is $n(n+1)/2$. Using this result we can sum the cycle counts given in Table □ to arrive at the total running time of $\frac{11}{2}n^2 + \frac{47}{2}n + 24$ cycles.

About Arithmetic Series Summation

The series, $1, 2, 3, 4, \dots$, is an *arithmetic series* and the summation

$$S_n = \sum_{i=1}^n i$$

is called the *arithmetic series summation*.

The summation can be solved as follows: First, we make the simple variable substitution $i=n-j$:

$$\begin{aligned} \sum_{i=1}^n i &= \sum_{n-j=1}^n (n-j) \\ &= \sum_{j=0}^{n-1} (n-j) \\ &= \sum_{j=0}^{n-1} n - \sum_{j=0}^{n-1} j \\ &= n \sum_{j=0}^{n-1} 1 - \sum_{j=1}^n j + n \end{aligned} \tag{2.12}$$

Note that the term in the first summation in Equation 2.12 is independent of j . Also, the second summation is identical to the left hand side. Rearranging Equation 2.12, and simplifying gives

$$\begin{aligned} 2 \sum_{i=1}^n i &= n \sum_{j=0}^{n-1} 1 + n \\ &= n^2 + n \\ &= n(n+1) \\ \sum_{i=1}^n i &= \frac{n(n+1)}{2}. \end{aligned}$$

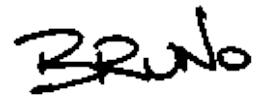
There is, of course, a simpler way to arrive at this answer. Consider the series, $1, 2, 3, 4, \dots, n$, and suppose n is even. The sum of the first and last element is $n+1$. So too is the sum of the second and second-last element, and the third and third-

last element, etc., and there are $n/2$ such pairs. Therefore, $S_n = \frac{n}{2}(n + 1)$.

And if n is odd, then $S_n = S_{n-1} + n$, where $n-1$ is even. So we can use the previous result for S_{n-1} to get $S_n = \frac{n-1}{2}n + n = n(n + 1)/2$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Geometric Series Summation Again

In this example we revisit the problem of computing a *geometric series summation*. We have already seen an algorithm to compute this summation in Section □ (Program □). This algorithm was shown to take $\frac{11}{2}n^2 + \frac{47}{2}n + 24$ cycles.

The problem of computing the geometric series summation is identical to that of computing the value of a polynomial in which all of the coefficients are one. This suggests that we could make use of *Horner's rule* as discussed in Section □. An algorithm to compute a geometric series summation using Horner's rule is given in Program □.

```
1 def geometricSeriesSum(x, n):
2     sum = 0
3     i = 0
4     while i <= n:
5         sum = sum * x + 1
6         i += 1
7     return sum
```

Program: Program to compute $\sum_{i=0}^n x^i$ using Horner's rule.

The executable statements in Program □ comprise lines 2-7. Table □ gives the running times, as given by the simplified model, for each of these statements.

Table:

Computing the
running time of
Program □.

statement	time
2	2
3	2

$$\begin{array}{rcl}
 4 & 3(n+2) \\
 5 & 6(n+1) \\
 6 & 4(n+1) \\
 7 & 2 \\
 \hline
 \text{TOTAL} & 13n+22
 \end{array}$$

In Programs □ and □ we have seen two different algorithms to compute the same geometric series summation. We determined the running time of the former to be $\frac{11}{2}n^2 + \frac{47}{2}n + 24$ cycles and of the latter to be $13n+22$ cycles. In particular, note that for all non-negative values of n , $(\frac{11}{2}n^2 + \frac{47}{2}n + 24) > 13n + 22$. Hence, according to our simplified model of the computer, Program □, which uses Horner's rule, *always* runs faster than Program □!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

About Geometric Series Summation

The series, $1, a, a^2, a^3, \dots$, is a *geometric series* and the summation

$$S_n = \sum_{i=0}^n a^i$$

is called the *geometric series summation*.

The summation can be solved as follows: First, we make the simple variable substitution $i=j-1$:

$$\begin{aligned}\sum_{i=0}^n a^i &= \sum_{j-1=0}^n a^{j-1} \\ &= \frac{1}{a} \sum_{j=1}^{n+1} a^j \\ &= \frac{1}{a} \left(\sum_{j=0}^n a^j + a^{n+1} - 1 \right)\end{aligned}\tag{2.13}$$

Note that the summation which appears on the right is identical to the left hand side. Rearranging Equation □, and simplifying gives

$$\sum_{i=0}^n a_i = \frac{a^{n+1} - 1}{a - 1}.\tag{2.14}$$

Example-Computing Powers

In this section we consider the running time to raise a number to a given integer power. That is, given a value x and non-negative integer n , we wish to compute the x^n . A naive way to calculate x^n would be to use a loop such as

```
result = 1
i = 0
while i <= n:
    result = result * x
    i += 1
```

While this may be fine for small values of n , for large values of n the running time may become prohibitive. As an alternative, consider the following recursive definition

$$x^n = \begin{cases} 1 & n = 0, \\ (x^2)^{\lfloor n/2 \rfloor} & n > 0, n \text{ is even}, \\ x(x^2)^{\lfloor n/2 \rfloor} & n > 0, n \text{ is odd}. \end{cases} \quad (2.15)$$

For example, using Equation □, we would determine x^{32} as follows

$$x^{32} = \left(\left(\left((x^2)^2 \right)^2 \right)^2 \right)^2,$$

which requires a total of five multiplication operations. Similarly, we would compute x^{31} as follows

$$x^{31} = \left(\left(\left((x^2) x \right)^2 x \right)^2 x \right)^2 x,$$

which requires a total of eight multiplication operations.

A recursive algorithm to compute x^n based on the direct implementation of Equation □ is given in Program □. Table □ gives the running time, as predicted by the simplified model, for each of the executable statements in Program □.

```

1 def power(x, n):
2     if n == 0:
3         return 1
4     elif n % 2 == 0: # n is even
5         return power(x * x, n / 2)
6     else: # n is odd
7         return x * power(x * x, n / 2)

```

Program: Program to compute x^n .

Table: Computing the running time of
Program □.

	time	
statement	$n=0$	$n>0$
	n is even	n is odd
2	3	3
3	2	--
4	--	5
5	--	$10 + T(\lfloor n/2 \rfloor)$
7	--	$12 + T(\lfloor n/2 \rfloor)$
TOTAL	5	$18 + T(\lfloor n/2 \rfloor)$
		$20 + T(\lfloor n/2 \rfloor)$

By summing the columns in Table □ we get the following recurrence for the running time of Program □

$$T(n) = \begin{cases} 5 & n = 0, \\ 18 + T(\lfloor n/2 \rfloor) & n > 0, n \text{ is even}, \\ 20 + T(\lfloor n/2 \rfloor) & n > 0, n \text{ is odd}. \end{cases} \quad (2.16)$$

As the first attempt at solving this recurrence, let us suppose that $n = 2^k$ for some $k > 0$. Clearly, since n is a power of two, it is even. Therefore, $\lfloor n/2 \rfloor = n/2 = 2^{k-1}$.

For $n = 2^k$, Equation □ gives

$$T(2^k) = 18 + T(2^{k-1}), \quad k > 0.$$

This can be solved by repeated substitution:

$$\begin{aligned} T(2^k) &= 18 + T(2^{k-1}) \\ &= 18 + 18 + T(2^{k-2}) \\ &= 18 + 18 + 18 + T(2^{k-3}) \\ &\vdots \\ &= 18j + T(2^{k-j}). \end{aligned}$$

The substitution stops when $k=j$. Thus,

$$\begin{aligned} T(2^k - 1) &= 18k + T(1) \\ &= 18k + 20 + T(0) \\ &= 18k + 20 + 5 \\ &= 18k + 25. \end{aligned}$$

Note that if $n = 2^k$, then $k = \log_2 n$. In this case, running time of Program \square is $T(n) = 18\log_2 n + 25$.

The preceding result is, in fact, the best case--in all but the last two recursive calls of the method, n was even. Interestingly enough, there is a corresponding worst-case scenario. Suppose $n = 2^k - 1$ for some value of $k > 0$. Clearly n is odd, since it is one less than 2^k which is a power of two and even. Now consider $\lfloor n/2 \rfloor$:

$$\begin{aligned} \lfloor n/2 \rfloor &= \lfloor (2^k - 1)/2 \rfloor \\ &= (2^k - 2)/2 \\ &= 2^{k-1} - 1. \end{aligned}$$

Hence, $\lfloor n/2 \rfloor$ is also odd!

For example, suppose n is 31 ($2^5 - 1$). To compute x^{31} , Program \square calls itself recursively to compute x^{15} , x^7 , x^3 , x^1 , and finally, x^0 --all but the last of which are odd powers of x .

For $n = 2^k - 1$, Equation \square gives

$$T(2^k - 1) = 20 + T(2^{k-1} - 1), \quad k > 1.$$

Solving this recurrence by repeated substitution we get

$$\begin{aligned}
 T(2^k - 1) &= 20 + T(2^{k-1} - 1) \\
 &= 20 + 20 + T(2^{k-2} - 1) \\
 &= 20 + 20 + 20 + T(2^{k-3} - 1) \\
 &\vdots \\
 &= 20j + T(2^{k-j} - 1).
 \end{aligned}$$

The substitution stops when $k=j$. Thus,

$$\begin{aligned}
 T(2^k - 1) &= 20k + T(2^0 - 1) \\
 &= 20k + 5.
 \end{aligned}$$

Note that if $n = 2^k - 1$, then $k = \log_2(n + 1)$. In this case, running time of Program \square is $T(n) = 20 \log_2(n + 1) + 5$.

Consider now what happens for an arbitrary value of n . Table \square shows the recursive calls made by Program \square in computing x^n for various values of n .

Table: Recursive calls made in Program \square .

$n^{\lfloor \log_2 n \rfloor + 1}$ powers computed recursively		
1	1	<u>1, 0</u>
2	2	<u>2, 1, 0</u>
3	2	<u>3, 1, 0</u>
4	3	<u>4, 2, 1, 0</u>
5	3	<u>5, 2, 1, 0</u>
6	3	<u>6, 3, 1, 0</u>
7	3	<u>7, 3, 1, 0</u>
8	4	<u>8, 4, 2, 1, 0</u>

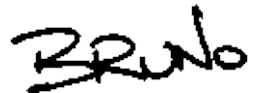
By inspection we determine that the number of recursive calls made in which the second argument is non-zero is $\lfloor \log_2 n \rfloor + 1$. Furthermore, depending on whether

the argument is odd or even, each of these calls contributes either 18 or 20 cycles. The pattern emerging in Table □ suggests that, on average just as many of the recursive calls result in an even number as result in an odd one. The final call (zero argument) adds another 5 cycles. So, on average, we can expect the running time of Program □ to be

$$T(n) = 19(\lfloor \log_2 n \rfloor + 1) + 5. \quad (2.17)$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Geometric Series Summation Yet Again

In this example we consider the problem of computing a *geometric series summation* for the last time. We have already seen two algorithms to compute this summation in Sections □ and □ (Programs □ and □).

An algorithm to compute a geometric series summation using the closed-form expression (Equation □) is given in Program □. This algorithm makes use of Program □ to compute x^{n+1} .

```
1 def geometricSeriesSum(x, n):
2     return (power(x, n + 1) - 1) / (x - 1)
```

Program: Program to compute $\sum_{i=0}^n x^i$ using the closed-form expression.

To determine the average running time of Program □ we will make use of Equation □, which gives the average running time for the power method which is called on line 2. In this case, the arguments are x and $n+1$, so the running time of the call to power is $19(\lfloor \log_2(n+1) \rfloor + 1) + 5$. Adding to this the additional work done on line 2 gives the average running time for Program □:

$$T(n) = 19(\lfloor \log_2(n+1) \rfloor + 1) + 18.$$

The running times of the three programs which compute the geometric series summation presented in this chapter are tabulated below in Table □ and are plotted for $1 \leq n \leq 100$ in Figure □. The plot shows that, according to our simplified model of the computer, Program □ has the best running time for $n < 4$. However as n increases, Program □ is clearly the fastest of the three and Program □ is the slowest for all values of n .

Table: Running times of Programs □, □ and □.

program	T(n)
Program □	$(\frac{11}{2}n^2 + \frac{47}{2}n + 24)$

Program \square $13n+22$
Program \square $19(\lfloor \log_2(n+1) \rfloor + 1) + 18$

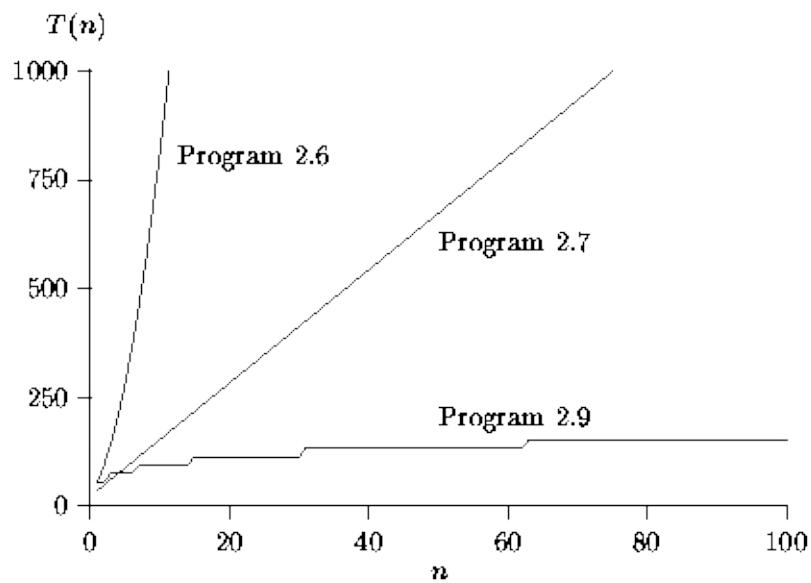


Figure: Plot of running time vs. n for Programs \square , \square and \square .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Exercises

1. Determine the running times predicted by the detailed model of the computer given in Section □ for each of the following program fragments:

```
1. i = 0
   while i < n:
       k += 1
       i += 1
```

```
2. i = 1
   while i < n:
       k += 1
       i = i * 2
```

```
3. i = n - 1
   while i != 0:
       k += 1
       i = i / 2
```

```
4. i = 0
   while i < n:
       if i % 2 == 0:
           k += 1
       i += 1
```

```
5. i = 0
   while i < n:
       j = 0
       while j < n:
           k += 1
           j += 1
       i += 1
```

```
6. i = 0
   while i < n:
       j = i
       while j < n:
           k += 1
           j += 1
       i += 1
```

```
7. i = 0
   while i < n:
```

```

j = 0
while j < i * i:
    k += 1
    j += 1
    i += 1

```

2. Repeat Exercise □, this time using the simplified model of the computer given in Section □.

3. Prove by induction the following summation formulas:

$$1. \sum_{i=0}^n i = \frac{n(n+1)}{2}$$

$$2. \sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$3. \sum_{i=0}^n i^3 = \frac{n^2(n+1)^2}{4}$$

4. Evaluate each of the following series summations:

$$1. \sum_{i=0}^n 2^i$$

$$2. \sum_{i=0}^n \left(\frac{1}{2}\right)^i$$

$$3. \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i$$

$$4. \sum_{i=-\infty}^n 2^i$$

5. Show that $\sum_{i=0}^{\infty} a^i = \frac{1}{1-a}$, for $0 \leq a < 1$. **Hint:** Let $S_n = \sum_{i=0}^n a^i$ and show that $\lim_{n \rightarrow \infty} (S_n - aS_n) = 1$.

6. Show that $\sum_{i=0}^{\infty} i/2^i = 2$. **Hint:** Let $S_n = \sum_{i=0}^n i/2^i$ and show that the difference $2S_n - S_n$ is (approximately) a geometric series summation.

7. Solve each of the following recurrences by repeated substitution:

$$1. T(n) = \begin{cases} 1 & n = 0, \\ T(n-1) + 1 & n > 0 \end{cases}$$

$$2. T(n) = \begin{cases} 1 & n \leq a, a > 0, \\ T(n-a) + 1 & n > a. \end{cases}$$

$$3. \quad T(n) = \begin{cases} 1 & n = 0, \\ 2T(n-1) + 1 & n > 0 \end{cases}$$

$$4. \quad T(n) = \begin{cases} 1 & n = 0, \\ 2T(n-1) + n & n > 0 \end{cases}$$

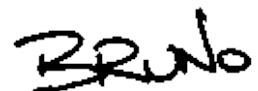
$$5. \quad T(n) = \begin{cases} 1 & n = 1, \\ T(n/2) + 1 & n > 1. \end{cases}$$

$$6. \quad T(n) = \begin{cases} 1 & n = 1, \\ 2T(n/2) + 1 & n > 1. \end{cases}$$

$$7. \quad T(n) = \begin{cases} 1 & n = 1, \\ 2T(n/2) + n & n > 1. \end{cases}$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Projects

1. Write a non-recursive method to compute the factorial of n according to Equation □. Calculate the running time predicted by the detailed model given in Section □ and the simplified model given in Section □.
2. Write a non-recursive method to compute x^n according to Equation □. Calculate the running time predicted by the detailed model given in Section □ and the simplified model given in Section □.
3. Write a program that determines the values of the timing parameters of the detailed model ($\tau_{\text{fetch}}, \tau_{\text{store}}, \tau_+, \tau_-, \tau_x, \tau_\div, \tau_<, \tau_{\text{call}}, \tau_{\text{return}}, \tau_{\text{new}}$, and $\tau_{[\cdot]}$) for the machine on which it is run.
4. Using the program written for Project □, determine the timing parameters of the detailed model for your computer. Then, measure the actual running times of Programs □, □ and □ and compare the measured results with those predicted by Equations □, □ and □ (respectively).
5. Given a sequence of n integers, $\{a_0, a_1, \dots, a_{n-1}\}$, and a small positive integer k , write an algorithm to compute

$$\sum_{i=0}^{n-1} 2^{ki} a_i,$$

without multiplication. **Hint:** Use Horner's rule and bitwise shifts.

6. Verify Equation □ experimentally as follows: Generate a large number of random sequences of length n , $\{a_0, a_1, a_2, \dots, a_{n-1}\}$. For each sequence, test the hypothesis that the probability that a_i is larger than all its predecessors in the sequence is $p_i = 1/(i+1)$. (For a good source of random numbers, see Section □).

Asymptotic Notation

Suppose we are considering two algorithms, A and B , for solving a given problem. Furthermore, let us say that we have done a careful analysis of the running times of each of the algorithms and determined them to be $T_A(n)$ and $T_B(n)$, respectively, where n is a measure of the problem size. Then it should be a fairly simple matter to compare the two functions $T_A(n)$ and $T_B(n)$ to determine which algorithm is *the best!*

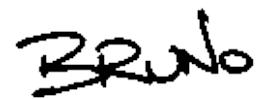
But is it really that simple? What exactly does it mean for one function, say $T_A(n)$, to be *better than* another function, $T_B(n)$? One possibility arises if we know the problem size *a priori*. For example, suppose the problem size is n_0 and $T_A(n_0) < T_B(n_0)$. Then clearly algorithm A is better than algorithm B for problem size n_0 .

In the general case, we have no *a priori* knowledge of the problem size. However, if it can be shown, say, that $T_A(n) \leq T_B(n)$ for all $n \geq 0$, then algorithm A is better than algorithm B regardless of the problem size.

Unfortunately, we usually don't know the problem size beforehand, nor is it true that one of the functions is less than or equal the other over the entire range of problem sizes. In this case, we consider the *asymptotic behavior* of the two functions for very large problem sizes.

-
- [An Asymptotic Upper Bound-Big Oh](#)
 - [An Asymptotic Lower Bound-Omega](#)
 - [More Notation-Theta and Little Oh](#)
 - [Asymptotic Analysis of Algorithms](#)
 - [Exercises](#)
 - [Projects](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

An Asymptotic Upper Bound-Big Oh

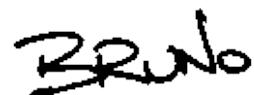
In 1892, P. Bachmann invented a notation for characterizing the asymptotic behavior of functions. His invention has come to be known as *big oh notation*:

Definition (Big Oh) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that `` $f(n)$ is big oh $g(n)$,'' which we write $f(n)=O(g(n))$, if there exists an integer n_0 and a constant $c>0$ such that for all integers $n \geq n_0$, $f(n) \leq cg(n)$.

- [A Simple Example](#)
 - [Big Oh Fallacies and Pitfalls](#)
 - [Properties of Big Oh](#)
 - [About Polynomials](#)
 - [About Logarithms](#)
 - [Tight Big Oh Bounds](#)
 - [More Big Oh Fallacies and Pitfalls](#)
 - [Conventions for Writing Big Oh Expressions](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



A Simple Example

Consider the function $f(n)=8n+128$ shown in Figure □. Clearly, $f(n)$ is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = O(n^2)$. According to Definition □, in order to show this we need to find an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cn^2$.

It does not matter what the particular constants are--as long as they exist! For example, suppose we choose $c=1$. Then

$$\begin{aligned} f(n) \leq cn^2 &\Rightarrow 8n + 128 \leq n^2 \\ &\Rightarrow 0 \leq n^2 - 8n - 128 \\ &\Rightarrow 0 \leq (n - 16)(n + 8). \end{aligned}$$

Since $(n+8) > 0$ for all values of $n \geq 0$, we conclude that $(n_0 - 16) \geq 0$. That is, $n_0 = 16$.

So, we have that for $c=1$ and $n_0 = 16$, $f(n) \leq cn^2$ for all integers $n \geq n_0$. Hence, $f(n) = O(n^2)$. Figure □ clearly shows that the function $f(n) = n^2$ is greater than the function $f(n)=8n+128$ to the right of $n=16$.

Of course, there are many other values of c and n_0 that will do. For example, $c=2$ and $n_0 = 2 + 4\sqrt{17} \approx 10.2$ will do, as will $c=4$ and $n_0 = 1 + \sqrt{33} \approx 6.7$. (See Figure □).

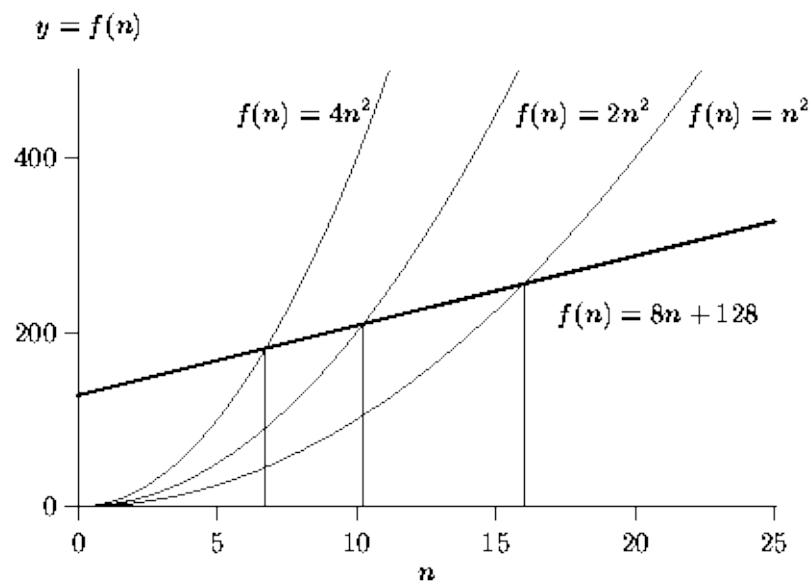


Figure: Showing that $f(n) = 8n + 128 = O(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Big Oh Fallacies and Pitfalls

Unfortunately, the way we write big oh notation can be misleading to the naive reader. This section presents two fallacies which arise because of a misinterpretation of the notation.

Fallacy Given that $f_1(n) = O(g(n))$ and $f_2(n) = O(g(n))$, then $f_1(n) = f_2(n)$.

Consider the equations:

$$\begin{aligned}f_1(n) &= h(n^2) \\f_2(n) &= h(n^2).\end{aligned}$$

Clearly, it is reasonable to conclude that $f_1(n) = f_2(n)$.

However, consider these equations:

$$\begin{aligned}f_1(n) &= O(n^2) \\f_2(n) &= O(n^2).\end{aligned}$$

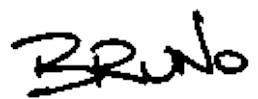
It does not follow that $f_1(n) = f_2(n)$. For example, $f_1(n) = n$ and $f_2(n) = n^2$ are both $O(n^2)$, but they are not equal.

Fallacy If $f(n)=O(g(n))$, then $g(n) = O^{-1}(f(n))$.

Consider functions f , g , and h , such that $f(n)=h(g(n))$. It is reasonable to conclude that $g(n) = h^{-1}(f(n))$ provided that $h(\cdot)$ is an invertible function. However, while we may write $f(n)=O(h(n))$, the equation $g(n) = O^{-1}(f(n))$ is nonsensical and meaningless. Big oh is not a mathematical function, so it has no inverse!

The reason for these difficulties is that we should read the notation $f(n) = O(n^2)$ as `` $f(n)$ is big oh n squared" not `` $f(n)$ equals big oh of n squared." The equal sign in the expression does not really denote mathematical equality! And the use of the functional form, $O(\cdot)$, does not really mean that O is a mathematical function!

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

Properties of Big Oh

In this section we examine some of the mathematical properties of big oh. In particular, suppose we know that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$.

- What can we say about the asymptotic behavior of the *sum* of $f_1(n)$ and $f_2(n)$? (Theorems \square and \square).
- What can we say about the asymptotic behavior of the *product* of $f_1(n)$ and $f_2(n)$? (Theorems \square and \square).
- How are $f_1(n)$ and $f_2(n)$ related when $g_1(n) = f_2(n)$? (Theorem \square).

The first theorem addresses the asymptotic behavior of the sum of two functions whose asymptotic behaviors are known:

Theorem If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))).$$

extbf{Proof} By Definition \square , there exist two integers, n_1 and n_2 and two constants c_1 and c_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n \geq n_1$ and $f_2(n) \leq c_2 g_2(n)$ for $n \geq n_2$.

Let $n_0 = \max(n_1, n_2)$ and $c_0 = 2 \max(c_1, c_2)$. Consider the sum $f_1(n) + f_2(n)$ for $n \geq n_0$:

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n), \quad n \geq n_0 \\ &\leq c_0(g_1(n) + g_2(n))/2 \\ &\leq c_0 \max(g_1(n), g_2(n)). \end{aligned}$$

Thus, $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.

According to Theorem \square , if we know that functions $f_1(n)$ and $f_2(n)$ are $O(g_1(n))$ and $O(g_2(n))$, respectively, the sum $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$. The meaning of $\max(g_1(n), g_2(n))$ in this context is the *function* $h(n)$ where $h(n) = \max(g_1(n), g_2(n))$ for integers all $n \geq 0$.

For example, consider the functions $g_1(n) = 1$ and $g_2(n) = 2 \cos^2(n\pi/2)$. Then

$$\begin{aligned}
h(n) &= \max(g_1(n), g_2(n)) \\
&= \max(1, 2\cos^2(n\pi/2)) \\
&= \begin{cases} 1 & n \text{ is even,} \\ 2 & n \text{ is odd.} \end{cases}
\end{aligned}$$

Theorem \square helps us simplify the asymptotic analysis of the sum of functions by allowing us to drop the \max required by Theorem \square in certain circumstances:

Theorem If $f(n) = f_1(n) + f_2(n)$ in which $f_1(n)$ and $f_2(n)$ are both non-negative for all integers $n \geq 0$ such that $\lim_{n \rightarrow \infty} f_2(n)/f_1(n) = L$ for some limit $L \geq 0$, then $f(n) = O(f_1(n))$.

extbfProof According to the definition of limits , the notation

$$\lim_{n \rightarrow \infty} \frac{f_2(n)}{f_1(n)} = L$$

means that, given any arbitrary positive value ϵ , it is possible to find a value n_0 such that for all $n \geq n_0$

$$\left| \frac{f_2(n)}{f_1(n)} - L \right| \leq \epsilon.$$

Thus, if we chose a particular value, say ϵ_0 , then there exists a corresponding n_0 such that

$$\begin{aligned}
\left| \frac{f_2(n)}{f_1(n)} - L \right| &\leq \epsilon_0, \quad n \geq n_0 \\
\frac{f_2(n)}{f_1(n)} - L &\leq \epsilon_0 \\
f_2(n) &\leq (\epsilon_0 + L)f_1(n).
\end{aligned}$$

Consider the sum $f(n) = f_1(n) + f_2(n)$:

$$\begin{aligned}
f(n) &= f_1(n) + f_2(n) \\
&\leq c_1 f_1(n) + c_2 f_2(n) \\
&\leq c_1 f_1(n) + c_2(\epsilon_0 + L)f_1(n), \quad n \geq n_0 \\
&\leq c_0 f_1(n)
\end{aligned}$$

where $c_0 = c_1 + c_2(\epsilon_0 + L)$. Thus, $f(n) = O(f_1(n))$.

Consider a pair of functions $f_1(n)$ and $f_2(n)$, which are known to be $O(g_1(n))$ and $O(g_2(n))$, respectively. According to Theorem \square , the sum $f(n) = f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$. However, Theorem \square says that, if $\lim_{n \rightarrow \infty} f_2(n)/f_1(n)$ exists, then the sum $f(n)$ is simply $O(f_1(n))$ which, by the transitive property (see Theorem \square below), is $O(g_1(n))$.

In other words, if the ratio $f_1(n)/f_2(n)$ asymptotically approaches a constant as n gets large, we can say that $f_1(n) + f_2(n)$ is $O(g_1(n))$, which is often a lot simpler than $O(\max(g_1(n), g_2(n)))$.

Theorem \square is particularly useful result. Consider $f_1(n) = n^3$ and $f_2(n) = n^2$.

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{f_2(n)}{f_1(n)} &= \lim_{n \rightarrow \infty} \frac{n^2}{n^3} \\ &= \lim_{n \rightarrow \infty} \frac{1}{n} \\ &= 0.\end{aligned}$$

From this we can conclude that $f_1(n) + f_2(n) = n^3 + n^2 = O(n^3)$. Thus, Theorem \square suggests that the sum of a series of powers of n is $O(n^m)$, where m is the largest power of n in the summation. We will confirm this result in Section \square below.

The next theorem addresses the asymptotic behavior of the product of two functions whose asymptotic behaviors are known:

Theorem If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n)).$$

extbfProof By Definition \square , there exist two integers, n_1 and n_2 and two constants c_1 and c_2 such that $f_1(n) \leq c_1 g_1(n)$ for $n \geq n_1$ and $f_2(n) \leq c_2 g_2(n)$ for $n \geq n_2$.

Furthermore, by Definition \square , $f_1(n)$ and $f_2(n)$ are both non-negative for all integers $n \geq 0$.

Let $n_0 = \max(n_1, n_2)$ and $c_0 = c_1 c_2$. Consider the product $f_1(n) \times f_2(n)$ for $n \geq n_0$:

$$\begin{aligned} f_1(n) \times f_2(n) &\leq c_1 g_1(n) \times c_2 g_2(n), \quad n \geq n_0 \\ &\leq c_0(g_1(n) \times g_2(n)). \end{aligned}$$

Thus, $f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n))$.

Theorem \square describes a simple, but extremely useful property of big oh. Consider the functions $f_1(n) = n^3 + n^2 + n + 1 = O(n^3)$ and $f_2(n) = n^2 + n + 1 = O(n^2)$. By Theorem \square , the asymptotic behavior of the product $f_1(n) \times f_2(n)$ is $O(n^3 \times n^2) = O(n^5)$. That is, we are able to determine the asymptotic behavior of the product without having to go through the gory details of calculating that $f_1(n) \times f_2(n) = n^5 + 2n^4 + 3n^3 + 3n^2 + 2n + 1$.

The next theorem is closely related to the preceding one in that it also shows how big oh behaves with respect to multiplication.

Theorem If $f_1(n) = O(g_1(n))$ and $g_2(n)$ is a function whose value is non-negative for integers $n \geq 0$, then

$$f_1(n) \times g_2(n) = O(g_1(n) \times g_2(n)).$$

extbf{Proof} By Definition \square , there exist integers n_0 and constant c_0 such that $f_1(n) \leq c_0 g_1(n)$ for $n \geq n_0$. Since $g_2(n)$ is never negative,

$$f_1(n) \times g_2(n) \leq c_0 g_1(n) \times g_2(n), \quad n \geq n_0.$$

Thus, $f_1(n) \times g_2(n) = O(g_1(n) \times g_2(n))$.

Theorem \square applies when we multiply a function, $f_1(n)$, whose asymptotic behavior is known to be $O(g_1(n))$, by another function $g_2(n)$. The asymptotic behavior of the result is simply $O(g_1(n) \times g_2(n))$.

One way to interpret Theorem \square is that it allows us to do the following mathematical manipulation:

$$\begin{aligned} f_1(n) = O(g_1(n)) &\Rightarrow f_1(n) \times g_2(n) = O(g_1(n)) \times g_2(n) \\ &\Rightarrow f_1(n) \times g_2(n) = O(g_1(n) \times g_2(n)). \end{aligned}$$

That is, Fallacy \square notwithstanding, we can multiply both sides of the ``equation''

by $\frac{g_2(n)}{g_2(n)}$ and the ``equality'' still holds. Furthermore, when we multiply $O(g_1(n))$ by $\frac{g_2(n)}{g_2(n)}$, we simply bring the $\frac{g_2(n)}{g_2(n)}$ inside the $O(\cdot)$.

The last theorem in this section introduces the *transitive property* of big oh:

Theorem (Transitive Property) If $f(n)=O(g(n))$ and $g(n)=O(h(n))$ then $f(n)=O(h(n))$.

extbfProof By Definition \square , there exist two integers, n_1 and n_2 and two constants c_1 and c_2 such that $f(n) \leq c_1 g(n)$ for $n \geq n_1$ and $g(n) \leq c_2 h(n)$ for $n \geq n_2$.

Let $n_0 = \max(n_1, n_2)$ and $c_0 = c_1 c_2$. Then

$$\begin{aligned} f(n) &\leq c_1 g(n), \quad n \geq n_1 \\ &\leq c_1 c_2 h(n), \quad n \geq n_0 \\ &\leq c_0 h(n). \end{aligned}$$

Thus, $f(n)=O(h(n))$.

The transitive property of big oh is useful in conjunction with Theorem \square .

Consider $f_1(n) = 5n^3$ which is clearly $O(n^3)$. If we add to $f_1(n)$ the function $f_2(n) = 3n^2$, then by Theorem \square , the sum $f_1(n) + f_2(n)$ is $O(f_1(n))$ because $\lim_{n \rightarrow \infty} f_2(n)/f_1(n) = 0$. That is, $f_1(n) + f_2(n) = O(f_1(n))$. The combination of the fact that $f_1(n) = O(n^3)$ and the transitive property of big oh, allows us to conclude that the sum is $O(n^3)$.

About Polynomials

In this section we examine the asymptotic behavior of polynomials in n . In particular, we will see that as n gets large, the term involving the highest power of n will dominate all the others. Therefore, the asymptotic behavior is determined by that term.

Theorem Consider a polynomial in n of the form

$$\begin{aligned} f(n) &= \sum_{i=0}^m a_i n^i \\ &= a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0 \end{aligned}$$

where $a_m > 0$. Then $f(n) = O(n^m)$.

extbfProof Each of the terms in the summation is of the form $a_i n^i$. Since n is non-negative, a particular term will be negative only if $a_i < 0$. Hence, for each term in the summation, $a_i n^i \leq |a_i| n^i$. Recall too that we have stipulated that the coefficient of the largest power of n is positive, i.e., $a_m > 0$.

$$\begin{aligned} f(n) &\leq \sum_{i=0}^m |a_i| n^i \\ &\leq n^m \sum_{i=0}^m |a_i| n^{i-m}, \quad n \geq 1 \\ &\leq n^m \sum_{i=0}^m |a_i| \frac{1}{n^{m-i}}. \end{aligned}$$

Note that for integers $n \geq 1$, $1/(n^{m-i}) \leq 1$ for $0 \leq i \leq m$. Thus

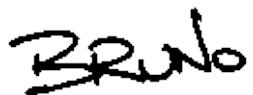
$$f(n) \leq \underbrace{n^m}_{g(n)} \underbrace{\sum_{i=0}^m |a_i|}_{c} \underbrace{\frac{1}{n^{m-i}}}_{n^0}. \quad (3.1)$$

From Equation \square we see that we have found the constants $n_0 = 1$ and $c = \sum_{i=0}^m |a_i|$, such that for all $n \geq n_0$, $f(n) = \sum_{i=0}^n a_i n^i \leq cn^m$. Thus, $f(n) = O(n^m)$.

This property of the asymptotic behavior of polynomials is used extensively. In fact, whenever we have a function, which is a polynomial in n ,
$$f(n) = a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0$$
 we will immediately ``drop'' the less significant terms (i.e., terms involving powers of n which are less than m), as well as the leading coefficient, a_m , to write $f(n) = O(n^m)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



About Logarithms

In this section we determine the asymptotic behavior of logarithms. Interestingly, despite the fact that $\log n$ diverges as n gets large, $\log n < n$ for all integers $n \geq 0$. Hence, $\log n = O(n)$. Furthermore, as the following theorem will show, $\log n$ raised to any integer power $k \geq 1$ is still $O(n)$.

Theorem For every integer $k \geq 1$, $\log^k n = O(n)$.

This result follows immediately from Theorem \square and the observation that for all integers $k \geq 1$,

$$\lim_{n \rightarrow \infty} \frac{\log^k n}{n} = 0. \quad (3.2)$$

This observation can be proved by induction as follows:

Base Case Consider the limit

$$\lim_{n \rightarrow \infty} \frac{\log^k n}{n}$$

for the case $k=1$. Using L'Hôpital's rule \diamond we see that

$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log n}{n} &= \lim_{n \rightarrow \infty} \frac{1}{n} \cdot \frac{1}{\ln 10} \\ &= 0 \end{aligned}$$

Inductive Hypothesis Assume that Equation \square holds for $k = 1, 2, \dots, m$. Consider the case $k=m+1$. Using L'Hôpital's rule \diamond we see that

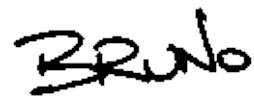
$$\begin{aligned} \lim_{n \rightarrow \infty} \frac{\log^{m+1} n}{n} &= \lim_{n \rightarrow \infty} \frac{m \log^m n \times \frac{1}{n \ln 10}}{1} \\ &= \frac{m}{\ln 10} \lim_{n \rightarrow \infty} \frac{\log^m n}{n} \\ &= 0 \end{aligned}$$

Therefore, by induction on m , Equation \square holds for all integers $k \geq 1$.

For example, using this property of logarithms together with the rule for determining the asymptotic behavior of the product of two functions (Theorem □), we can determine that since $\log n = O(n)$, then $n \log n = O(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Tight Big Oh Bounds

Big oh notation characterizes the asymptotic behavior of a function by providing an upper bound on the rate at which the function grows as n gets large.

Unfortunately, the notation does not tell us how close the actual behavior of the function is to the bound. That is, the bound might be very close (tight) or it might be overly conservative (loose).

The following definition tells us what makes a bound tight, and how we can test to see whether a given asymptotic bound is the best one available.

Definition (Tightness) Consider a function $f(n)=O(g(n))$. If for every function $h(n)$ such that $f(n)=O(h(n))$ it is also true that $g(n)=O(h(n))$, then we say that $g(n)$ is a *tight asymptotic bound* on $f(n)$.

For example, consider the function $f(n)=8n+128$. In Section □, it was shown that $f(n) = O(n^2)$. However, since $f(n)$ is a polynomial in n , Theorem □ tells us that $f(n)=O(n)$. Clearly $O(n)$ is a tighter bound on the asymptotic behavior of $f(n)$ than is $O(n^2)$.

By Definition □, in order to show that $g(n)=n$ is a tight bound on $f(n)$, we need to show that for every function $h(n)$ such that $f(n)=O(h(n))$, it is also true that $g(n)=O(h(n))$.

We will show this result using proof by contradiction: Assume that $g(n)$ is *not* a tight bound for $f(n)=8n+128$. Then there exists a function $h(n)$ such that $f(n)=8n+128=O(h(n))$, but for which $g(n) \neq O(h(n))$. Since $8n+128=O(h(n))$, by the definition of big oh there exist positive constants c and n_0 such that $8n+128 \leq ch(n)$ for all $n \geq n_0$.

Clearly, for all $n \geq 0$, $n \leq 8n+128$. Therefore, $g(n) \leq ch(n)$. But then, by the definition of big oh, we have that $g(n)=O(h(n))$ --a contradiction! Therefore, the bound $f(n)=O(n)$ is a tight bound.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

More Big Oh Fallacies and Pitfalls

The purpose of this section is to dispel some common misconceptions about big oh. The next fallacy is related to the selection of the constants c and n_0 used to show a big oh relation.

Fallacy Consider non-negative functions $f(n)$, $\frac{g_1(n)}{f(n)}$, and $\frac{g_2(n)}{f(n)}$, such that $f(n) = g_1(n) \times g_2(n)$. Since $\frac{f(n)}{g_1(n)} \leq c$ for all integers $n \geq 0$ if $c = g_2(n)$, then by Definition \square $f(n) = O(g_1(n))$.

This fallacy often results from the following line of reasoning: Consider the function $f(n) = n \log n$. Let $c = \log n$ and $n_0 = 1$. Then $f(n)$ must be $O(n)$, since $f(n) \leq cn$ for all $n \geq n_0$. However, this line of reasoning is false because according to Definition \square , c must be a *positive constant*, not a function of n .

The next fallacy involves a misunderstanding of the notion of the *asymptotic upper bound*.

Fallacy Given non-negative functions $f_1(n)$, $f_2(n)$, $\frac{g_1(n)}{f_1(n)}$, and $\frac{g_2(n)}{f_2(n)}$, such that $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, and for all integers $n \geq 0$, $g_1(n) < g_2(n)$, then $f_1(n) < f_2(n)$.

This fallacy arises from the following line of reasoning. Consider the function $f_1(n) = O(n^2)$ and $f_2(n) = O(n^3)$. Since $n^2 \leq n^3$ for all values of $n \geq 1$, we might be tempted to conclude that $f_1(n) \leq f_2(n)$. In fact, such a conclusion is erroneous. For example, consider $f_1(n) = n$ and $f_2(n) = n^2 + 1$. Clearly, the former is $O(n^2)$ and the latter is $O(n^3)$. Clearly too, $f_2(n) \geq f_1(n)$ for all values of $n \geq 0$!

The previous fallacy essentially demonstrates that while we may know how the asymptotic upper bounds on two functions are related, we don't necessarily know, in general, the relative behavior of the two bounded functions.

This fallacy often arises in the comparison of the performance of algorithms. Suppose we are comparing two algorithms, A and B , to solve a given problem

and we have determined that the running times of these algorithms are $T_A(n) = O(g_1(n))$ and $T_B(n) = O(g_2(n))$, respectively. Fallacy \square demonstrates that it is an error to conclude from the fact that $g_1(n) \leq g_2(n)$ for all $n \geq 0$ that algorithm A will solve the problem faster than algorithm B for all problem sizes.

But what about any one specific problem size? Can we conclude that for a given problem size, say n_0 , that algorithm A is faster than algorithm B? The next fallacy addresses this issue.

Fallacy Given non-negative functions $f_1(n)$, $f_2(n)$, $g_1(n)$, and $g_2(n)$, such that $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, and for all integers $n \geq 0$, $g_1(n) < g_2(n)$, there exists an integer n_0 for which then $f_1(n_0) < f_2(n_0)$.

This fallacy arises from a similar line of reasoning as the preceding one.

Consider the function $f_1(n) = O(n^2)$ and $f_2(n) = O(n^3)$. Since $n^2 \leq n^3$ for all values of $n \geq 1$, we might be tempted to conclude that there exists a value n_0 for which $f_1(n_0) \leq f_2(n_0)$. Such a conclusion is erroneous. For example, consider $f_1(n) = n^2$ and $f_2(n) = n + 1$. Clearly, the former is $O(n^2)$ and the latter is $O(n^3)$. Clearly too, since $f_2(n) \geq f_1(n)$ for all values of $n \geq 0$, there does not exist any value $n_0 \geq 0$ for which $f_1(n_0) \leq f_2(n_0)$.

The final fallacy shows that not all functions are *commensurate*:

Fallacy Given two non-negative functions $f(n)$ and $g(n)$ then either $f(n)=O(g(n))$ or $g(n)=O(f(n))$.

This fallacy arises from thinking that the relation $O(\cdot)$ is like \leq and can be used to compare any two functions. However, not all functions are commensurate. \diamond

Consider the following functions:

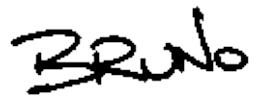
$$f(n) = \begin{cases} n & n \text{ is even,} \\ 0 & n \text{ is odd.} \end{cases}$$

$$g(n) = \begin{cases} 0 & n \text{ is even,} \\ n & n \text{ is odd.} \end{cases}$$

Clearly, there does not exist a constant c for which $f(n) \leq cg(n)$ for any even integer n , since the $g(n)$ is zero and $f(n)$ is not. Conversely, there does not exist a constant c for which $g(n) \leq cf(n)$ for any odd integer n , since the $f(n)$ is zero and $g(n)$ is not. Hence, neither $f(n)=O(g(n))$ nor $g(n)=O(f(n))$ is true.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Conventions for Writing Big Oh Expressions

Certain conventions have evolved which concern how big oh expressions are normally written:

- First, it is common practice when writing big oh expressions to drop all but the most significant terms. Thus, instead of $O(n^2 + n \log n + n)$ we simply write $O(n^2)$.
- Second, it is common practice to drop constant coefficients. Thus, instead of $O(3n^2)$, we simply write $O(n^2)$. As a special case of this rule, if the function is a constant, instead of, say $O(1024)$, we simply write $O(1)$.

Of course, in order for a particular big oh expression to be the most useful, we prefer to find a *tight* asymptotic bound (see Definition □). For example, while it is not wrong to write $f(n) = n = O(n^3)$, we prefer to write $f(n)=O(n)$, which is a tight bound.

Certain big oh expressions occur so frequently that they are given names. Table □ lists some of the commonly occurring big oh expressions and the usual name given to each of them.

Table: The names of common big oh expressions.

expression	name
$O(1)$	constant
$O(\log n)$	logarithmic
$O(\log^2 n)$	log squared
$O(n)$	linear
$O(n \log n)$	$n \log n$

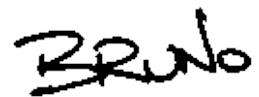
$O(n^2)$ quadratic

$O(n^3)$ cubic

$O(2^n)$ exponential

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



An Asymptotic Lower Bound-Omega

The big oh notation introduced in the preceding section is an asymptotic *upper bound*. In this section, we introduce a similar notation for characterizing the asymptotic behavior of a function, but in this case it is a *lower bound*.

Definition (Omega) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that `` $f(n)$ is omega $g(n)$," which we write $f(n) = \Omega(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \geq cg(n)$.

The definition of omega is almost identical to that of big oh. The only difference is in the comparison--for big oh it is $f(n) \leq cg(n)$; for omega, it is $f(n) \geq cg(n)$. All of the same conventions and caveats apply to omega as they do to big oh.

-
- [A Simple Example](#)
 - [About Polynomials Again](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



A Simple Example

Consider the function $f(x) = 5x^2 - 64x + 256$ which is shown in Figure □. Clearly, $f(n)$ is non-negative for all integers $n \geq 0$. We wish to show that $f(n) = \Omega(n^2)$. According to Definition □, in order to show this we need to find an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \geq cn^2$.

As with big oh, it does not matter what the particular constants are--as long as they exist! For example, suppose we choose $c=1$. Then

$$\begin{aligned} f(n) \geq cn^2 &\Rightarrow 5n^2 - 64n + 256 \geq n^2 \\ &\Rightarrow 4n^2 - 64n + 256 \geq 0 \\ &\Rightarrow 4(n - 8)^2 \geq 0. \end{aligned}$$

Since $(n - 8)^2 \geq 0$ for all values of $n \geq 0$, we conclude that $n_0 = 0$.

So, we have that for $c=1$ and $n_0 = 0$, $f(n) \geq cn^2$ for all integers $n \geq n_0$. Hence, $f(n) = \Omega(n^2)$. Figure □ clearly shows that the function $f(n) = n^2$ is less than the function $f(n) = 5n^2 - 64n + 256$ for all values of $n \geq 0$. Of course, there are many other values of c and n_0 that will do. For example, $c=2$ and $n_0 = 16$.

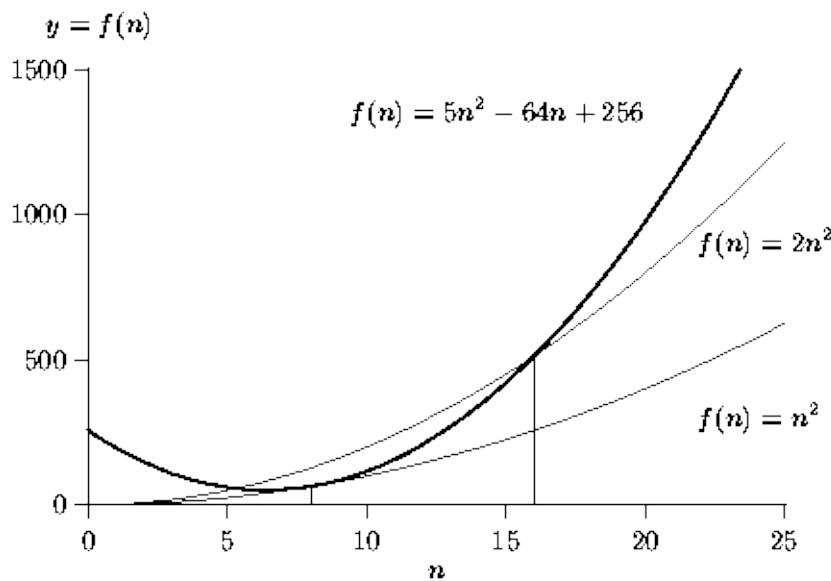
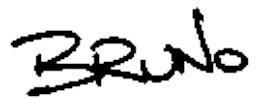


Figure: Showing that $f(n) = 4n^2 - 64n + 288 = \Omega(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



About Polynomials Again

In this section we reexamine the asymptotic behavior of polynomials in n . In Section □ we showed that $f(n) = O(n^m)$. That is, $f(n)$ grows asymptotically no more quickly than n^m . This time we are interested in the asymptotic lower bound rather than the asymptotic upper bound. We will see that as n gets large, the term involving n^m also dominates the lower bound in the sense that $f(n)$ grows asymptotically *as quickly* as n^m . That is, that $f(n) = \Omega(n^m)$.

Theorem Consider a polynomial in n of the form

$$\begin{aligned} f(n) &= \sum_{i=0}^m a_i n^i \\ &= a_m n^m + a_{m-1} n^{m-1} + \cdots + a_2 n^2 + a_1 n + a_0 \end{aligned}$$

where $a_m > 0$. Then $f(n) = \Omega(n^m)$.

extbfProof We begin by taking the term $a_m n^m$ out of the summation:

$$\begin{aligned} f(n) &= \sum_{i=0}^m a_i n^i \\ &= a_m n^m + \sum_{i=0}^{m-1} a_i n^i. \end{aligned}$$

Since, n is a non-negative integer and $a_m > 0$, the term $a_m n^m$ is positive. For each of the remaining terms in the summation, $a_i n^i \geq -|a_i| n^i$. Hence

$$\begin{aligned} f(n) &\geq a_m n^m - \sum_{i=0}^{m-1} |a_i| n^i \\ &\geq a_m n^m - n^{m-1} \sum_{i=0}^{m-1} |a_i| n^{i-(m-1)}, \quad n \geq 1 \\ &\geq a_m n^m - n^{m-1} \sum_{i=0}^{m-1} |a_i| \frac{1}{n^{(m-1)-i}}. \end{aligned}$$

Note that for integers $n \geq 1$, $1/(n^{(m-1)-i}) \leq 1$ for $0 \leq i \leq (m-1)$. Thus

$$\begin{aligned} f(n) &\geq a_m n^m - n^{m-1} \sum_{i=0}^{m-1} |a_i|, \quad n \geq 1 \\ &\geq n^m \left(a_m - \frac{1}{n} \sum_{i=0}^{m-1} |a_i| \right). \end{aligned}$$

Consider the term in parentheses on the right. What we need to do is to find a positive constant c and an integer n_0 so that for all integers $n \geq n_0$ this term is greater than or equal to c :

$$a_m - \frac{1}{n} \sum_{i=0}^{m-1} |a_i| \geq a_m - \frac{1}{n_0} \sum_{i=0}^{m-1} |a_i|$$

We choose the value n_0 for which the term is greater than zero:

$$\begin{aligned} a_m - \frac{1}{n_0} \sum_{i=0}^{m-1} |a_i| &> 0 \\ n_0 &> \frac{1}{a_m} \sum_{i=0}^{m-1} |a_i| \end{aligned}$$

The value $n_0 = \left\lceil \frac{1}{a_m} \sum_{i=0}^{m-1} |a_i| \right\rceil + 1$ will suffice! Thus

$$\begin{aligned} f(n) &\geq \underbrace{n^m}_{g(n)} \underbrace{\left(a_m - \frac{1}{n_0} \sum_{i=0}^{m-1} |a_i| \right)}_c, \quad n \geq n_0 \\ n_0 &= \left\lceil \frac{1}{a_m} \sum_{i=0}^{m-1} |a_i| \right\rceil + 1. \end{aligned} \tag{3.3}$$

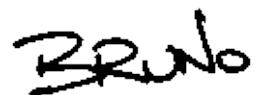
From Equation \square we see that we have found the constants n_0 and c , such that for all $n \geq n_0$, $f(n) = \sum_{i=0}^n a_i n^m \geq c n^m$. Thus, $f(n) = \Omega(n^m)$.

This property of the asymptotic behavior of polynomials is used extensively. In fact, whenever we have a function, which is a polynomial in n , $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_2 n^2 + a_1 n + a_0$ we will immediately ``drop'' the less significant terms (i.e., terms involving powers of n which are less than m),

as well as the leading coefficient, a_m , to write $f(n) = \Omega(n^m)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



More Notation-Theta and Little Oh

This section presents two less commonly used forms of asymptotic notation. They are:

- A notation, $\Theta(\cdot)$, to describe a function which is both $O(g(n))$ and $\Omega(g(n))$, for the same $g(n)$. (Definition □).
- A notation, $o(\cdot)$, to describe a function which is $O(g(n))$ but not $\Theta(g(n))$, for the same $g(n)$. (Definition □).

Definition (Theta) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that `` $f(n)$ is theta $g(n)$," which we write $f(n) = \Theta(g(n))$, if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Recall that we showed in Section □ that a polynomial in n , say $f(n) = a_m n^m + a_{m-1} n^{m-1} + \dots + a_2 n^2 + a_1 n + a_0$, is $O(n^m)$. We also showed in Section □ that such a polynomial is $\Omega(n^m)$. Therefore, according to Definition □, we will write $f(n) = \Theta(n^m)$.

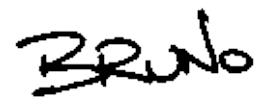
Definition (Little Oh) Consider a function $f(n)$ which is non-negative for all integers $n \geq 0$. We say that `` $f(n)$ is little oh $g(n)$," which we write $f(n) = o(g(n))$, if and only if $f(n)$ is $O(g(n))$ but $f(n)$ is not $\Theta(g(n))$.

Little oh notation represents a kind of *loose asymptotic bound* in the sense that if we are given that $f(n) = o(g(n))$, then we know that $g(n)$ is an asymptotic upper bound since $f(n) = O(g(n))$, but $g(n)$ is *not* an asymptotic lower bound since $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$. implies that $f(n) \neq \Omega(g(n))$. ◇

For example, consider the function $f(n) = n+1$. Clearly, $f(n) = O(n^2)$. Clearly too, $f(n) \neq \Omega(n^2)$, since no matter what c we choose, for large enough n , $cn^2 \geq n+1$. Thus, we may write $f(n) = n+1 = o(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Asymptotic Analysis of Algorithms

The previous chapter presents a detailed model of the computer which involves a number of different timing parameters-- τ_{fetch} , τ_{store} , τ_+ , τ_- , τ_{\times} , τ_{\div} , $\tau_{<}$, τ_{call} , τ_{return} , τ_{new} , and $\tau_{[\cdot]}$. We show that keeping track of the details is messy and tiresome. So we simplify the model by measuring time in clock cycles, and by assuming that each of the parameters is equal to one cycle. Nevertheless, keeping track of and carefully counting all the cycles is still a tedious task.

In this chapter we introduce the notion of asymptotic bounds, principally big oh, and examine the properties of such bounds. As it turns out, the rules for computing and manipulating big oh expressions greatly simplify the analysis of the running time of a program when all we are interested in is its asymptotic behavior.

For example, consider the analysis of the running time of Program \square , which is just Program \square again, an algorithm to evaluate a polynomial using Horner's rule.

```
1 def Horner(a, n, x):
2     result = a[n]
3     i = n - 1
4     while i >= 0:
5         result = result * x + a[i]
6         i -= 1
7     return result
```

Program: Program \square again.

Table: Computing the running time of Program \square .

statement	detailed model	simple big oh model	
2	$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$	5	$O(1)$
3	$2\tau_{\text{fetch}} + \tau_- + \tau_{\text{store}}$	4	$O(1)$
4	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n + 1)$	$3n+3$	$O(n)$

5	$(5\tau_{\text{fetch}} + \tau_{[]} + \tau_+ + \tau_x + \tau_{\text{store}}) \times n$	9n	$O(n)$
6	$(2\tau_{\text{fetch}} + \tau_- + \tau_{\text{store}}) \times n$	4n	$O(n)$
7	$\tau_{\text{fetch}} + \tau_{\text{return}}$	2	$O(1)$
TOTAL	$(9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_- + \tau_{[]}) \times n$ $+ (\tau_+ + \tau_x + \tau_-) \times n$ $+ (8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[]} + \tau_- + \tau_- + \tau_{\text{return}})$	$16n + 14$	$O(n)$

Table □ shows the running time analysis of Program □ done in three ways--a detailed analysis, a simplified analysis, and an asymptotic analysis. In particular, note that all three methods of analysis are in agreement: Lines 2, 3, and 7 execute in a constant amount of time; 4, 5, and 6 execute in an amount of time which is proportional to n , plus a constant.

The most important observation to make is that, regardless of what the actual constants are, the asymptotic analysis always produces the same answer! Since the result does not depend upon the values of the constants, the asymptotic bound tells us something fundamental about the running time of the algorithm. And this fundamental result *does not depend upon the characteristics of the computer and compiler actually used to execute the program!*

Of course, you don't get something for nothing. While the asymptotic analysis may be significantly easier to do, all that we get is an upper bound on the running time of the algorithm. In particular, we know nothing about the *actual* running time of a particular program. (Recall Fallacies □ and □).

- [Rules For Big Oh Analysis of Running Time](#)
- [Example-Prefix Sums](#)
- [Example-Fibonacci Numbers](#)
- [Example-Bucket Sort](#)
- [Reality Check](#)
- [Checking Your Analysis](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Rules For Big Oh Analysis of Running Time

In this section we present some simple rules for determining a big-oh upper bound on the running time of the basic compound statements in a Python program.

Rule 3.1 (Sequential Composition) The worst-case running time of a sequence of Python statements such as

```
S1
S2
⋮
Sm
```

is $O(\max(T_1(n), T_2(n), \dots, T_m(n)))$, where the running time of S_i , the i^{th} statement in the sequence, is $O(T_i(n))$.

Rule 3.1 follows directly from Theorem 3.1. The total running time of a sequence of statements is equal to the sum of the running times of the individual statements. By Theorem 3.1, when computing the sum of a series of functions it is the largest one (the `max`) that determines the bound.

Rule 3.2 (Iteration—while) The worst-case running time of a Python `while` loop such as

```
while S1:
    S2
```

is $O(\max(T_1(n) \times (I(n) + 1), T_2(n) \times I(n)))$, where the running time of statement S_i is $O(T_i(n))$, for $i = 1$ and 2 , and $I(n)$ is the number of iterations executed in the worst case.

Rule 3.3 (Iteration—`for`) The worst-case running time of a Python `for` loop such as

```
for i in range(S1):
    S2
```

is the same as the equivalent Python `while` loop

```
i = 0
while S1:
    S2
    i += 1
```

which is $O(\max(T_1(n) \times (I(n) + 1), T_2(n) \times I(n)))$, where the running time of statement S_i is $O(T_i(n))$, for $i = 1$ and 2 , and $I(n)$ is the number of iterations executed in the worst case.

Consider the following simple *counted do loop*.

```
for i in range(n):
    S2
```

Here S_1 is the expression ``n'', so its running time is constant ($T_1(n) = O(1)$). Also, the number of iterations is $I(n) = n$. According to Rule \square , the running time of the `for` loop is $O(\max(I(n) + 1, T_2(n) \times I(n)))$, which simplifies to

$O(\max(n + 1, T_2(n) \times n))$. Furthermore, if the loop body *does anything at all*, its running time must be $T_2(n) = \Omega(1)$. Hence, the loop body will dominate the calculation of the maximum, and the running time of the loop is simply $O(n \times T_2(n))$.

If we don't know the exact number of iterations executed, $I(n)$, we can still use Rule \square provided we have an upper bound, $I(n) = O(f(n))$, on the number of iterations executed. In this case, the running time is $O(\max(f(n) + 1, T_2(n) \times f(n)))$.

Rule 3.4 (Conditional Execution) The worst-case running time of a Python `if-then-else` such as

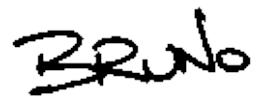
```
if S1:
    S2
else:
    S3
```

is $O(\max(T_1(n), T_2(n), T_3(n)))$, where the running time of statement S_i is $O(T_i(n))$, for $i = 1, 2, 3$.

Rule \square follows directly from the observation that the total running time for an if-then-else statement will never exceed the sum of the running time of the conditional test, S_1 , plus the larger of the running times of the *then part*, S_2 , and the *else part*, S_3 .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Prefix Sums

In this section, we will determine a tight big-oh bound on the running time of a program to compute the series of sums S_0, S_1, \dots, S_{n-1} , where

$$S_j = \sum_{i=0}^j a_i.$$

An algorithm to compute this series of summations is given in Program □. Table □ summarizes the running time calculation.

```
1 def prefixSums(a, n):
2     j = n - 1
3     while j >= 0:
4         sum = 0
5         i = 0
6         while i <= j:
7             sum += a[i]
8             i += 1
9         a[j] = sum
10        j -= 1
```

Program: Program to compute $\sum_{i=0}^j a_i$ for $0 \leq j < n$.

Table: Computing the running time of Program □.

statement	time
2	$O(1)$
3	$O(1) \times O(n)$ iterations
4	$O(1) \times O(n)$ iterations
5	$O(1) \times O(n)$ iterations
6	$O(1) \times O(n^2)$ iterations
7	$O(1) \times O(n^2)$ iterations

8	$O(1) \times O(n^2)$ iterations
9	$O(1) \times O(n)$ iterations
10	$O(1) \times O(n)$ iterations
<hr/> TOTAL	$O(n^2)$

Usually the easiest way to analyze program which contains nested loops is to start with the body of the inner-most loop. In Program □, the inner-most loop comprises lines 6-8. In all, a constant amount of work is done--this includes the conditional test (line 6), the loop body (line 7) and the incrementing of the loop index (line 8).

For a given value of j , the inner-most loop is done a total $j+1$ times. And since the outer loop is done for $j = n - 1, n - 2, \dots, 0$, in the worst case, the inner-most loop is done n times. Therefore, the contribution of the inner loop to the running time of one iteration of the outer loop is $O(n)$.

The rest of the outer loop (lines 4, 5, 9 and 10) does a constant amount of work in each iteration. This constant work is dominated by the $O(n)$ of the inner loop. The outer loop does exactly n iterations. Therefore, the total running time of the program is $O(n^2)$.

But is this a tight big oh bound? We might suspect that it is not, because of the worst-case assumption we made in the analysis concerning the number of times the inner loop is executed. The inner-most loop is done exactly $j+1$ times for $j = n - 1, n - 2, \dots, 0$. However, we did the calculation assuming the inner loop is done $O(n)$ times, in each iteration of the outer loop. Unfortunately, in order to determine whether our answer is a tight bound, we must determine more precisely the actual running time of the program.

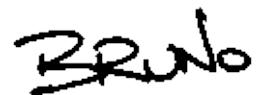
However, there is one approximate calculation that we can easily make. If we observe that the running time will be dominated by the work done in the inner-most loop, and that the work done in one iteration of the inner-most loop is constant, then all we need to do is to determine exactly the number of times the inner loop is actually executed. This is given by:

$$\begin{aligned}
 \sum_{j=0}^{n-1} j + 1 &= \sum_{j=1}^n j \\
 &= \frac{n(n+1)}{2} \\
 &= \Theta(n^2)
 \end{aligned}$$

Therefore, the result $T(n) = O(n^2)$ is a tight, big-oh bound on the running time of Program \square .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Fibonacci Numbers

In this section we will compare the asymptotic running times of two different programs that both compute Fibonacci numbers. ◇ The *Fibonacci numbers* are the series of numbers F_0, F_1, \dots , given by

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ F_{n-1} + F_{n-2} & n \geq 2. \end{cases} \quad (3.4)$$

Fibonacci numbers are interesting because they seem to crop up in the most unexpected situations. However, in this section, we are merely concerned with writing an algorithm to compute F_n given n .

Fibonacci numbers are easy enough to compute. Consider the sequence of Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The next number in the sequence is computed simply by adding together the last two numbers--in this case it is $5+8=13$. Program □ is a direct implementation of this idea. The running time of this algorithm is clearly $O(n)$ as shown by the analysis in Table □.

```
1 def Fibonacci(n):
2     previous = -1
3     result = 1
4     i = 0
5     while i <= n:
6         sum = result + previous
7         previous = result
8         result = sum
9         i += 1
10    return result
```

Program: Non-recursive program to compute Fibonacci numbers.

Table: Computing the running time of Program □.

statement	time
2	$O(1)$
3	$O(1)$
4	$O(1)$
5	$O(1) \times (n + 2)$ iterations
6	$O(1) \times (n + 1)$ iterations
7	$O(1) \times (n + 1)$ iterations
8	$O(1) \times (n + 1)$ iterations
9	$O(1) \times (n + 1)$ iterations
10	$O(1)$
TOTAL	$O(n)$

Recall that the Fibonacci numbers are defined recursively: $F_n = F_{n-1} + F_{n-2}$. However, the algorithm used in Program □ is non-recursive --it is *iterative* . What happens if instead of using the iterative algorithm, we use the definition of Fibonacci numbers to implement directly a recursive algorithm ? Such an algorithm is given in Program □ and its running time is summarized in Table □.

```

1 def Fibonacci(n):
2     if n == 0 or n == 1:
3         return n
4     else:
5         return Fibonacci(n - 1) + Fibonacci(n - 2)

```

Program: Recursive program to compute Fibonacci numbers.

Table: Computing the running time of Program □.

time
$n \geq 2$

statement $n < 2$

2	$O(1)$	$O(1)$
3	$O(1)$	--
5	--	$T(n-1) + T(n-2) + O(1)$
TOTAL	$O(1)$	$T(n-1) + T(n-2) + O(1)$

From Table \square we find that the running time of the recursive Fibonacci algorithm is given by the recurrence

$$T(n) = \begin{cases} O(1) & n < 2, \\ T(n-1) + T(n-2) + O(1) & n \geq 2. \end{cases}$$

But how do you solve a recurrence containing big oh expressions?

It turns out that there is a simple trick we can use to solve a recurrence containing big oh expressions *as long as we are only interested in an asymptotic bound on the result*. Simply drop the $O(\cdot)$'s from the recurrence, solve the recurrence, and put the $O(\cdot)$ back! In this case, we need to solve the recurrence

$$T(n) = \begin{cases} 1 & n < 2, \\ T(n-1) + T(n-2) + 1 & n \geq 2. \end{cases}$$

In the previous chapter, we used successfully repeated substitution to solve recurrences. However, in the previous chapter, all of the recurrences only had one instance of $T(\cdot)$ on the right-hand-side--in this case there are two. As a result, repeated substitution won't work.

There is something interesting about this recurrence: It looks very much like the definition of the Fibonacci numbers. In fact, we can show by induction on n that $T(n) \geq F_{n+1}$ for all $n \geq 0$.

extbfProof (By induction).

Base Case There are two base cases:

$$\begin{aligned} T(0) &= 1, & F_1 &= 1 \implies T(0) \geq F_1, \text{ and} \\ T(1) &= 1, & F_2 &= 1 \implies T(1) \geq F_2. \end{aligned}$$

Inductive Hypothesis Suppose that $T(n) \geq F_{n+1}$ for $n = 0, 1, 2, \dots, k$ for some $k \geq 1$. Then

$$\begin{aligned} T(k+1) &= T(k) + T(k-1) + 1 \\ &\geq F_{k+1} + F_k + 1 \\ &\geq F_{k+2} + 1 \\ &\geq F_{k+2}. \end{aligned}$$

Hence, by induction on k , $T(n) \geq F_{n+1}$ for all $n \geq 0$.

So, we can now say with certainty that the running time of the recursive Fibonacci algorithm, Program \square , is $T(n) = \Omega(F_{n+1})$. But is this good or bad? The following theorem shows us how bad this really is!

Theorem (Fibonacci numbers) The Fibonacci numbers are given by the closed form expression

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \quad (3.5)$$

where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$.

extbfProof (By induction).

Base Case There are two base cases:

$$\begin{aligned} F_0 &= \frac{1}{\sqrt{5}}(\phi^0 - \hat{\phi}^0) \\ &= 0 \\ F_1 &= \frac{1}{\sqrt{5}}(\phi^1 - \hat{\phi}^1) \\ &= \frac{1}{\sqrt{5}}((1 + \sqrt{5})/2 - (1 - \sqrt{5})/2) \\ &= 1 \end{aligned}$$

Inductive Hypothesis Suppose that Equation \square holds for $n = 0, 1, 2, \dots, k$ for some $k \geq 1$. First, we make the following observation:

$$\begin{aligned}
\phi^2 &= ((1 + \sqrt{5})/2)^2 \\
&= 1 + (1 + \sqrt{5})/2 \\
&= 1 + \phi.
\end{aligned}$$

Similarly,

$$\begin{aligned}
\hat{\phi}^2 &= ((1 - \sqrt{5})/2)^2 \\
&= 1 + (1 - \sqrt{5})/2 \\
&= 1 + \hat{\phi}.
\end{aligned}$$

Now, we can show the main result:

$$\begin{aligned}
F_{n+1} &= F_n + F_{n-1} \\
&= \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) + \frac{1}{\sqrt{5}}(\phi^{n-1} - \hat{\phi}^{n-1}) \\
&= \frac{1}{\sqrt{5}}(\phi^{n-1}(1 + \phi) - \hat{\phi}^{n-1}(1 + \hat{\phi})) \\
&= \frac{1}{\sqrt{5}}(\phi^{n-1}\phi^2 - \hat{\phi}^{n-1}\hat{\phi}^2) \\
&= \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1})
\end{aligned}$$

Hence, by induction, Equation \square correctly gives F_n for all $n \geq 0$.

Theorem \square gives us that $F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n)$ where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$.

Consider $|\hat{\phi}| < 1$. A couple of seconds with a calculator should suffice to convince you that $F_n \geq \phi^n - 1$. Consequently, as n gets large, $|\hat{\phi}^n|$ is vanishingly small. Therefore, $F_n = \Omega(\phi^n)$. In asymptotic terms, we write $F_n = \Omega((3/2)^n)$. Now, since $\phi \approx 1.62 > (3/2)$, we can write that $F_n = \Omega((3/2)^n)$.

Returning to Program \square , recall that we have already shown that its running time is $T(n) = \Omega(F_{n+1})$. And since $F_n = \Omega((3/2)^n)$, we can write that $T(n) = \Omega((3/2)^{n+1}) = \Omega((3/2)^n)$. That is, the running time of the recursive Fibonacci program grows *exponentially* with increasing n . And that is really bad in comparison with the linear running time of Program \square !

Figure \square shows the actual running times of both the non-recursive and recursive

algorithms for computing Fibonacci numbers.◊ Because the largest Python plain integer is 2147483647, it is only possible to compute Fibonacci numbers up to $F_{46} = 1\,836\,311\,903$ before the Python virtual machine starts computing with long integers (at which point our model of computation no longer applies).

The graph shows that up to about $n=30$, the running times of the two algorithms are comparable. However, as n increases past 30, the exponential growth rate of Program □ is clearly evident. In fact, the actual time taken by Program □ to compute F_{46} was in excess of SOMETHING!

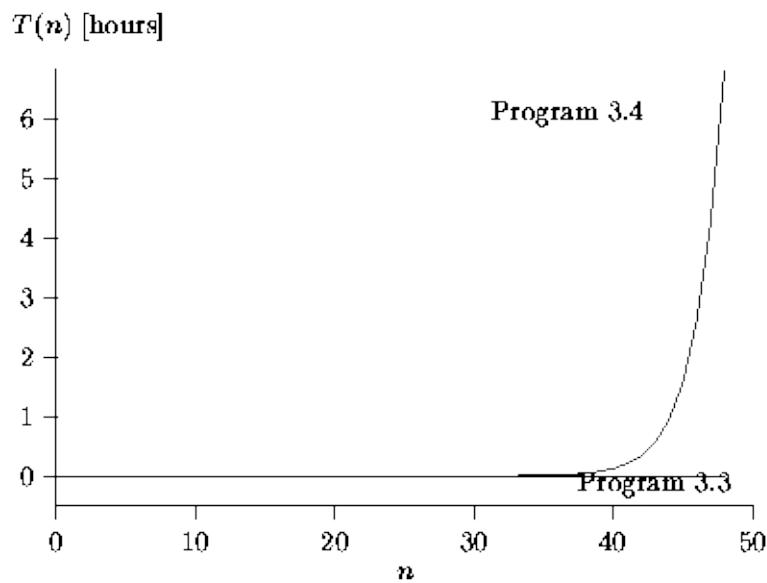


Figure: Actual running times of Programs □ and □.

Example-Bucket Sort

So far all of the asymptotic running time analyses presented in this chapter have resulted in tight big oh bounds. In this section we consider an example which illustrates that a cursory big oh analysis does not always result in a tight bound on the running time of the algorithm.

In this section we consider an algorithm to solve the following problem: Sort an array of n integers a_0, a_1, \dots, a_{n-1} , each of which is known to be between 0 and $m-1$ for some fixed m . An algorithm for solving this problem, called a *bucket sort* , is given in Program □.

```
1 def bucketSort(a, n, buckets, m):
2     for j in range(m):
3         buckets[j] = 0
4     for i in range(n):
5         buckets[a[i]] += 1
6     i = 0
7     for j in range(m):
8         for k in range(buckets[j]):
9             a[i] = j
10            i += 1
```

Program: Bucket sort.

A bucket sort works as follows: An array of m counters, or *buckets* , is used. Each of the counters is set initially to zero. Then, a pass is made through the input array, during which the buckets are used to keep a count of the number of occurrences of each value between 0 and $m-1$. Finally, the sorted result is produced by first placing the required number of zeroes in the array, then the required number of ones, followed by the twos, and so on, up to $m-1$.

The analysis of the running time of Program □ is summarized in Table □. Clearly, the worst-case running time of the first loop (lines 7-8) is $O(m)$ and that of the second loop (lines 9-10) is $O(n)$.

Table: Computing the running time of

Table: Computing the running time of

Program □.

time

statement cursory analysis careful analysis

2-3	$O(m)$	$O(m)$
4-5	$O(n)$	$O(n)$
6-10	$O(mn)$	$O(m+n)$
TOTAL	$O(mn)$	$O(m+n)$

Consider nested loops on lines 7-10. Exactly m iterations of the outer loop are done--the number of iterations of the outer loop is fixed. But the number of iterations of the inner loop depends on $\text{bucket}[j]$ --the value of the counter. Since there are n numbers in the input array, in the worst case a counter may have the value n . Therefore, the running time of lines 7-10 is $O(mn)$ and this running time dominates all the others, so the running time of Program □ is $O(mn)$. (This is the *cursory analysis* column of Table □).

Unfortunately, the cursory analysis has not produced a tight bound. To see why this is the case, we must consider the operation of Program □ more carefully. In particular, since we are sorting n items, the final answer will only contain n items. Therefore, line 9 will be executed exactly n times--not mn times as the cursory result suggests.

Consider the inner loop at line 8. During the j^{th} iteration of the outer loop, the inner loop does $\text{bucket}[j]$ iterations. According to Rule □ this gives the worst-case running time of $O(\text{bucket}[j] + 1)$. Therefore, the total running time is

$$\begin{aligned} \sum_{j=0}^{m-1} (\text{bucket}[j] + 1) &= \sum_{j=0}^{m-1} \text{bucket}[j] + \sum_{j=0}^{m-1} 1 \\ &= n + m. \end{aligned}$$

So, the running time of lines 6-10 is $O(m+n)$ and therefore running time of Program □ is $O(m+n)$. (This is the *careful analysis* column of Table □).

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Reality Check

``Asymptotic analysis is nice in theory," you say, ``but of what practical value is it when I don't know what c and n_0 are?" Fallacies \square and \square showed us that if we have two programs, A and B , that solve a given problem, whose running times are $T_A = O(n^2)$ and $T_B = O(n^3)$ say, we cannot conclude in general that we should use algorithm A rather than algorithm B to solve a particular instance of the problem. Even if the bounds are both known to be tight, we still don't have enough information. What we do know for sure is that *eventually*, for large enough n , program A is the better choice.

In practice we need not be so conservative. It is almost always the right choice to select program A . To see why this is the case, consider the times shown in Table \square . This table shows the running times computed for a very conservative scenario. We assume that the constant of proportionality, c , is one cycle of a 1 GHz clock. This table shows the running times we can expect even if only one instruction is done for each element of the input.

Table: Actual lower bounds assuming
a 1 GHz clock, $c = 1$ cycle and $n_0 = 0$.

	$n=1$	$n=8$	$n = 1K$	$n = 1024K$
$\Omega(1)$	1 ns	1 ns	1 ns	1 ns
$\Omega(\log n)$	1 ns	3 ns	10 ns	20 ns
$\Omega(n)$	1 ns	8 ns	102 ns	1.05 ms
$\Omega(n \log n)$	1 ns	24 ns	1.02 μ s	21 ms
$\Omega(n^2)$	1 ns	64 ns	10.2 μ s	18.3 minutes
$\Omega(n^3)$	1 ns	512 ns	1.07 s	36.5 years
$\Omega(2^n)$	1 ns	256 ns	10^{292} years	10^{10^8} years

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Checking Your Analysis

Having made an asymptotic analysis of the running time of an algorithm, how can you verify that the implementation of the algorithm performs as predicted by the analysis? The only practical way to do this is to conduct an experiment-- write out the algorithm in the form of a computer program, run the program, and measure its actual running time for various values of the parameter, n say, used to characterize the size of the problem.

However, several difficulties immediately arise:

- How do you compare the results of the analysis which, by definition, only applies asymptotically, i.e., as n gets arbitrarily large, with the actual running time of a program which, of necessity, must be measured for fixed and finite values of n ?
- How do you explain it when the results of your analysis do not agree with the observed behavior of the program?

Suppose you have conducted an experiment in which you measured the actual running time of a program, $T(n)$, for a number of different values of n .

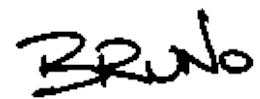
Furthermore, suppose that on the basis of an analysis of the algorithm you have concluded that the worst-case running time of the program is $O(f(n))$. How do you tell from the measurements made that the program behaves as predicted?

One way to do this follows directly from the definition of big oh: there exists $c > 0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$. This suggests that we should compute the ratio $T(n)/f(n)$ for each of value of n in the experiment and observe how the ratio behaves as n increases. If this ratio diverges, then $f(n)$ is probably too small; if this ratio converges to zero, then $f(n)$ is probably too big; and if the ratio converges to a constant, then the analysis is probably correct.

What if $f(n)$ turns out too large? There are several possibilities:

- The function $f(n)$ is not a *tight* bound. That is, the analysis is still correct, but the bound is not the tightest bound possible.
- The analysis was for the *worst case* but the worst case did not arise in the set of experiments conducted.
- A mistake was made and the analysis is wrong.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

Exercises

1. Consider the function $f(n) = 3n^2 - n + 4$. Using Definition \square show that $f(n) = O(n^2)$.
2. Consider the function $f(n) = 3n^2 - n + 4$. Using Definition \square show that $f(n) = \Omega(n^2)$.
3. Consider the functions $f(n) = 3n^2 - n + 4$ and $g(n) = n \log n + 5$. Using Theorem \square show that $f(n) + g(n) = O(n^2)$.
4. Consider the functions $f(n) = \sqrt{n}$ and $g(n) = \log n$. Using Theorem \square show that $f(n) + g(n) = O(\sqrt{n})$.
5. For each pair of functions, $f(n)$ and $g(n)$, in the following table, indicate if $f(n)=O(g(n))$ and if $g(n)=O(f(n))$.

$f(n)$	$g(n)$
$10n$	$n^2 - 10n$
n^3	$n^2 \log n$
$n \log n$	$n + \log n$
$\log n$	$\sqrt[3]{n}$
$\ln n$	$\log n$
$\log(n+1)$	$\log n$
$\log \log n$	$\log n$
2^n	10^n
n^m	m^n
$\cos(n\pi/2)$	$\sin(n\pi/2)$
n^2	$(n \cos n)^2$

6. Show that the Fibonacci numbers (see Equation \square) satisfy the identities

$$F_{2n-1} = (F_n)^2 + (F_{n-1})^2$$

$$F_{2n} = (F_n)^2 + 2F_n F_{n-1}$$

for $n \geq 1$.

7. Prove each of the following formulas:

$$1. \sum_{i=0}^n i = O(n^2)$$

$$2. \sum_{i=0}^n i^2 = O(n^3)$$

$$3. \sum_{i=0}^n i^3 = O(n^4)$$

8. Show that $\sum_{i=0}^n a^i = O(1)$, where $0 \leq a < 1$ and $n \geq 0$.

9. Show that $\sum_{i=1}^n \frac{1}{i} = O(\log n)$.

10. Solve each of the following recurrences:

$$1. T(n) = \begin{cases} O(1) & n = 0, \\ aT(n-1) + O(1) & n > 0, \quad a > 1. \end{cases}$$

$$2. T(n) = \begin{cases} O(1) & n = 0, \\ aT(n-1) + O(n) & n > 0, \quad a > 1. \end{cases}$$

$$3. T(n) = \begin{cases} O(1) & n = 1, \\ aT(\lfloor n/a \rfloor) + O(1) & n > 1, \quad a \geq 2. \end{cases}$$

$$4. T(n) = \begin{cases} O(1) & n = 1, \\ aT(\lfloor n/a \rfloor) + O(n) & n > 1, \quad a \geq 2. \end{cases}$$

11. Derive tight, big oh expressions for the running times of example-a,example-b,example-c,example-d,example-f,example-g,example-h,example-i.
12. Consider the Python program fragments given below. Assume that n , m , and k are non-negative integers and that the methods e , f , g , and h have the following characteristics:
- The worst case running time for $e(n, m, k)$ is $O(1)$ and it returns a value between 1 and $(n+m+k)$.
 - The worst case running time for $f(n, m, k)$ is $O(n+m)$.
 - The worst case running time for $g(n, m, k)$ is $O(m+k)$.
 - The worst case running time for $h(n, m, k)$ is $O(n+k)$.

Determine a tight, big oh expression for the worst-case running time of each of the following program fragments:

1. $f(n, 10, 0);$
 $g(n, m, k);$
 $h(n, m, 1000000);$

```
2. for i in range(n):
    f(n, m, k);

3. for i in range (e(n, 10, 100)):
    f(n, 10, 0);

4. for i in range (e(n, m, k)):
    f(n, 10, 0);

5. for i in range(n):
    for j in range(i, n):
        f(n, m, k);
```

13. Consider the following Python program fragment. What value does `f` compute? (Express your answer as a function of n). Give a tight, big oh expression for the worst-case running time of the method `f`.

```
def f(n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + i
    return sum
```

14. Consider the following Python program fragment. (The method `f` is given in Exercise □). What value does `g` compute? (Express your answer as a function of n). Give a tight, big oh expression for the worst-case running time of the method `g`.

```
def g(n):
    sum = 0
    for i in range(1, n + 1):
        sum = sum + i + f(i)
    return sum
```

15. Consider the following Python program fragment. (The method `f` is given in Exercise □ and the method `g` is given in Exercise □). What value does `h` compute? (Express your answer as a function of n). Give a tight, big oh expression for the worst-case running time of the method `h`.

```
def h(n):
    return f(n) + g(n)
```

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Projects

1. Write a Python method that takes a single integer argument n and has a worst-case running time of $O(n)$.
2. Write a Python method that takes a single integer argument n and has a worst-case running time of $O(n^2)$.
3. Write a Python method that takes two integer arguments n and k and has a worst-case running time of $O(n^k)$.
4. Write a Python method that takes a single integer argument n and has a worst-case running time of $O(\log n)$.
5. Write a Python method that takes a single integer argument n and has a worst-case running time of $O(n \log n)$.
6. Write a Python method that takes a single integer argument n and has a worst-case running time of $O(2^n)$.
7. The generalized Fibonacci numbers of order $k \geq 2$ are given by

$$F_n^{(k)} = \begin{cases} 0 & 0 \leq n < k - 1, \\ 1 & n = k - 1, \\ \sum_{i=1}^k F_{n-i}^{(k)} & n \geq k. \end{cases} \quad (3.6)$$

Write both *recursive* and *non-recursive* methods that compute $F_n^{(k)}$. Measure the running times of your algorithms for various values of k and n .

Foundational Data Structures

In this book we consider a variety of *abstract data types* (ADTs) , including stacks, queues, deques, ordered lists, sorted lists, hash and scatter tables, trees, priority queues, sets, and graphs. In just about every case, we have the option of implementing the ADT using an array or using a some kind of linked data structure.

Because they are the base upon which almost all of the ADTs are built, we call the *array* and the *linked list* the *foundational data structures* . It is important to understand that we do not view the array or the linked list as ADTs, but rather as alternatives for the implementation of ADTs.

In this chapter we consider arrays first. We review the support for arrays in Python and and present an `Array` class implementation that provides resizeable arrays with arbitrary subscript ranges. We also discuss multi-dimensional arrays and matrices. Next, we consider a number of linked list implementation alternatives and we discuss in detail the implementation of a singly-linked list class, `LinkedList`. It is important to become familiar with the `Array` and `LinkedList` classes, as they are used extensively throughout the remainder of the book.

- [Python Lists and Arrays](#)
 - [Multi-Dimensional Arrays](#)
 - [Singly-Linked Lists](#)
 - [Exercises](#)
 - [Projects](#)
-

Bruno

Python Lists and Arrays

Probably the most common way to aggregate data in a Python program is to use a Python list. A Python list is an object that contains an ordered collection of objects. For example,

```
a = [0, 0, 0, 0, 0]
```

creates a Python list that comprises five plain integers (all zero) and assigns it to the variable `a`.

The elements of a Python list are accessed using integer-valued indices. The first or leftmost element of a Python list always has index zero. Thus, the five elements of list `a` are `a[0], a[1], ..., a[4]`. Python also supports the use of negative indices to index into a list from the right. The last or rightmost element of a Python list always has index `-1`. Thus, the five elements of list `a` can also be accessed as `a[-5], a[-4], ..., a[-1]`.

Python provides a built-in function called `len` that returns the length of any object (that has a length). When applied to a Python list, the `len` function returns the number of elements in that list. Thus, `len(a)` has the value 5.

Python checks at run-time that the index used to access a list element is valid. Valid indices fall between `-n` and `n-1`, where `n` is the length of the list. If an invalid index expression is used, an `IndexError` exception is raised.

It is important to remember that in Python, an assignment statement assigns a name to an object. In particular, the sequence of statements

```
a = [0, 0, 0, 0, 0]
b = a
```

causes the variable `b` to refer to the same list object as variable `a`. Furthermore, the sequence of statements

```
x = 57
a[0] = x
```

`a[1] = x`

causes `x`, `a[0]`, and `a[1]` all to refer to the same object.

How is a Python list represented in the memory of the computer? The specification of the Python language leaves many of the details up to the system implementers[49]. However, Figure □ illustrates the typical implementation scenario.

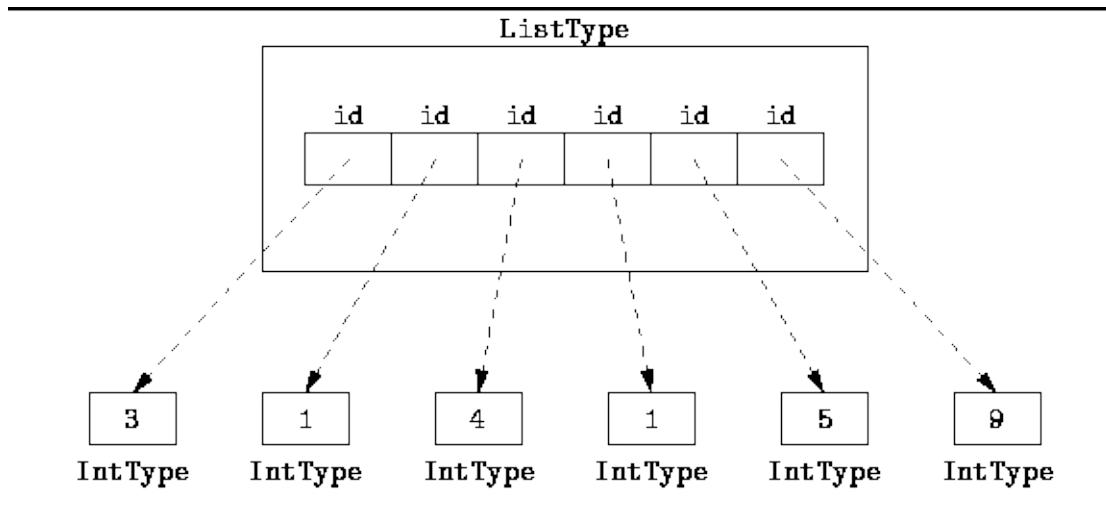


Figure: Memory representation of a Python list.

A Python list represents a collection of objects. In this case, the objects are all plain integers. The list object actually contains an array of the identities (or addresses) of the objects in the collection. The array elements (the identities) typically occupy consecutive memory locations. That way, given i it is possible to find the identity of $a[i]$ in constant time.

On the basis of Figure □, we can now estimate the total storage required to represent a Python list. Let $S(n)$ be the total storage (memory) needed to represent a list of n objects. $S(n)$ is given by

$$S(n) \geq n \text{sizeof}(\text{id})$$

where the function `sizeof(X)` is the number of bytes used for the memory representation of an instance of an object of type `X`.

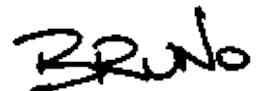
In the Python virtual machine, object identities (or addresses) are typically represented using fixed-size integers. Hence, $\text{sizeof}(\text{id}) = O(1)$. In practice, a

Python list may contain additional data items. For example, it is reasonable to expect that there is a datum that records the position in memory of the first array element. In any event, the overhead associated with a fixed number of additional data items is $O(1)$. Therefore, $S(n)=O(n)$.

- [Extending Python Lists - An Array Class](#)
 - [init Method](#)
 - [copy Method](#)
 - [getitem and setitem Methods](#)
 - [Array Properties](#)
 - [Resizing an Array](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



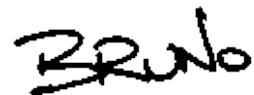
Extending Python Lists - An Array Class

While the Python programming language does indeed provide built-in support for lists, that support is not without its shortcomings: For example, the first (or leftmost) index of a list is always zero (or $-n$) and the rightmost index of a list is always $n-1$ (or -1). However, in certain applications, it is desireable to index list elements starting with a non-zero base index. Another issue arise from the fact that negative indices are allowed. In many applications, only non-negative indices are required and an attempt to use a negative index is evidence of a programming error. Because Python allows negative indices, such errors are not detected at run time and may lead to incorrect program execution.

One way to address these deficiencies is to define a new class with the desired functionality. We do this by defining an `Array` class with two instance attributes, `_data` and `_baseIndex`. The first is a Python list and the second is a plain integer which records the lower bound for array indices.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



__init__ Method

Program □ gives the code for the Array class `__init__` method. The `__init__` method takes three arguments, `self`, `length` and `baseIndex`. The `length` argument gives the desired array length and the `baseIndex` argument gives the lower bound for array indices. The `__init__` method creates list of of the desired length and then sets the `_baseIndex`. Note that the default base index is zero and the default array length is zero.

```
1  class Array(object):
2
3      def __init__(self, length = 0, baseIndex = 0):
4          assert length >= 0
5          self._data = [ None for i in xrange(length) ]
6          self._baseIndex = baseIndex
7
8      # ...
```

Program: Array class `__init__` method.

In Python, when a list is allocated, two things happen. First, memory is allocated for the list object and its elements. Second, each element of the list is initialized with the appropriate default value (in this case all of the list elements refer to the `None` object).

For now, we shall assume that the first step takes a constant amount of time. Since there are $n = \text{length}$ elements to be initialized, the second step takes $O(n)$ time. Therefore, the running time of the Array class `__init__` method is $O(n)$.

__copy__ Method

Program □ defines the `__copy__` method of the `Array` class. This method provides a way to create a copy of a given array. Specifically, the `__copy__` method creates a *shallow copy*. A shallow copy creates a copy of the `Array` object but does not copy the objects contained in the array.

```
1  class Array(object):
2
3      def __copy__(self):
4          result = Array(len(self._data))
5          for i, datum in enumerate(self._data):
6              result._data[i] = datum
7          result._baseIndex = self._baseIndex
8          return result
9
10 # ...
```

Program: `Array` class `__copy__` method.

The `__copy__` method is intended to be used in conjunction with the Python `copy` module like this: like this:

```
from copy import copy

a = Array(5)
b = copy(a)
```

The purpose of the `copy` function in the `copy` module is to create a shallow copy of an object. If the object provides a `__copy__` method, the `copy` function calls that method to create the copy.

Note that after the `copy`, `a` and `b` refer to distinct `Array` instances. However, the elements of the array are shared. For example, `a[0]` and `b[0]` both refer to the same object instance.

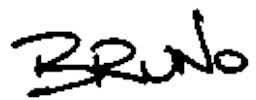
Program □ shows a simple implementation of the `__copy__` method. To determine its running time, we need to consider carefully the execution of this method.

First, a new Array instance is created. As discussed above, this operation takes $O(n)$ in the worst case, where n is the new array length.

Next, there is a loop which copies one-by-one the elements of the input array to the newly allocated array. Clearly this operation takes $O(n)$ time to perform. Finally, the `_baseIndex` instance attribute is copied in $O(1)$ time. Altogether, the running time of the `__copy__` method is $T(n)=O(n)$, where n is the size of the array being copied.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



__getitem__ and __setitem__ Methods

The elements of a Python list are accessed by enclosing the index expression between brackets [and] like this:

```
a[2] = b[3]
```

In order to be able to use the same syntax to access the elements of a Array object, we define the __getitem__ and __setitem__ methods as shown in Program □

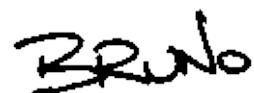
```
1  class Array(object):
2
3      def getOffset(self, index):
4          offset = index - self._baseIndex
5          if offset < 0 or offset >= len(self._data):
6              raise IndexError
7          return offset
8
9      def __getitem__(self, index):
10         return self._data[self.getOffset(index)]
11
12     def __setitem__(self, index, value):
13         self._data[self.getOffset(index)] = value
14
15     # ...
```

Program: Array class __getitem__ and __setitem__ methods.

Both __getitem__ and __setitem__ invoke the getOffset method to translate the given index by subtracting from it the value of the _baseIndex instance attribute. In this way arbitrary subscript ranges are supported. Since the overhead of this subtraction is constant, the running times of the __getitem__ and __setitem__ methods are $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Array Properties

Program □ defines two properties of Arrays--`data` and `baseIndex`. These properties provide the means to *inspect* the contents of an Array object (using the `fget` *accessor* methods) and the means to *modify* the contents of an Array object (using the `fset` *accessor* methods).

Clearly, the running times of each of the `baseIndex` property `fget` and `fset` accessors is a constant. Similarly, the running time of the `data` property `fget` accessor is also a constant.

```
1  class Array(object):
2
3      def getData(self):
4          return self._data
5
6      data = property(
7          fget = lambda self: self.getData())
8
9      def getBaseIndex(self):
10         return self._baseIndex
11
12     def setBaseIndex(self, baseIndex):
13         self._baseIndex = baseIndex
14
15     baseIndex = property(
16         fget = lambda self: self.getBaseIndex(),
17         fset = lambda self, value: self.setBaseIndex(value))
18
19     # ...
```

Program: Array class `data` and `baseIndex` properties.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Resizing an Array

Program □ defines the `length` property of the `Array` class. The `fget` accessor simply invokes the `__len__` method which returns the length of the array. The `fset` accessor of the `length` property calls the `setLength` method which provides the means to change the size of an array at run time. This method can be used both to increase and to decrease the size of an array.

```
1  class Array(object):
2
3      def __len__(self):
4          return len(self._data)
5
6      def setLength(self, value):
7          if len(self._data) != value:
8              newData = [None for i in xrange(value)]
9              m = min(len(self._data), value)
10             for i in xrange(m):
11                 newData[i] = self._data[i]
12             self._data = newData
13
14     length = property(
15         fget = lambda self: self.__len__(),
16         fset = lambda self, value: self.setLength(value))
17
18     # ...
```

Program: `Array` class `length` property.

The running time of this algorithm depends only on the new array length. Let n be the original size of the array and let m be the new size of the array. Consider the case where $m \neq n$. The method first allocates and initializes a new array of size m . Next, it copies at most $\min(m, n)$ elements from the old array to the new array. Therefore,

$$T(m, n) = O(m) + O(\min(m, n)) = O(m)$$

Bruno

Multi-Dimensional Arrays

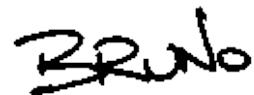
A *multi-dimensional array* of dimension n (i.e., an n -dimensional array or simply n -D array) is a collection of items which is accessed via n subscript expressions. For example, the $(i, j)^{th}$ element of a two-dimensional array x is accessed by writing $x[i, j]$.

Python does not provide built-in support for multi-dimensional arrays. In this section, we will examine the implementation of a multi-dimensional array class, `MultiDimensionalArray`, that is based on the one-dimensional array class discussed in Section □.

-
- [Array Subscript Calculations](#)
 - [An Implementation](#)
 - [MultiDimensionalArray class init Method](#)
 - [MultiDimensionalArray class getitem and setitem Methods](#)
 - [Matrices](#)
 - [Dense Matrices](#)
 - [Canonical Matrix Multiplication](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Array Subscript Calculations

The memory of a computer is essentially a one-dimensional array--the memory address is the array subscript. Therefore, a natural way to implement a multi-dimensional array is to store its elements in a one-dimensional array. In order to do this, we need a mapping from the n subscript expressions used to access an element of the multi-dimensional array to the one subscript expression used to access the one-dimensional array. For example, suppose we wish to represent a 2×3 array of ints, a , using a one-dimensional array like this:

```
b = Array(6)
```

Then we need to determine which element of b , say $b[k]$, will be accessed given a reference of the form $a[i, j]$. That is, we need the mapping f such that $k = f(i, j)$.

The mapping function determines the way in which the elements of the array are stored in memory. The most common way to represent an array is in *row-major order*, also known as *lexicographic order*. For example, consider the 2×3 two-dimensional array. The row-major layout of this array is shown in Figure □.

position	value
$b[0]$	$a[0,0]$
$b[1]$	$a[0,1]$
$b[2]$	$a[0,2]$
$b[3]$	$a[1,0]$
$b[4]$	$a[1,1]$
$b[5]$	$a[1,2]$

row 0

row 1

Figure: Row-major order layout of a 2D array.

In row-major layout, it is the right-most subscript expression (the column index) that increases the fastest. As a result, the elements of the rows of the matrix end up stored in contiguous memory locations. In Figure □, the first element of the first row is at position $b[0]$. The first element of the second row is at position $b[3]$, since there are 3 elements in each row.

We can now generalize this to an arbitrary n -dimensional array. Suppose we have an n -D array a with dimensions

$$\delta_0 \times \delta_1 \times \cdots \times \delta_{n-1}.$$

Then, the position of the element $a[i_0, i_1, \dots, i_{n-1}]$ is given by

$$\sum_{j=0}^{n-1} f_j i_j \quad (4.1)$$

where

$$f_j = \begin{cases} 1 & j = n - 1, \\ \prod_{k=j+1}^{n-1} \delta_k & 0 \leq j < n - 1. \end{cases} \quad (4.2)$$

The running time required to calculate the position appears to be $O(n^2)$ since the position is the sum of n terms and for each term we need to compute f_j , which requires $O(n)$ multiplications in the worst case. However, the factors f_j are determined solely from the dimensions of the array. Therefore, we need only compute the factors once. Assuming that the factors have been precomputed, the position calculation can be done in $O(n)$ time using the following algorithm:

```
offset = 0
for j in range(n):
    offset += f_j * i_j
```

An Implementation

In this section we illustrate the implementation of a multi-dimensional array using a one-dimensional array. We do this by defining a class called `MultiDimensionalArray` that is very similar to the `Array` class defined in Section □.

Altogether three instance attributes are used to implement the `MultiDimensionalArray` class. The first, `_dimensions` is an array of length n , where n is number of dimensions and $\text{dimension}[i]$ is the size of the i^{th} dimension (δ_i).

The second instance attribute, `_factors`, is also an array of length n . The j^{th} element of the `factors` array corresponds to the factor f_j given by Equation □.

The third instance attribute, `_data`, is a one-dimensional array used to hold the elements of the multi-dimensional array in row-major order.

MultiDimensionalArray class `__init__` Method

The `__init__` method for the `MultiDimensionalArray` class is defined in Program □. The `dimensions` argument is a tuple which represents the dimensions of the array. For example, to create a $3 \times 5 \times 7$ three-dimensional array, we create a `MultiDimensionalArray` like this:

```
a = MultiDimensionalArray(3, 5, 7)
```

```
1  class MultiDimensionalArray(object):
2
3      def __init__(self, *dimensions):
4          self._dimensions = Array(len(dimensions))
5          self._factors = Array(len(dimensions))
6          product = 1
7          i = len(dimensions) - 1
8          while i >= 0:
9              self._dimensions[i] = dimensions[i]
10             self._factors[i] = product
11             product *= self._dimensions[i]
12             i -= 1
13             self._data = Array(product)
14
15     # ...
```

Program: `MultiDimensionalArray` class `__init__` method.

The `__init__` method copies the dimensions of the array into the `_dimensions` array, and then it computes the `_factors` array. These operations take $O(n)$, where n is the number of dimensions. The `__init__` method then allocates a one-dimensional array of length m given by

$$m = \prod_{i=0}^{n-1} \delta_i.$$

The worst-case running time of the `__init__` method is $O(m+n)$.

Bruno

MultiDimensionalArray class `__getitem__` and `__setitem__` Methods

The elements of a multi-dimensional array are accessed using the `__getitem__` and `__setitem__` methods of the `MultiDimensionalArray` class. For example, you can access the $(i, j, k)^{th}$ element of a three-dimensional array `a` like this:

```
value = a[i, j, k]
```

which invokes the `__getitem__` method with the tuple (i, j, k) as the index expression. Similarly, you can modify the $(i, j, k)^{th}$ element like this:

```
a[i, j, k] = value
```

which invokes the `setitem` method with the tuple (i, j, k) as the index expression.

Program □ shows how that `__getitem__` and `__setitem__` methods are both implemented using a `getOffset` method. The `getOffset` method takes a tuple of n indices and computes the position of the corresponding element in the one-dimensional array according to Equation □. This computation takes $O(n)$ time in the worst case, where n is the number of dimensions. Consequently, the running times of the get and set accessors are also $O(n)$.

```
 1 class MultiDimensionalArray(object):
 2
 3     def getOffset(self, indices):
 4         if len(indices) != len(self._dimensions):
 5             raise IndexError
 6         offset = 0
 7         for i, dim in enumerate(self._dimensions):
 8             if indices[i] < 0 or indices[i] >= dim:
 9                 raise IndexError
10             offset += self._factors[i] * indices[i]
11         return offset
12
13     def __getitem__(self, indices):
14         return self._data[self.getOffset(indices)]
15
16     def __setitem__(self, indices, value):
17         self._data[self.getOffset(indices)] = value
18
19     # ...
```

Program: MultiDimensionalArray class `__getitem__` and `__setitem__` methods.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Matrices

Two-dimensional arrays of floating-point numbers arise in many different scientific computations. Such arrays are usually called *matrices*.

Mathematicians have studied the properties of matrices for many years and have developed an extensive repertoire of operations on matrices. In this section we consider two-dimensional matrices and examine the implementation of simple, matrix multiplication.

The dimensions of a two-dimensional matrix are referred to as the *rows* and the *columns* of the matrix. Program □ defines a `Matrix` class which provides two properties, `numberOfRows` and `numberOfColumns`.

```
 1  class Matrix(object):
 2
 3      def __init__(self, numberOfRows, numberOfColumns):
 4          assert numberOfRows >= 0
 5          assert numberOfColumns >= 0
 6          super(Matrix, self).__init__()
 7          self._numberOfRows = numberOfRows
 8          self._numberOfColumns = numberOfColumns
 9
10      def getNumberOfRows(self):
11          return self._numberOfRows
12
13      numberOfRows = property(
14          fget = lambda self: self.getNumberOfRows())
15
16      def getNumberOfColumns(self):
17          return self._numberOfColumns
18
19      numberOfColumns = property(
20          fget = lambda self: self.getNumberOfColumns())
21
22      # ...
```

Program: `Matrix` class.

The `Matrix` class has two plain integer instance attributes, `_numberOfRows` and `_numberOfColumns`, which record the dimensions of the matrix.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Dense Matrices

The simplest way to implement a matrix is to use a multi-dimensional array with two dimensions as shown in Program □. The `DenseMatrix` class extends the `Matrix` class discussed in the preceding section. The `DenseMatrix` class adds a instance attribute called `_array` which is a multi-dimensional array. The `__init__` method of the `DenseMatrix` class takes two arguments, `m = rows` and `n = cols`, and constructs the corresponding $m \times n$ multi-dimensional array. Clearly, the running time of the `__init__` method is $O(mn)$.

```
 1  class DenseMatrix(Matrix):
 2
 3      def __init__(self, rows, cols):
 4          super(DenseMatrix, self).__init__(rows, cols)
 5          self._array = MultiDimensionalArray(rows, cols)
 6
 7      def __getitem__(self, (i, j)):
 8          return self._array[i,j]
 9
10      def __setitem__(self, (i, j), value):
11          self._array[i,j] = value
12
13      # ...
```

Program: `DenseMatrix` class `__init__`, `__getitem__` and `__setitem__` methods.

Program □ also defines the `__getitem__` and `__setitem__` methods for the `DenseMatrix` class. These methods each take an ordered pair, (i,j) , as the index expression and use the pair as the index into the multi-dimensional array. Because the number of dimensions is fixed at two, the running time for each of these methods is $O(1)$.

Canonical Matrix Multiplication

Given an $m \times n$ matrix A and an $n \times p$ matrix B , the product $C=AB$ is an $m \times p$ matrix. The elements of the result matrix are given by

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}. \quad (4.3)$$

Accordingly, in order to compute the produce matrix, C , we need to compute mp summations each of which is the sum of n product terms. An algorithm to compute the matrix product is given in Program □. The algorithm given is a direct implementation of Equation □.

```
 1 class DenseMatrix(Matrix):
 2
 3     def __mul__(self, mat):
 4         assert self.numberOfColumns == mat.numberOfRows
 5         result = DenseMatrix(
 6             self.numberOfRows, mat.numberOfColumns)
 7         for i in xrange(self.numberOfRows):
 8             for j in xrange(mat.numberOfColumns):
 9                 sum = 0
10                 for k in xrange(self.numberOfColumns):
11                     sum += self[i,k] * mat[k,j]
12                 result[i,j] = sum
13         return result
14
15     # ...
```

Program: `DenseMatrix` class `__times__` method.

The algorithm begins by checking to see that the matrices to be multiplied have compatible dimensions. That is, the number of columns of the first matrix must be equal to the number of rows of the second one. This check takes $O(1)$ time in the worst case.

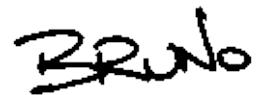
Next a matrix in which the result will be formed is constructed (line 6). The running time for this is $O(mp)$. For each value of i and j , the innermost loop (lines 10-11) does n iterations. Each iteration takes a constant amount of time.

The body of the middle loop (lines 9-12) takes time $O(n)$ for each value of i and j . The middle loop is done for p iterations, giving the running time of $O(np)$ for each value of i . Since, the outer loop (lines 7-12) does m iterations, its overall running time is $O(mnp)$. Finally, the result matrix is returned on line 13. This takes a constant amount of time.

In summary, we have shown that lines 4-5 are $O(1)$; line 6 is $O(mp)$; lines 7-12 are $O(mnp)$; and line 13 is $O(1)$. Therefore, the running time of the canonical matrix multiplication algorithm is $O(mnp)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Singly-Linked Lists

The singly-linked list is the most basic of all the linked data structures. A singly-linked list is simply a sequence of objects, each of which refers to its successor in the list. Despite this obvious simplicity, there are myriad implementation variations. Figure □ shows several of the most common singly-linked list variants.

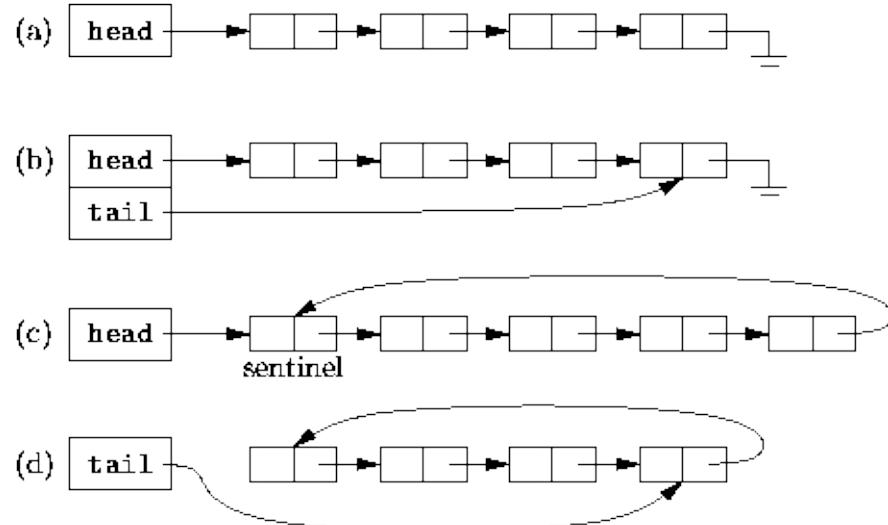


Figure: Singly-linked list variations.

The basic singly-linked list is shown in Figure □ (a). Each element of the list refers to its successor and the last element refers to None. One variable, labeled `head` in Figure □ (a), is used to keep track of the list.

The basic singly-linked list is inefficient in those cases when we wish to add elements to both ends of the list. While it is easy to add elements at the head of the list, to add elements at the other end (*the tail*) we need to locate the last element. If the basic basic singly-linked list is used, the entire list needs to be traversed in order to find its tail.

Figure □ (b) shows a way in which to make adding elements to the tail of a list more efficient. The solution uses a second variable, `tail`, which refers to the

last element of the list. Of course, this time efficiency comes at the cost of the additional space used to store the variable `tail`.

The singly-linked list labeled (c) in Figure □ illustrates two common programming tricks. There is an extra element at the head of the list called a *sentinel*. This element is never used to hold data and it is always present. The principal advantage of using a sentinel is that it simplifies the programming of certain operations. For example, since there is always a sentinel standing guard, we never need to modify the `head` variable. Of course, the disadvantage of a sentinel such as that shown in (c) is that extra space is required, and the sentinel needs to be created when the list is initialized.

The list (c) is also a *circular list*. Instead of using a reference `None` to demarcate the end of the list, the last element of the list refers to the sentinel. The advantage of this programming trick is that insertion at the head of the list, insertion at the tail of the list, and insertion at an arbitrary position of the list are all identical operations.

Of course, it is also possible to make a circular, singly-linked list that does not use a sentinel. Figure □ (d) shows a variation in which a single variable is used to keep track of the list, but this time the variable, `tail`, refers to the last element of the list. Since the list is circular in this case, the first element follows the last element of the list. Therefore, it is relatively simple to insert both at the head and at the tail of this list. This variation minimizes the storage required, at the expense of a little extra time for certain operations.

Figure □ illustrates how the empty list (i.e., the list containing no list elements) is represented for each of the variations given in Figure □. Notice that the sentinel is always present in list variant (c). On the other hand, in the list variants which do not use a sentinel, a reference to `None` is used to indicate the empty list.

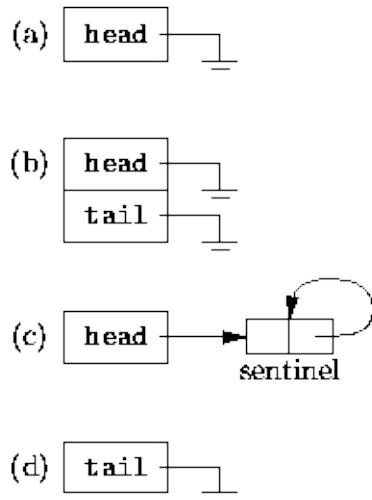


Figure: Empty singly-linked lists.

In the following sections, we will present the implementation details of a generic singly-linked list. We have chosen to present variation (b)--the one which uses a head and a tail--since it supports append and prepend operations efficiently.

-
- [An Implementation](#)
 - [List Elements](#)
 - [LinkedList Class __init__ Method](#)
 - [purge Method](#)
 - [LinkedList Properties](#)
 - [first and last Properties](#)
 - [prepend Method](#)
 - [append Method](#)
 - [copy Method](#)
 - [extract Method](#)
 - [insertAfter and insertBefore Methods](#)
-

An Implementation

Figure □ illustrates the singly-linked list scheme we have chosen to implement. Two related structures are used. The elements of the list are represented using instances of the `Element` class which comprises three instance attributes, `_list`, `_datum` and `_next`. The main structure is an instance of the `LinkedList` class which also comprises two instance attributes, `_head` and `_tail`, which refer to the first and last list elements, respectively. The `_list` instance attribute of every `Element` contains a reference to the `LinkedList` instance with which it is associated. The `_datum` instance attribute is used to refer to the objects in the list and the `_next` instance attribute refers to the next list element.

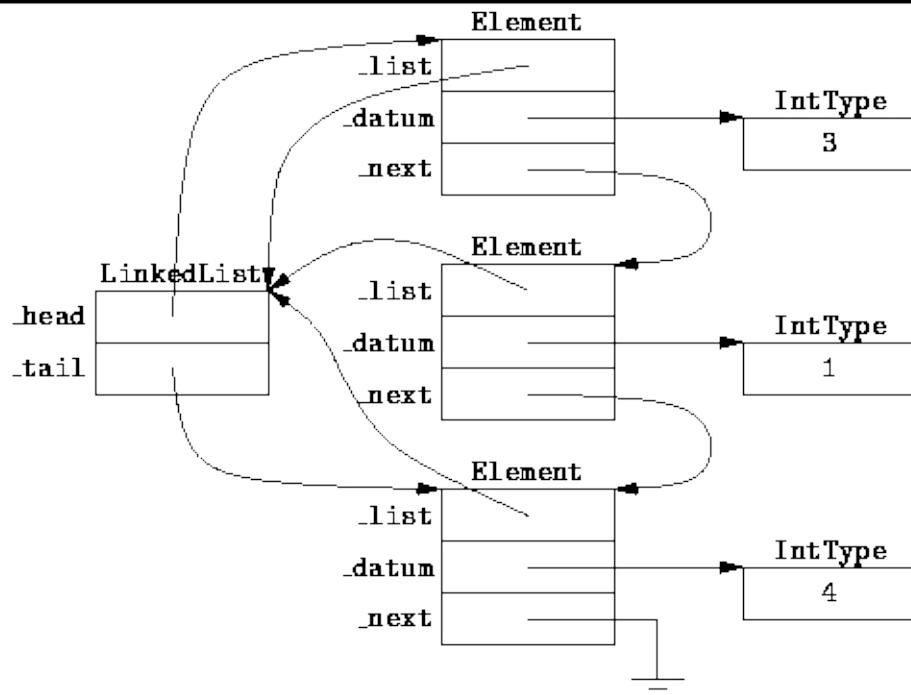


Figure: Memory representation of a linked list.

Program □ defines the `LinkedList.Element` class. It is used to represent the elements of a linked list. It has three instance attributes, `_list`, `_datum` and `_next`, an `__init__` method and two properties, `datum` and `next`.

```
1 class LinkedList(object):
2
3     class Element(object):
4
5         def __init__(self, list, datum, next):
6             self._list = list
7             self._datum = datum
8             self._next = next
9
10        def getDatum(self):
11            return self._datum
12
13        datum = property(
14            fget = lambda self: self.getDatum())
15
16        def getNext(self):
17            return self._next
18
19        next = property(
20            fget = lambda self: self.getNext())
21
22    # ...
```

Program: `LinkedList.Element` class.

We can calculate the total storage required, $S(n)$, to hold a linked list of n items from the class definitions given in Program □ as follows:

$$\begin{aligned} S(n) &= \text{sizeof}(LinkedList) + n \text{sizeof}(LinkedList.Element) \\ &= 2\text{sizeof}(id) + 3n\text{sizeof}(id) \end{aligned}$$

In Python object identifiers (addresses) occupy a constant amount of space. Therefore, $S(n)=O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



List Elements

The definitions of the methods of the `LinkedList.Element` class are given in Program □. Altogether, there are three methods--`__init__`, `getDatum` and `getNext`.

The `__init__` method simply initializes the instance attributes to the provided values. Assigning a value to the `_list`, `_datum` and `_next` instance attributes takes a constant amount of time. Therefore, the running time of the `init` method is $O(1)$.

The `getDatum` and `getNext` methods simply return the values of the corresponding instance attributes. Clearly, the running times of each of these methods is $O(1)$.

LinkedList Class `__init__` Method

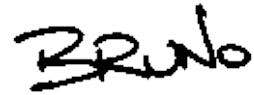
The code for the `LinkedList` class `init` method is given in Program □. Since the instance attributes `_head` and `_tail` are set to `None`, the list is empty by default. The running time of the `__init__` method is clearly constant. That is, $T(n)=O(1)$.

```
1  class LinkedList(object):
2
3      def __init__(self):
4          self._head = None
5          self._tail = None
6
7      # ...
```

Program: `LinkedList` class `__init__` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



purge Method

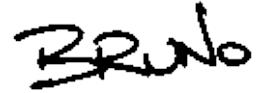
Program □ gives the code for the purge method of the `LinkedList` class. The purpose of this method is to discard the current list contents and to make the list empty again. Clearly, the running time of `purge` is $O(1)$.

```
1  class LinkedList(object):
2
3      def purge(self):
4          self._head = None
5          self._tail = None
6
7      # ...
```

Program: `LinkedList` class `purge` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



LinkedList Properties

Three `LinkedList` properties are defined in Program □. The `head` and `tail` properties provide accessors for the corresponding instance attributes of `LinkedList`. The `isEmpty` property provides an accessor that returns a `bool` result which indicates whether the list is empty. Clearly, the running time of each accessor is $O(1)$.

```
 1 class LinkedList(object):
 2
 3     def getHead(self):
 4         return self._head
 5
 6     head = property(
 7         fget = lambda self: self.getHead())
 8
 9     def getTail(self):
10         return self._tail
11
12     tail = property(
13         fget = lambda self: self.getTail())
14
15     def isEmpty(self):
16         return self._head is None
17
18     isEmpty = property(
19         fget = lambda self: self.isEmpty())
20
21     # ...
```

Program: `LinkedList` class properties.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



first and last Properties

Two more `LinkedList` properties are defined in Program □. The `first` property provides an accessor that returns the first list element. Similarly, the `last` property provides an accessor that returns the last list element. The code for both methods is almost identical. In the event that the list is empty, a `ContainerEmpty` exception is raised.

```
 1 class LinkedList(object):
 2
 3     def getFirst(self):
 4         if self._head is None:
 5             raise ContainerEmpty
 6         return self._head._datum
 7
 8     first = property(
 9         fget = lambda self: self.getFirst())
10
11     def getLast(self):
12         if self._tail is None:
13             raise ContainerEmpty
14         return self._tail._datum
15
16     last = property(
17         fget = lambda self: self.getLast())
18
19     # ...
```

Program: `LinkedList` class `first` and `last` properties.

We will assume that in a bug-free program, neither the `first` nor the `last` property accessors will be called for an empty list. In that case, the running time of each of these methods is constant. That is, $T(n)=O(1)$.

prepend Method

To *prepend* an element to a linked list is to insert that element in front of the first element of the list. The prepended list element becomes the new head of the list. Program □ gives the algorithm for the prepend method of the `LinkedList` class.

```
1  class LinkedList(object):
2
3      def prepend(self, item):
4          tmp = self.Element(self, item, self._head)
5          if self._head is None:
6              self._tail = tmp
7              self._head = tmp
8
9      # ...
```

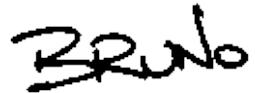
Program: `LinkedList` class prepend method.

The `prepend` method first creates a new `LinkedList.Element`. Its `_datum` instance attribute is initialized with the value to be prepended to the list, `item`; and the `_next` instance attribute refers to the first element of the existing list by initializing it with the current value of `_head`. If the list is initially empty, both `_head` and `_tail` refer to the new element. Otherwise, just `_head` needs to be updated.

Note, the `LinkedList.Element` instance is initialized by calling its `__init__` method. In Section □ the running time of the `__init__` method was determined to be $O(1)$. And since the body of the `prepend` method adds only a constant amount of work, the running time of the `prepend` method is also $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



append Method

The append method, the definition of which is given in Program □, adds a new `LinkedList.Element` at the tail-end of the list. The appended element becomes the new tail of the list.

```
1 class LinkedList(object):
2
3     def append(self, item):
4         tmp = self.Element(self, item, None)
5         if self._head is None:
6             self._head = tmp
7         else:
8             self._tail._next = tmp
9             self._tail = tmp
10
11     # ...
```

Program: `LinkedList` class append method.

The append method first allocates a new `LinkedList.Element`. Its `_datum` instance attribute is initialized with the value to be appended, and the `_next` instance attribute is set to `None`. If the list is initially empty, both `_head` and `_tail` refer to the new element. Otherwise, the new element is appended to the existing list, and the just `_tail` pointer is updated.

The running time analysis of the append method is essentially the same as for prepend. I.e, the running time is $O(1)$.

__copy__ Method

The code for the `__copy__` method of the `LinkedList` class is given in Program □. The `__copy__` method is used to create a shallow copy of a given list.

```
1  class LinkedList(object):
2
3      def __copy__(self):
4          result = LinkedList()
5          ptr = list._head
6          while ptr is not None:
7              result.append(ptr._datum)
8              ptr = ptr._next
9          return result
10
11      # ...
```

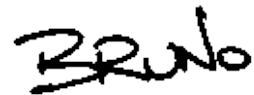
Program: `LinkedList` class `__copy__` method.

The `__copy__` method first creates a new empty `LinkedList` instance. Then, it traverses the elements of the given list one-by-one calling the `append` method to append the items to the new list.

In Section □ the running time for the `append` method was determined to be $O(1)$. If the resulting list has n elements, the `append` method will be called n times. Therefore, the running time of the `__copy__` method is $O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



extract Method

In this section we consider the extract method of the `LinkedList` class. The purpose of this method is to delete the specified element from the linked list.

```
1  class LinkedList(object):
2
3      def extract(self, item):
4          ptr = self._head
5          prevPtr = None
6          while ptr is not None and ptr._datum is not item:
7              prevPtr = ptr
8              ptr = ptr._next
9          if ptr is None:
10              raise KeyError
11          if ptr == self._head:
12              self._head = ptr._next
13          else:
14              prevPtr._next = ptr._next
15          if ptr == self._tail:
16              self._tail = prevPtr
17
18      # ...
```

Program: `LinkedList` class extract method.

The `extract` method searches sequentially for the item to be deleted. In the absence of any *a priori* knowledge, we do not know in which list element the item to be deleted will be found. In fact, the specified item may not even appear in the list!

If we assume that the item to be deleted *is* in the list, and if we assume that there is an equal probability of finding it in each of the possible positions, then on average we will need to search half way through the list before the item to be deleted is found. In the worst case, the item will be found at the tail--assuming it is in the list.

If the item to be deleted does not appear in the list, the algorithm shown in Program □ raises a `KeyError` exception . A simpler alternative might be to do nothing--after all, if the item to be deleted is not in the list, then we are already done! However, attempting to delete an item which is not there, is more likely to

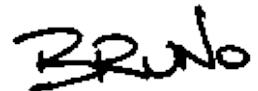
indicate a logic error in the programming. It is for this reason that an exception is raised.

In order to determine the running time of the extract method, we first need to determine the time to find the element to be deleted. If the item to be deleted *is not* in the list, then the running time of Program □ up to the point where it raises the exception (line 10) is $T(n)=O(n)$.

Now consider what happens if the item to be deleted *is* found in the list. In the worst-case the item to be deleted is at the tail. Thus, the running time to find the element is $O(n)$. Actually deleting the element from the list once it has been found is a short sequence of relatively straight-forward manipulations. These manipulations can be done in constant time. Therefore, the total running time is $T(n)=O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



insertAfter and insertBefore Methods

Consider the methods `insertAfter` and `insertBefore` of the `LinkedList.Element` class shown in Program □. Both methods take a single argument that specifies an item to be inserted into the list. The given item is inserted either in front of or immediately following this list element.

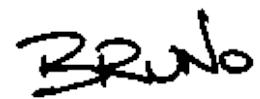
```
 1  class LinkedList(object):
 2
 3      class Element(object):
 4
 5          def insertAfter(self, item):
 6              self._next = LinkedList.Element(
 7                  self._list, item, self._next)
 8              if self._list._tail is self:
 9                  self._list._tail = self._next
10
11          def insertBefore(self, item):
12              tmp = LinkedList.Element(self._list, item, self)
13              if self is self._list._head:
14                  self._list._head = tmp
15              else:
16                  prevPtr = self._list._head
17                  while prevPtr is not None \
18                      and prevPtr._next is not self:
19                      prevPtr = prevPtr._next
20                  prevPtr._next = tmp
21
22      # ...
```

Program: `LinkedList.Element` class `insertAfter` and `insertBefore` methods.

The `insertAfter` method is almost identical to `append`. Whereas `append` inserts an item after the tail, `insertAfter` inserts an item after an arbitrary list element. Nevertheless, the running time of `insertAfter` is identical to that of `append`, i.e., it is $O(1)$.

To insert a new item *before* a given list element, it is necessary to traverse the linked list starting from the head to locate the list element that precedes the given list element. In the worst case, the given element is at the tail of the list and the entire list needs to be traversed. Therefore, the running time of the `insertBefore` method is $O(n)$.

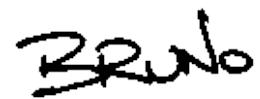
Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

Exercises

1.
 1. How much space does the `Array` class declared in Program □ use to store an array of `IntTypes` of length N ?
 2. How much space does the `LinkedList` class declared in Program □ use to store a list of n `IntTypes`?
 3. For what value of N/n do the two classes use the same amount of space?
2. The array subscripting methods defined in Program □ don't test explicitly the index expression to see if it is in the proper range. Explain why the test is not required in this implementation.
3. The `baseIndex` property `setBaseIndex` accessor of the `Array` class defined in Program □ simply changes the value of the `baseIndex` instance attribute. As a result, after the base is changed, all the array elements appear to have moved. How might the method be modified so that the elements of the array don't change their apparent locations when the base is changed?
4. Equation □ is only correct if the subscript ranges in each dimension start at zero. How does the formula change when each dimension is allowed to have an arbitrary subscript range?
5. The alternative to *row-major* layout of multi-dimensional arrays is called *column-major order*. In column-major layout the leftmost subscript expression increases fastest. For example, the elements of the columns of a two-dimensional matrix end up stored in contiguous memory locations. Modify Equation □ to compute the correct position for column-major layout.
6. Write a `plus` method for the `DenseMatrix` class defined in Program □ that implements the usual matrix addition semantics.
7. Which methods are affected if we drop the `_tail` member variable from the `LinkedList` class. Determine new running times for the affected methods.
8. How does the implementation of the `prepend` method of the `LinkedList` class defined in Program □ change when a circular list with a sentinel is used as shown in Figure □ (c).
9. How does the implementation of the `append` method of the `LinkedList` class defined in Program □ change when a circular list with a sentinel is used as shown in Figure □ (c).

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

Projects

1. Complete the implementation of the `Array` class given in Program □ to Program □. Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.
2. Complete the implementation of the `LinkedList` class given in Program □ to Program □. Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.
3. Change the implementation of the `LinkedList` class given in Program □ to Program □ by removing the `_tail` instance attribute. That is, implement the singly-linked list variant shown in Figure □ (a). Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.
4. Change the implementation of the `LinkedList` class given in Program □ to Program □ so that it uses a circular, singly-linked list with a sentinel as shown in Figure □ (c). Write a test suite to verify all of the functionality. Try to exercise every line of code in the implementation.
5. The `MultiDimensionalArray` class given in Program □ to Program □ only supports subscript ranges starting at zero. Modify the implementation to allow an arbitrary subscript base in each dimension.
6. Design and implement a three-dimensional matrix class `Matrix3D` based on the two-dimensional class `DenseMatrix` given in Program □ to Program □
7. A row vector is a $1 \times n$ matrix and a column vector is an $n \times 1$ matrix. Define and implement classes `RowVector` and `ColumnVector` as classes derived from the base class `Array` given in Program □ to Program □. Show how these classes can be combined to implement the `Matrix` interface declared in Program □.

Data Types and Abstraction

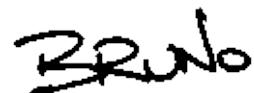
It is said that ``computer science is [the] science of *abstraction*[[2](#)].'' But what exactly is abstraction? Abstraction is ``the idea of a quality thought of apart from any particular object or real thing having that quality''[[10](#)]. For example, we can think about the size of an object without knowing what that object is. Similarly, we can think about the way a car is driven without knowing its model or make.

Abstraction is used to suppress irrelevant details while at the same time emphasizing relevant ones. The benefit of abstraction is that it makes it easier for the programmer to think about the problem to be solved.

-
- [Abstract Data Types](#)
 - [Design Patterns](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Abstract Data Types

An object in a programming language such as C++ , Java , and Python , is an abstraction. The abstraction comprises a number of *attributes*--name , address , value , lifetime , scope , type , and size . Each attribute has an associated value. For example, if we create an integer object in Python, `x = int(5)`, we say that the name attribute has value ``x'' and that the type attribute has value ``int''.

Unfortunately, the terminology can be somewhat confusing: The word ``value'' has two different meanings--in one instance it denotes one of the attributes and in the other it denotes the quantity assigned to an attribute. For example, after the statement `x = int(5)`, the *value attribute* of the object named `x` has the *value* five.

The *name* of an object is the textual label used to refer to that object in the text of the source program. The *address* of an object denotes its location in memory. The *value* attribute is the quantity which that object represents. The *lifetime* of an object is the interval of time during the execution of the program in which the object is said to exist. The *scope* of an object is the set of statements in the text of the source program in which the object is said to be *visible* . The *type* of an object denotes the set of values which can be assigned to the *value* attribute and the set of operations which can be performed on the object. Finally, the *size* attribute denotes the amount of storage required to represent the object.

The process of assigning a value to an attribute is called *binding* . When a value is assigned to an attribute, that attribute is said to be *bound* to the value.

Depending on the semantics of the programming language, and on the attribute in question, the binding may be done statically by the compiler or dynamically at run-time. For example, in Python the *type* of an object is usually determined at compile time--*static binding* . On the other hand, the *value* of an object is usually not determined until run-time--*dynamic binding* .

In this chapter we are concerned primarily with the *type* attribute of an object. The type of an object specifies two sets:

- a set of values; and,
- a set of operations.

For example, when we create an object, say `x`, of type `int`, we know that `x` can represent an integer in the range $[-2^{31}, 2^{31} - 1]$ and that we can perform operations using `x` such as addition, subtraction, multiplication, and division.

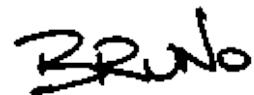
The type `int` is an *abstract data type* in the sense that we can think about the qualities of an `int` apart from any real thing having that quality. In other words, we don't need to know *how* `ints` are represented nor how the operations are implemented to be able to use them or reason about them.

In designing *object-oriented* programs, one of the primary concerns of the programmer is to develop an appropriate collection of abstractions for the application at hand, and then to define suitable abstract data types to represent those abstractions. In so doing, the programmer must be conscious of the fact that defining an abstract data type requires the specification of *both* a set of values and a set of operations on those values.

Indeed, it has been only since the advent of the so-called *object-oriented programming languages* that we see programming languages which provide the necessary constructs to properly declare abstract data types. For example, in Python, the `class` construct is the means by which both a set of values and an associated set of operations is declared. Compare this with the `struct` construct of C or Pascal's `record`, which only allow the specification of a set of values!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Design Patterns

An experienced programmer is in a sense like concert musician--he has mastered a certain *repertoire* of pieces which he is prepared to play at any time. For the programmer, the repertoire comprises a set of abstract data types with which he is familiar and which he is able to use in his programs as the need arises.

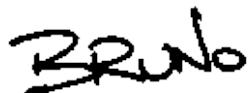
The chapters following this present a basic repertoire of abstract data types. In addition to defining the abstractions, we show how to implement them in Python and we analyze the performance of the algorithms.

The repertoire of basic abstract data types has been designed as a hierarchy of Python classes. This section presents an overview of the class hierarchy and lays the groundwork for the following chapters.

- [Class Hierarchy](#)
 - [Abstract Objects and the `__builtin__.object` Class](#)
 - [Containers](#)
 - [Iterators](#)
 - [Visitors](#)
 - [Searchable Containers](#)
 - [Associations](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Class Hierarchy

The Python class hierarchy which is used to represent the basic repertoire of abstract data types is shown in Figure □. Two kinds of classes are shown in Figure □; *abstract Python classes*, which look like this `AbstractClass`, and *concrete Python classes*, which look like this `ConcreteClass`. Arrows in the figure indicate the *specializes* relation between classes; an arrow points from a derived class to the base class from which it is derived.

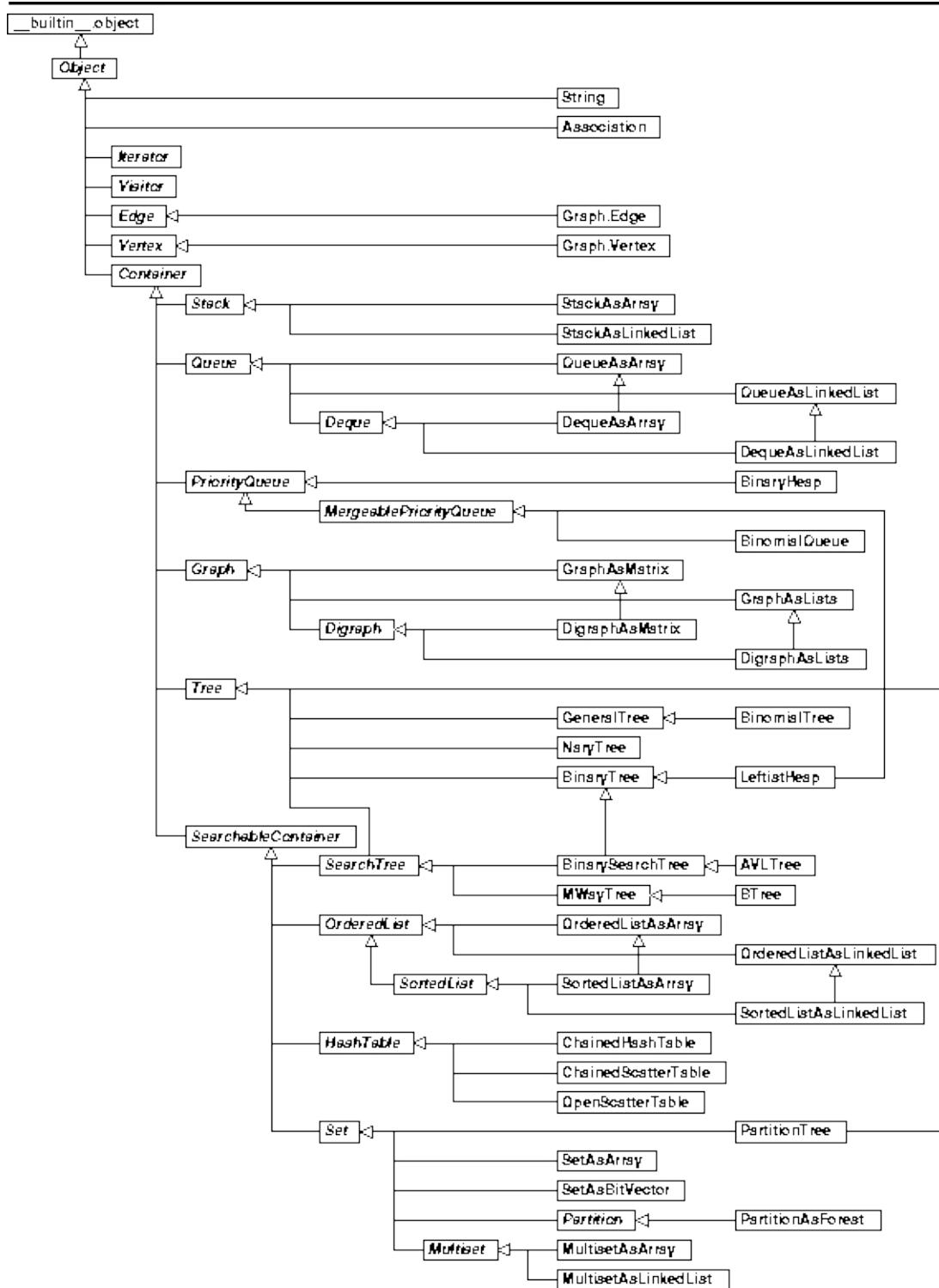


Figure: Object class hierarchy.

The distinction between an *abstract class* and a *concrete class* is purely one of convention. These concepts are not built into the Python language. Nevertheless,

it is possible to write Python programs in a way that makes it clear that a class is an abstract class.

An *abstract class* is a class which defines only part of an implementation. Consequently, it does not make sense to create object instances of abstract classes. By convention, an abstract class may contain zero or more *abstract methods* or *abstract properties*. As with classes, the distinction between abstract methods or properties and concrete methods or properties is purely one of convention. An *abstract method* or property is one for which no implementation is given.

An abstract class is intended to be used as the *base class* from which other classes are *derived*. The derived classes are expected to *override* the abstract methods and properties of the base classes. By defining abstract methods in the base class, it is possible to understand how an object of a derived class can be used. We don't need to know how a particular object instance is implemented, nor do we need to know of which derived class it is an instance.

This design pattern uses the idea of *polymorphism*. Polymorphism literally means ``having many forms.'' The essential idea is that a base class is used to define the set of values and the set of operations--the abstract data type. Then, various different implementations (*many forms*) of the abstract data type can be made. We do this by defining *abstract classes* that contain shared implementation features and then by deriving concrete classes from the abstract base classes.

The remainder of this section presents the top levels of the class hierarchy shown in Figure □. The top levels define those attributes of objects which are common to all of the classes in the hierarchy. The lower levels of the hierarchy are presented in subsequent chapters where the abstractions are defined and various implementations of those abstractions are elaborated.

Abstract Objects and the `__builtin__.object` Class

All Python classes are ultimately derived from the base class called `object`.[◇] The `object` class is defined in the Python `__builtin__` module. The following code fragment identifies some of the methods defined in the `__builtin__.object` class[◇]:

```
class object:
    def __new__(...): ...
    def __init__(...): ...
    def __getattribute__(...): ...
    def __setattr__(...): ...
    def __delattr__(...): ...
    def __hash__(...): ...
    def __repr__(...): ...
    def __str__(...): ...
    ...
```

-
- [Abstract Objects](#)
 - [Abstract Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Abstract Objects

Most of the classes described in this book are derived from a common base class called `Object`. As shown in Figure □, the `Object` class is an *abstract* class that extends the `__builtin__.object` class

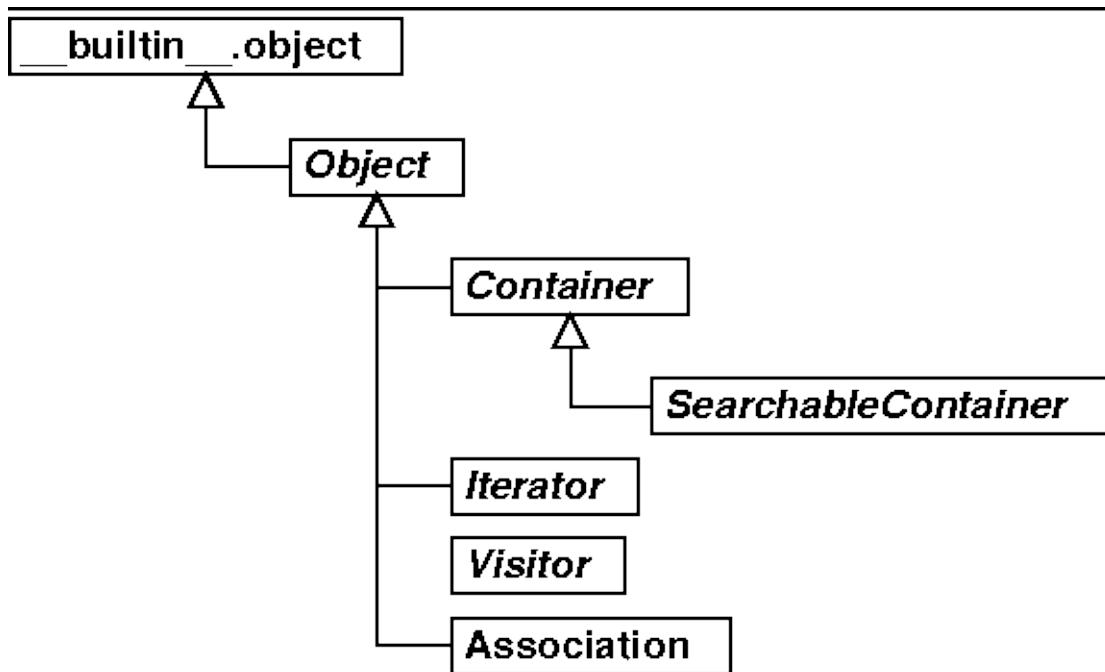


Figure: Object class hierarchy.

Program □ gives the Python code that defines the `Object` class.

```
 1 class Object(object):
 2
 3     def __init__(self):
 4         super(Object, self).__init__()
 5
 6     def __cmp__(self, obj):
 7         if isinstance(self, obj.__class__):
 8             return self._compareTo(obj)
 9         elif isinstance(obj, self.__class__):
10             return -obj._compareTo(self)
11         else:
12             return cmp(self.__class__.__name__,
13                        obj.__class__.__name__)
14
15     def _compareTo(self, obj):
16         pass
17     _compareTo = abstractmethod(_compareTo)
18     # ...
```

Program: Object class definition.

Notice that the `Object` class defines a method called `__cmp__`. This instance method takes a specified object and compares it with the given object instance. The method returns an integer that is less than, equal to, or greater than zero depending on whether this object instance is less than, equal to, or greater than the specified object instance `obj`, respectively.

The `__cmp__` method is special in several respects. First, it is the method that is called by the standard `cmp` function. I.e., given two `Object` instances, the following statements are equivalent:

```
result = cmp(obj1, obj2)
result = obj1.__cmp__(obj2)
```

Second, the `__cmp__` method is automatically called whenever one of the comparison operators, (`<`, `<=`, `==`, `!=`, `>`, and `>=`) is used to compare two `Object` instances. For example, the following statements are equivalent:

```
result = obj1 < obj2
result = obj1.__cmp__(obj2) < 0
```

Notice how the `__cmp__` method of the `Object` class is implemented. When comparing two object instances, `obj1` and `obj2`, the comparison works as follows:

1. If the class of obj1 is a subclass of the class of obj2, then the result of the comparison is obj1._compareTo(obj2).
2. If the class of obj2 is a subclass of the class of obj1, then the result of the comparison is obj2._compareTo(obj1).
3. If the classes are unrelated, the the result is obtained by comparing the *names* of the classes.

The `_compareTo` method is declared to be an *abstract* method. This means that it one must be provided in the implementation of every concrete class derived from the `Object` base class. The implementation of the `_compareTo` method for a given class is simplified because it will always be the case that the given class is a subclass of the class of the `_compareTo` argument.

The use of polymorphism in the way shown gives the programmer enormous leverage. The fact most objects are derived from the `Object` base class, together with the fact that every concrete class must implement an appropriate `_compareTo` method, ensures that the comparison operators can be used to compare any pair of objects and that the comparisons always work as expected.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Abstract Methods

The `_compareTo` method given in Program □ is an abstract method. In particular, the body of the method does nothing. It is expected that the `_compareTo` method will be overridden in a class derived from the `Object` class. However, this is only a convention--the Python language does nothing to ensure that the method is overridden. And if the method is not overridden, it remains possible for a Python program to call the abstract method.

To prevent the calling of abstract methods, we introduce the `abstractmethod descriptor` class shown in Program □. The idea is to wrap a function object that is supposed to be an abstract method in an instance of the `abstractmethod` class as shown on line 17 of Program □.

```
 1  class abstractmethod(object):
 2
 3      def __init__(self, func):
 4          assert inspect.isfunction(func)
 5          self._func = func
 6
 7      def __get__(self, obj, type):
 8          return self.method(obj, self._func, type)
 9
10      class method(object):
11
12          def __init__(self, obj, func, cls):
13              self._self = obj
14              self._func = func
15              self._class = cls
16              self.__name__ = func.__name__
17
18          def __call__(self, *args, **kwargs):
19              msg = "Abstract method %s of class %s called." % (
20                  self._func.__name__, self._class.__name__)
21              raise TypeError, msg
```

Program: `abstractmethod` class definition.

The `abstractmethod` class does its magic by defining a special `__get__` method.

Python uses the `__get__` method of a class to bind a method to an instance. For example, suppose you have an instance `obj` of some class `cls`. When you call a method like this:

```
obj.func(*args)
```

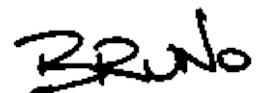
Python effectively does this:

```
meth = __get__(func, obj, cls)
meth.__call__(*args)
```

In Program □, we implement a `__get__` method that returns an instance of the nested `abstractmethod.method` class. This class in turn implements a `__call__` method that raises a `TypeError` exception. The effect of this is that whenever an abstract method is called an exception is raised.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Containers

A container is an object that contains within it other objects. Many of the data structures presented in this book can be viewed as containers. For this reason, we develop a common abstract base class that is extended by the various data structure classes.

The Container base class is defined in Program □ to Program □. The Container class defines the concrete methods `__init__`, `getCount`, `getIsEmpty`, `getIsFull`, `accept` and the abstract methods `purge` and `__iter__` and the properties `count`, `isFull` and `isEmpty`.

Conspicuous by their absence are methods for putting objects into a container and for taking them out again. These methods have been omitted from the Container class, because the precise nature of these methods depends on the type of container implemented.

-
- [Container `__init__`, hookiter and purge methods](#)
 - [Container Properties](#)
-

Container `__init__`, `hookiter` and `purge` methods

A container may be empty or it may contain one or more other objects.

Typically, a container has finite capacity. As shown in Program □, a single instance attribute called `_count` is used to keep track of the number of objects held in the container. The `_count` instance attribute is set initially to zero. It is the responsibility of the derived class to update this instance attribute as required.

```
 1  class Container(Object):
 2
 3      def __init__(self):
 4          super(Container, self).__init__()
 5          self._count = 0
 6
 7      def purge(self):
 8          pass
 9      purge = abstractmethod(purge)
10
11      def __iter__(self):
12          pass
13      __iter__ = abstractmethod(__iter__)
14
15      # ...
```

Program: Container class.

The purpose of the `purge` method is to discard all of the contents of a container. This method is declared abstract because the manner in which the `purge` method is implemented depends on the type of the container. After a container is purged, the value of the `_count` instance attribute should be zero.

The purpose of the `__iter__` method is to return an iterator that enumerates the objects in the container. Iterators are discussed in Section □.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Container Properties

Program □ defines the methods `getCount`, `getIsEmpty` and `getIsFull`. The `getCount` method simply returns the number of objects in the container.

```
 1 class Container(Object):
 2
 3     def getCount(self):
 4         return self._count
 5
 6     count = property(
 7         fget = lambda self: self.getCount())
 8
 9     def getIsEmpty(self):
10         return self.count == 0
11
12     isEmpty = property(
13         fget = lambda self: self.getIsEmpty())
14
15     def getIsFull(self):
16         return False
17
18     isFull = property(
19         fget = lambda self: self.getIsFull())
20
21     # ...
```

Program: Container methods and properties.

The `getIsEmpty` and `getIsFull` methods are bool-valued methods which indicate whether a given container is empty or full, respectively. Notice that the `getIsEmpty` get accessor does not directly access the `count` instance attribute. Instead it uses `Count` property. As long as the `Count` property has the correct semantics, the `IsEmpty` property will too.

In some cases, a container is implemented in a way which makes its capacity finite. When this is the case, it is necessary to be able to determine when the container is full. `IsFull` is a bool-valued property that provides a get accessor that returns the value `True` if the container is full. The default version always

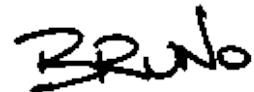
returns False.

Program □ also defines the properties `count`, `isEmpty` and `isFull`. Properties are Python object attributes that are referenced like instance attributes, but which are implemented using methods. For example, a reference to the `count` property results in a call to the `getCount` method. Similarly, a reference to the `isEmpty` property results in a call to the `getIsEmpty` method, and a reference to the `isFull` property results in a call to the `getIsFull` method.

Notice that the properties are implemented as lambda expressions. The reason for this is that when a Python property is created it is bound to a specific method instance. The problem with this is that if a property is bound to a specific method instance in a base class, and a derived method overrides the method, the property still remains bound to the original base class method. Thus, we would have to also override the property in the derived class in order to get the desired behavior. By binding the property to a lambda function that calls the desired method, when the method is overridden in a derived class, the property need not also be overridden in that derived class.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Iterators

In this section we introduce an abstraction called an *iterator*. An iterator provides the means to access one-by-one all the objects in a container.

A Python iterator is any class that provides two methods called `__iter__` and `next` and that implements the *iterator protocol* which we describe below. In order to use an iterator to access the objects in a container, that container must provide iteration support. Specifically, that class must implement a method called `__iter__` that returns the corresponding iterator for that class.

The idea is that for every concrete container class derived from the `Container` base class we will also implement a related class derived from the abstract `Iterator` class shown in Program □

```
 1  class Iterator(Object):
 2
 3      def __init__(self, container):
 4          super(Object, self).__init__()
 5          self._container = container
 6
 7      def __iter__(self):
 8          return self
 9
10      def next(self):
11          pass
12          next = abstractmethod(next)
13
14      # ...
```

Program: Iterator class.

The `Iterator` class comprises three methods, `__init__`, `__iter__`, and `next`. The `__init__` and `__iter__` methods are straightforward. The abstract `next` method provided in a derived class is required to implement the iterator protocol.

In order to understand the iterator protocol, it is best to consider first an example which illustrates the use of an iterator. Consider a concrete container class, say `SomeContainer`, that implements the `Container` interface. The following code

fragment illustrates the use of an iterator to access one-by-one the objects contained in the container:

```
c = SomeContainer()  
# ...  
i = iter(c) # Equivalent to c.__iter__()  
while True:  
    try:  
        obj = i.next()  
        print obj  
    except StopIteration:  
        break
```

In order to have the desired effect, the `next` method must behave as follows:

1. Each time the `next` method is called, it returns the next element in the container.
2. When all the elements in the container have been returned, the `next` method raises the `StopIteration` exception.

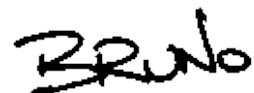
Given these semantics for the `next` method, the program fragment shown above systematically visits all of the objects in the container and prints each one on its own line of the console.

A certain amount of care is required when defining and using iterators. In order to simplify the implementation of iterators, we shall assume that while an iterator is in use, the associated container will not be modified.

-
- [Iterators and the Python `for` statement](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Iterators and the Python for statement

The Python compiler automatically generates code to use an iterator when the `for` statement is used. Thus, given an object `c` that is an instance of a concrete class `SomeContainer` derived from the abstract `Container` base class, we can use the `for` statement to enumerate the objects in the container as follows:

```
c = new SomeContainer();
# ...
for obj in c:
    print obj
```

One of the advantages of using an iterator object that is separate from the container is that it is possible to have more than one iterator associated with a given container. For example, consider the following code fragment:

```
c = SomeContainer()
# ...
for obj1 in c:
    for obj2 in c:
        if obj1 == obj2:
            print obj1, obj2
```

This code implicitly uses two distinct iterators, one for each `for` loop. This code compares all ordered-pairs of objects in the container `c` and prints out those which are equal.

Visitors

In this section we introduce an abstraction called a *visitor*. A visitor provides the means to access one-by-one all the objects in a container and to perform a given operation on those objects. Program □ defines the abstract class `Visitor` which is the base class from which all visitors are derived.

```
 1  class Visitor(object):
 2
 3      def __init__(self):
 4          super(Visitor, self).__init__()
 5
 6      def visit(self, obj):
 7          pass
 8
 9      def getIsDone(self):
10          return False
11
12      isDone = property(
13          fget = lambda self: self.getIsDone())
```

Program: Visitor interface.

The `Container` class defines an abstract method called `accept` as shown in Program □. The idea is that the `accept` method of the `Container` class takes as its argument a any object that is an instance of a class derived from the `Visitor` class.

```
 1  class Container(object):
 2
 3      def accept(self, visitor):
 4          assert isinstance(visitor, Visitor)
 5          for obj in self:
 6              visitor.visit(obj)
 7
 8      # ...
```

Program: Container class accept method.

But what is a visitor? As shown in Program □, a visitor is an object that has a

`visit` method and an `isDone` property. Of these, the `visit` method is the most interesting. The `visit` method takes as its argument an `Object` instance.

The interaction between a container and a visitor goes like this: The container is passed a visitor by calling the container's `accept` method. That is, the container ``accepts'' the visitor. What does a container do with a visitor? It calls the `visit` method of that visitor one-by-one for each object contained in the container.

The interaction between a `Container` and its `Visitor` are best understood by considering an example. The following code fragment gives an implementation of the `Accept` method in some concrete class, say `SomeContainer`, that implements the `Container` interface:

```
class SomeContainer(Container):

    def accept(self, visitor):
        for i in self:
            visitor.visit(i)

    # ...
```

The `accept` method calls `visit` for each object `i` in the container. Note, the `visit` method in the `Visitor` class is abstract. What a visitor actually does with an object depends on the actual class of visitor used.

Suppose that we want to print all of the objects in the container. One way to do this is to create a `PrintingVisitor` which prints every object it visits, and then to pass the visitor to the container by calling the `accept` method. The following code shows how we can declare the `PrintingVisitor` class which prints an object on the console.

```
class PrintingVisitor(Visitor):

    def visit(self, obj):
        print obj

    # ...
```

Finally, given an object `c` that is an instance of a concrete class `SomeContainer` derived from the abstract `Container` class, we can call the `accept` method as follows:

```
c = SomeContainer()
```

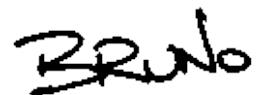
```
// ...
c.accept(PrintingVisitor())
```

The effect of this call is to call the `visit` method of the visitor for each object in the container.

- [The `isDone` Property](#)
 - [The Container Class `str` Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





The `isDone` Property

As shown in Program □, the `Visitor` class also includes the property `isDone`. The `isDone` property calls the `getIsDone` accessor to determine whether a visitor has finished its work. That is, the `getIsDone` method returns the `bool` value `True` if the visitor ``is done.''

The idea is this: Sometimes a visitor does not need to visit all the objects in a container. That is, in some cases, the visitor may be finished its task after having visited only a some of the objects. The `isDone` property can be used by the container to terminate the `accept` method early like this:

```
class SomeContainer(Container):

    def accept(self, visitor):
        for i in self:
            if visitor.isDone:
                return
            visitor.visit(i)
    # ...
```

To illustrate the usefulness of `isDone`, consider a visitor which visits the objects in a container with the purpose of finding the first object that matches a given target object. Having found the first matching object in the container, the visitor is done and does not need to visit any more contained objects.

The following code fragment defines a visitor which finds the first object in the container that matches a given object.

```
class MatchingVisitor(Visitor):

    def __init__(self, target):
        self.target = target
        self.found = None

    def visit(self, obj):
        if not self.isDone and obj == target:
            self.found = obj
```

```
def isDone(self):  
    return found not is None:
```

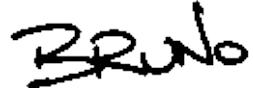
The `__init__` method of the `MatchingVisitor` visitor takes an `Object` instance that is the target of the search. That is, we wish to find an object in a container that matches the target. For each object the `MatchingVisitor` visitor visits, it compares that object with the target and makes `found` point at that object if it matches. Clearly, the `MatchingVisitor` visitor is done when the `found` pointer is non-zero.

Suppose we have a container `c` that is an instance of a concrete container class, `SomeContainer`, derived from the abstract `Container` class; and an object `x` that is an instance of a concrete object class, `SomeObject`, derived from the abstract `Object` class. Then, we can call use the `MatchingVisitor` visitor as follows:

```
c = SomeContainer()  
x = SomeObject()  
// ...  
c.accept(MatchingVisitor(x))
```

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





The Container Class `__str__` Method

One of the methods defined in the Python object class is the `__str__` method. Consequently, every Python object supports the `__str__` method. The `__str__` method is required to return a string that represents the object ``textually.'' It is typically invoked in situations where it is necessary to print out a human-readable representation of an object.

Program □ defines the `__str__` method of the abstract Container class. This method is provided to simplify the implementation of classes derived from the Container class. The default behavior is to print out the name of the class and then to print each of the elements in the container, by using the `accept` method together with a visitor.

```
 1  class Container(Object):
 2
 3      class StrVisitor(Visitor):
 4
 5          def __init__(self):
 6              self._string = ""
 7              self._comma = False
 8
 9          def visit(self, object):
10              if self._comma:
11                  self._string = self._string + ", "
12              self._string = self._string + str(object)
13              self._comma = True
14
15          def __str__(self):
16              return self._string
17
18      def __str__(self):
19          visitor = Container.StrVisitor()
20          self.accept(visitor)
21          return "%s (%s)" % (self.__class__.__name__, str(visitor))
22
23      # ...
```

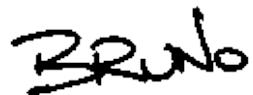
Program: Container class `__str__` method.

The `strvisitor` is a visitor. It uses its `_string` instance attribute to accumulate the textual representations of the objects it visits. (It also makes sure to put in commas as required).

The final result returned by the `Container` class `__str__` method consists of the name of the container class, followed by a comma-separated list of the contents of that container enclosed in braces { and }.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Searchable Containers

A *searchable container* is an extension of the container abstraction. It adds to the set of methods provided for containers methods for putting objects in and taking objects out, for testing whether a given object is in the container, and a method to search the container for a given object.

The definition of the SearchableContainer abstract class is shown in Program □. The SearchableContainer abstract class extends the Container abstract class given in Program □. It adds four new abstract methods (and overrides the `__init__` method).

```
 1  class SearchableContainer(Container):
 2
 3      def __init__(self):
 4          super(SearchableContainer, self).__init__()
 5
 6      def __contains__(self, obj):
 7          pass
 8          __contains__ = abstractmethod(__contains__)
 9
10      def insert(self, obj):
11          pass
12          insert = abstractmethod(insert)
13
14      def withdraw(self, obj):
15          pass
16          withdraw = abstractmethod(withdraw)
17
18      def find(self, obj):
19          pass
20          find = abstractmethod(find)
```

Program: SearchableContainer abstract class.

The `__contains__` method is a bool-valued method which takes as its argument any object derived from the `Object` abstract base class. The purpose of this method is to test whether the given object instance is in the container.

The `__contains__` method is treated specially in Python. It is automatically

called whenever the operator `in` is called. Thus, the following two statements are equivalent:

```
result = c.__contains__(obj)
result = c in obj
```

The purpose of the `insert` method is to put an object into the container. The `insert` method takes an object and inserts it into the container. Similarly, the `withdraw` method is used to remove an object from a container. The argument refers to the object to be removed.

The final method, `find`, is used to locate an object in a container and to return a reference to that object. In this case, it is understood that the search is to be done using the comparison methods defined in the `Object` abstract base class. That is, the `find` method is *not* to be implemented as a search of the container for the given object instance but rather as a search of the container for an object that compares equal to the given object.

This is an important subtlety in the semantics of `find`: The search is not for the given object, but rather for an object that compares equal to the given object. These semantics are particularly useful when using *associations*, which are defined in Section □.

In the event that the `find` method fails to find an object equal to the specified object, then it will return the object `None`. Therefore, the user of the `find` method should test explicitly the returned value to determine whether the search was successful. Also, the `find` method does *not* remove the object it finds from the container. An explicit call of the `withdraw` method is needed to actually remove the object from the container.

Associations

An association is an ordered pair of objects. The first element of the pair is called the *key* ; the second element is the *value* associated with the given key.

Associations are useful for storing information in a database for later retrieval. For example, a database can be viewed as a container that holds key-and-value pairs. The information associated with a given key is retrieved from the database by searching the database for an the ordered pair in which the key matches the given key.

Program □ introduces the Association class. The Association class concrete extension of the abstract Object class given in Program □.

```
 1  class Association(Object):
 2
 3      def __init__(self, *args):
 4          if len(args) == 1:
 5              self._tuple = (args[0], None)
 6          elif len(args) == 2:
 7              self._tuple = args
 8          else:
 9              raise ValueError
10
11      # ...
```

Program: Association class `__init__` method.

As shown in Program □, we implement an association using a Python tuple . The `__init__` method always creates a tuples with exactly two elements. The first element of the tuple represents the key of the association and the second element of the tuple represents the value associated with the given key.

Two properties of an Association are defined in Program □. The `key` property returns the value of the first element of the tuple and the `value` property returns the value of the second element of the tuple.

```
 1 class Association(object):
 2
 3     def getKey(self):
 4         return self._tuple[0]
 5
 6     key = property(
 7         fget = lambda self: self.getKey())
 8
 9     def getValue(self):
10         return self._tuple[1]
11
12     value = property(
13         fget = lambda self: self.getValue())
14
15     # ...
```

Program: Association properties.

The remaining methods of the `Association` class are defined in Program □. The `_compareTo` method is used to compare associations. Its argument is an object that is assumed to be an instance of the `Association` class. The `_compareTo` method is one place where an association distinguishes between the key and the value. In this case, the result of the comparison is based solely on the keys of the two associations--the values have no role in the comparison.

```
 1 class Association(object):
 2
 3     def _compareTo(self, assoc):
 4         assert isinstance(self, assoc.__class__)
 5         return cmp(self.key, assoc.key)
 6
 7     def __str__(self):
 8         return "Association %s" % str(self._tuple)
 9
10     # ...
```

Program: Association methods.

Program □ also defines a `__str__` method. The purpose of the `__str__` method is to return a textual representation of the association. In this case, the implementation is trivial and needs no further explanation.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Exercises

1. Specify the set of values and the set of operations provided by each of the following Python types:
 1. `int`,
 2. `float`, and
 3. `str`.
2. What are the features of Python that facilitate the creation of *user-defined* data types.
3. Explain how each of the following Python features supports *polymorphism*:
 1. classes,
 2. inheritance, and
 3. special methods like `__cmp__` and `__ismember__`.
4. Suppose we define two concrete classes, `A` and `B`, both of which are derived from the abstract `Object` class declared in Program □. Furthermore, let `a` and `b` be instances of classes `A` and `B` (respectively) declared as follows:

```
class A(Object):
    ...
class B(Object):
    ...
a = A()
b = B()
```

Give the sequence of methods called in order to evaluate a comparison such as ```a < b`''. Is the result of the comparison `True` or `False`? Explain.

5. Let `c` be an instance of some concrete class derived from the `Container` class given in Program □. Explain how the statement

```
print str(c)
```

prints the contents of the container on the console.

6. Suppose we have a container `c` (i.e., an instance of some concrete class derived from the `Container` class defined in Program □) which among other things happens to contain itself. What happens when we invoke the `__str__` method on `c`?
7. Iterators and visitors provide two ways to do the same thing--to visit one-by-one all the objects in a container. Give an implementation for the `accept` method of the `Container` class that uses an iterator.

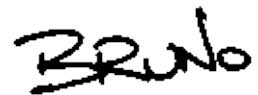
8. Is it possible to implement an iterator using a visitor? Explain.
9. Suppose we have a container which we know contains only plain integers.
Design a Visitor which computes the sum of all the integers in the container.
10. Consider the following pair of Associations:

```
a = Association(3, 4)  
b = Association(3)
```

Give the sequence of methods called in order to evaluate a comparison such as ``a == b''. Is the result of the comparison True or False? Explain.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Projects

1. Design and implement suitable wrapper classes that extend the `Object` base class for the Python types `bool`, `int`, `str`, and `tuple`.
2. Using *visitors*, devise an implementation for the `__contains__` method and the `find` method of the `SearchableContainer` class declared in Program □.
3. Using an *iterator*, devise an implementation for the `__contains__` method and the `find` method of the `SearchableContainer` class declared in Program □.
4. Devise a scheme using visitors whereby all of the objects contained in one searchable container can be removed from it and transferred to another container.
5. A *bag* is a simple container that can hold a collection of objects. Design and implement a concrete class called `Bag` that extends the `SearchableContainer` class defined in Program □. Use the `ArrayList` class given in Chapter □ to keep track of the contents of the bag.
6. Repeat Project □, this time using the `LinkedList` class given in Chapter □.
7. In Java it is common to use an *enumeration* as the means to iterate through the objects in a container. In Python we can define an enumeration like this:

```
class Enumeration(Object):
    def hasMoreElements():
        pass
    hasMoreElements = abstractmethod(hasMoreElements)

    def nextElement():
        pass
    nextElement = abstractmethod(nextElement)
```

Given an enumeration `e` from some container `c`, the contents of `c` can be printed like this:

```
while (e.hasMoreElements()):
    print e.nextElement()
```

Devise a wrapper class to encapsulate a Python iterator and provide the functionality of a Java enumeration.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Stacks, Queues, and Deques

In this chapter we consider several related abstract data types--stacks, queues, and deques. Each of these can be viewed as a pile of items. What distinguishes each of them is the way in which items are added to or removed from the pile.

In the case of a *stack*, items are added to and removed from the top of the pile. Consider the pile of papers on your desk. Suppose you add papers only to the top of the pile or remove them only from the top of the pile. At any point in time, the only paper that is visible is the one on top. What you have is a *stack*.

Now suppose your boss comes along and asks you to complete a form immediately. You stop doing whatever it is you are doing, and place the form on top of your pile of papers. When you have filled-out the form, you remove it from the top of the stack and return to the task you were working on before your boss interrupted you. This example illustrates that a *stack* can be used to keep track of partially completed tasks.

A *queue* is a pile in which items are added at one end and removed from the other. In this respect, a queue is like the line of customers waiting to be served by a bank teller. As customers arrive, they join the end of the queue while the teller serves the customer at the head of the queue. As a result, a *queue* is used when a sequence of activities must be done on a *first-come, first-served* basis.

Finally, a *deque* extends the notion of a queue. In a deque, items can be added to or removed from either end of the queue. In a sense, a deque is the more general abstraction of which the stack and the queue are just special cases.

As shown in Figure □, we view stacks, queues, and deques as containers. This chapter presents a number of different implementation alternatives for stacks, queues, and deques. All of the concrete classes presented are extensions of the abstract Container class defined in Chapter □.

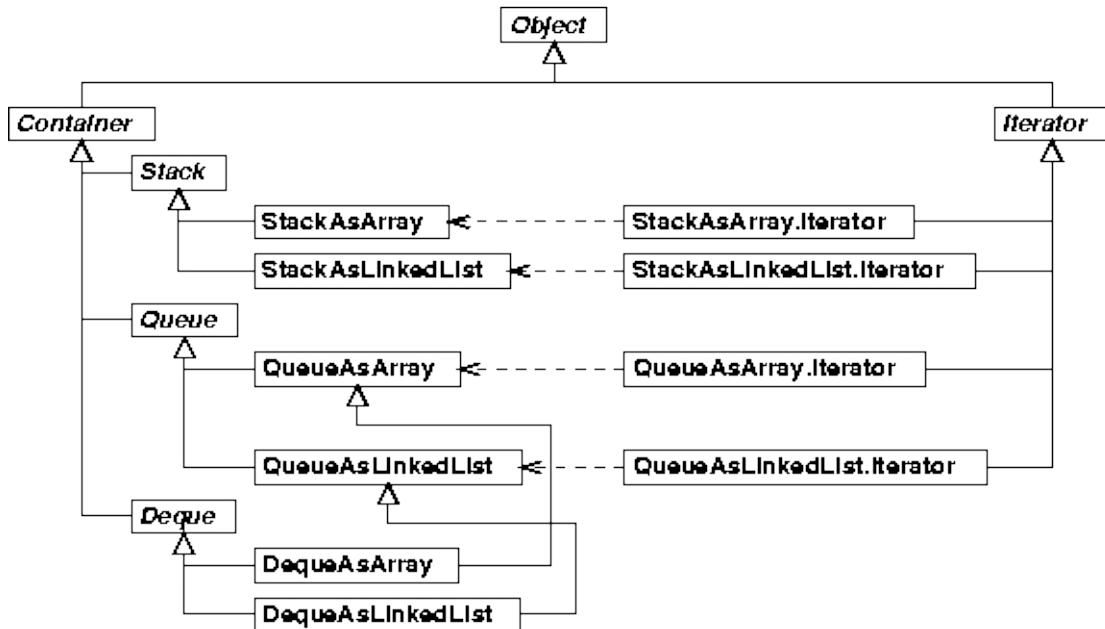


Figure: Object class hierarchy.

- [Stacks](#)
- [Queues](#)
- [Deques](#)
- [Exercises](#)
- [Projects](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Stacks

The simplest of all the containers is a *stack*. A stack is a container which provides exactly one method, `push`, for putting objects into the container; and one method, `pop`, for taking objects out of the container. Figure □ illustrates the basic idea.

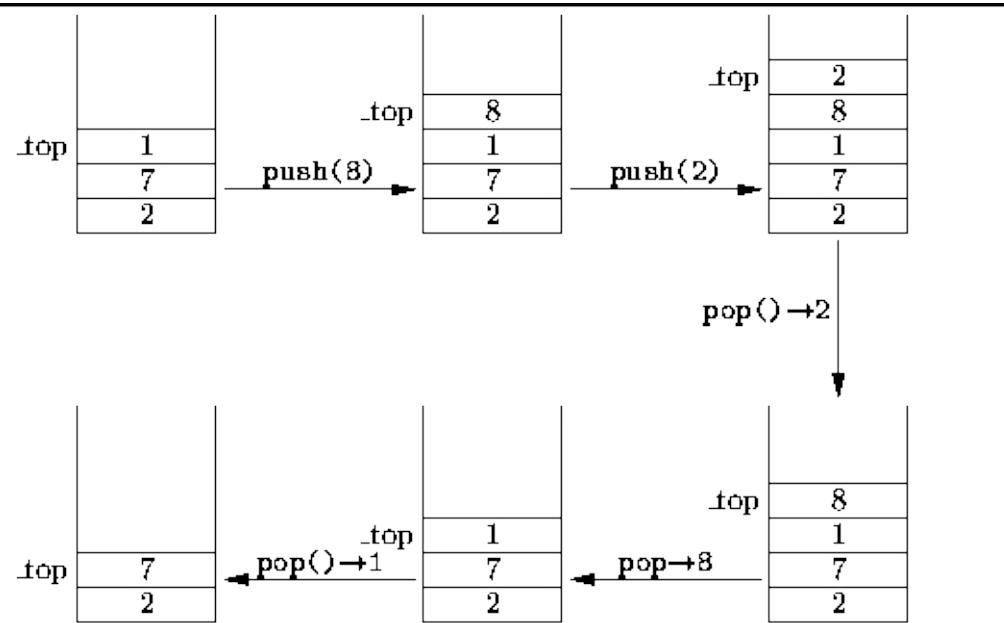


Figure: Basic stack operations.

Objects which are stored in a stack are kept in a pile. The last item put into the stack is at the top. When an item is pushed into a stack, it is placed at the top of the pile. When an item is popped, it is always the top item which is removed. Since it is always the last item to be put into the stack that is the first item to be removed, a stack is a *last-in, first-out* or *LIFO* data structure.

In addition to the `push` and `pop` operations, the typical stack implementation also has a property called `top` that returns the item at the top of the stack without removing it from the stack.

Program □ defines the `Stack` class. The abstract `Stack` class extends the abstract `Container` class defined in Program □-Program □. Hence, it comprises all of the methods inherited from `Container` plus the `push` and `pop` methods and the `Top`

property.

```
 1 class Stack(Container):
 2
 3     def __init__(self):
 4         super(Stack, self).__init__()
 5
 6     def getTop(self):
 7         pass
 8     getTop = abstractmethod(getTop)
 9
10     top = property(
11         fget = lambda self: self.getTop())
12
13     def push(self, obj):
14         pass
15     push = abstractmethod(push)
16
17     def pop(self):
18         pass
19     pop = abstractmethod(pop)
```

Program: Abstract Stack class.

When implementing a data structure, the first issue to be addressed is which foundational data structure(s) to use. Often, the choice is between an array-based implementation and a linked-list implementation. The next two sections present an array-based implementation of stacks followed by a linked-list implementation.

-
- [Array Implementation](#)
 - [Linked-List Implementation](#)
 - [Applications](#)
-

Array Implementation

This section describes an array-based implementation of stacks. Program □ introduces the `StackAsArray` class. The `StackAsArray` class is a concrete class that extends the abstract `Stack` class defined in Program □.

```
1  class StackAsArray(Stack):
2
3      def __init__(self, size = 0):
4          super(StackAsArray, self).__init__()
5          self._array = Array(size)
6
7      def purge(self):
8          while self._count > 0:
9              self._array[self._count] = None
10             self._count -= 1
11
12     #...
```

Program: `StackAsArray __init__` and `purge` methods.

-
- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [push, pop, and getTop Methods](#)
 - [accept Method](#)
 - [__iter__ Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

[Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.](#)





Instance Attributes

The `StackAsArray` class contains one instance attribute--an array of objects called `_array`. In addition, the `StackAsArray` class inherits the `_count` instance attribute from the `Container` class. In the array implementation of the stack, the elements contained in the stack occupy positions $0, 1, \dots, {}^{\text{-count}} - 1$ of the array.



[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

__init__ and purge Methods

The definitions of the `StackAsArray` `__init__` and `purge` methods are given in Program □. In addition to `self`, the `__init__` method takes a single parameter, `size`, which specifies the maximum number of items that can be stored in the stack. The variable `_array` is initialized to be an array of length `size`. The `__init__` method requires $O(n)$ time to construct the array, where `n = size`.

The purpose of the `purge` method is to remove all the contents of a container. In this case, the objects in the stack occupy the first `_count` positions of the array. To empty the stack, the `purge` method simply assigns `None` to the first `_count` positions of the array. Clearly, the running time for the `purge` method is $O(n)$, where `n = count`.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



push, pop, and getTop Methods

Program □ defines the push, pop and getTop methods of the StackAsArray class. The first of these, push, adds an element to the stack. In addition to self, it takes as its argument the object, obj, to be pushed onto the stack.

```
 1  class StackAsArray(Stack):
 2
 3      def push(self, obj):
 4          if self._count == len(self._array):
 5              raise ContainerFull
 6          self._array[self._count] = obj
 7          self._count += 1
 8
 9      def pop(self):
10          if self._count == 0:
11              raise ContainerEmpty
12          self._count -= 1
13          result = self._array[self._count]
14          self._array[self._count] = None
15          return result
16
17      def getTop(self):
18          if self._count == 0:
19              raise ContainerEmpty
20          return self._array[self._count - 1]
21
22      #...
```

Program: StackAsArray class push, pop, and getTop methods.

The push method first checks to see if there is room left in the stack. If no room is left, it raises a ContainerFull exception. Otherwise, it simply puts the object into the array, and then increases the _count variable by one. In a correctly functioning program, stack overflow should not occur. If we assume that overflow does not occur, the running time of the push method is clearly $O(1)$.

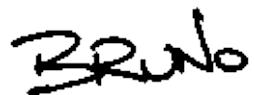
The pop method removes an item from the stack and returns that item. The pop method first checks if the stack is empty. If the stack is empty, it raises a

`ContainerEmpty` exception. Otherwise, it simply decreases `_count` by one and returns the item found at the top of the stack. In a correctly functioning program, stack underflow will not occur normally. The running time of the `pop` method is $O(1)$.

Finally, the `getTop` method returns the top item in the stack. The `getTop` method does not modify the stack. In particular, it does *not* remove the top item from the stack. The `getTop` method first checks if the stack is empty. If the stack is empty, it raises a `ContainerEmpty` exception. Otherwise, it returns the top item, which is found at position $_{\text{count}} - 1$ in the array. Assuming stack underflow does not occur normally, the running time of the `getTop` method is $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





accept Method

Program □ defines the accept method for the StackAsArray class. As discussed in Chapter □, the purpose of the accept method of a container is to accept a visitor and to cause it to visit one-by-one all of the contained objects.

```
1  class StackAsArray(Stack):
2
3      def accept(self, visitor):
4          assert isinstance(visitor, Visitor)
5          for i in xrange(self._count):
6              visitor.visit(self._array[i])
7              if visitor.isDone:
8                  return
9
10     #...
```

Program: StackAsArray class accept method.

The body of the accept method is simply a loop which calls the visit method for each object in the stack. The running time of the accept method depends on the running time of the visit method. Let T_{visit} be the running time of the visit method. In addition to the time for the method call, each iteration of the loop incurs a constant overhead. Consequently, the total running time for accept is $nT_{\text{visit}} + O(n)$, where n is the number of objects in the container. And if $T_{\text{visit}} = O(1)$, the total running time is to $O(n)$.



`__iter__` Method

As discussed in Section □, the `__iter__` method of a Container returns an Iterator. In Python, an iterator is implicitly used when you write a `for` loop like this:

```
stack = new StackAsArray(57)
stack.push(3)
stack.push(1)
stack.push(4)
for obj in stack:
    print obj
```

The Python `for` loop given above is implemented as if it was coded like this:

```
it = iter(stack) # Equivalent to stack.__iter__()
while true:
    try:
        obj = it.next()
        print obj
    except StopIteration:
        break
```

This code creates an instance of the `StackAsArray` class and assigns it to the variable `stack`. Next, several `ints` are pushed onto the stack. Finally, an iterator is used to systematically print out all of the objects in the stack.

Program □ defines the `__iter__` method of the `StackAsArray` class. The `__iter__` method returns a new instance of the nested class `StackAsArray.Iterator` that implements the iterator protocol (lines 3-14).

```
 1 class StackAsArray(Stack):
 2
 3     class Iterator(Iterator):
 4
 5         def __init__(self, stack):
 6             super(StackAsArray.Iterator, self).__init__(stack)
 7             self._position = 0
 8
 9         def next(self):
10             if self._position >= self._container._count:
11                 raise StopIteration
12             obj = self._container._array[self._position]
13             self._position = self._position + 1
14             return obj
15
16         def __iter__(self):
17             return self.Iterator(self)
18
19     #...
```

Program: StackAsArray class `__iter__` method.

The `Iterator` class has two instance attributes, `_container` and `_position`. The `_container` instance attribute refers to the stack whose elements are being enumerated. The `_position` instance attribute is used to keep track of the position in the array of the current object.

The `_next` method is called in the body of the loop to advance the iterator to the next object in the stack. The `_next` method increments the `_position` instance attribute and then returns the object in the stack specified at that position. The `_next` method resets the position to -1 and raises a `StopIteration` exception when there are not more elements. Clearly, the running time of `_next` is $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Linked-List Implementation

In this section we will examine a linked-list implementation of stacks that makes use of the `LinkedList` data structure developed in Chapter □. Program □ introduces the `StackAsLinkedList` class. The `StackAsLinkedList` class is a concrete class that extends the abstract `Stack` class defined in Program □.

```
1  class StackAsLinkedList(Stack):
2
3      def __init__(self):
4          super(StackAsLinkedList, self).__init__()
5          self._list = LinkedList()
6
7      def purge(self):
8          self._list.purge()
9          self._count = 0
10
11      # ...
```

Program: `StackAsLinkedList` `__init__` and `purge` methods.

-
- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [push, pop and getTop Methods](#)
 - [accept Method](#)
 - [__iter__ Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Instance Attributes

The implementation of the `StackAsLinkedList` class makes use of one instance attribute--an instance of the `LinkedList` class called `_list`. In addition, the `StackAsLinkedList` class inherits the `_count` instance attribute from the `Container` class. This list is used to keep track of the objects in the stack. As a result, there are as many elements in the linked list as there are objects in the stack.

__init__ and purge Methods

The definitions of the `__init__` and the `purge` methods of the `StackAsLinkedList` class are shown in Program □. With a linked-list implementation, it is not necessary to preallocate storage space for the objects in the stack. Space is allocated dynamically and incrementally on the basis of demand.

The `__init__` method simply creates an empty `LinkedList` and assigns it to the `_list` instance attribute. Since an empty list can be created in constant time, the running time of the `StackAsLinkedList __init__` method is $O(1)$.

The `purge` method of the `StackAsLinkedList` class simply calls the `purge` method of the `LinkedList` class. The `purge` method of the `LinkedList` class discards all the elements of the list in constant time. Consequently, the running time of the `purge` method is also $O(1)$.



push, pop and getTop Methods

The push, pop and getTop methods of the StackAsLinkedList class are defined in Program □.

```
 1  class StackAsLinkedList(Stack):
 2
 3      def push(self, obj):
 4          self._list.prepend(obj)
 5          self._count += 1
 6
 7      def pop(self):
 8          if self._count == 0:
 9              raise ContainerEmpty
10          result = self._list.first
11          self._list.extract(result)
12          self._count -= 1
13          return result
14
15      def getTop(self):
16          if self._count == 0:
17              raise ContainerEmpty
18          return self._list.first
19
20      # ...
```

Program: StackAsLinkedList class push, pop and getTop methods.

The implementation of push is trivial. It takes as its argument the object, `obj`, to be pushed onto the stack and simply prepends that object to the linked list `_list`. Then, one is added to the `_count` variable. The running time of the push method is constant, since the `prepend` method has a constant running time, and updating the `_count` only takes $O(1)$ time.

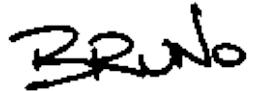
The pop method is implemented using the `first` property and the `extract` method of the `LinkedList` class. The `first` property is used to get the first item in the linked list. The `first` property `get` method runs in constant time. The `extract` method is then called to extract the first item from the linked list. In the worst case, `extract` requires $O(n)$ time to extract an item from a linked list of

length n . But the worst-case time arises only when it is the *last* element of the list which is to be extracted. In the case of the `pop` method, it is always the *first* element that is extracted. This can be done in constant time. Assuming that the exception which is raised when `pop` is called on an empty list does not occur, the running time for `pop` is $O(1)$.

The definition of the `getTop` method is quite simple. It returns the first object in the linked list. Provided the linked list is not empty, the running time of `getTop` is $O(1)$. If the linked list is empty, the `getTop` method raises a `ContainerEmpty` exception.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



accept Method

The accept method of the StackAsLinkedList class is defined in Program □. The accept method takes a visitor and calls its visit method one-by-one for all of the objects on the stack.

```
1  class StackAsLinkedList(Stack):
2
3      def accept(self, visitor):
4          assert isinstance(visitor, Visitor)
5          ptr = self._list.head
6          while ptr is not None:
7              visitor.visit(ptr.datum)
8              if visitor.isDone:
9                  return
10             ptr = ptr.next
11
12     # ...
```

Program: StackAsLinkedList class accept method.

The implementation of the accept method for the StackAsLinkedList class mirrors that of the StackAsArray class shown in Program □. In this case, the linked list is traversed from front to back, i.e., from the top of the stack to the bottom. As each element of the linked list is encountered, the visit method is called. If T_{visit} is the running time of the visit method, the total running time for accept is $nT_{\text{visit}} + O(n)$, where $n = \text{count}$ is the number of objects in the container. If we assume that $T_{\text{visit}} = O(1)$, the total running time is $O(n)$.



`__iter__` Method

Program □ defines the `__iter__` method of the `StackAsLinkedList` class. The `__iter__` method returns an instance of the nested class `StackAsLinkedList.Iterator` that implements the iterator protocol (lines 3-16).

```
 1  class StackAsLinkedList(Stack):
 2
 3      class Iterator(Iterator):
 4
 5          def __init__(self, stack):
 6              super(StackAsLinkedList.Iterator, self).__init__(
 7                  stack)
 8              self._position = stack._list.head
 9
10
11          def next(self):
12              if self._position is None:
13                  raise StopIteration
14              element = self._position
15              self._position = self._position.next
16              return element.datum
17
18          def __iter__(self):
19              return self.Iterator(self)
20
21      # ...
```

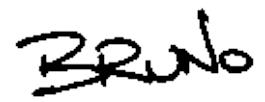
Program: `StackAsLinkedList` class `__iter__` method.

The `Iterator` class has two instance attributes, `_container` and `_position`. The `_container` instance attribute refers to the stack whose elements are being enumerated. The `_position` instance attribute is used to keep track of the position in the linked list of the next object to be enumerated.

The purpose of the `_next` method is to advance the iterator to the next object in the stack and to raise a `StopIteration` exception when there are no more elements to be enumerated. In Program □ elements remain as long as the `_position` is not `None`. Clearly, the running time of `_next` is $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Applications

Consider the following expression:

$$(5 + 9) \times 2 + 6 \times 5 \quad (6.1)$$

In order to determine the value of this expression, we first compute the sum $5+9$ and then multiply that by 2. Then we compute the product 6×5 and add it to the previous result to get the final answer. Notice that the order in which the operations are to be done is crucial. Clearly if the operations are not done in the correct order, the wrong result is computed.

The expression above is written using the usual mathematical notation. This notation is called *infix* notation. What distinguishes this notation is the way that expressions involving binary operators are written. A *binary operator* is an operator which has exactly two operands, such as $+$ and \times . In infix notation, binary operators appear *in between* their operands.

Another characteristic of *infix* notation is that the order of operations is determined by *operator precedence*. For example, the \times (multiplication) operator has higher precedence than does the $+$ (addition) operator. When an evaluation order is desired that is different from that provided by the precedence, *parentheses*, ``(`` and ``)`', are used to override precedence rules. An expression in parentheses is evaluated first.

As an alternative to infix, the Polish logician *Jan Łukasiewicz* introduced notations which require neither parentheses nor operator precedence rules. The first of these, the so-called *Polish notation*, places the binary operators before their operands. For Equation □ we would write:

$+ \times + 5 9 2 \times 6 5$

This is also called *prefix* notation, because the operators are written in front of their operands.

While prefix notation is completely unambiguous in the absence of parentheses, it is not very easy to read. A minor syntactic variation on prefix is to write the

operands as a comma-separated list enclosed in parentheses as follows:

$$+ (\times (+ (5, 9), 2), \times (6, 5))$$

While this notation seems somewhat foreign, in fact it is precisely the notation that is used for function calls in Python:

```
__add__(__mul__(__add__(5, 9) , 2), __mul__(6, 5))
```

The second form of Łukasiewicz notation is the so-called *Reverse-Polish notation* (RPN). Equation \square is written as follows in RPN:

$$5\ 9\ +\ 2\ \times\ 6\ 5\ \times\ + \quad (6.2)$$

This notation is also called *postfix* notation for the obvious reason--the operators are written *after* their operands.

Postfix notation, like prefix notation, does not make use of operator precedence nor does it require the use of parentheses. A postfix expression can always be written without parentheses that expresses the desired evaluation order. For example, the expression $1 + 2 \times 3$, in which the multiplication is done first, is written $1\ 2\ 3\ \times\ +$; whereas the expression $(1 + 2) \times 3$ is written $1\ 2\ +\ 3\ \times$.

-
- [Evaluating Postfix Expressions](#)
 - [Implementation](#)
-



Evaluating Postfix Expressions

One of the most useful characteristics of a postfix expression is that the value of such an expression can be computed easily with the aid of a stack of values. The components of a postfix expression are processed from left to right as follows:

1. If the next component of the expression is an operand, the value of the component is pushed onto the stack.
2. If the next component of the expression is an operator, then its operands are in the stack. The required number of operands are popped from the stack; the specified operation is performed; and the result is pushed back onto the stack.

After all the components of the expression have been processed in this fashion, the stack will contain a single result which is the final value of the expression. Figure □ illustrates the use of a stack to evaluate the RPN expression given in Equation □.

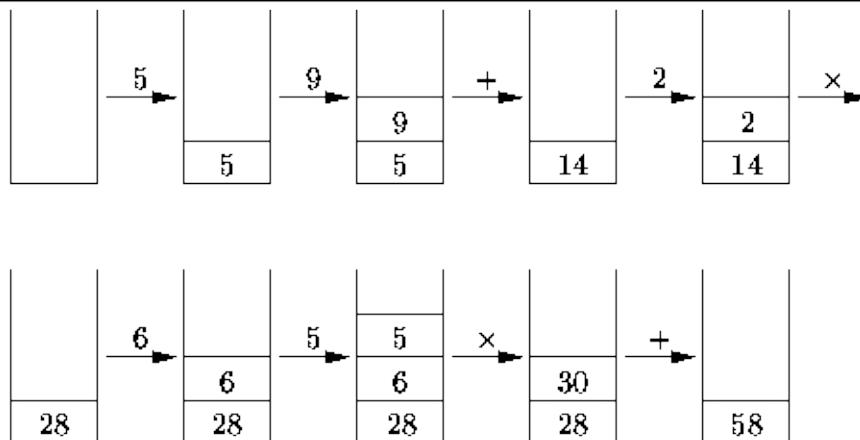


Figure: Evaluating the RPN expression in Equation □ using a stack.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

Program □ gives the implementation of a simple RPN calculator. The purpose of this example is to illustrate the use of the Stack class. The program shown accepts very simplified RPN expressions: The expression may contain only integers, the addition operator, +, and the multiplication operator, *. In addition, the operator = pops the top value off the stack and prints it on the console. Furthermore, the calculator does its computation entirely with integers.

```
 1  class Algorithms(object):
 2
 3      def calculator(input, output):
 4          stack = StackAsLinkedList()
 5          for line in input.readlines():
 6              for word in line.split():
 7                  if word == "+":
 8                      arg2 = stack.pop()
 9                      arg1 = stack.pop()
10                      stack.push(arg1 + arg2)
11                  elif word == "*":
12                      arg2 = stack.pop()
13                      arg1 = stack.pop()
14                      stack.push(arg1 * arg2)
15                  elif word == "=":
16                      arg = stack.pop()
17                      output.write(str(arg) + "\n")
18                  else:
19                      stack.push(int(word))
20      calculator = staticmethod(calculator)
```

Program: Stack application--a single-digit, RPN calculator.

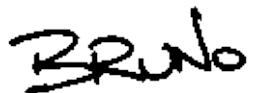
Notice that the stack variable of the calculator method may be any object that extends the abstract Stack class. Consequently, the calculator does not depend on the stack implementation used! For example, if we wish to use a stack implemented using an array, we can simply replace line 4 with the following:

```
stack = StackAsArray(10)
```

The running time of the calculator method depends upon the number of symbols, operators, and operands, in the expression being evaluated. If there are n symbols, the body of the inner for loop is executed n times. It should be fairly obvious that the amount of work done per symbol is constant, regardless of the type of symbol encountered. This is the case for both the StackAsArray and the StackAsLinkedList stack implementations. Therefore, the total running time needed to evaluate an expression comprised of n symbols is $O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Queues

In the preceding section we saw that a stack comprises a pile of objects that can be accessed only at one end--the top. In this section we examine a similar data structure called a *single-ended queue*. Whereas in a stack we add and remove elements at the same end of the pile, in a single-ended queue we add elements at one end and remove them from the other. Since it is always the first item to be put into the queue that is the first item to be removed, a queue is a *first-in, first-out* or *FIFO* data structure. Figure □ illustrates the basic queue operations.

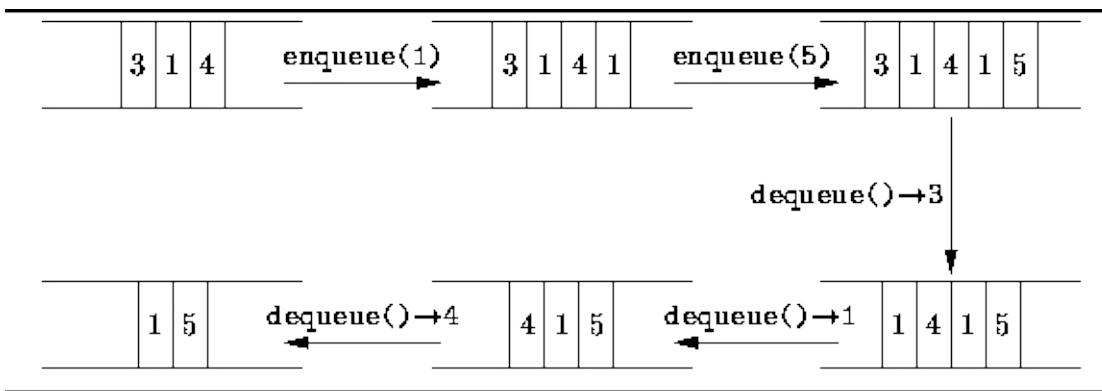


Figure: Basic queue operations.

Program □ defines the Queue class. The abstract Queue class extends the abstract Container class defined in Program □. Hence, it comprises all the methods inherited from Container plus the methods, enqueue and equeue, and the head property. As we did with stacks, we examine two queue implementations--an array-based one and a linked-list one.

```
 1 class Queue(Container):
 2
 3     def __init__(self):
 4         super(Queue, self).__init__()
 5
 6     def getHead(self):
 7         pass
 8     getHead = abstractmethod(getHead)
 9
10     head = property(
11         fget = lambda self: self.getHead())
12
13     def enqueue(self, obj):
14         pass
15     enqueue = abstractmethod(enqueue)
16
17     def dequeue(self):
18         pass
19     dequeue = abstractmethod(dequeue)
```

Program: Abstract Queue class.

-
- [Array Implementation](#)
 - [Linked-List Implementation](#)
 - [Applications](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Array Implementation

Program □ introduces the QueueAsArray class. The QueueAsArray class is a concrete class that extends the abstract Queue class defined in Program □.

```
1  class QueueAsArray(Queue):
2
3      def __init__(self, size = 0):
4          super(QueueAsArray, self).__init__()
5          self._array = Array(size)
6          self._head = 0
7          self._tail = size - 1
8
9      def purge(self):
10         while self._count > 0:
11             self._array[self._head] = None
12             self._head = self._head + 1
13             if self._head == len(self._array):
14                 self._head = 0
15             self._count -= 1
16
17     # ...
```

Program: QueueAsArray __init__ and purge methods.

-
- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [enqueue, dequeue and getHead Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Instance Attributes

QueueAsArray objects comprise three instance attributes--`_array`, `_head`, and `_tail`. The first, `_array`, is an array that is used to hold the contents of the queue. The objects contained in the queue will be held in a contiguous range of array elements as shown in Figure □ (a). The instance attributes `_head` and `_tail` denote the left and right ends, respectively, of this range.

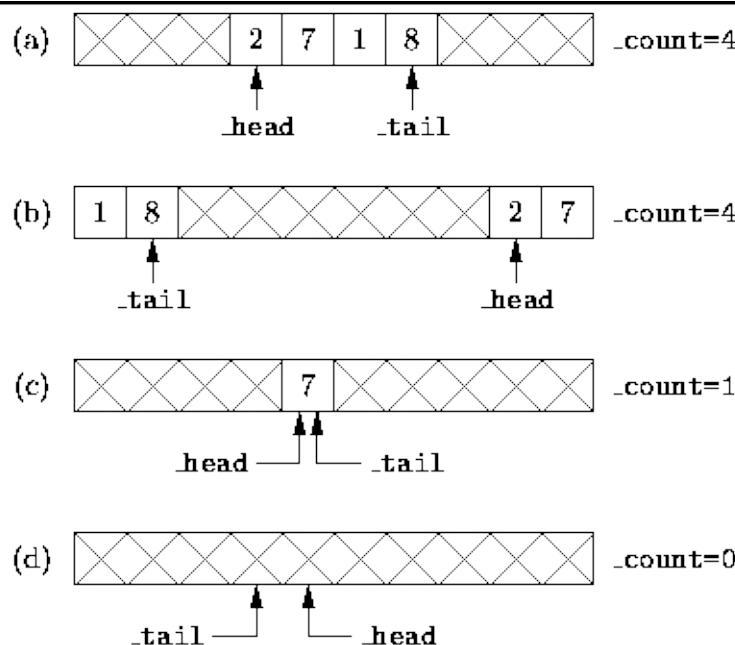


Figure: Array implementation of a queue.

In general, the region of contiguous elements will not necessarily occupy the leftmost array positions. As elements are deleted at the head, the position of the left end will change. Similarly, as elements are added at the tail, the position of the right end will change. In some circumstances, the contiguous region of elements will wrap around the ends of the array as shown in Figure □ (b).

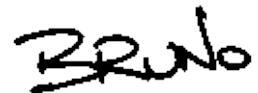
As shown in Figure □, the leftmost element is `_array[_head]`, and the rightmost element is `_array[_tail]`. When the queue contains only one element, `_head = _tail` as shown in Figure □ (c).

Finally, Figure □(b) shows that if the queue is empty, the `_head` position will actually be to the right of the `_tail` position. However, this is also the situation which arises when the queue is completely full! The problem is essentially this: Given an array of length n , then $0 \leq \text{head} < n$ and $0 \leq \text{tail} < n$. Therefore, the difference between the `_head` and `_tail` satisfies $0 \leq |\text{head} - \text{tail}| \leq n - 1$. Since there are only n distinct differences, there can be only n distinct queue lengths, 0, 1, ..., $n-1$. It is not possible to distinguish the queue which is empty from the queue which has n elements solely on the basis of the `_head` and `_tail` instance attributes.

There are two options for dealing with this problem: The first is to limit the number of elements in the queue to be at most $n-1$. The second is to use another instance attribute, `_count`, to keep track explicitly of the actual number of elements in the queue rather than to infer the number from the `_head` and `_tail` variables. The second approach has been adopted in the implementation given below.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





`__init__` and `purge` Methods

The definitions of the `QueueAsArray` class `__init__` and `purge` methods are given in Program □. In addition to `self`, the `__init__` takes a single parameter, `size`, which specifies the maximum number of items that can be stored in the queue. The `__init__` method initializes the instance attributes as follows: The `_array` instance attribute is initialized to an array of length `_size` and the `_head` and `_tail` instance attributes, are initialized to represent the empty queue. The total running time for the `QueueAsArray __init__` method is $O(n)$, where `n = size`.

The purpose of the `purge` method is to remove all the contents of a container. In this case, the objects in the queue occupy contiguous array positions between `_head` and `_tail`. To empty the queue, the `purge` method walks through the occupied array positions assigning to each one the value `None` as it goes. Clearly, the running time for the `purge` method is $O(n)$, where `n = count`.



enqueue, dequeue and getHead Methods

Program □ defines the enqueue, Dequeue and getHead methods of the QueueAsArray class.

```
 1  class QueueAsArray(Queue):
 2
 3      def getHead(self):
 4          if self._count == 0:
 5              raise ContainerEmpty
 6          return self._array[self._head]
 7
 8      def enqueue(self, obj):
 9          if self._count == len(self._array):
10              raise ContainerFull
11          self._tail = self._tail + 1
12          if self._tail == len(self._array):
13              self._tail = 0
14          self._array[self._tail] = obj
15          self._count += 1
16
17      def dequeue(self):
18          if self._count == 0:
19              raise ContainerEmpty
20          result = self._array[self._head]
21          self._array[self._head] = None
22          self._head = self._head + 1
23          if self._head == len(self._array):
24              self._head = 0
25          self._count -= 1
26          return result
27
28      # ...
```

Program: QueueAsArray class enqueue, Dequeue and getHead methods.

The getHead method returns the object found at the head of the queue, having first checked to see that the queue is not empty. If the queue is empty, it raises a ContainerEmpty exception. Under normal circumstances, we expect that the queue will not be empty. Therefore, the normal running time of this accessor is

$O(1)$.

In addition to `self`, the `enqueue` method takes a single argument which is the object to be added to the tail of the queue. The `enqueue` method first checks that the queue is not full--a `ContainerFull` exception is raised when the queue is full. Next, the position at which to insert the new element is determined by increasing the `_tail` instance attribute by one modulo the length of the array. Finally, the object to be enqueued is put into the array at the correct position and the `_count` is adjusted accordingly. Under normal circumstances (i.e., when the exception is not thrown), the running time of `Enqueue` is $O(1)$.

The `dequeue` method removes an object from the head of the queue and returns that object. First, it checks that the queue is not empty and raises an exception when it is. If the queue is not empty, the method first sets aside the object at the head in the local variable `result`; it increases the `_head` instance attribute by one modulo the length of the array; adjusts the `_count` accordingly; and returns `result`. All this can be done in a constant amount of time so the running time of `dequeue` is a $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Linked-List Implementation

This section presents a queue implementation which makes use of the singly-linked list data structure, `LinkedList`, that is defined in Chapter 2. Program 2 introduces the `QueueAsLinkedList` class. The `QueueAsLinkedList` extends the abstract `Queue` class.

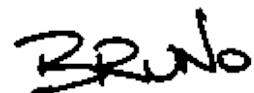
```
1  class QueueAsLinkedList(Queue):
2
3      def __init__(self):
4          super(QueueAsLinkedList, self).__init__()
5          self._list = LinkedList()
6
7      def purge(self):
8          self._list.purge()
9          self._count = 0
10
11      # ...
```

Program: `QueueAsLinkedList` class `__init__` and `purge` methods.

-
- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [enqueue, dequeue and getHead Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



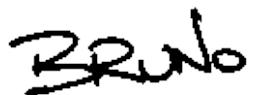
[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Instance Attributes

Just like the `StackAsLinkedList` class, the implementation of the `QueueAsLinkedList` class requires only one instance attribute--`_list`. The `_list` instance attribute is an instance of the `LinkedList` class. It is used to keep track of the elements in the queue.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





`__init__` and `purge` Methods

Program `□` defines the `QueueAsLinkedList` `__init__` and `purge` methods. In the case of the linked list implementation, it is not necessary to preallocate storage. The `__init__` method simply initializes the `_list` object as an empty list. The running time of the `__init__` method is $O(1)$.

The `purge` method empties the queue by invoking the `purge` method provided by the `LinkedList` class and then sets the `_count` instance attribute to zero. Since a linked-list can be purged in constant time, the total running time for the `purge` method is $O(1)$.



enqueue, dequeue and getHead Methods

The enqueue, dequeue and getHead methods of the QueueAsLinkedList class are given in Program □.

```
 1  class QueueAsLinkedList(Queue):
 2
 3      def getHead(self):
 4          if self._count == 0:
 5              raise ContainerEmpty
 6          return self._list.first
 7
 8      def enqueue(self, obj):
 9          self._list.append(obj)
10          self._count += 1
11
12      def dequeue(self):
13          if self._count == 0:
14              raise ContainerEmpty
15          result = self._list.first
16          self._list.extract(result)
17          self._count -= 1
18          return result
19
20      # ...
```

Program: QueueAsLinkedList class enqueue, dequeue and getHead methods.

The getHead method returns the object at the head of the queue. The head of the queue is in the first element of the linked list. In Chapter □ we saw that the running time of `LinkedList.first` is a constant, Therefore, the normal running time for the head method is $O(1)$.

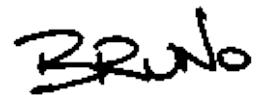
The enqueue method takes a single argument--the object to be added to the tail of the queue. The method simply calls the `LinkedList.append` method. Since the running time for append is $O(1)$, the running time of enqueue is also $O(1)$.

The dequeue method removes an object from the head of the queue and returns that object. First, it verifies that the queue is not empty and throws an exception

when it is. If the queue is not empty, dequeue saves the first item in the linked list in the local variable `result`. Then that item is extracted from the list. Using the `LinkedList` class from Chapter □, the time required to extract the first item from a list is $O(1)$ regardless of the number of items in the list. As a result, the running time of `dequeue` is also $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Applications

The FIFO nature of queues makes them useful in certain algorithms. For example, we will see in Chapter □ that a queue is an essential data structure for many different graph algorithms. In this section we illustrate the use of a queue in the *breadth-first traversal* of a tree.

Figure □ shows an example of a tree. A tree is comprised of *nodes* (indicated by the circles) and *edges* (shown as arrows between nodes). We say that the edges point from the *parent* node to a *child* node. The *degree* of a node is equal to the number of children of that node. For example, node A in Figure □ has degree three and its children are nodes B, C, and D. A child and all of its descendants is called a *subtree*.

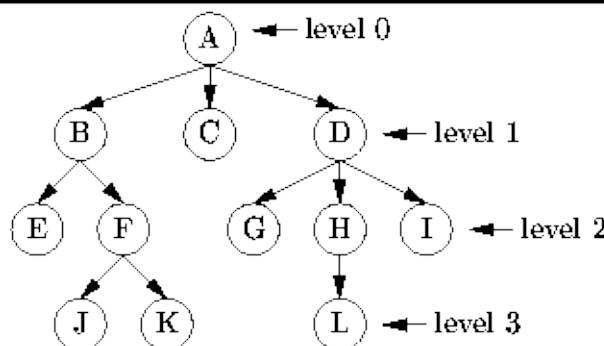


Figure: A tree.

One way to represent such a tree is to use a collection of linked structures. Consider the following class definition which is an abridged version of the abstract Tree class described in Chapter □.

```
class Tree:  
    key = property(fget = ...)  
    degree = property(fget = ...)  
    def getSubtree(self, i): pass  
    getSubtree = abstractmethod(getSubtree)  
    # ...
```

Each node in a tree is represented by an instance of a class derived from the abstract Tree class. The key property provides a get accessor that returns an

object which represents the contents of the node. E.g. in Figure □, each node carries a label so the `key` property would return a `str` value that represents that label. The `degree` property provides a get accessor that returns the degree of the node and the `getSubtree` method takes an `int` argument `i` and returns the corresponding child of that node.

One of the essential operations on a tree is a *tree traversal*. A traversal *visits* one-by-one all the nodes in a given tree. To *visit a node* means to perform some computation using the information contained in that node--e.g., print the key. The standard tree traversals are discussed in Chapter □. In this section we consider a traversal which is based on the levels of the nodes in the tree.

Each node in a tree has an associated level which arises from the position of that node in the tree. For example, node A in Figure □ is at level 0, nodes B, C, and D are at level 1, etc. A *breadth-first traversal* visits the nodes of a tree in the order of their levels. At each level, the nodes are visited from left to right. For this reason, it is sometimes also called a *level-order traversal*. The breadth-first traversal of the tree in Figure □ visits the nodes from A to L in alphabetical order.

One way to implement a breadth-first traversal of a tree is to make use of a queue as follows: To begin the traversal, the root node of the tree is enqueued. Then, we repeat the following steps until the queue is empty:

1. Dequeue and visit the first node in the queue.
2. Enqueue its children in order from left to right.

Figure □ illustrates the breadth-first traversal algorithm by showing the contents of the queue immediately prior to each iteration.

A
B C D
C D E F
D E F
E F G H I
F G H I
G H I J K
H I J K
I J K L
J K L
K L
L

Figure: Queue contents during the breadth-first traversal of the tree in Figure □.

- [Implementation](#)

Implementation

Program □ defines the method `breadthFirstTraversal`. This method takes as its argument any instance of a class derived from the abstract `Tree` class. The idea is that the method is passed the root of the tree to be traversed. The algorithm makes use of the `QueueAsLinkedList` data structure, which was defined in the preceding section, to hold the appropriate tree nodes.

The running time of the `breadthFirstTraversal` algorithm depends on the number of nodes in the tree which is being traversed. Each node of the tree is enqueued exactly once--this requires a constant amount of work. Furthermore, in each iteration of the loop, each node is dequeued exactly once--again a constant amount of work. As a result, the running time of the `breadthFirstTraversal` algorithm is $O(n)$ where n is the number of nodes in the traversed tree.

```
 1  class Algorithms(object):
 2
 3      def breadthFirstTraversal(tree):
 4          queue = QueueAsLinkedList()
 5          queue.enqueue(tree)
 6          while not queue.isEmpty():
 7              t = queue.dequeue()
 8              print t.key
 9              for i in xrange(t.degree):
10                  subTree = t.getSubtree(i)
11                  queue.enqueue(subTree)
12      breadthFirstTraversal = staticmethod(breadthFirstTraversal)
```

Program: Queue application--breadth-first tree traversal.

Deques

In the preceding section we saw that a queue comprises a pile of objects into which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue which provides a means to insert and remove items at both ends of the pile. This data structure is a *deque*. The word *deque* is an acronym derived from *double-ended queue*. ◇

Figure □ illustrates the basic deque operations. A deque provides three operations which access the head of the queue, Head, EnqueueHead and DequeueHead, and three operations to access the tail of the queue, Tail, EnqueueTail and DequeueTail.

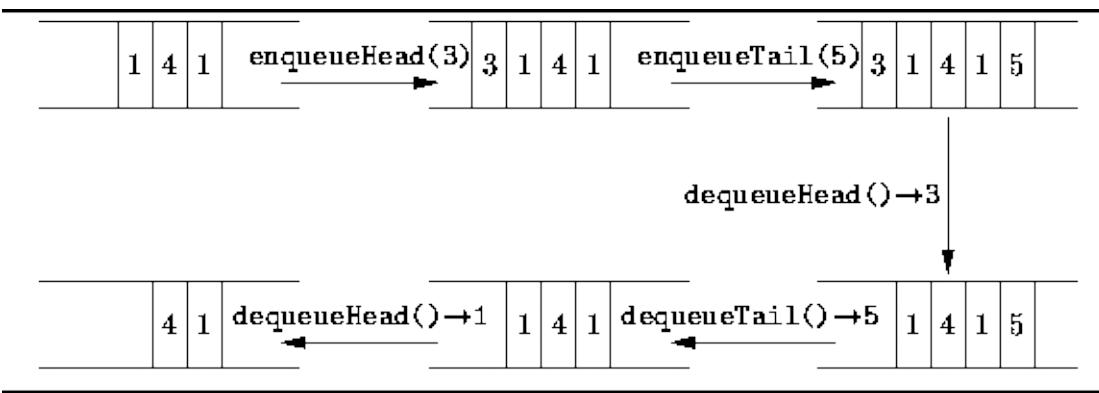


Figure: Basic deque operations.

Program □ defines the `Deque` class. The abstract `Deque` class extends the abstract `Container` class defined in Program □. Hence, it comprises all of the methods inherited from `Container` plus the methods, `enqueueHead`, `dequeueHead`, `enqueueTail`, and `dequeueTail`, and the properties, `head` and `tail`.

```
 1 class Deque(Queue):
 2
 3     def __init__(self):
 4         super(Deque, self).__init__()
 5
 6     def getHead(self):
 7         pass
 8     getHead = abstractmethod(getHead)
 9
10     head = property(
11         fget = lambda self: self.getHead())
12
13     def getTail(self):
14         pass
15     getTail = abstractmethod(getTail)
16
17     tail = property(
18         fget = lambda self: self.getTail())
19
20     def enqueueHead(self, obj):
21         pass
22     enqueueHead = abstractmethod(enqueueHead)
23
24     def dequeueHead(self):
25         return self.dequeue()
26
27     def enqueueTail(self, object):
28         self.enqueue(object)
29
30     def dequeueTail(self):
31         pass
32     dequeueTail = abstractmethod(dequeueTail)
```

Program: Abstract Deque class.

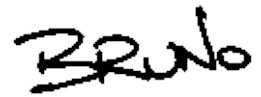
The `dequeueHead` method of the `Deque` class is trivial to implement--it simply calls the `dequeue` method inherited from the `Queue` class. Similarly, the `enqueueTail` method simply calls the `enqueue` method inherited from the `Queue` class.

- [Array Implementation](#)
- [Linked List Implementation](#)

- [Doubly-Linked and Circular Lists](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

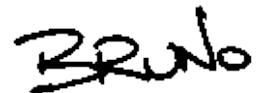
Array Implementation

Program □ introduces an array implementation of a deque. The `DequeAsArray` class extends the abstract `Deque` class defined in Program □. The `QueueAsArray` class provides almost all the required functionality.

- [enqueueHead Method](#)
 - [dequeueTail and getTail Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



enqueueHead Method

Program □ defines the enqueueHead method.

```
1 class DequeAsArray(QueueAsArray, Deque):
2
3     def __init__(self, size = 0):
4         super(DequeAsArray, self).__init__(size)
5
6     def enqueueHead(self, obj):
7         if self._count == len(self._array):
8             raise ContainerFull
9         if self._head == 0:
10            self._head = len(self._array) - 1
11        else:
12            self._head = self._head - 1
13        self._array[self._head] = obj;
14        self._count += 1
15
16    # ...
```

Program: DequeAsArray class enqueueHead method.

In addition to `self`, the `enqueueHead` method takes a single argument which is the object to be added to the head of the deque. The `enqueueHead` method first checks that the deque is not full--a `ContainerFull` exception is raised when the deque is full. Next, the position at which to insert the new element is determined by decreasing the `_head` instance attribute by one modulo the length of the array. Finally, the object to be enqueued is put into the array at the correct position and the `_count` is adjusted accordingly. Under normal circumstances (i.e., when the exception is not raised), the running time of `EnqueueHead` is $O(1)$.



dequeueTail and getTail Methods

Program □ defines the dequeueTail and getTail methods of the DequeAsArray class.

```
 1  class DequeAsArray(QueueAsArray, Deque):
 2
 3      def getTail(self):
 4          if self._count == 0:
 5              raise ContainerEmpty
 6          return self._array[self._tail];
 7
 8      def dequeueTail(self):
 9          if self._count == 0:
10              raise ContainerEmpty
11          result = self._array[self._tail];
12          self._array[self._tail] = None
13          if self._tail == 0:
14              self._tail = len(self._array) - 1
15          else:
16              self._tail = self._tail - 1
17          self._count -= 1
18          return result;
19
20      # ...
```

Program: DequeAsArray class dequeueTail and getTail methods.

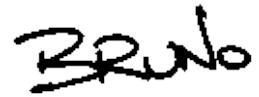
The getTail method that returns the object found at the tail of the deque, having first checked to see that the deque is not empty. If the deque is empty, it raises a ContainerEmpty exception. Under normal circumstances, we expect that the deque will not be empty. Therefore, the normal running time of this method is $O(1)$.

The DequeueTail method removes an object from the tail of the deque and returns that object. First, it checks that the deque is not empty and throws an exception when it is. If the deque is not empty, the method sets aside the object at the tail in the local variable `result`; it decreases the `_tail` instance attribute by one modulo the length of the array; adjusts the `_count` accordingly; and

returns result. All this can be done in a constant amount of time so the running time of DequeueTail is a constant.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Linked List Implementation

Program □ defines a linked-list implementation of a deque. The `DequeAsLinkedList` class extends the abstract `Deque` class defined in Program □. The `QueueAsLinkedList` implementation provides almost all of the required functionality.

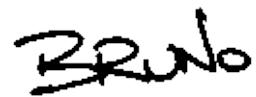
```
1  class DequeAsLinkedList(QueueAsLinkedList, Deque):
2
3      def __init__(self):
4          super(DequeAsLinkedList, self).__init__()
5
6      def enqueueHead(self, obj):
7          self._list.prepend(obj);
8          self._count += 1
9
10     # ...
```

Program: `DequeAsLinkedList` class `enqueueHead` method.

-
- [enqueueHead Method](#)
 - [dequeueTail and getTail Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



enqueueHead Method

In addition to `self`, the `EnqueueHead` method takes a single argument--the object to be added to the head of the deque. The method simply calls the `LinkedList.prepend` method. Since the running time for `prepend` is $O(1)$, the running time of `enqueueHead` is also $O(1)$.



dequeueTail and getTail Methods

Program □ defines and DequeueTail and getTail methods of the DequeAsArrayList class.

```
 1  class DequeAsLinkedList(QueueAsLinkedList, Deque):
 2
 3      def getTail(self):
 4          if self._count == 0:
 5              raise ContainerEmpty
 6          return self._list.last
 7
 8      def dequeueTail(self):
 9          if self._count == 0:
10              raise ContainerEmpty
11          result = self._list.last
12          self._list.extract(result)
13          self._count -= 1
14          return result
15
16      # ...
```

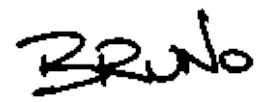
Program: DequeAsLinkedList class dequeueTail and getTail methods.

The getTail method returns the object at the tail of the deque. The tail of the deque is in the last element of the linked list. In Chapter □ we saw that the running time of `LinkedList.last` property is a constant, Therefore, the normal running time for this accesor is $O(1)$.

The dequeueTail method removes an object from the tail of the deque and returns that object. First, it verifies that the deque is not empty and throws an exception when it is. If the deque is not empty, dequeueTail saves the last item in the linked list in the local variable `result`. Then that item is extracted from the linked list. When using the `LinkedList` class from Chapter □, the time required to extract the last item from a list is $O(n)$, where $n = \text{count}$ is the number of items in the list. As a result, the running time of `DequeueTail` is $O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Doubly-Linked and Circular Lists

In the preceding section we saw that the running time of `dequeueHead` is $O(1)$, but that the running time of `dequeueTail` is $O(n)$, for the linked-list implementation of a deque. This is because the linked list data structure used, `LinkedList`, is a *singly-linked list*. Each element in a singly-linked list contains a single reference--a reference to the successor (`next`) element of the list. As a result, deleting the head of the linked list is easy: The new head is the successor of the old head.

However, deleting the tail of a linked list is not so easy: The new tail is the predecessor of the original tail. Since there is no reference from the original tail to its predecessor, the predecessor must be found by traversing the linked list from the head. This traversal gives rise to the $O(n)$ running time.

In a *doubly-linked list*, each list element contains two references--one to its successor and one to its predecessor. There are many different variations of doubly-linked lists: Figure 1 illustrates three of them.

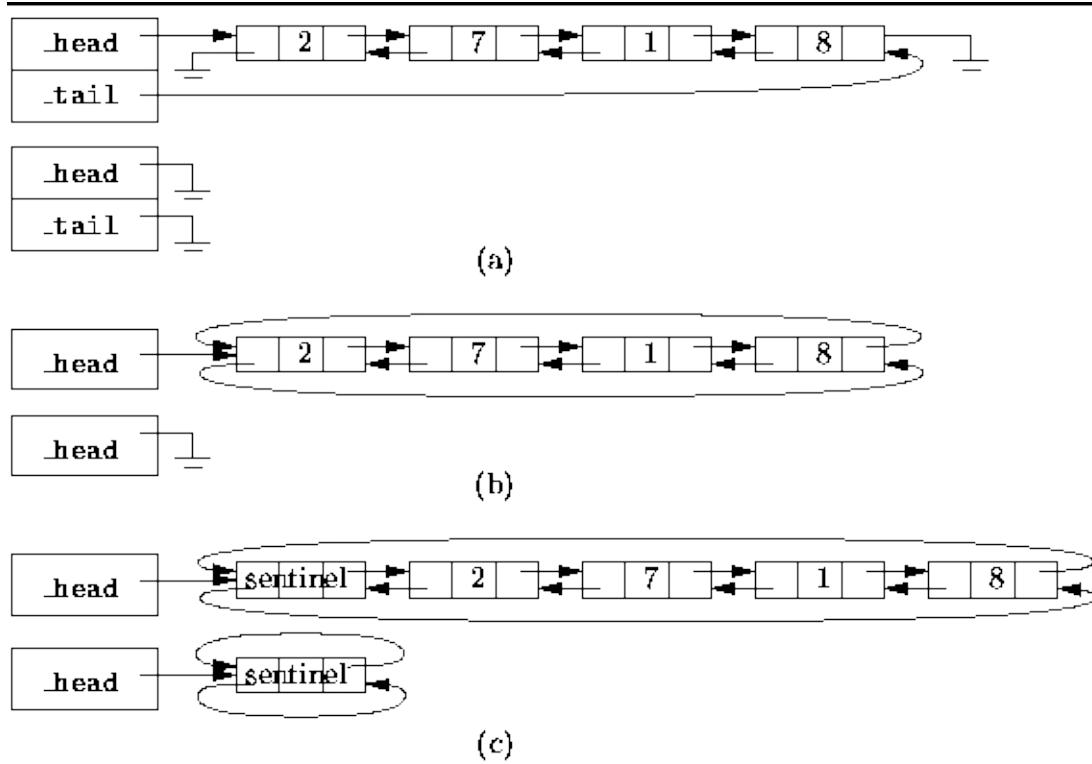


Figure: Doubly-linked and circular list variations.

Figure □ (a) shows the simplest case: Two variables, say `_head` and `_tail`, are used to keep track of the list elements. One of them refers to the first element of the list, the other refers to the last. The first element of the list has no predecessor, therefore that reference is null. Similarly, the last element has no successor and the corresponding reference is also null. In effect, we have two overlapping singly-linked lists which go in opposite directions. Figure □ also shows the representation of an empty list. In this case the head and tail variables are both null.

A *circular, doubly-linked list* is shown in Figure □ (b). A circular list is formed by making use of variables which would otherwise be null: The last element of the list is made the predecessor of the first element; the first element, the successor of the last. The upshot is that we no longer need both a head and tail variable to keep track of the list. Even if only a single variable is used, both the first and the last list elements can be found in constant time.

Finally, Figure □ (c) shows a circular, doubly-linked list which has a single sentinel. This variation is similar to the preceding one in that both the first and the last list elements can be found in constant time. This variation has the advantage that no special cases are required when dealing with an empty list. Figure □ shows that the empty list is represented by a list with exactly one element--the sentinel. In the case of the empty list, the sentinel is both its own successor and predecessor. Since the sentinel is always present, and since it always has both a successor and a predecessor, the code for adding elements to the empty list is identical to that for adding elements to a non-empty list.

Exercises

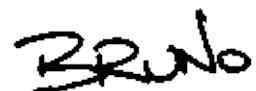
1. The array-based stack implementation introduced in Program □ uses a fixed length array. As a result, it is possible for the stack to become full.
 1. Rewrite the push method so that it doubles the length of the array when the array is full.
 2. Rewrite the pop method so that it halves the length of the array when the array is less than half full.
 3. Show that the *average* time for both push and pop operations is $O(1)$.
Hint: Consider the running time required to push $n = 2^k$ items onto an empty stack, where $k \geq 0$.
2. Consider a sequence S of push and pop operations performed on a stack that is initially empty. The sequence S is a valid sequence of operations if at no point is a pop operation attempted on an empty stack and if the stack is empty at the end of the sequence. Design a set of rules for generating a valid sequence.
3. Devise an implementation of the *queue* abstract data type *using two stacks*. Give algorithms for the enqueue and dequeue operations, and derive tight big-oh expressions for the running times of your implementation.
4. Write each of the following *infix* expressions in *postfix* notation:
 1. $a + b \times c \div d$,
 2. $a + b \times (c \div d)$,
 3. $(a + b) \times c \div d$,
 4. $(a + b) \times (c \div d)$,
 5. $(a + b \times c) \div d$, and
 6. $(c \div d) \times (a + b)$.
5. Write each of the following *postfix* expressions in *infix* notation:
 1. $w \ x \ y \div \ z \times -$,
 2. $w \ x \ y \ z \times \div -$,
 3. $w \ x - y \div z \times$,
 4. $w \ x - y \ z \times \div$,
 5. $w \ x \ y \div - z \times$, and
 6. $y \ z \times w \ x - \div$.
6. Devise an algorithm which translates a *postfix* expression to a *prefix*

expression. **Hint:** Use a stack of strings.

7. The array-based queue implementation introduced in Program □ uses a fixed length array. As a result, it is possible for the queue to become full.
 1. Rewrite the enqueue method so that it doubles the length of the array when the array is full.
 2. Rewrite the dequeue method so that it halves the length of the array when the array is less than half full.
 3. Show that the *average* time for both enqueue and dequeue operations is $O(1)$.
8. Stacks and queues can be viewed as special cases of deques. Show how all the operations on stacks and queues can be mapped to operations on a deque. Discuss the merits of using a deque to implement a stack or a queue.
9. Suppose we add a new operation to the stack ADT called `findMinimum` that returns a reference to the smallest element in the stack. Show that it is possible to provide an implementation for `findMinimum` that has a worst case running time of $O(1)$.
10. The *breadth-first traversal* method shown in Program □ visits the nodes of a tree in the order of their levels in the tree. Modify the algorithm so that the nodes are visited in reverse. **Hint:** Use a stack.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Projects

1. Enhance the functionality of the RPN calculator given in Program □ in the following ways:
 1. Use double-precision, floating-point arithmetic.
 2. Provide the complete repertoire of basic operators: +, -, ×, and ÷.
 3. Add an exponentiation operator and a unary negation operator.
 4. Add a *clear* method that empties the operand stack and a *print* method that prints out the contents of the operand stack.
2. Modify Program □ so that it accepts expressions written in *prefix* (Polish) notation. **Hint:** See Exercise □.
3. Write a program to convert a *postfix* expression into an *infix* expression using a stack. One way to do this is to modify the RPN calculator program given in Program □ to use a stack of infix expressions. A binary operator should pop two strings from the stack and then push a string which is formed by concatenating the operator and its operands in the correct order. For example, suppose the operator is ``*'' and the two strings popped from the stack are "(b+c)" and "a". Then the result that gets pushed onto the stack is the string "a*(b+c)".
4. Devise a scheme using a stack to convert an *infix* expression to a *postfix* expression. **Hint:** In a postfix expression operators appear *after* their operands whereas in an infix expression they appear *between* their operands. Process the symbols in the prefix expression one-by-one. Output operands immediately, but save the operators in a stack until they are needed. Pay special attention to the precedence of the operators.
5. Modify your solution to Project □ so that it immediately evaluates the infix expression. That is, create an *infixCalculator* method in the style of Program □.
6. Consider a string of characters, *S*, comprised only of the characters (,), [,], , and . We say that *S* is balanced if it has one of the following forms:
 - , i.e., *S* is the string of length zero,
 - ,
 - ,
 - ,
 - ,

where both T and U are balanced strings, In other words, for every left parenthesis, bracket or brace, there is a corresponding right parenthesis, bracket or brace. For example, " $\{()\[()\]\}$ " is balanced, but " $([)]$ " is not. Write a program that uses a stack of characters to test whether a given string is balanced.

7. Design and implement a `MultipleStack` class which provides $m \geq 1$ stacks in a single container. The declaration of the class should look something like this:

```
class MultipleStack(Container):
    def __init__(self, numberofStacks): ...
    def push(self, obj, whichStack): ...
    def pop(self, whichStack): ...
    # ...
```

- In addition to `self`, the `__init__` method takes a single integer argument that specifies the number of stacks in the container.
- In addition to `self`, the `push` method takes two arguments. The first gives the object to be pushed and the second specifies the stack on which to push it.
- In addition to `self`, the `pop` method takes a single integer argument which specifies the stack to pop.

Choose one of the following implementation approaches:

1. Keep all the stack elements in a single array.
 2. Use an array of `Stack` objects.
 3. Use a linked list of `Stack` objects.
8. Design and implement a class called `DequeAsDoublyLinkedList` that implements the `Deque` interface using a doubly-linked list. Select one of the approaches shown in Figure □.
9. In Section □, the `DequeAsArray` class extends the `QueueAsArray` class. Redesign the `DequeAsArray` and `QueueAsArray` components of the class hierarchy making `DequeAsArray` the base class and deriving `QueueAsArray` from it.

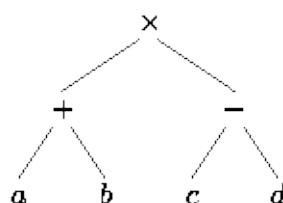
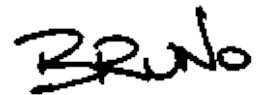


Figure: Expression tree for $(a + b) \times (c - d)$.

10. Devise an approach for evaluating an arithmetic expression using a *queue* (rather than a stack). **Hint:** Transform the expression into a tree as shown in Figure □ and then do a *breadth-first traversal* of the tree *in reverse* (see Exercise □). For example, the expression $(a + b) \times (c - d)$ becomes $d c b a - + \times$. Evaluate the resulting sequence from left to right using a queue in the same way that a postfix expression is evaluated using a stack.
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Ordered Lists and Sorted Lists

The most simple, yet one of the most versatile containers is the *list*. In this chapter we consider lists as *abstract data types*. A list is a series of items. In general, we can insert and remove items from a list and we can visit all the items in a list in the order in which they appear.

In this chapter we consider two kinds of lists--ordered lists and sorted lists. In an *ordered list* the order of the items is significant. Consider a list of the titles of the chapters in this book. The order of the items in the list corresponds to the order in which they appear in the book. However, since the chapter titles are not sorted alphabetically, we cannot consider the list to be sorted. Since it is possible to change the order of the chapters in book, we must be able to do the same with the items of the list. As a result, we may insert an item into an ordered list at any position.

On the other hand, a *sorted list* is one in which the order of the items is defined by some collating sequence. For example, the index of this book is a sorted list. The items in the index are sorted alphabetically. When an item is inserted into a sorted list, it must be inserted at the correct position.

As shown in Figure □, two abstract classes are used to represent the different list abstractions--`OrderedList` and `SortedList`. The various list abstractions can be implemented in many ways. In this chapter we examine implementations based on the *array* and the *linked list* foundational data structures presented in Chapter □.

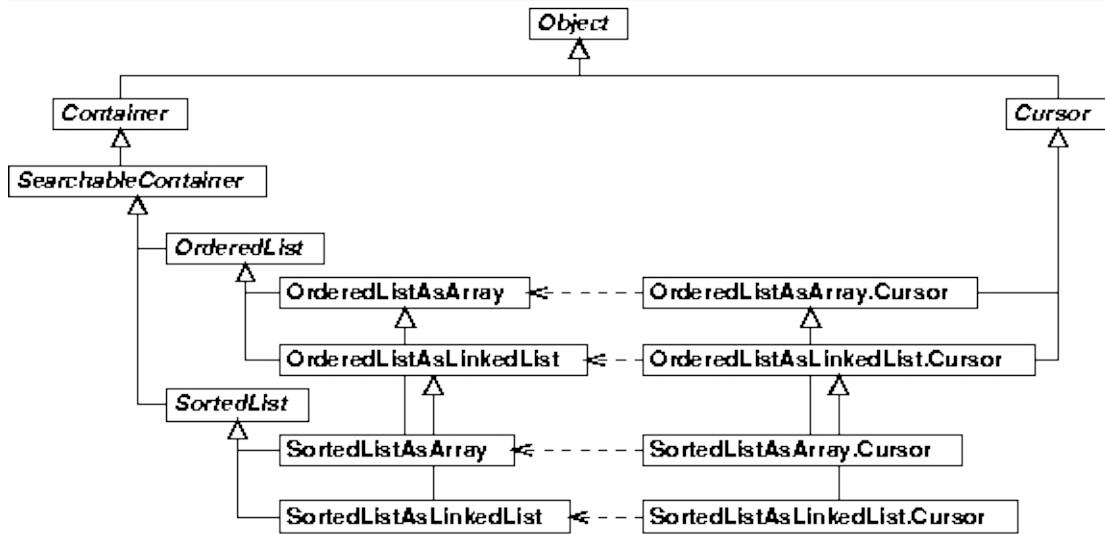


Figure: Object class hierarchy.

- [Ordered Lists](#)
- [Sorted Lists](#)
- [Exercises](#)
- [Projects](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Ordered Lists

An *ordered list* is a list in which the order of the items is significant. However, the items in an ordered lists are not necessarily *sorted*. Consequently, it is possible to *change* the order of items and still have a valid ordered list.

Program □ defines the `OrderedList` class. The abstract `OrderedList` class extends the abstract `Container` class defined in Program □. Recall that a searchable container is a container that supports the following additional operations:

insert

used to put objects into a the container;

withdraw

used to remove objects from the container;

find

used to locate objects in the container;

contains

used to test whether a given object instance is in the container.

The abstract `OrderedList` class adds the following operations:

getitem

used to access the object at a given position in the ordered list, and

findPosition

used to find the position of an object in the ordered list.

```
1 class OrderedList(SearchableContainer):
2
3     def __init__(self):
4         super(OrderedList, self).__init__()
5
6     def __getitem__(self, i):
7         pass
8     __getitem__ = abstractmethod(__getitem__)
9
10    def findPosition(self, obj):
11        pass
12    findPosition = abstractmethod(findPosition)
```

Program: Abstract OrderedList class.

In addition to `self`, the `findPosition` method of the abstract `List` class takes an object, `obj`, and searches the list for an object that matches the given one. The return value is an instance of a class derived from the `Cursor` class. Program □ defines the abstract `Cursor` class.

```
1 class Cursor(Object):
2
3     def __init__(self, list):
4         super(Cursor, self).__init__()
5         self._list = list
6
7     def getDatum(self):
8         pass
9     getDatum = abstractmethod(getDatum)
10
11    datum = property(
12        fget = lambda self: self.getDatum())
13
14    def insertAfter(self, obj):
15        pass
16    insertAfter = abstractmethod(insertAfter)
17
18    def insertBefore(self, obj):
19        pass
20    insertBefore = abstractmethod(insertBefore)
21
22    def withdraw(self, obj):
23        pass
24    withdraw = abstractmethod(withdraw)
```

Program: Abstract Cursor class.

A cursor ``remembers'' the position of an item in a list. The abstract Program \square class given in Program \square defines the following operations:

datum

used to access the object in the ordered list at the current cursor position;

insertAfter

used to insert an object into the ordered list after the current cursor position;

insertBefore

used to insert an object into the ordered list before the current cursor position; and

withdraw

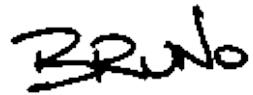
used to remove from the ordered list the object at the current cursor position.

As we did in the previous chapter with stacks, queues, and deques, we will examine two ordered list implementations--an array-based one and a linked-list one. Section \square presents an implementation using the `ArrayList` class; Section \square , an implementation using on the `LinkedList` class.

- [Array Implementation](#)
 - [Linked-List Implementation](#)
 - [Performance Comparison: `OrderedListAsArrayList` vs. `ListAsLinkedList`](#)
 - [Applications](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Array Implementation

This section presents an array-based implementation of ordered lists. Program □ introduces the `OrderedListAsArrayList` class. The `OrderedListAsArrayList` class extends the abstract `OrderedList` class defined in Program □.

```
1  class OrderedDictAsArrayList(OrderedList):
2
3      def __init__(self, size = 0):
4          super(OrderedListAsArrayList, self).__init__()
5          self._array = Array(size)
6
7      # ...
```

Program: `OrderedListAsArrayList` class `__init__` method.

-
- [Instance Attributes](#)
 - [Creating a List and Inserting Items](#)
 - [Finding Items in a List](#)
 - [Removing Items from a List](#)
 - [Positions of Items in a List](#)
 - [Finding the Position of an Item and Accessing by Position](#)
 - [Inserting an Item at an Arbitrary Position](#)
 - [Removing Arbitrary Items by Position](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Instance Attributes

The `OrderedListAsArray` class comprises one instance attribute, `_array`, which is an array of objects. In addition, the `OrderedListAsArray` class inherits the instance attribute `_count` from `Container`. The `_array` variable is used to hold the items in the ordered list. Specifically, the items in the list are stored in array positions $0, 1, \dots, \text{count} - 1$. In an ordered list the position of an item is significant. The item at position 0 is the first item in the list; the item at position $\text{count} - 1$, the last.

An item at position $i+1$ is the *successor* of the one at position i . That is, the one at $i+1$ *follows* or comes *after* the one at i . Similarly, an item at position i is the *predecessor* of the one at position $i+1$; the one at position i is said to *precede* or to come *before* the one at $i+1$.



Creating a List and Inserting Items

Program □ defines the `__init__` method of the `OrderedListAsArrayList` class. In addition to `self`, the `__init__` method takes a single argument which specifies the length of array to use in the representation of the ordered list. Thus if we use an array-based implementation, we need to know when a list is declared what will be the maximum number of items in that list. The `__init__` method initializes the `_array` variable as an array with the specified length. The running time of the `__init__` method is clearly $O(n)$, where $n = \text{size}$.

Program □ defines the `insert` method of the `OrderedListAsArrayList` class. The `insert` method is implemented by all searchable containers. Its purpose is to put an object into the container. The obvious question which arises is, where should the inserted item be placed in the ordered list? The simple answer is, at the end.

```
1 class OrderedListAsArrayList(OrderedList):
2
3     def insert(self, obj):
4         if self._count == len(self._array):
5             raise ContainerFull
6         self._array[self._count] = obj
7         self._count += 1
8
9     # ...
```

Program: `OrderedListAsArrayList` class `insert` method.

In Program □ we see that the `insert` method simply adds the new item to the end of the list, provided there is still room in the array. Normally, the array will not be full, so the running time of this method is $O(1)$.



Finding Items in a List

Program □ defines two `OrderedListAsArray` class methods which search for an object in the ordered list. The `__contains__` method tests whether a particular object instance is in the ordered list. The `find` method locates in the list an object which *matches* its argument.

```
 1  class OrderedDictAsArray(OrderedList):
 2
 3      def __contains__(self, obj):
 4          for i in xrange(self._count):
 5              if self._array[i] is obj:
 6                  return True
 7          return False
 8
 9      def find(self, obj):
10          for i in xrange(self._count):
11              if self._array[i] == obj:
12                  return self._array[i]
13          return None
14
15      # ...
```

Program: `OrderedListAsArray` class `__contains__` and `find` methods.

The `__contains__` method is a bool-valued method. In addition to `self`, the `__contains__` method takes a single argument, `obj`. This method compares the argument one-by-one with the contents of the `_array`. Note that this method tests whether a *particular object instance* is contained in the ordered list using the Python `is` operator. In the worst case, the object sought is not in the list. In this case, the running time of the method is $O(n)$, where $n = \text{count}$ is the number of items in the ordered list.

The `find` method also does a search of the ordered list. However, it uses the `==` operator to compare the items. Thus, the `Find` method searches the list for an object which matches its argument. The `Find` method returns the object found. If no match is found, it returns `None`. The running time of this method depends on the time required for the comparison operator, $T\langle -_{eq} \rangle$. In the worst case, the

object sought is not in the list. In this case the running time is $n \times T(\text{eq}) + O(n)$. For simplicity, we will assume that the comparison takes a constant amount of time. Hence, the running time of the method is also $O(n)$, where $n = \text{count}$ is the number of items in the list.

It is important to understand the subtle distinction between the search done by the `__contains__` method and that done by `find`. The `__contains__` method searches for a specific object instance while `find` simply looks for a matching object. Consider the following:

```
obj1 = [57]
obj2 = [57]
list = OrderedListAsArrayList(1)
list.insert(obj1)
```

This code fragment creates two tuples, both of which contain a single integer, 57. Only the first object, `obj1`, is inserted into the ordered list `list`. Consequently, evaluating the expression

`obj1 in list`

results in the value `true`; whereas evaluating the expression

`obj2 in list`

results in the value `false`.

On the other hand, if a search is done using the `find` method like this:

```
obj3 = list.find(obj2)
```

the search will be successful! After the call, `obj3` and `obj1` refer to the same object.



Removing Items from a List

Objects are removed from a searchable container using the `withdraw` method. Program □ defines the `withdraw` method for the `OrderedListAsArray` class. In addition to `self`, this method takes a single argument which is the object to be removed from the container. It is the specific object instance which is removed from the container, not simply one which matches (i.e., compares equal to) the argument.

```
1  class OrderedDictAsArray(OrderedList):
2
3      def withdraw(self, obj):
4          if self._count == 0:
5              raise ContainerEmpty
6          i = 0
7          while i < self._count and self._array[i] is not obj:
8              i += 1
9          if i == self._count:
10              raise KeyError
11          while i < self._count - 1:
12              self._array[i] = self._array[i + 1]
13              i += 1
14          self._array[i] = None
15          self._count -= 1
16
17      # ...
```

Program: `OrderedListAsArray` class `withdraw` method.

The `withdraw` method first needs to find the position of the item to be removed from the list. An exception is raised if the list is empty, or if the object to be removed is not in the list. The number of iterations needed to find an object depends on its position. If the object to be removed is found at position i , then the search phase takes $O(i)$ time.

Removing an object from position i of an ordered list which is stored in an array requires that all of the objects at positions $i+1, i+2, \dots, \text{count} - 1$, be moved one position to the left. Altogether, $\text{count} - 1 - i$ objects need to be moved. Hence,

this phase takes $O(\text{count} - i)$ time.

The running time of the `withdraw` method is the sum of the running times of the two phases, $O(i) + O(\text{count} - i)$. Hence, the total running time is $O(n)$, where $n = \text{count}$ is the number of items in the ordered list.

Care must be taken when using the `withdraw` method. Consider the following:

```
obj1 = [57]
obj2 = [57]
list = OrderedListAsArray(1)
list.insert(obj1)
```

To remove `obj1` from the ordered list, we may write

```
list.withdraw(obj1)
```

However, the call

```
list.withdraw(obj2)
```

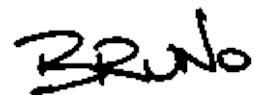
will fail because `obj2` is not actually in the list. If for some reason we have lost track of `obj1`, we can always write:

```
list.withdraw(list.find(obj2))
```

which first locates the object in the ordered list (`obj1`) which matches `obj2` and then deletes that object.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Positions of Items in a List

As shown in Program □, objects that implement the `cursor` interface can be used to access, insert, and delete objects in an ordered list. Program □ defines the nested class called `OrderedListAsArray.Cursor` that extends the abstract `Cursor` class. The idea is that instances of this class are used by the `OrderedListAsArray` class to represent the abstraction of a *position* in an ordered list.

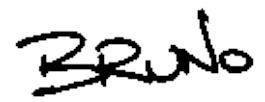
```
1  class OrderedListAsArray(OrderedList):
2
3      class Cursor(Cursor):
4
5          def __init__(self, list, offset):
6              super(OrderedListAsArray.Cursor, self).__init__(list)
7              self._offset = offset
8
9          def getDatum(self):
10             if self._offset < 0 \
11                 or self._offset >= self._list._count:
12                 raise IndexError
13             return self._list._array[self._offset]
14
15             # ...
16
17     # ...
```

Program: `OrderedListAsArray.Cursor` class `__init__` and `getDatum` methods.

The `Cursor` class has two instance attributes, `_list` and `_offset`. The `_list` instance attribute refers to an `OrderedListAsArray` instance and the `_offset` instance attribute records an offset in that list's array of objects. The `__init__` method simply assigns the given values to the `_list` and `_offset` instance attributes. Program □ also defines the `getDatum` method of the `Cursor` class. This method returns the item in the array at the position record in the `_offset` instance attribute, provided that position is valid. The running time of the accessor is simply $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Finding the Position of an Item and Accessing by Position

Program □ defines two more operations of the `OrderedListAsArray` class, `findPosition` and `__getitem__`. In addition to `self`, the `findPosition` method takes as its argument an object, `obj`. The purpose of this method is to search the ordered list for an item which matches the object, and to return the position in the form of a `Cursor`. In this case, the result is an instance of the `OrderedListAsArray.Cursor` class.

```

1  class OrderedDictAsArray(OrderedList):
2
3      def findPosition(self, obj):
4          i = 0
5          while i < self._count and self._array[i] != obj:
6              i += 1
7          return self.Cursor(self, i)
8
9      def __getitem__(self, offset):
10         if offset < 0 or offset >= self._count:
11             raise IndexError
12         return self._array[offset]
13
14     # ...

```

Program: `OrderedListAsArray` class `findPosition` and `__getitem__` methods.

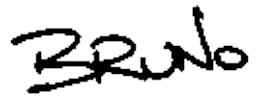
The search algorithm used in `findPosition` is identical to that used in the `find` method (Program □). The `findPosition` uses the `==` operator to locate a contained object which is equal to the search target. Note that if no match is found, the `_offset` is set to the value `_count`, which is one position to the right of the last item in the ordered list. The running time of `findPosition` is identical to that of `find`: $n \times T\{eq\} + O(n)$, where $n = count$.

In addition to `self`, the `__getitem__` method takes an `int` argument and returns the object in the ordered list at the specified position. In this case, the position is specified using an integer-valued subscript expression. The implementation of this method is trivial--it simply indexes into the array. Assuming the specified

offset is valid, the running time of this method is $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Inserting an Item at an Arbitrary Position

Two methods for inserting an item at an arbitrary position in an ordered list are declared in Program □--`insertBefore` and `insertAfter`. In addition to `self`, both of these take one argument, `obj`, the object to be inserted. The effects of these two methods are illustrated in Figure □.

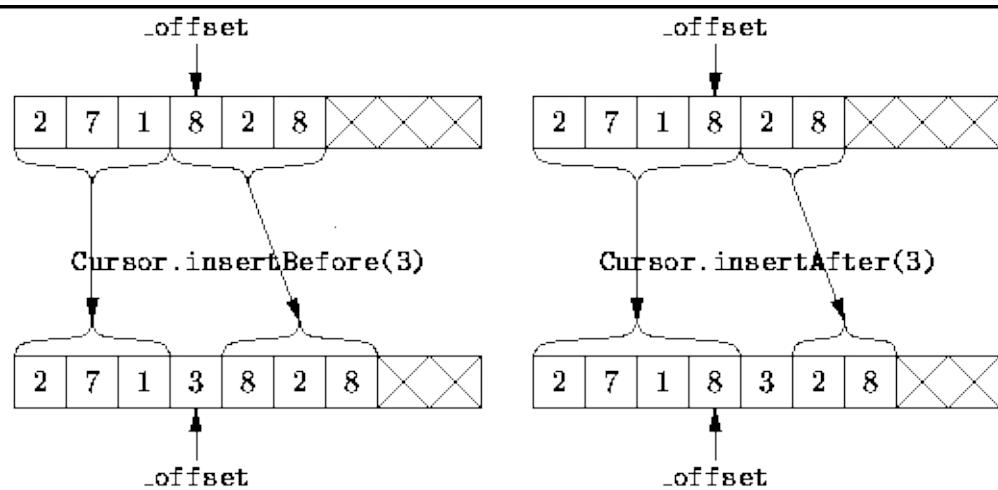


Figure: Inserting an item in an ordered list implemented as an array.

Figure □ shows that in both cases a number of items to the right of the insertion point need to be moved over to make room for the item that is being inserted into the ordered list. In the case of `insertBefore`, items to the right *including the item at the point of insertion* are moved; for `insertAfter`, only items to the right of the point of insertion are moved, and the new item is inserted in the array location following the insertion point.

Program □ gives the implementation of the `insertAfter` method for the `OrderedListAsArrayList.Cursor` class. The code for the `insertBefore` method is identical except for one line as explained below.

```
1 class OrderedListAsArray(OrderedList):
2
3     class Cursor(Cursor):
4
5         def insertAfter(self, obj):
6             if self._offset < 0 \
7                 or self._offset >= self._list._count:
8                 raise IndexError
9             if self._list._count == len(self._list._array):
10                 raise ContainerFull
11             insertPosition = self._offset + 1
12             i = self._list._count
13             while i > insertPosition:
14                 self._list._array[i] = self._list._array[i - 1]
15                 i -= 1
16             self._list._array[insertPosition] = obj
17             self._list._count += 1
18
19         # ...
20
21     # ...
```

Program: OrderedListAsArray.Cursor class insertAfter method.

In addition to `self`, the `insertAfter` method takes one argument, `obj`. The method begins by performing some simple tests to ensure that the position is valid and that there is room left in the array to do the insertion.

On line 11 the array index where the new item will ultimately be stored is computed. For `insertAfter` the index is $\text{offset} + 1$ as shown in Program □. In the case of `insertBefore`, the value required is simply `_offset`. The loop on lines 13-15 moves items over and then object being inserted is put in the array on line 16.

If we assume that no exceptions are raised, the running time of `insertAfter` is dominated by the loop which moves list items. In the worst case, all the items in the array need to be moved. Thus, the running time of both the `insertAfter` and `insertBefore` method is $O(n)$, where $n = \text{count}$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Removing Arbitrary Items by Position

The final method of the `OrderedListAsArray.Cursor` class that we will consider is the `withdraw` method. The desired effect of this method is to remove from the ordered list the item at the position specified by the cursor.

Figure □ shows the way in which to delete an item from an ordered list which implemented with an array. All of the items remaining in the list to the right of the deleted item need to be shifted to the left in the array by one position.

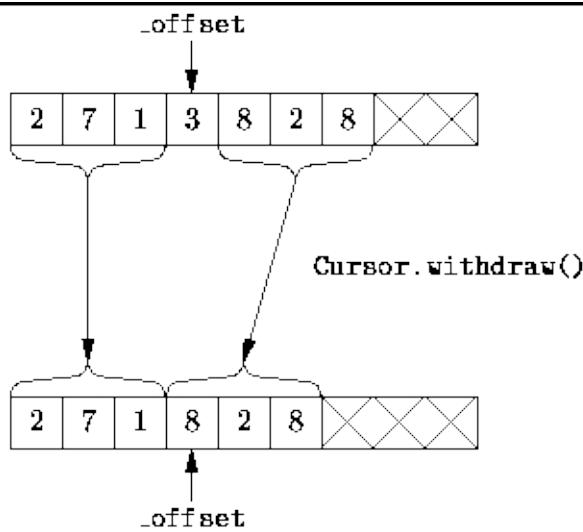


Figure: Withdrawing an item from an ordered list implemented as an array.

Program □ gives the implementation of the `withdraw` method. After checking the validity of the position, all of the items following the item to be withdraw are moved one position to the left in the array.

```
1  class OrderedListAsArray(OrderedList):
2
3      class Cursor(Cursor):
4
5          def withdraw(self):
6              if self._offset < 0 \
7                  or self._offset >= self._list._count:
8                  raise IndexError
9              if self._list._count == 0:
10                  raise ContainerEmpty
11              i = self._offset
12              while i < self._list._count - 1:
13                  self._list._array[i] = self._list._array[i + 1]
14                  i += 1
15                  ++i
16              self._list._array[i] = None
17              self._list._count -= 1
18
19          # ...
20
21      # ...
```

Program: OrderedListAsArray.Cursor class withdraw method.

The running time of the withdraw method depends on the position in the array of the item being deleted and on the number of items in the ordered lists. In the worst case, the item to be deleted is in the first position. In this case, the work required to move the remaining items left is $O(n)$, where $n = \text{_count}$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Linked-List Implementation

This section presents a linked-list implementation of ordered lists. Program □ introduces the `OrderedListAsLinkedList` class. The `OrderedListAsLinkedList` class extends the abstract `OrderedList` class defined in Program □.

```
1  class OrderedListAsLinkedList(OrderedList):
2
3      def __init__(self):
4          super(OrderedListAsLinkedList, self).__init__()
5          self._linkedList = LinkedList()
6
7      # ...
```

Program: `OrderedListAsLinkedList` class `__init__` method.

-
- [Instance Attributes](#)
 - [Inserting and Accessing Items in a List](#)
 - [Finding Items in a List](#)
 - [Removing Items from a List](#)
 - [Positions of Items in a List](#)
 - [Finding the Position of an Item and Accessing by Position](#)
 - [Inserting an Item at an Arbitrary Position](#)
 - [Removing Arbitrary Items by Position](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Instance Attributes

Objects of the `OrderedListAsLinkedList` class contain one instance attribute, `_linkedList`, which is a linked list of objects. The `_linkedList` is used to hold the items in the ordered list. Since a linked list is used, there is no notion of an inherent limit on the number of items which can be placed in the ordered list. Items can be inserted until the available memory is exhausted.



Inserting and Accessing Items in a List

Program □ gives the code for the `__init__` method of the `OrderedListAsLinkedList` class. The `__init__` method simply creates an empty linked list. Clearly, the running time of the `__init__` method is $O(1)$.

Program □ gives the code for the `insert` and `__getitem__` methods of the `OrderedListAsLinkedList` class. The `insert` method takes an object and adds it to the ordered list. As in the case of the `ArrayAsLinkedList` class, the object is added at the end of the ordered list. This is done simply by calling the `append` method from the `LinkedList` class.

```
1  class OrderedListAsLinkedList(OrderedList):
2
3      def insert(self, obj):
4          self._linkedList.append(obj)
5          self._count += 1
6
7      def __getitem__(self, offset):
8          if offset < 0 or offset >= self._count:
9              raise IndexError
10         ptr = self._linkedList.head
11         i = 0
12         while i < offset and ptr is not None:
13             ptr = ptr.next
14             i += 1
15         return ptr.datum
16
17     # ...
```

Program: `OrderedListAsLinkedList` class `insert` and `__getitem__` methods.

The running time of the `insert` method is determined by that of `append`. In Chapter □ this was shown to be $O(1)$. The only other work done by the `insert` method is to add one to the `_count` variable. Consequently, the total running time for `insert` is $O(1)$.

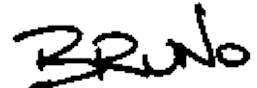
Program □ also defines the `__getitem__` method. In addition to `self`, the `__getitem__` method takes an argument of type `int`. This method is used to

access elements of the ordered list by their position in the list. In this case, the position is specified by a non-negative, integer-valued index. Since there is no way to access directly the i^{th} element of linked list, the implementation of this method comprises a loop which traverses the list to find the i^{th} item. The method returns a reference to the i^{th} item, provided $0 \leq i < _count$. Otherwise, i is not a valid subscript value and the method raises an exception.

The running time of the accessor method depends on the number of items in the list and on the value of the subscript expression. In the worst case, the item sought is at the end of the ordered list. Therefore, the worst-case running time of this algorithm, assuming the subscript expression is valid, is $O(n)$, where $n = _count$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Finding Items in a List

Program □ defines the `__contains__` and `find` methods of the `ListAsLinkedList` class. The implementations of these methods are almost identical. However, they differ in two key aspects--the comparison used and the return value.

```
 1  class OrderedListAsLinkedList(OrderedList):
 2
 3      def __contains__(self, obj):
 4          ptr = self._linkedList.head
 5          while ptr is not None:
 6              if ptr.datum is obj:
 7                  return True
 8              ptr = ptr.next
 9          return False
10
11      def find(self, arg):
12          ptr = self._linkedList.head
13          while ptr is not None:
14              obj = ptr.datum
15              if obj == arg:
16                  return obj
17              ptr = ptr.next
18          return None
19
20      # ...
```

Program: `OrderedListAsLinkedList` class `__contains__` and `find` methods.

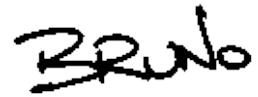
The `__contains__` method tests whether a particular object instance is contained in the ordered list. It returns a `bool` value indicating whether the object is present. The running time of this method is clearly $O(n)$, where $n = \text{count}$, the number of items in the ordered list.

The `find` method locates an object which matches a given object. The match is determined by using the `==` operator. `find` returns a reference to the matching object if one is found. Otherwise, it returns `None`. The running time for this method, is $n \times T(\text{eq}) + O(n)$, where $T(\text{eq})$ is the time required to do the

comparison, and $n = \text{count}$ is the number of items in the ordered list. This simplifies to $O(n)$ when the comparison can be done in constant time.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Removing Items from a List

The `withdraw` method is used to remove a specific object instance from an ordered list. The implementation of the `withdraw` method for the `OrderedListAsLinkedList` class is given in Program □.

```
1  class OrderedListAsLinkedList(OrderedList):
2
3      def withdraw(self, obj):
4          if self._count == 0:
5              raise ContainerEmpty
6          self._linkedlist.extract(obj)
7          self._count -= 1
8
9      # ...
```

Program: `OrderedListAsLinkedList` class `withdraw` method.

The implementation of `withdraw` is straight-forward: It simply calls the `extract` method provided by the `LinkedList` class to remove the specified object from `linkedlist`. The running time of the `withdraw` method is dominated by that of `extract` which was shown in Chapter □ to be $O(n)$, where n is the number of items in the linked list.



Positions of Items in a List

Program □ gives the definition of a the `OrderedListAsLinkedList.Cursor` nested class. This class extends the abstract `Cursor` class defined in Program □. The purpose of this class is to record the position of an item in an ordered list implemented as a linked list.

```
1  class OrderedListAsLinkedList(OrderedList):
2
3      class Cursor(Cursor):
4
5          def __init__(self, list, element):
6              super(OrderedListAsLinkedList.Cursor, self) \
7                  .__init__(list)
8              self._element = element
9
10         def getDatum(self):
11             return self._element.datum
12
13         # ...
14
15     # ...
```

Program: `OrderedListAsLinkedList.Cursor` class `__init__` and `getDatum` methods.

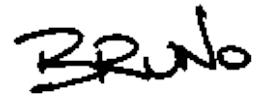
The `Cursor` class has two instance attributes, `_list` and `_element`. The `_list` instance attribute refers to an `OrderedListAsLinkedList` instance and the `_element` refers to the linked-list element in which a given item appears. Notice that this version of `Cursor` is fundamentally different from the array version. In the array version, the position was specified by an offset, i.e, by an *ordinal number* that shows the position of the item in the ordered sequence. In the linked-list version, the position is specified by a reference to the element of the linked list in which the item is stored. Regardless of the implementation, both kinds of position provide exactly the same functionality because they both implement methods defined in the abstract `Cursor` class.

The `getDatum` method of the `OrderedListAsLinkedList.Cursor` class is also

defined in Program □. This method dereferences the `_element` instance attribute to obtain the required item in the ordered list. The running time is clearly $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Finding the Position of an Item and Accessing by Position

The `findPosition` method of the `OrderedListAsLinkedList` class is used to determine the position of an item in an ordered list implemented as a linked list. Its result is an instance of the nested `Cursor` class. The `findPosition` method is defined in Program □

```
1  class OrderedDictAsLinkedList(OrderedList):
2
3      def findPosition(self, obj):
4          ptr = self._linkedList.head
5          while ptr is not None:
6              if ptr.datum == obj:
7                  break
8              ptr = ptr.next
9          return self.Cursor(self, ptr)
10
11      # ...
```

Program: `OrderedListAsLinkedList` class `findPosition` method

In addition to `self`, the `findPosition` method takes as its argument an object that is the target of the search. The search algorithm used by `findPosition` is identical to that of `find`, which is given in Program □. Consequently, the running time is the same: $n \times T\{=\} + O(n)$, where $T\{=\}$ is the time required to match two objects, and $n = _count$ is the number of items in the ordered list.

Inserting an Item at an Arbitrary Position

Having determined the position of an item in an ordered list, we can make use of that position to insert items into the middle of the list. Two methods are specifically provided for this purpose--`insertAfter` and `insertBefore`. In addition to `self`, both of these take a single argument--the object to be inserted into the list.

```
1  class OrderedListAsLinkedList(OrderedList):
2
3      class Cursor(Cursor):
4
5          def insertAfter(self, obj):
6              self._element.insertAfter(obj)
7              self._list._count += 1
8
9          # ...
10
11     # ...
```

Program: `OrderedListAsLinkedList.Cursor` class `insertAfter` method.

Program □ gives the implementation for the `insertAfter` method of the `OrderedListAsLinkedList.Cursor` class. This method simply calls the `insertAfter` method provided by the `LinkedList` class. Assuming no exceptions are thrown, the running time for this method is $O(1)$.

The implementation of `insertBefore` is not shown--its similarity with `insertAfter` should be obvious. Since it must call the `insertBefore` method provided by the `LinkedList` class, we expect the worst case running time to be $O(n)$, where $n = _count$.

Removing Arbitrary Items by Position

The final method to be considered is the `withdraw` method of the `OrderedListAsLinkedList.Cursor` class. This method removes an arbitrary item from an ordered list, where the position of that item is specified by a cursor instance. The code for the `withdraw` method is given in Program □.

```
1  class OrderedDictAsLinkedList(OrderedList):
2
3      class Cursor(Cursor):
4
5          def withdraw(self):
6              self._list._linkedList.extract(self._element.datum)
7              self._list._count -= 1
8
9          # ...
10
11     # ...
```

Program: `OrderedListAsLinkedList.Cursor` class `withdraw` method.

The item in the linked list at the position specified by the cursor is removed by calling the `extract` method provided by the `LinkedList` class. The running time of the `withdraw` method depends on the running time of the `extract` of the `LinkedList` class. The latter was shown to be $O(n)$ where n is the number of items in the linked list. Consequently, the total running time is $O(n)$.

Performance Comparison: `OrderedListAsArray` vs. `ListAsLinkedList`

The running times calculated for the various operations of the two ordered list implementations, `OrderedListAsArray` and `OrderedListAsLinkedList`, are summarized below in Table □. With the exception of two operations, the running times of the two implementations are asymptotically identical.

Table: Running times of operations on ordered lists.

method	ordered list implementation	
	OrderedList- AsArray	OrderedList- AsLinkedList
<code>__contains__</code>	$O(n)$	$O(n)$
<code>find</code>	$O(n)$	$O(n)$
<code>findPosition</code>	$O(n)$	$O(n)$
<code>__getitem__</code>	$O(1)$	\neq $O(n)$
<code>insert</code>	$O(1)$	$O(1)$
<code>withdraw</code>	$O(n)$	$O(n)$
<code>Cursor.datum</code>	$O(1)$	$O(1)$
<code>Cursor.insertAfter</code>	$O(n)$	\neq $O(1)$
<code>Cursor.insertBefore</code>	$O(n)$	$O(n)$
<code>Cursor.withdraw</code>	$O(n)$	$O(n)$

The two differences are the `__getitem__` method and the `insertAfter` method. The `__getitem__` operation can be done constant time when using an array, but it requires $O(n)$ in a linked list. Conversely, `insertAfter` requires $O(n)$ time when using an array, but can be done in constant time in the singly-linked list.

Table □ does not tell the whole story. The other important difference between the two implementations is the amount of space required. Consider first the array

implementation, `OrderedListAsArray`. The storage needed for an `OrderedListAsArray` which can hold *at most M ComparableObjects* is given by:

$$\begin{aligned}\text{sizeof}(\text{OrderedListAsArray}) &= \text{sizeof}(_count) + \text{sizeof}(_array) \\ &= 2\text{sizeof}(id) + \text{sizeof}(int) \\ &\quad + \text{sizeof}(\text{Array}(M))\end{aligned}$$

The storage required for an array was discussed in Chapter □:

$$\begin{aligned}\text{sizeof}(\text{Array}(M)) &= \text{sizeof}(_baseIndex) + \text{sizeof}(_data) \\ &= 2\text{sizeof}(id) + \text{sizeof}(int) + M\text{sizeof}(id)\end{aligned}$$

Therefore, the storage needed for an `OrderedListAsArray` is:

$$\text{sizeof}(\text{OrderedListAsArray}) = 2\text{sizeof}(int) + (M + 4)\text{sizeof}(id)$$

Notice that we do not include in this calculation the space required for the objects themselves. Since we cannot know the types of the contained objects, we cannot calculate the space required by those objects.

A similar calculation can also be done for the `OrderedListAsLinkedList` class. In this case, we assume that the actual number of contained objects is *n*. The total storage required is given by:

$$\begin{aligned}\text{sizeof}(\text{OrderedListAsLinkedList}) &= \text{sizeof}(_count) \\ &\quad + \text{sizeof}(_linkedList) \\ &= 2\text{sizeof}(id) + \text{sizeof}(int) \\ &\quad + \text{sizeof}(\text{LinkedList}()) \\ &\quad + n\text{sizeof}(\text{LinkedList.Element})\end{aligned}$$

The storage required for a linked list was discussed in Chapter □:

$$\begin{aligned}\text{sizeof}(\text{LinkedList}()) &= \text{sizeof}(_head) + \text{sizeof}(_tail) \\ &= 2\text{sizeof}(id) \\ \text{sizeof}(\text{LinkedList.Element}()) &= \text{sizeof}(_list) + \text{sizeof}(_datum) \\ &\quad + \text{sizeof}(_next) \\ &= 3\text{sizeof}(id)\end{aligned}$$

Therefore, the storage needed for an `OrderedListAsLinkedList` is:

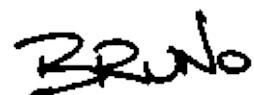
$$\text{sizeof}(\text{OrderedListAsLinkedList}) = \text{sizeof}(int) + (3n + 4)\text{sizeof}(id)$$

If we assume that integers and object references require four bytes each, the storage requirement for the `OrderedListAsArray` class becomes $4M+24$ bytes; and for the `OrderedListAsLinkedList` class, $12n+20$ bytes. That is, the storage needed for the array implementation is $O(M)$, where M is the maximum length of the ordered list; whereas, the storage needed for the linked list implementation is $O(n)$, where n is the actual number of items in the ordered list. Equating the two expressions, we get that the break-even point occurs at $n = (M + 1)/3 \approx M/3$. That is, if the array is less than one third full, the array version uses more memory space; and if the array is more than one third full, the linked list version uses more memory space.

It is not just the amount of memory space used that should be considered when choosing an ordered list implementation. We must also consider the implications of the existence of the limit M . The array implementation requires *a priori* knowledge about the maximum number of items to be put in the ordered list. The total amount of storage then remains constant during the course of execution. On the other hand, the linked list version has no pre-determined maximum length. It is only constrained by the total amount of memory available to the program. Furthermore, the amount of memory used by the linked list version varies during the course of execution. We do not have to commit a large chunk of memory for the duration of the program.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Applications

The applications of lists and ordered lists are myriad. In this section we will consider only one--the use of an ordered list to represent a polynomial. In general, an n^{th} -order polynomial in x , for non-negative integer n , has the form

$$\sum_{i=0}^n a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where $a_n \neq 0$. The term a_i is the *coefficient* of the i^{th} power of x . We shall assume that the coefficients are real numbers.

An alternative representation for such a polynomial consists of a sequence of ordered pairs:

$$\{(a_0, 0), (a_1, 1), (a_2, 2), \dots, (a_n, n)\}.$$

Each ordered pair, (a_i, i) , corresponds to the term $a_i x^i$ of the polynomial. That is, the ordered pair is comprised of the coefficient of the i^{th} term together with the subscript of that term, i . For example, the polynomial $31 + 41x + 59x^2$ can be represented by the sequence $\{(31, 0), (41, 1), (59, 2)\}$.

Consider now the 100^{th} -order polynomial $x^{100} + 1$. Clearly, there are only two nonzero coefficients: $a_{100} = 1$ and $a_0 = 1$. The advantage of using the sequence of ordered pairs to represent such a polynomial is that we can omit from the sequence those pairs that have a zero coefficient. We represent the polynomial $x^{100} + 1$ by the sequence $\{(1, 100), (1, 0)\}$.

Now that we have a way to represent polynomials, we can consider various operations on them. For example, consider the polynomial

$$p(x) = \sum_{i=0}^n a_i x^i.$$

We can compute its *derivative* with respect to x by *differentiating* each of the terms to get

$$p'(x) = \sum_{i=0}^{n-1} a'_i x^i,$$

where $a'_i = (i + 1)a_{i+1}$. In terms of the corresponding sequences, if $p(x)$ is represented by the sequence

$$\{(a_0, 0), (a_1, 1), (a_2, 2), \dots, (a_i, i), \dots, (a_n, n)\},$$

then its derivative is the sequence

$$\{(a_1, 0), (2a_2, 1), (3a_3, 2), \dots, (a_i, i - 1), \dots, (na_n, n - 1)\}.$$

This result suggests a very simple algorithm to differentiate a polynomial which is represented by a sequence of ordered pairs:

1. Drop the ordered pair that has a zero exponent.
2. For every other ordered pair, multiply the coefficient by the exponent, and then subtract one from the exponent.

Since the representation of an n^{th} -order polynomial has at most $n+1$ ordered pairs, and since a constant amount of work is necessary for each ordered pair, this is inherently an $\Omega(n)$ algorithm.

Of course, the worst-case running time of the polynomial differentiation will depend on the way that the sequence of ordered pairs is implemented. We will now consider an implementation that makes use of the `OrderedListAsLinkedList` class. To begin with, we need a class to represent the terms of the polynomial. Program □ gives the definition of the `Term` class and several of its methods.

```

1  class Polynomial(Container):
2
3      class Term(Object):
4
5          def __init__(self, coefficient, exponent):
6              self._coefficient = coefficient
7              self._exponent = exponent
8
9          def _compareTo(self, term):
10             assert isinstance(self, term.__class__)
11             if self._exponent == term._exponent:
12                 return cmp(self._coefficient, term._coefficient)
13             else:
14                 return cmp(self._exponent, term._exponent)
15
16         def differentiate(self):
17             if self._exponent > 0:
18                 self._coefficient *= self._exponent
19                 self._exponent -= 1
20             else:
21                 self._coefficient = 0
22
23         # ...
24
25     # ...

```

Program: `Polynomial.Term` class.

Each `Term` instance has two instance attributes, `_coefficient` and `_exponent`, which correspond to the elements of the ordered pair as discussed above. The former is a `float` and the latter, an `int`.

The `Term` class extends the abstract `Object` class introduced in Program □. Program □ defines three methods: `__init__`, `_compareTo`, and `differentiate`. In addition to `self`, the `__init__` method simply takes a pair of arguments and initializes the corresponding instance attributes accordingly.

The `_compareTo` method is used to compare two `Term` instances. Consider two terms, ax^i and bx^j . We define the relation \prec on terms of a polynomial as follows:

$$ax^i \prec bx^j \iff (i < j) \vee (i = j \wedge a < b)$$

Note that the relation \prec does not depend on the value of the variable x . The `_compareTo` method implements the \prec relation.

Finally, the `differentiate` method does what its name says: It differentiates a term with respect to x . Given a term such as $(a_0, 0)$, it computes the result $(0,0)$; and given a term such as (a_i, i) where $i > 0$, it computes the result $(ia_i, i - 1)$.

We now consider the representation of a polynomial. Program □ defines the abstract `Polynomial` class. The class comprises three abstract methods--`addTerm`, `differentiate`, and `__add__`. The `addTerm` method is used to add terms to a polynomial. The `differentiate` method differentiates the polynomial. The `__add__` method is used to compute the sum of two polynomials.

```

1  class Polynomial(Container):
2
3      def __init__(self):
4          super(Polynomial, self).__init__()
5
6      def addTerm(self, term):
7          pass
8      addTerm = abstractmethod(addTerm)
9
10     def differentiate(self):
11         pass
12     differentiate = abstractmethod(differentiate)
13
14     def __add__(self, polynomial):
15         pass
16     __add__ = abstractmethod(__add__)
17
18     # ...

```

Program: Abstract Polynomial class.

Program □ introduces the `PolynomialAsOrderedList` class. This concrete class extends the abstract `Polynomial` class. It has a single instance attribute called `_list`. In this case `_list`, an instance of the `OrderedListAsLinkedList` class, is used to contain the terms of the polynomial.

```
1 class PolynomialAsOrderedList(Polynomial):
2
3     def __init__(self):
4         self._list = OrderedListAsLinkedList()
5
6     def addTerm(self, term):
7         self._list.insert(term)
8
9     def accept(self, visitor):
10        assert isinstance(visitor, Visitor)
11        self._list.accept(visitor)
12
13    def find(self, term):
14        return self._list.find(term)
15
16    def withdraw(self, term):
17        self._list.withdraw(term)
18
19    # ...
```

Program: PolynomialAsOrderedList class.

Program □ defines the method `differentiate` which has the effect of changing the polynomial to its derivative with respect to x . To compute this derivative, it is necessary to call the `differentiate` method of the `Term` class for each term in the polynomial. Since the polynomial is implemented as a container, there is an `accept` method which can be used to perform a given operation on all of the objects in that container. In this case, we define a visitor, `DifferentiatingVisitor`, which assumes its argument is an instance of the `Term` class and differentiates it.

```
1 class Polynomial(Container):
2
3     class DifferentiatingVisitor(Visitor):
4
5         def __init__(self):
6             super(Polynomial.DifferentiatingVisitor, self) \
7                 .__init__()
8
9         def visit(self, term):
10             term.differentiate()
11
12     def differentiate(self):
13         self.accept(self.DifferentiatingVisitor())
14         zeroTerm = self.find(self.Term(0, 0))
15         if zeroTerm is not None:
16             self.withdraw(zeroTerm)
17
18     # ...
```

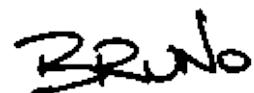
Program: Polynomial class differentiate method.

After the terms in the polynomial have been differentiated, it is necessary to check for the term $(0,0)$ which arises from differentiating $(a_0, 0)$. The `find` method is used to locate the term, and if one is found the `withdraw` method is used to remove it.

The analysis of the running time of the polynomial `differentiate` method is straightforward. The running time required to differentiate a term is clearly $O(1)$. So too is the running time of the `visit` method of the `DifferentiatingVisitor`. The latter method is called once for each contained object. In the worst case, given an n^{th} -order polynomial, there are $n+1$ terms. Therefore, the time required to differentiate the terms is $O(n)$. Locating the zero term is $O(n)$ in the worst case, and so too is deleting it. Therefore, the total running time required to differentiate a n^{th} -order polynomial is $O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Sorted Lists

The next type of searchable container that we consider is a *sorted list*. A sorted list is like an ordered list: It is a searchable container that holds a sequence of objects. However, the position of an item in a sorted list is not arbitrary. The items in the sequence appear in order, say, from the smallest to the largest. Of course, for such an ordering to exist, the relation used to sort the items must be a *total order*◊.

Program □ defines the `SortedList` class. The abstract `SortedList` class extends the abstract `OrderedList` class defined in Program □.

```
1  class SortedList(OrderedList):
2
3      def __init__(self):
4          super(SortedList, self).__init__()
```

Program: Abstract `SortedList` class.

In addition to the basic repertoire of operations supported by all searchable containers, sorted lists provide the following operations (inherited from the `OrderedList` class):

getitem

used to access the object at a given position in the sorted list; and

findPosition

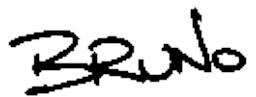
used to find the position of an object in the sorted list.

Sorted lists are very similar to ordered lists. As a result, we can make use of the code for ordered lists when implementing sorted lists. Specifically, we will consider an array-based implementation of sorted lists that is derived from the `OrderedListAsArrayList` class defined in Section □, and a linked-list implementation of sorted lists that is derived from the `OrderedListAsLinkedList` class given in Section □.

- [Array Implementation](#)
 - [Linked-List Implementation](#)
 - [Performance Comparison: SortedListAsArray vs. SortedListAsList](#)
 - [Applications](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Array Implementation

The `SortedListAsArrayList` class is introduced in Program □. The `SortedListAsArrayList` class extends the `OrderedListAsArrayList` class introduced in Program □ and it also extends the abstract `SortedList` class defined in Program □.

```
1  class SortedListAsArrayList(OrderedListAsArrayList, SortedList):
2
3      def __init__(self, size = 0):
4          super(SortedListAsArrayList, self).__init__(size)
5
6      # ...
```

Program: `SortedListAsArrayList` class.

There are no additional instance attributes required to implement the `SortedListAsArrayList` class. That is, the instance attributes provided by the base class `OrderedListAsArrayList` are sufficient.

-
- [Inserting Items in a Sorted List](#)
 - [Locating Items in an Array-Binary Search](#)
 - [Finding Items in a Sorted List](#)
 - [Removing Items from a List](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Inserting Items in a Sorted List

When inserting an item into a sorted list we have as a *precondition* that the list is already sorted. Furthermore, once the item is inserted, we have the *postcondition* that the list must still be sorted. Therefore, all the items initially in the list that are larger than the item to be inserted need to be moved to the right by one position as shown in Figure □.

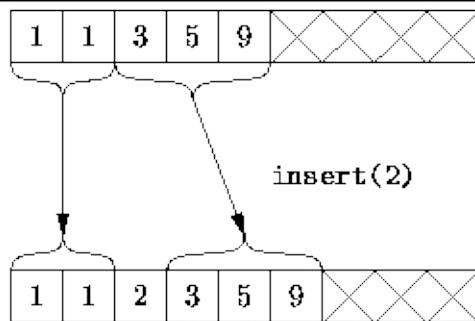


Figure: Inserting an item into a sorted list implemented as an array.

Program □ defines the `insert` method for the `SortedListAsArrayList` class. In addition to `self`, this method takes as its argument the object to be inserted in the list. Recall that the `insert` method provided by the `ListAsLinkedList` class simply adds items at the end of the array. While this is both efficient and easy to implement, it is not suitable for the `SortedListAsArrayList` class since the items in the array must be end up in order.

```
1 class SortedListAsArray(OrderedListAsArray, SortedList):
2
3     def insert(self, obj):
4         if self._count == len(self._array):
5             raise ContainerFull
6         i = self._count
7         while i > 0 and self._array[i - 1] > obj:
8             self._array[i] = self._array[i - 1]
9             i -= 1
10        self._array[i] = obj
11        self._count += 1
12
13    # ...
```

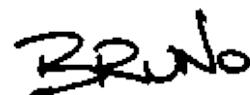
Program: SortedListAsArray class insert method.

The `insert` method given in Program □ first checks that there is still room in the array for one more item. Then, to insert the item into the list, all the items in the list that are larger than the one to be inserted are moved to the right. This is accomplished by the loop on lines 7-9. Finally, the item to be inserted is recorded in the appropriate array position on line 10.

In the worst case, the item to be inserted is smaller than all the items already in the sorted list. In this case, all $n = _count$ items must be moved one position to the right. Therefore, the running time of the `insert` method is $O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Locating Items in an Array-Binary Search

Given a sorted array of items, an efficient way to locate a given item is to use a *binary search*. The `findOffset` method of the `SortedListAsArrayList` class defined in Program □ uses a binary search to locate an item in the array which matches a given item.

```
 1  class SortedListAsArrayList(OrderedListAsArrayList, SortedList):
 2
 3      def findOffset(self, obj):
 4          left = 0
 5          right = self._count - 1
 6          while left <= right:
 7              middle = (left + right) / 2
 8              if obj > self._array[middle]:
 9                  left = middle + 1
10              elif obj < self._array[middle]:
11                  right = middle - 1
12              else:
13                  return middle
14          return -1
15
16      # ...
```

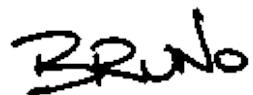
Program: `SortedListAsArrayList` class `findOffset` method.

The binary search algorithm makes use of a *search interval* to determine the position of an item in the sorted list. The search interval is a range of array indices in which the item being sought is expected to be found. The initial search interval is $0 \dots \text{count} - 1$. The interval is iteratively narrowed by comparing the item sought with the item found in the array at the middle of the search interval. If the middle item matches the item sought, then we are done. Otherwise, if the item sought is less than the middle item, then we can discard the middle item and the right half of the interval; if the item sought is greater than the middle item, we can discard the middle item and the left half of the interval. At each step, the size of the search interval is approximately halved. The algorithm terminates either when the item sought is found, or if the size of the search interval becomes zero.

In the worst case, the item sought is not in the sorted list. Specifically, the worst case occurs when the item sought is smaller than any item in the list because this case requires two comparisons in each iteration of the binary search loop. In the worst case, $\lceil \log n \rceil + 2$ iterations are required. Therefore, the running time of the `FindOffset` method is $(\lceil \log n \rceil + 2) \times (T_{\text{LT}} + T_{\text{GT}}) + O(\log n)$, where T_{LT} and T_{GT} represents the running times required to compare two `ComparableObject` instances. If we assume that $T_{\text{LT}} = O(1)$ and $T_{\text{GT}} = O(1)$, then the total running time is simply $O(\log n)$, where $n = \text{_count}$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Finding Items in a Sorted List

Program □ defines the two methods used to locate items in a sorted list. Both of these methods make use of the `findOffset` method described above.

```

1  class SortedListAsArray(OrderedListAsArray, SortedList):
2
3      def find(self, obj):
4          offset = self.findOffset(obj)
5          if offset >= 0:
6              return self._array[offset]
7          else:
8              return None
9
10     class Cursor(OrderedListAsArray.Cursor):
11
12         def __init__(self, list, offset):
13             super(SortedListAsArray.Cursor, self) \
14                 .__init__(list, offset)
15
16         def insertAfter(self, obj):
17             raise TypeError
18
19         def insertBefore(self, obj):
20             raise TypeError
21
22         def findPosition(self, obj):
23             return self.Cursor(self, self.findOffset(obj))
24
25     # ...

```

Program: `SortedListAsArray` class `find` and `findPosition` methods.

The `find` method takes a given object and finds the object contained in the sorted list which matches (i.e., compares equal to) the given one. It calls `findOffset` to determine by doing a binary search the array index at which the matching object is found. `find` returns the matching object, if one is found; otherwise, it returns `None`. The total running time of `find` is dominated by `findOffset`. Therefore, the running time is $O(\log n)$.

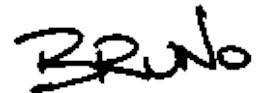
The `findPosition` method also takes an object, but it returns a `Cursor` instead. `findPosition` determines the position in the array of an object which matches its second argument.

The implementation of `findPosition` is trivial: It calls `findOffset` to determine the position at which the matching object is found and returns an instance of the nested class `Cursor`. (The `Cursor` class is derived from the class of the same name shown in Program □). The `Cursor` overrides the inherited `insertAfter` and `insertBefore` methods with methods that raise a `TypeError` exception. These insert operations are not provided for sorted lists because they allow arbitrary insertion, but arbitrary insertions do not necessarily result in sorted lists.

The total running time of the `findPosition` method is dominated by `findOffset`. Therefore like `find`, the running time of `findPosition` is $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Removing Items from a List

The purpose of the `withdraw` method is to remove an item from the sorted list. Program □ defines the `withdraw` method which takes an object and removes it from the sorted list.

```
1  class SortedListAsArray(OrderedListAsArray, SortedList):
2
3      def withdraw(self, obj):
4          if self._count == 0:
5              raise ContainerEmpty
6          offset = self.findOffset(obj)
7          if offset < 0:
8              raise KeyError
9          i = offset
10         while i < self._count:
11             self._array[i] = self._array[i + 1]
12             i += 1
13         self._array[i] = None
14         self._count -= 1
15
16     # ...
```

Program: `SortedListAsArray` class `withdraw` method.

The `withdraw` method makes use of `findOffset` to determine the array index of the item to be removed. Removing an object from position i of an ordered list which is stored in an array requires that all of the objects at positions $i+1, i+2, \dots, \text{count} - 1$, be moved one position to the left. The worst case is when $i=0$. In this case, $\text{count} - 1$ items need to be moved to the left.

Although the `withdraw` method is able to make use of `findOffset` to locate the position of the item to be removed in $O(\log n)$ time, the total running time is dominated by the left shift, which is $O(n)$ in the worst case. Therefore, the running time of `withdraw` is $O(n)$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Linked-List Implementation

This section presents a linked-list implementation of sorted lists that is derived from the `OrderedListAsLinkedList` class given in Section □. The `SortedListAsLinkedList` class is introduced in Program □. The `SortedListAsLinkedList` extends the `OrderedListAsLinkedList` class introduced in Program □ and it also extends the abstract `SortedList` class defined in Program □.

```
1  class SortedListAsLinkedList(
2      OrderedListAsLinkedList, SortedList):
3
4      def __init__(self):
5          super(SortedListAsLinkedList, self).__init__()
6
7      # ...
```

Program: `SortedListAsLinkedList` class.

There are no additional instance attributes defined in the `SortedListAsLinkedList` class. The inherited instance attributes are sufficient to implement a sorted list. In fact, the functionality inherited from the `ListAsLinkedList` class is almost sufficient--the only method of which the functionality must change is the `insert` operation.

-
- [Inserting Items in a Sorted List](#)
 - [Other Operations on Sorted Lists](#)
-



Inserting Items in a Sorted List

Program □ gives the implementation of the `insert` method of the `SortedListAsLinkedList` class. In addition to `self`, this method takes a single argument: the object to be inserted into the sorted list. The algorithm used for the insertion is as follows: First, the existing sorted, linked list is traversed in order to find the linked list element which is greater than or equal to the object to be inserted into the list. The traversal is done using two variables--`prevPtr` and `ptr`. During the traversal, the latter keeps track of the current element and the former keeps track of the previous element.

By keeping track of the previous element, it is possible to efficiently insert the new item into the sorted list by calling the `insertAfter` method of the `LinkedList` class. In Chapter □, the `InsertAfter` method was shown to be $O(1)$.

In the event that the item to be inserted is smaller than the first item in the sorted list, then rather than using the `insertAfter` method, the `prepend` method is used. The `Prepend` method was also shown to be $O(1)$.

In the worst case, the object to be inserted into the linked list is larger than all of the objects already present in the list. In this case, the entire list needs to be traversed before doing the insertion. Consequently, the total running time for the `insert` operation of the `SortedListAsLinkedList` class is $O(n)$, where $n = \text{count}$.

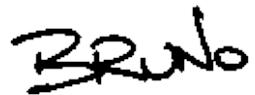
.

```
1 class SortedListAsLinkedList(
2     OrderedListAsLinkedList, SortedList):
3
4     def insert(self, obj):
5         prevPtr = None
6         ptr = self._linkedList.head
7         while ptr is not None:
8             if ptr.datum >= obj:
9                 break
10            prevPtr = ptr
11            ptr = ptr.next
12        if prevPtr is None:
13            self._linkedList.prepend(obj)
14        else:
15            prevPtr.insertAfter(obj)
16        self._count += 1
17
18    # ...
```

Program: SortedListAsLinkedList class insert method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Other Operations on Sorted Lists

Unfortunately, it is not possible to do a binary search in a linked list. As a result, it is not possible to exploit the sortedness of the list in the implementation of any of the other required operations on sorted lists. The methods inherited from the `OrderedListAsLinkedList` provide all of the needed functionality.

Performance Comparison: `SortedListAsArray` vs. `SortedListAsList`

The running times calculated for the various methods of the two sorted list implementations, `SortedListAsArray` and `SortedListAsLinkedList`, are summarized below in Table □. With the exception of two methods, the running times of the two implementations are asymptotically identical.

Table: Running times of operations on sorted lists.

method	sorted list implementation	
	SortedList- AsArray	SortedList- AsLinkedList
<code>__contains__</code>	$O(n)$	$O(n)$
<code>find</code>	$O(\log n)$	\neq $O(n)$
<code>findPosition</code>	$O(\log n)$	\neq $O(n)$
<code>__getitem__</code>	$O(1)$	\neq $O(n)$
<code>insert</code>	$O(n)$	$O(n)$
<code>withdraw</code>	$O(n)$	$O(n)$
<code>Cursor.datum</code>	$O(1)$	$O(1)$
<code>Cursor.withdraw</code>	$O(n)$	$O(n)$

Neither the `SortedListAsArray` nor `SortedListAsLinkedList` implementations required any additional instance attributes beyond those inherited from their respective base classes, `OrderedListAsArray` and `OrderedListAsLinkedList`. Consequently, the space requirements analysis of the sorted list implementations is identical to that of the ordered list implementations given in Section □.

Bruno

Applications

In Section □ we saw that an n^{th} -order polynomial,

$$\sum_{i=0}^n a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where $a_n \neq 0$, can be represented by a sequence of ordered pairs thus:

$$\{(a_0, 0), (a_1, 1), (a_2, 2), \dots, (a_n, n)\}.$$

We also saw that it is possible to make use of an *ordered list* to represent such a sequence and that given such a representation, we can write an algorithm to perform differentiation.

As it turns out, the order of the terms in the sequence does not affect the differentiation algorithm. The correct result is always obtained and the running time is unaffected regardless of the order of the terms in the sequence.

Unfortunately, there are operations on polynomials whose running time depends on the order of the terms. For example, consider the addition of two polynomials:

$$(a_0 + a_1 x + a_2 x^2) + (b_3 x^3 + b_2 x^2 + b_1 x) = \\ (a_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + (b_3)x^3$$

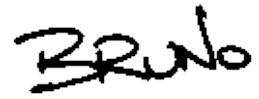
To perform the addition all the terms involving x raised to the same power need to be grouped together.

If the terms of the polynomials are in an arbitrary order, then the grouping together of the corresponding terms is time consuming. On the other hand, if the terms are ordered, say, from smallest exponent to largest, then the summation can be done rather more efficiently. A single pass through the polynomials will suffice. It makes sense to represent each of the polynomials as a *sorted list* of terms using, say, the `SortedListAsLinkedList` class.

- [Implementation](#)
 - [Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

To begin with, we need to represent the terms of the polynomial. Program □ extends the definition of the `Term` class introduced in Program □--some additions are needed to support the the implementation of polynomial addition.

```
 1  class Polynomial(Container):
 2
 3      class Term(Object):
 4
 5          def __copy__(self):
 6              return Polynomial.Term(
 7                  self._coefficient, self._exponent)
 8
 9          def getCoefficient(self):
10              return self._coefficient
11
12          coefficient = property(
13              fget = lambda self: self.getCoefficient())
14
15          def getExponent(self):
16              return self._exponent
17
18          exponent = property(
19              fget = lambda self: self.getExponent())
20
21          def __add__(self, term):
22              assert self._exponent == term._exponent
23              return Polynomial.Term(
24                  self._coefficient + term._coefficient,
25                  self._exponent)
26
27          # ...
28
29          # ...
```

Program: `Term` methods.

A number of additional operations are declared in Program □. The first method, `__copy__`, creates a copy of a given term.

Two properties, `coefficient` and `exponent`, use the methods `getCoefficient` and `getExponent` (respectively) to return the corresponding instance attributes of a `Term` instance. Clearly, the running time of each of these operations is $O(1)$.

The final method, `__add__`, provides the means to add two `Terms` together. The result of the addition is another `Term`. The working assumption is that the terms to be added have identical exponents. If the exponents are allowed to differ, the result of the addition is a polynomial which cannot be represented using a single term! To add terms with like exponents, we simply need to add their respective coefficients. Therefore, the running time of the `Term` addition operator is $O(1)$.

We now turn to the polynomial itself. Program □ introduces the `PolynomialAsSortedList` class. This class extends the abstract `Polynomial` class defined in Program □. It has a single instance attribute of type `SortedListAsLinkedList`. We have chosen in this implementation to use the linked-list sorted list implementation to represent the sequence of terms.

```
1 class PolynomialAsSortedList(Polynomial):
2
3     def __init__(self):
4         super(PolynomialAsSortedList, self).__init__()
5         self._list = SortedListAsLinkedList()
6
7     def nextTerm(self, iter):
8         try:
9             return iter.next()
10        except StopIteration:
11            return None
12
13     def __add__(self, poly):
14         result = PolynomialAsSortedList()
15         p1 = iter(self._list)
16         p2 = iter(poly._list)
17         term1 = self.nextTerm(p1)
18         term2 = self.nextTerm(p2);
19         while term1 is not None and term2 is not None:
20             if term1.exponent < term2.exponent:
21                 result.addTerm(copy(term1))
22                 term1 = self.nextTerm(p1)
23             elif term1.exponent > term2.exponent:
24                 result.addTerm(copy(term2))
25                 term2 = self.nextTerm(p2)
26             else:
27                 sum = term1 + term2
28                 if sum.coefficient != 0:
29                     result.addTerm(sum)
30                 term1 = self.nextTerm(p1)
31                 term2 = self.nextTerm(p2)
32         while term1 is not None:
33             result.addTerm(copy(term1))
34             term1 = self.nextTerm(p1)
35         while term2 is not None:
36             result.addTerm(copy(term2))
37             term2 = self.nextTerm(p2)
38         return result
39
40     # ...
```

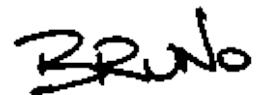
Program: PolynomialAsSortedList class `__init__` and `__add__` methods.

Program □ defines the `__add__` method. This method adds two Polynomials to obtain a third. It is intended to be used like this:

```
p1 = PolynomialAsSortedList()
p2 = PolynomialAsSortedList()
# ...
p3 = p1 + p2
```

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Analysis

The proof of the correctness of Program □ is left as an exercise for the reader (Exercise □). We discuss here the running time analysis of the algorithm, as there are some subtle points to remember which lead to a result that may be surprising.

Consider the addition of a polynomial $p(x)$ with its arithmetic complement $-p(x)$. Suppose $p(x)$ has n terms. Clearly $-p(x)$ also has n terms. The sum of the polynomials is the zero polynomial. An important characteristic of the zero polynomial is that it *has no terms!* In this case, exactly n iterations of the main loop are done (lines 19-31). Furthermore, zero iterations of the second and the third loops are required (lines 32-34 and 35-37). Since the result has no terms, there will be no calls to the `addTerm` method. Therefore, the amount of work done in each iteration is a constant. Consequently, the best case running time is $O(n)$.

Consider now the addition of two polynomials, $p(x)$ and $q(x)$, having l and m terms, respectively. Furthermore, suppose that largest exponent in $p(x)$ is less than the smallest exponent in $q(x)$. Consequently, there is no power of x which the two polynomials have in common. In this case, since $p(x)$ has the lower-order terms, exactly l iterations of the main loop (lines 19-31) are done. In each of these iterations, exactly one new term is inserted into the result by calling the `addTerm` method. Since all of the terms of $p(x)$ will be exhausted when the main loop is finished, there will be no iterations of the second loop (lines 32-34). However, there will be exactly m iterations of the third loop (lines 35-37) in each of which one new term is inserted into the result by calling the `addTerm` method.

Altogether, $l+m$ calls to the `addTerm` will be made. It was shown earlier that the running time for the `insert` method is $O(k)$, where k is the number of items in the sorted list. Consequently, the total running time for the $l+m$ insertions is

$$\sum_{k=0}^{l+m-1} O(k) = O((l + m)^2).$$

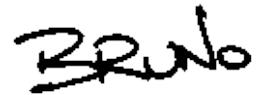
Consequently, the worst case running time for the polynomial addition given in Program \square is $O(n^2)$, where $n=l+m$. This is somewhat disappointing. The implementation is not optimal because it fails to take account of the order in which the terms of the result are computed. That is, the `addTerm` method repeatedly searches the sorted list for the correct position at which to insert the next term. But we know that correct position is at the end! By replacing in Program \square all of the calls to the `addTerm` method by

```
self._list._linkedList.append(...)
```

the total running time can be reduced to $O(n)$ from $O(n^2)$!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exercises

1. Devise an algorithm to reverse the contents of an ordered list. Determine the running time of your algorithm.
2. Devise an algorithm to append the contents of one ordered list to the end of another. Assume that both lists are represented using arrays. What is the running time of your algorithm?
3. Repeat Exercise □, but this time assume that both lists are represented using linked lists. What is the running time of your algorithm?
4. Devise an algorithm to merge the contents of two sorted lists. Assume that both lists are represented using arrays. What is the running time of your algorithm?
5. Repeat Exercise □, but this time assume that both lists are represented using linked lists. What is the running time of your algorithm?
6. The `withdraw` method can be used to remove items from a list one at a time. Suppose we want to provide an additional a method, `withdrawAll`, that takes one argument and withdraws all the items in a list that *match* the given argument. We can provide an implementation of the `withdrawAll` method in the `SearchableContainer` class like this:

```
class SearchableContainer(Container):  
  
    def withdrawAll(self, obj):  
        while True:  
            match = self.find(obj)  
            if match is None:  
                break  
            self.withdraw(match)
```

...

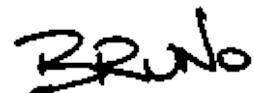
Determine the worst-case running time of this method for each of the following cases:

1. an array-based implementation of an ordered list,
 2. a linked-list implementation of an ordered list,
 3. an array-based implementation of a sorted list, and
 4. a linked-list implementation of a sorted list.
7. Devise an $O(n)$ algorithm, to remove from an ordered list all the items that match a given item. Assume the list is represented using an array.

8. Repeat Exercise □, but this time assume the ordered list is represented using a linked list.
9. Consider an implementation of the `OrderedList` interface that uses a doubly-linked list such as the one shown in Figure □ (a). Compare the running times of the operations for this implementation with those given in Table □.
10. Derive an expression for the amount of space used to represent an ordered list of n elements using a doubly-linked list such as the one shown in Figure □ (a). Compare this with the space used by the array-based implementation. Assume that integers and pointers each occupy four bytes.
11. Consider an implementation of the `SortedList` interface that uses a doubly-linked list such as the one shown in Figure □ (a). Compare the running times of the operations for this implementation with those given in Table □.
12. Verify that Program □ correctly computes the sum of two polynomials.
13. Write an algorithm to multiply a polynomial by a scalar. **Hint:** Use a visitor.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Projects

1. Write a visitor to solve each of the following problems:
 1. Find the smallest element of a list.
 2. Find the largest element of a list.
 3. Compute the sum of all the elements of a list.
 4. Compute the product of all the elements of a list.
2. Design and implement a class called `orderedListAsDoublyLinkedList` which represents an ordered list using a doubly-linked list. Select one of the approaches shown in Figure □.
3. Consider the `Polynomial` class given in Program □. Implement a method that computes the value of a polynomial, say $p(x)$, for a given value of x .
Hint: Use a visitor that visits all the terms in the polynomial and accumulates the result.
4. Devise and implement an algorithm to multiply two polynomials. **Hint:** Consider the identity

$$\left(\sum_{i=0}^n a_i x^i \right) \times \left(\sum_{j=0}^m b_j x^j \right) = \sum_{i=0}^n a_i x^i \left(\sum_{j=0}^m b_j x^j \right).$$

Write a method to multiply a `Polynomial` by a `Term` and use the polynomial addition operator defined in Program □.

5. Devise and implement an algorithm to compute the k^{th} power of a polynomial, where k is a positive integer. What is the running time of your algorithm?
6. For some calculations it is necessary to have very large integers, i.e., integers with an arbitrarily large number of digits. We can represent such integers using lists. Design and implement a class for representing arbitrarily large integers. Your implementation should include operations to add, subtract, and multiply such integers, and to compute the k^{th} power of such an integer, where k is a *small* positive integer. **Hint:** Base your design on the `Polynomial` class given in Program □.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Hashing, Hash Tables, and Scatter Tables

A very common paradigm in data processing involves storing information in a table and then later retrieving the information stored there. For example, consider a database of driver's license records. The database contains one record for each driver's license issued. Given a driver's license number, we can look up the information associated with that number.

Similar operations are done by the Python compiler. The compiler uses a *symbol table* to keep track of the user-defined symbols in a Python program. As it compiles a program, the compiler inserts an entry in the symbol table every time a new symbol is declared. In addition, every time a symbol is used, the compiler looks up the attributes associated with that symbol to see that it is being used correctly and to guide the generation of the Python byte code.

Typically the database comprises a collection of key-and-value pairs. Information is retrieved from the database by searching for a given key. In the case of the driver's license database, the key is the driver's license number and in the case of the symbol table, the key is the name of the symbol.

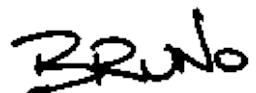
In general, an application may perform a large number of insertion and/or look-up operations. Occasionally it is also necessary to remove items from the database. Because a large number of operations will be done we want to do them as quickly as possible.

-
- [Hashing-The Basic Idea](#)
 - [Hashing Methods](#)
 - [Hash Function Implementations](#)
 - [Hash Tables](#)
 - [Scatter Tables](#)

- [Scatter Table using Open Addressing](#)
 - [Applications](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Hashing-The Basic Idea

In this chapter we examine data structures which are designed specifically with the objective of providing efficient insertion and find operations. In order to meet the design objective, certain concessions are made. Specifically, we do not require that there be any specific ordering of the items in the container. In addition, while we still require the ability to remove items from the container, it is not our primary objective to make removal as efficient as the insertion and find operations.

Ideally we would build a data structure for which both the insertion and find operations are $O(1)$ in the worst case. However, this kind of performance can only be achieved with complete *a priori* knowledge. We need to know beforehand specifically which items are to be inserted into the container. Unfortunately, we do not have this information in the general case. So, if we cannot guarantee $O(1)$ performance in the *worst case*, then we make it our design objective to achieve $O(1)$ performance in the *average case*.

The constant time performance objective immediately leads us to the following conclusion: Our implementation must be based in some way on an array rather than a linked list. This is because we can access the k^{th} element of an array in constant time, whereas the same operation in a linked list takes $O(k)$ time.

In the previous chapter, we consider two searchable containers--the *ordered list* and the *sorted list*. In the case of an ordered list, the cost of an insertion is $O(1)$ and the cost of the find operation is $O(n)$. For a sorted list the cost of insertion is $O(n)$ and the cost of the find operation is $O(\log n)$ for the array implementation.

Clearly, neither the ordered list nor the sorted list meets our performance objectives. The essential problem is that a search, either linear or binary, is always necessary. In the ordered list, the find operation uses a linear search to locate the item. In the sorted list, a binary search can be used to locate the item because the data is sorted. However, in order to keep the data sorted, insertion becomes $O(n)$.

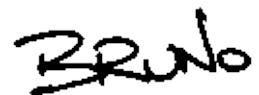
In order to meet the performance objective of constant time insert and find operations, we need a way to do them *without performing a search*. That is,

given an item x , we need to be able to determine directly from x the array position where it is to be stored.

- [Example](#)
 - [Keys and Hash Functions](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example

We wish to implement a searchable container which will be used to contain character strings from the set of strings K ,

$$K = \{"ett", "två"\sup{1}, "tre", "fyra", "fem", "sex"\sup{2}, \\ "sju", "åtta", "nio", "tio", "elva", "tolv"\}.$$

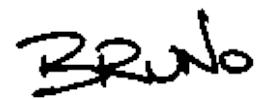
Suppose we define a function $h : K \rightarrow \mathbb{Z}$ as given by the following table:

x	$h(x)$
"ett"	1
"två"	2
"tre"	3
"fyra"	4
"fem"	5
"sex"	6
"sju"	7
"åtta"	8
"nio"	9
"tio"	10
"elva"	11
"tolv"	12

Then, we can implement a searchable container using an array of length $n=12$. To insert item x , we simply store it at position $h(x)-1$ of the array. Similarly, to locate item x , we simply check to see if it is found at position $h(x)-1$. If the function $h(\cdot)$ can be evaluated in constant time, then both the insert and the find operations are $O(1)$.

We expect that any reasonable implementation of the function $h(\cdot)$ will run in constant time, since the size of the set of strings, K , is a constant! This example illustrates how we can achieve $O(1)$ performance in the worst case when we have complete, *a priori* knowledge.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with "Bruno" written in a larger, more prominent style than the "R." preceding it.

Keys and Hash Functions

We are designing a container which will be used to hold some number of items of a given set K . In this context, we call the elements of the set K *keys*. The general approach is to store the keys in an array. The position of a key in the array is given by a function $h(\cdot)$, called a *hash function*, which determines the position of a given key directly from that key.

In the general case, we expect the size of the set of keys, $|K|$, to be relatively large or even unbounded. For example, if the keys are 32-bit integers, then $|K| = 2^{32}$. Similarly, if the keys are arbitrary character strings of arbitrary length, then $|K|$ is unbounded.

On the other hand, we also expect that the actual number of items stored in the container to be significantly less than $|K|$. That is, if n is the number of items actually stored in the container, then $n \ll |K|$. Therefore, it seems prudent to use an array of size M , where M is as least as great as the maximum number of items to be stored in the container.

Consequently, what we need is a function $h : K \mapsto \{0, 1, \dots, M - 1\}$. This function maps the set of values to be stored in the container to subscripts in an array of length M . This function is called a *hash function*.

In general, since $|K| \geq M$, the mapping defined by hash function will be a *many-to-one mapping*. That is, there will exist many pairs of distinct keys x and y , such that $x \neq y$, for which $h(x) = h(y)$. This situation is called a *collision*. Several approaches for dealing with collisions are explored in the following sections.

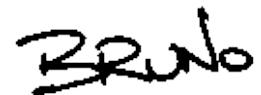
What are the characteristics of a good hash function?

- A good hash function avoids collisions.
 - A good hash function tends to spread keys evenly in the array.
 - A good hash function is easy to compute.
-

- [Avoiding Collisions](#)
 - [Spreading Keys Evenly](#)
 - [Ease of Computation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Avoiding Collisions

Ideally, given a set of $n \leq M$ distinct keys, $\{k_1, k_2, \dots, k_n\}$, the set of hash values $\{h(k_1), h(k_2), \dots, h(k_n)\}$ contains no duplicates. In practice, unless we know something about the keys chosen, we cannot guarantee that there will not be collisions. However, in certain applications we have some specific knowledge about the keys that we can exploit to reduce the likelihood of a collision. For example, if the keys in our application are telephone numbers, and we know that the telephone numbers are all likely to be from the same geographic area, then it makes little sense to consider the area codes in the hash function, the area codes are all likely to be the same.



Spreading Keys Evenly

Let p_i be the probability that the hash function $h(\cdot) = i$. A hash function which spreads keys evenly has the property that for $0 \leq i < M$, $p_i = 1/M$. In other words, the hash values computed by the function $h(\cdot)$ are *uniformly distributed*. Unfortunately, in order to say something about the distribution of the hash values, we need to know something about the distribution of the keys.

In the absence of any information to the contrary, we assume that the keys are equiprobable. Let K_i be the set of keys that map to the value i . That is, $K_i = \{k \in K : h(k) = i\}$. If this is the case, the requirement to spread the keys uniformly implies that $|K_i| = |K|/M$. An equal number of keys should map into each array position.

Ease of Computation

This does not mean necessarily that it is easy for someone to compute the hash function, nor does it mean that it is easy to write the algorithm to compute the function; it means that the running time of the hash function should be $O(1)$.

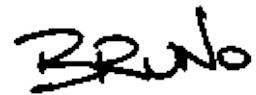
Hashing Methods

In this section we discuss several hashing methods. In the following discussion, we assume that we are dealing with integer-valued keys, i.e., $K = \mathbb{Z}$. Furthermore, we assume that the value of the hash function falls between 0 and $M-1$.

- [Division Method](#)
 - [Middle Square Method](#)
 - [Multiplication Method](#)
 - [Fibonacci Hashing](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Division Method

Perhaps the simplest of all the methods of hashing an integer x is to divide x by M and then to use the remainder modulo M . This is called the *division method of hashing* . In this case, the hash function is

$$h(x) = |x| \bmod M.$$

Generally, this approach is quite good for just about any value of M . However, in certain situations some extra care is needed in the selection of a suitable value for M . For example, it is often convenient to make M an even number. But this means that $h(x)$ is even if x is even; and $h(x)$ is odd if x is odd. If all possible keys are equiprobable, then this is not a problem. However if, say, even keys are more likely than odd keys, the function $h(x) = x \bmod M$ will not spread the hashed values of those keys evenly.

Similarly, it is often tempting to let M be a power of two. For example, $M = 2^k$ for some integer $k > 1$. In this case, the hash function $h(x) = x \bmod 2^k$ simply extracts the bottom k bits of the binary representation of x . While this hash function is quite easy to compute, it is not a desirable function because it does not depend on all the bits in the binary representation of x .

For these reasons M is often chosen to be a prime number. For example, suppose there is a bias in the way the keys are created that makes it more likely for a key to be a multiple of some small constant, say two or three. Then making M a prime increases the likelihood that those keys are spread out evenly. Also, if M is a prime number, the division of x by that prime number depends on all the bits of x , not just the bottom k bits, for some small constant k .

The division method is extremely simple to implement. The following Python code illustrates how to do it:

```
M = 1021 # a prime

def h(x):
    return abs(x) % M
```

In this case, it appears that M has a constant value. However, an advantage of the division method is that the value of M need not be static--its value can be determined at run time. In any event, the running time of this implementation is clearly a constant.

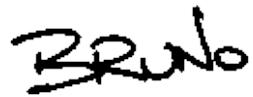
A potential disadvantage of the division method is due to the property that consecutive keys map to consecutive hash values:

$$\begin{aligned} h(i) &= i \\ h(i+1) &= i+1 \pmod{M} \\ h(i+2) &= i+2 \pmod{M} \\ &\vdots \end{aligned}$$

While this ensures that consecutive keys do not collide, it does mean that consecutive array locations will be occupied. We will see that in certain implementations this can lead to degradation in performance. In the following sections we consider hashing methods that tend to scatter consecutive keys.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Middle Square Method

In this section we consider a hashing method which avoids the use of division. Since integer division is usually slower than integer multiplication, by avoiding division we can potentially improve the running time of the hashing algorithm. We can avoid division by doing the arithmetic by using shifts and bitwise operations instead.

The *middle-square hashing method* works as follows. First, we assume that M is a power of two, say $M = 2^k$ for some $k \geq 1$. Then, to hash an integer x , we use the following hash function:

$$h(x) = \left\lfloor \frac{M}{W} (x^2 \bmod W) \right\rfloor.$$

For W we use the *word size* of the machine. That is $W = 2^w$.

Notice that since M and W are both powers of two, the ratio $\frac{W/M}{(x^2 \bmod W)} = 2^{w-k}$ is also a power two. Therefore, in order to multiply the term $(x^2 \bmod W)$ by M/W we simply shift it to the right by $w-k$ bits! In effect, we are extracting k bits from the middle of the square of the key--hence the name of the method.

The following code fragment illustrates the middle-square method of hashing:

```
k = 10 # M==1024
w = 32
mask = (1 << k) - 1

def h(x):
    return ((x * x) >> (w - k)) & mask
```

We assume that x is an `int`. In Python, an `int` represents a 32-bit quantity. The product of two `ints` is at most a 64-bit quantity. The final result is obtained by shifting the product $w-k$ bits to the right, where w is the number of bits in an integer and then masking to obtain the bottom k bits. Note, the final result is an integer between 0 and $M-1$.

The middle-square method does a pretty good job when the integer-valued keys are equiprobable. The middle-square method also has the characteristic that it

scatters consecutive keys nicely. However, since the middle-square method only considers a subset of the bits in the middle of x^2 , keys which have a large number of leading zeroes will collide. For example, consider the following set of keys:

$$\{x \in \mathbb{Z} : |x| < \sqrt{W/M}\}.$$

This set contains all keys x such that $|x| < 2^{(w-k)/2}$. For all of these keys $h(x)=0$.

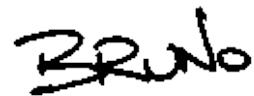
A similar line of reasoning applies for keys which have a large number of trailing zeroes. Let W be an even power of two. Consider the set of keys

$$\{x \in \mathbb{Z} : x = \pm n\sqrt{W}, \quad n \in \mathbb{Z}\}.$$

The least significant $w/2$ bits of the keys in this set are all zero. Therefore, the least significant w bits of x^2 are also zero and as a result $h(x)=0$ for all such keys!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Multiplication Method

A very simple variation on the middle-square method that alleviates its deficiencies is the so-called, *multiplication hashing method*. Instead of multiplying the key x by itself, we multiply the key by a carefully chosen constant a , and then extract the middle k bits from the result. In this case, the hashing function is

$$h(x) = \left\lfloor \frac{M}{W} (ax \bmod W) \right\rfloor.$$

What is a suitable choice for the constant a ? If we want to avoid the problems that the middle-square method encounters with keys having a large number of leading or trailing zeroes, then we should choose an a that has neither leading nor trailing zeroes.

Furthermore, if we choose an a that is *relatively prime* to W , then there exists another number a' such that $aa' \equiv 1 \pmod{W}$. In other words, a' is the *inverse* of a modulo W , since the product of a and its inverse is one. Such a number has the nice property that if we take a key x , and multiply it by a to get ax , we can recover the original key by multiplying the product again by a' , since $axa'=aa'x=1x$.

There are many possible constants which have the desired properties. One possibility which is suited for 32-bit arithmetic (i.e., $W = 2^32$) is $a = 2654435769$. The binary representation of a is

10 011 110 001 101 110 111 100 110 111 001.

This number has neither many leading nor trailing zeroes. Also, this value of a and $W = 2^32$ are relatively prime and the inverse of a modulo W is $a' = 340573321$.

The following code fragment illustrates the multiplication method of hashing:

```
k = 10 # M==1024
w = 32
mask = (1 << k) - 1
a = 2654435769L

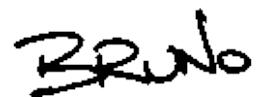
def h(x):
```

```
return ((x * a) >> (w - k)) & mask
```

The code is a simple modification of the middle-square version. Nevertheless, the running time remains $O(1)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Fibonacci Hashing

The final variation of hashing to be considered here is called the *Fibonacci hashing method*. In fact, Fibonacci hashing is exactly the multiplication hashing method discussed in the preceding section using a very special value for a . The value we choose is closely related to the number called the golden ratio.

The *golden ratio* is defined as follows: Given two positive numbers x and y , the ratio $\phi = x/y$ is the golden ratio if the ratio of x to y is the same as that of $x+y$ to x . The value of the golden ratio can be determined as follows:

$$\begin{aligned}\frac{x}{y} = \frac{x+y}{x} &\Rightarrow 0 = x^2 - xy - y^2 \\ &\Rightarrow 0 = \phi^2 - \phi - 1 \\ &\Rightarrow \phi = \frac{1 + \sqrt{5}}{2}\end{aligned}$$

There is an intimate relationship between the golden ratio and the Fibonacci numbers. In Section □ it was shown that the n^{th} Fibonacci number is given by

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n),$$

where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$!

The Fibonacci hashing method is essentially the multiplication hashing method in which the constant a is chosen as the integer that is relatively prime to W which is closest to W/ϕ . The following table gives suitable values of a for various word sizes.

W	$a \approx W/\phi$
2^{16}	40503
2^{32}	2654435769
2^{64}	11400714819323198485

Why is W/ϕ special? It has to do with what happens to consecutive keys when they are hashed using the multiplicative method. As shown in Figure □,

consecutive keys are spread out quite nicely. In fact, when we use $a \approx W/\phi$ to hash consecutive keys, the hash value for each subsequent key falls in between the two widest spaced hash values already computed. Furthermore, it is a property of the golden ratio, ϕ , that each subsequent hash value divides the interval into which it falls according to the golden ratio!

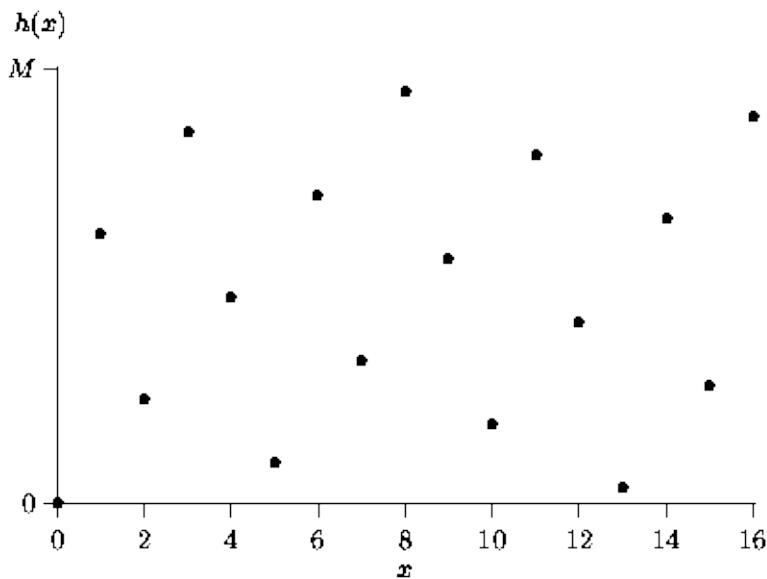


Figure: Fibonacci hashing.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Hash Function Implementations

The preceding section presents methods of hashing integer-valued keys. In reality, we cannot expect that the keys will always be integers. Depending on the application, the keys might be letters, character strings or even more complex data structures such as Associations or Containers.

In general given a set of keys, K , and a positive constant, M , a hash function is a function of the form

$$h : K \mapsto \{0, 1, \dots, M - 1\}.$$

In practice is it convenient to implement the hash function h as the composition of two functions f and g . The function f maps keys into integers:

$$f : K \mapsto \mathbb{Z},$$

where \mathbb{Z} is the set of integers. The function g maps non-negative integers into $\{0, 1, \dots, M - 1\}$:

$$g : \mathbb{Z} \mapsto \{0, 1, \dots, M - 1\}.$$

Given appropriate functions f and g , the hash function h is simply defined as the composition of those functions:

$$h = g \circ f$$

That is, the hash value of a key x is given by $g(f(x))$.

By decomposing the function h in this way, we can separate the problem into two parts: The first involves finding a suitable mapping from the set of keys K to the non-negative integers. The second involves mapping non-negative integers into the interval $[0, M-1]$. Ideally, the two problems would be unrelated. That is, the choice of the function f would not depend on the choice of g and *vice versa*. Unfortunately, this is not always the case. However, if we are careful, we can design the functions in such a way that $h = g \circ f$ is a good hash function.

The hashing methods discussed in the preceding section deal with integer-valued keys. But this is precisely the domain of the function g . Consequently, we have already examined several different alternatives for the function g . On the other hand, the choice of a suitable function for f depends on the characteristics of its domain.

In the following sections, we consider various different domains (sets of keys) and develop suitable hash functions for each of them. Each domain considered is implemented as a Python class that provides a method called `__hash__`. The `__hash__` method corresponds to the function f which maps keys into integers.

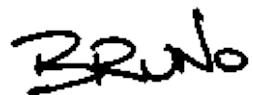
The Python built-in `hash` function automatically calls the `__hash__` method of a class that provides one. Thus, if `obj` is an instance of a class that provides a `__hash__` method, the following Python statements are equivalent:

```
hash(obj)  
obj.__hash__()
```

- [Integer Keys](#)
 - [Floating-Point Keys](#)
 - [Character String Keys](#)
 - [Hashing Containers](#)
 - [Using Associations](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Integer Keys

Of all the Python types, integers (`ints` and `longs`) are the simplest to hash. A suitable function f is:

$$f(x) = x \bmod 2^{31}.$$

Program □ introduces the class `Integer` which extends the built-in `int` class as well as the abstract `Object` class introduced in Program □. In this case, the `__hash__` method simply returns the bitwise and of `self` and `sys.maxint`.

Because `sys.maxint` has the value $2^{31} - 1$, this has the effect of discards all but the least significant 31 bits, that is, division module 2^{31} .

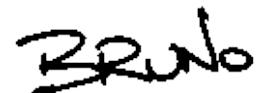
The running time of the `__hash__` method is $O(1)$ if the integer is a plain integer (a 32-bit integer). If the integer is a long integer, the running time of the `__hash__` method depends on the number of digits in that long. If the long value is n , the running time is $O(\log n)$.

```
1  class Integer(int, Object):
2
3      def __hash__(self):
4          return self & sys.maxint
5
6      # ...
```

Program: `Integer` class `__hash__` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Floating-Point Keys

Dealing with floating-point number involves a little more work. In Python the floating-point numbers are represented using the IEEE double-precision format (64 bits). We seek a function f which maps a floating-point value into a non-negative integer. A characteristic of floating-point numbers that must be dealt with is the extremely wide range of values which can be represented. For example, when using IEEE floating-point, the smallest double precision quantity that can be represented is 5×10^{-324} and the largest is $\approx 1.80 \times 10^{308}$. Somehow we need to map values in this large domain into the range of an `int`.

Every non-zero floating-point quantity x can be written uniquely as

$$x = (-1)^s \times m \times 2^e,$$

where $s \in \{0, 1\}$, $0.5 \leq m < 1$ and $-1023 \leq e \leq 1024$. The quantity s is called the *sign* , m is called the *mantissa* or *significant* and e is called the *exponent* . This suggests the following definition for the function f :

$$f(x) = \begin{cases} 0 & x = 0, \\ \lfloor 2(m - \frac{1}{2})W \rfloor & x \neq 0, \end{cases} \quad (8.1)$$

where $W = 2^{31}$.

This hashing method is best understood by considering the conditions under which a collision occurs between two distinct floating-point numbers x and y . Let m_x and m_y be the mantissas of x and y , respectively. The collision occurs when $f(x)=f(y)$.

$$\begin{aligned} f(x) = f(y) &\Rightarrow f(x) - f(y) = 0 \\ &\Rightarrow \lfloor 2(2m_x - \frac{1}{2})W \rfloor - \lfloor 2(m_y - \frac{1}{2})W \rfloor = 0 \\ &\Rightarrow |2(m_x - \frac{1}{2})W - 2(m_y - \frac{1}{2})W| \leq 1 \\ &\Rightarrow |m_x - m_y| \leq \frac{1}{2W} \end{aligned}$$

Thus, x and y collide if their mantissas differ by less than $1/2W$. Notice that the sign of the number is not considered. Thus, x and $-x$ collide Also, the exponent is

not considered. Therefore, if x and y collide, then so too do x and $y \times 2^k$ for all permissible values of k .

Program □ introduces the class `Float` which extends the built-in `float` class as well as the abstract `Object` class introduced in Program □. In this case, the `__hash__` method computes the hash function defined in Equation □.

```

1  class Float(float, Object):
2
3      def __hash__(self):
4          (m, e) = math.frexp(self)
5          mprime = int((abs(m) - 0.5) * (1L << 52))
6          return mprime >> 20
7
8      # ...

```

Program: `Float` class `__hash__` method.

This implementation uses the `frexp` function provided in the standard Python `math` module. This function returns the mantissa, m , and the exponent, e , of a given floating-point number. In the IEEE standard floating-point format the least-significant 52 bits of a 64-bit floating-point number represent the quantity $m' = (m - \frac{1}{2}) \times 2^{53}$. Since $W = 2^{31}$, we can rewrite Equation □ as follows:

$$\begin{aligned} f(n) &= 2(m - \frac{1}{2})W \\ &= 2^{53}(m - \frac{1}{2})/2^{20} \\ &= m'/2^{20}. \end{aligned}$$

Thus, we can compute the hash function by computing the integer m' and then shifting the binary representation of that integer 20 bits to the right as shown in Program □. Clearly the running time of the `__hash__` method is $O(1)$.

Character String Keys

Strings of characters are represented in Python as instances of the `str` class. A character string is simply a sequence of characters. Since such a sequence may be arbitrarily long, to devise a suitable hash function we must find a mapping from an unbounded domain into the finite range of a 32-bit integer.

We can view a character string, s , as a sequence of n characters,

$$\{s_0, s_1, \dots, s_{n-1}\},$$

where n is the length of the string. (The length of a string can be determined using the Python built-in method `len`). One very simple way to hash such a string would be to simply sum the numeric values associated with each character:

$$f(s) = \sum_{i=0}^{n-1} s_i \quad (8.2)$$

As it turns out, this is not a particularly good way to hash character strings. Given that the integer value of a Python character is an 8-bit quantity, $0 \leq s_i \leq 2^8 - 1$, for all $0 \leq i < n$. As a result, $0 \leq f(s) < n(2^8 - 1)$. For example, given a string of length $n=5$, the value of $f(s)$ falls between zero and 1 275. In fact, the situation is even worse, in North America we typically use only the *ASCII* character set. The ASCII character set uses only the least-significant seven bits of a `char`. If the string is comprised of only ASCII characters, the result falls in the range between zero and 640.

Essentially the problem with a function f which produces a result in a relatively small interval is the situation which arises when that function is composed with the function $g(x) = x \bmod M$. If the size of the range of the function f is less than M , then $h = g \circ f$ does not spread its values uniformly on the interval $[0, M-1]$. For example, if $M=1031$ only the first 640 values (62% of the range) are used!

Alternatively, suppose we have *a priori* knowledge that character strings are limited to length $n=4$. Then, we can construct an integer by concatenating the binary representations of each of the characters. For example, given

$s = \{s_0, s_1, s_2, s_3\}$, we can construct an integer with the function

$$f(s) = s_0B^3 + s_1B^2 + s_2B + s_0. \quad (8.3)$$

where $B = 2^7$. Since B is a power of two, this function is easy to write in Python:

```
def f(s):
    return ord(s[0]) << 21 | ord(s[1]) \
        << 14 | ord(s[2]) << 7 | ord(s[3])
```

While this function certainly has a larger range, it still has a problem--it cannot deal strings of arbitrary length.

Equation \square can be generalized to deal with strings of arbitrary length as follows:

$$f(s) = \sum_{i=0}^{n-1} B^{n-i-1} s_i$$

This function produces a unique integer for every possible string. Unfortunately, the range of $f(s)$ is unbounded. A simple modification of this algorithm suffices to bound the range:

$$f(s) = \left(\sum_{i=0}^{n-1} B^{n-i-1} s_i \right) \bmod W, \quad (8.4)$$

where $W = 2^w$ such that w is word size of the machine. Unfortunately, since W and B are both powers of two, the value computed by this hash function depends only on the last W/B characters in the character string. For example, for $W = 2^32$ and $B = 2^7$, this result depends only on the last five characters in the string--all character strings having exactly the same last five characters collide!

Writing the code to compute Equation \square is actually quite straightforward if we realize that $f(s)$ can be viewed as a polynomial in B , the coefficients of which are s_0, s_1, \dots, s_n . Therefore, we can use *Horner's rule* (see Section \square) to compute $f(s)$ as follows:

```
def f(s):
    result = 0
    for c in s:
        result = result * B + ord(c)
    return result
```

This implementation can be simplified even further if we make use of the fact that $B = 2^b$, where $b=7$. Since B is a power of two, in order to multiply the variable `result` by B all we need to do is to shift it left by b bits. Furthermore, having just shifted `result` left by b bits, we know that the least significant b bits of the result are zero. And since each character has no more than $b=7$ bits, we can replace the addition operation with an *exclusive or* operation.

```
def f(s):
    result = 0
    for c in s:
        result = result << b ^ ord(c)
    return result
```

Of the 128 characters in the 7-bit ASCII character set, only 97 characters are printing characters including the space, tab, and newline characters (see Appendix □). The remaining characters are control characters which, depending on the application, rarely occur in strings. Furthermore, if we assume that letters and digits are the most common characters in strings, then only 62 of the 128 ASCII codes are used frequently. Notice, the letters (both upper and lower case) all fall between 0101_8 and 0172_8 . All the information is in the least significant six bits. Similarly, the digits fall between 060_8 and 071_8 --these differ in the least significant four bits. These observations suggest that using $B = 2^6$ should work well. That is, for $W = 2^3 \cdot 2$, the hash value depends on the last five characters plus two bits of the sixth-last character.

We have developed a hashing scheme which works quite well given strings which differ in the trailing letters. For example, the strings "temp1", "temp2", and "temp3", all produce different hash values. However, in certain applications the strings differ in the leading letters. For example, the two *Internet domain names* "ece.uwaterloo.ca" and "cs.uwaterloo.ca" collide when using Equation □. Essentially, the effect of the characters that differ is lost because the corresponding bits have been shifted out of the hash value.

```

1 class String(str, Object):
2
3     shift = 6
4
5     mask = ~0 << (31 - shift)
6
7     def __hash__(self):
8         result = 0
9         for c in self:
10             result = ((result & String.mask) ^
11                     result << String.shift ^ ord(c)) & sys.maxint
12         return result
13
14     # ...

```

Program: String class `__hash__` method.

This suggests a final modification which shown in Program □. Instead of losing the $b=6$ most significant bits when the variable `result` is shifted left, we retain those bits and *exclusive or* them back into the shifted `result` variable. Using this approach, the two strings "ece.uwaterloo.ca" and "cs.uwaterloo.ca" produce different hash values.

Table □ lists a number of different character strings together with the hash values obtained using Program □. For example, to hash the string "fyra", the following computation is performed (all numbers in octal):

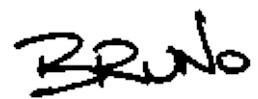
146	f
171	y
162	r
141a	
147706341	

Table: Sample
character string keys
and the hash values
obtained using
Program □.

x	Hash(x) (octal)
"ett"	01446564
"två"	05360614565
"tre"	01656345
"fyra"	0147706341
"fem"	01474455
"sex"	01624470
"sju"	01625365
"åtta"	01564656541
"nio"	01575057
"tio"	01655057
"elva"	0144556741
"tolv"	0165565566

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Hashing Containers

As explained in Section □, a container is an object which contains other objects. In this section we show how to define a `__hash__` method for the abstract Container class defined in Program □-Program □. The method given computes a suitable hash function on any container.

Given a container c which contains n objects, o_1, o_2, \dots, o_n , we can define the hash function $f(c)$ as follows:

$$f(c) = \left(\sum_{i=1}^n h(o_i) \right) \bmod W \quad (8.5)$$

That is, to hash a container, simply compute the sum of the hash values of the contained objects.

Program □ gives the code for the `__hash__` method of the abstract Container class. This method implicitly uses an iterator to enumerate all of the objects contained in the container. It calls the built-in hash function on each object in the container and accumulates the result. Recall that the `hash` function calls an object's `__hash__` method.

```
1  class Container(object):
2
3      def __hash__(self):
4          result = hash(self.__class__)
5          for obj in self:
6              result = (result + hash(obj)) & sys.maxint
7          return result
8
9      # ...
```

Program: Container class `__hash__` method.

Every concrete class derived from the Container class provides an appropriate iterator implementation. However, it is *not* necessary for any derived class to redefine the behavior of the `__hash__` method--the behavior inherited from the abstract Container class is completely generic and should suffice for all concrete container classes.

There is a slight problem with Equation □. Different container types that happen to contain identical objects produce exactly the same hash value. For example, an empty stack and an empty list both produce the same hash value. We have avoided this situation in Program □ by adding to the sum the value obtained from hashing the class of the container itself.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Using Associations

Hashing provides a way to determine the position of a given object directly from that object itself. Given an object x we determine its position by evaluating the appropriate hash function, $h(x)$. We find the location of object x in exactly the same way. But of what use is this ability to find an object if, in order to compute the hash function $h(x)$, we must be able to access the object x in the first place?

In practice, when using hashing we are dealing with *keyed data*. Mathematically, keyed data consists of ordered pairs

$$A = \{(k, v) : k \in K, v \in V\},$$

where K is a set of keys, and V is a set of values. The idea is that we will access elements of the set A using the key. That is, the hash function for elements of the set A is given by

$$f_A((k, v)) = f_K(k),$$

where f_K is the hash function associated with the set K .

For example, suppose we wish to use hashing to implement a database which contains driver's license records. Each record contains information about a driver, such as his name, address, and perhaps a summary of traffic violations. Furthermore, each record has a unique driver's license number. The driver's license number is the key and the other information is the value associated with that key.

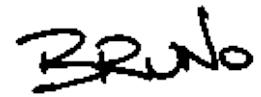
In Section □ the `Association` class was declared which comprises two instance attributes, a key and a value. Given this declaration, the definition of the hash method for `Associations` is trivial. As shown in Program □, it simply calls the built-in hash function on the key instance attribute.

```
1  class Association(Object):
2
3      def __hash__(self):
4          return hash(self.key)
5
6      # ...
```

Program: Association class `__hash__` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Hash Tables

A *hash table* is a searchable container. As such, it provides methods for putting an object into the container, finding an object in the container, and removing an object from the container.

Program □ introduces the `HashTable` class. The abstract `HashTable` class extends the the abstract `SearchableContainer` class defined in Program □.

```
 1  class HashTable(SearchableContainer):
 2
 3      def __init__(self):
 4          super(HashTable, self).__init__()
 5
 6      def __len__():
 7          pass
 8      __len__ = abstractmethod(__len__)
 9
10      def getLoadFactor(self):
11          return self.count / len(self)
12
13      loadFactor = property(
14          fget = lambda self: self.getLoadFactor())
15
16      # ...
```

Program: Abstract `HashTable` class.

Program □ declares the `__len__` method to be an abstract method. Concrete classes derived from `HashTable` must override this method. Program □ also defines the property called `loadFactor`. The purpose of this property is explained in Section □.

Program □ continues the definition of the `HashTable` class. Three methods, `f`, `g` and `h`, are defined.

```
1 class HashTable(SearchableContainer):
2
3     def f(self, obj):
4         return hash(obj)
5
6     def g(self, x):
7         return abs(x) % len(self)
8
9     def h(self, obj):
10        return self.g(self.f(obj))
11
12    # ...
```

Program: Abstract HashTable class f , g and h methods.

The methods f , g , and h correspond to the composition $h = g \circ f$ discussed in Section □. The f method takes as an object and calls the built-in hash function on that object to compute an integer. The g method uses the *division method* of hashing defined in Section □ to map an integer into the interval $[0, M-1]$, where M is the length of the hash table. Finally, the h method computes the composition of f and g .

Figure □ shows the class hierarchy for the classes discussed in this chapter. These classes illustrate various ways of implementing hash tables. In all cases, the underlying implementation of the hash table makes use of an array. The position of an object in the array is determined by hashing the object. The main problem to be resolved is how to deal with collisions--two different objects cannot occupy the same array position at the same time. In the next section, we consider an approach which solves the problem of collisions by keeping objects that collide in a linked list.

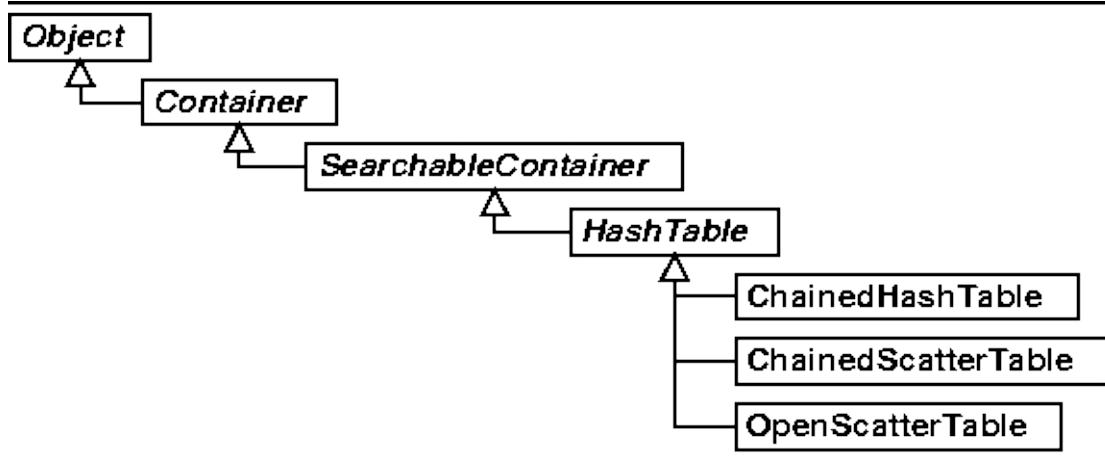


Figure: Object class hierarchy.

- [Separate Chaining](#)
 - [Average Case Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Separate Chaining

Figure □ shows a hash table that uses *separate chaining* to resolve collisions. The hash table is implemented as an array of linked lists. To insert an item into the table, it is appended to one of the linked lists. The linked list to it is appended is determined by hashing that item.

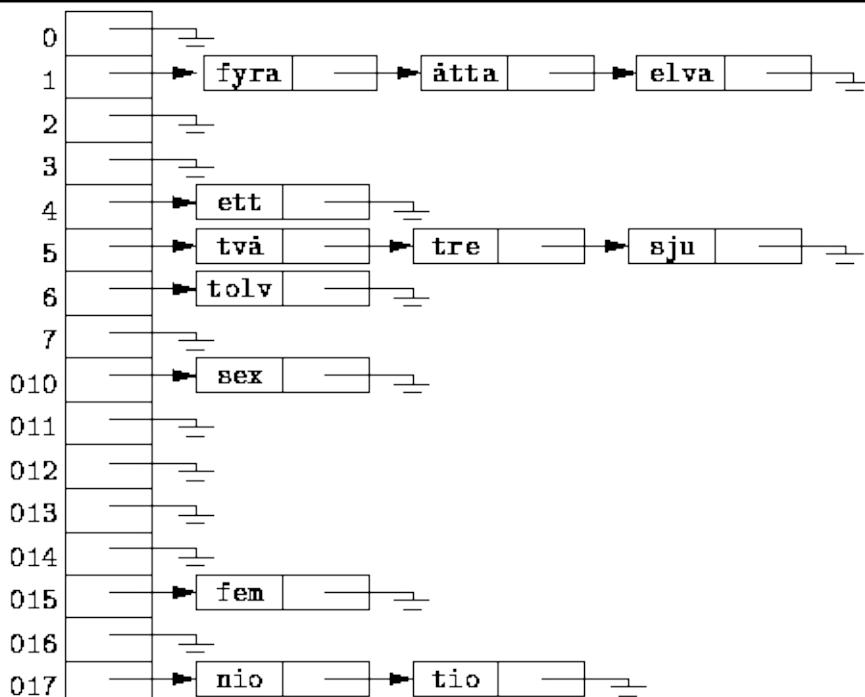


Figure: Hash table using separate chaining.

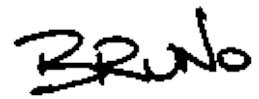
Figure □ illustrates an example in which there are $M=16$ linked lists. The twelve character strings "ett"- "tolv" have been inserted into the table using the hashed values and in the order given in Table □. Notice that in this example since $M=16$, the linked list is selected by the least significant four bits of the hashed value given in Table □. In effect, it is only the last letter of a string which determines the linked list in which that string appears.

-
- [Implementation](#)
 - [__init__, len and purge Methods](#)

- [Inserting and Removing Items](#)
 - [Finding an Item](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Program 1 introduces the ChainedHashTable class. The ChainedHashTable class extends the abstract HashTable class introduced in Program 1. The ChainedHashTable class contains a single instance attribute called `_array`. It is an Array of LinkedLists. (The Array and LinkedList classes are described in Chapter 1).

```
1  class ChainedHashTable(HashTable):
2
3      def __init__(self, length):
4          super(ChainedHashTable, self).__init__()
5          self._array = Array(length)
6          for i in xrange(len(self._array)):
7              self._array[i] = LinkedList()
8
9      def __len__(self):
10         return len(self._array)
11
12     def purge(self):
13         for i in xrange(len(self._array)):
14             self._array[i].purge()
15         self._count = 0
16     # ...
```

Program: ChainedHashTable class `__init__`, `__len__` and `purge` methods.



__init__, __len__ and purge Methods

The `__init__`, `__len__` and `purge` methods of the `ChainedHashTable` class are defined in Program □. In addition to `self`, the `__init__` method takes a single argument which specifies the size of hash table desired. It creates an array of the specified length and then initializes the elements of the array. Each element of the array is assigned an empty linked list. The running time for the `ChainedHashTable __init__` method is $O(M)$ where M is the size of the hash table.

The `__len__` method returns the length of the `_array` instance attribute. Clearly its running time is $O(1)$

The purpose of the `purge` method is to make the container empty. It does this by invoking the `purge` method one-by-one on each of the linked lists in the array. The running time of the `purge` method is $O(M)$, where M is the size of the hash table.



Inserting and Removing Items

Program □ gives the code for inserting and removing items from a ChainedHashTable.

```

1  class ChainedHashTable(HashTable):
2
3      def insert(self, obj):
4          self._array[self.h(obj)].append(obj)
5          self._count += 1
6
7      def withdraw(self, obj):
8          self._array[self.h(obj)].extract(obj)
9          self._count -= 1
10     # ...

```

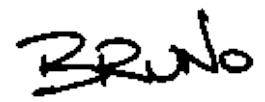
Program: ChainedHashTable class insert and withdraw methods.

The implementations of the `insert` and `withdraw` methods are remarkably simple. For example, the `insert` method first calls the hash method `h` to compute an array index which is used to select one of the linked lists. The `append` method provided by the `LinkedList` class is used to add the object to the selected linked list. The total running time for the `insert` operation is $\mathcal{T}(\text{hash}) + O(1)$, where $\mathcal{T}(\text{hash})$ is the running time of the hash function. Notice that if the hash function runs in constant time, then so too does hash table insertion operation!

The `withdraw` method is almost identical to the `insert` method. Instead of calling the `append`, it calls the linked list `extract` method to remove the specified object from the appropriate linked list. The running time of `withdraw` is determined by the time of the `extract` operation. In Chapter □ this was shown to be $O(n)$ where n is the number of items in the linked list. In the worst case, all of the items in the `ChainedHashTable` have collided with each other and ended up in the same list. That is, in the worst case if there are n items in the container, all n of them are in a single linked list. In this case, the running time of the `withdraw` operation is $\mathcal{T}(\text{hash}) + O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Finding an Item

The definition of the `find` method of the `ChainedHashTable` class is given in Program □. In addition to `self`, the `find` method takes as its argument an object. The purpose of the `find` operation is to return the object in the container that is equal to the given object.

```
1  class ChainedHashTable(HashTable):
2
3      def find(self, obj):
4          ptr = self._array[self.h(obj)].head
5          while ptr is not None:
6              datum = ptr.datum
7              if obj == datum:
8                  return datum
9              ptr = ptr.next
10         return None
11     # ...
```

Program: `ChainedHashTable` class `find` method.

The `find` method simply hashes its argument to select the linked list in which it should be found. Then, it traverses the linked list to locate the target object. As for the `withdraw` operation, the worst case running time of the `find` method occurs when all the objects in the container have collided, and the item that is being sought does not appear in the linked list. In this case, the running time of the `find` operation is $nT(\text{eq}) + T(\text{hash}) + O(n)$.

Average Case Analysis

The previous section has shown that in the worst case, the running time to insert an object into a separately chained hash table is $O(1)$, and the time to find or delete an object is $O(n)$. But these bounds are no better than the same operations on plain lists! Why have we gone to all the trouble inventing hash tables?

The answer lies not in the worst-case performance, but in the average expected performance. Suppose we have a hash table of size M . Let there be exactly n items in the hash table. We call the quantity $\lambda = n/M$ the *load factor*. The load factor is simply the ratio of the number of items in the hash table to the array length.

Program \square gives the implementation for `getLoadFactor` method of the `HashTable` class. This method computes $\lambda = n/M$ by accessing the `count` property to determine n and by calling the `len` function to determine M .

Consider a chained hash table. Let n_i be the number of items in the i^{th} linked list, for $i = 0, 1, \dots, M - 1$. The average length of a linked list is

$$\begin{aligned} \frac{1}{M} \sum_{i=0}^{M-1} n_i &= \frac{n}{M} \\ &= \lambda. \end{aligned}$$

The average length of a linked list is exactly the load factor!

If we are given the load factor λ , we can determine the *average* running times for the various operations. The average running time of `insert` is the same as its worst case time, $O(1)$ --this result does not depend on λ . On the other hand, the average running time for `withdraw` does depend on λ . It is $T(\text{hash}) + O(1) + O(\lambda)$ since the time required to delete an item from a linked list of length λ is $O(\lambda)$.

To determine the average running time for the `find` operation, we need to make an assumption about whether the item that is being sought is in the table. If the item is not found in the table, the search is said to be *unsuccessful*. The average

running time for an unsuccessful search is

$$T\langle \text{hash} \rangle + \lambda T\langle \text{eq} \rangle + O(1) + O(\lambda).$$

On the other hand, if the search target is in the table, the search is said to be *successful*. The average number of comparisons needed to find an arbitrary item in a linked list of length λ is

$$\frac{1}{\lambda} \sum_{i=1}^{\lambda} i = \frac{\lambda + 1}{2}.$$

Thus, the average running time for a successful search is

$$T\langle \text{hash} \rangle + ((\lambda + 1)/2)T\langle \text{eq} \rangle + O(1) + O(\lambda).$$

So, while any one search operation can be as bad as $O(n)$, if we do a large number of random searches, we expect that the average running time will be $O(\lambda)$. In fact, if we have a sufficiently good hash function and a reasonable set of objects in the container, we can expect that those objects are distributed throughout the table. Therefore, any one search operation will not be very much worse than the worst case.

Finally, if we know how many objects will be inserted into the hash table *a priori*, then we can choose a table size M which is larger than the maximum number of items expected. By doing this, we can ensure that $\lambda = n/M \leq 1$. That is, a linked list contains no more than one item on average. In this case, the average time for withdraw is $T\langle \text{hash} \rangle + O(1)$ and for find it is $T\langle \text{hash} \rangle + T\langle \text{eq} \rangle + O(1)$.

Scatter Tables

The separately chained hash table described in the preceding section is essentially a linked-list implementation. We have seen both linked-list and array-based implementations for all of the data structures considered so far and hash tables are no exception. Array-based hash tables are called *scatter tables*.

The essential idea behind a scatter table is that all of the information is stored within a fixed size array. Hashing is used to identify the position where an item should be stored. When a collision occurs, the colliding item is stored somewhere else in the array.

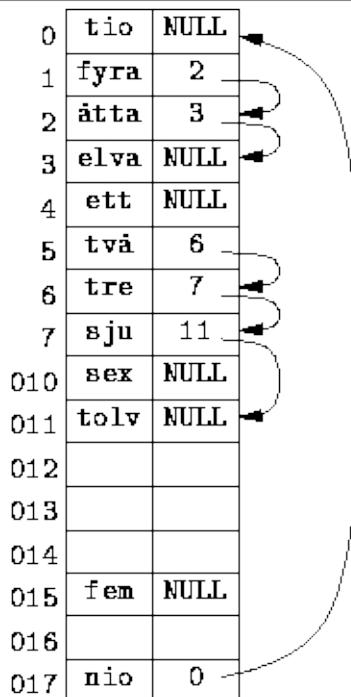
One of the motivations for using scatter tables can be seen by considering again the linked-list hash table shown in Figure □. Since most of the linked lists are empty, much of the array is unused. At the same time, for each item that is added to the table, dynamic memory is consumed. Why not simply store the data in the unused array positions?

-
- [Chained Scatter Table](#)
 - [Average Case Analysis](#)
-

Chained Scatter Table

Figure □ illustrates a *chained scatter table*. The elements of a chained scatter table are ordered pairs. Each array element contains a key and a ``pointer.'' All keys are stored in the table itself. Consequently, there is a fixed limit on the number of items that can be stored in a scatter table.

0	tio	NULL
1	fyra	2
2	åtta	3
3	elva	NULL
4	ett	NULL
5	två	6
6	tre	7
7	sju	11
010	sex	NULL
011	tolv	NULL
012		
013		
014		
015	fem	NULL
016		
017	nio	0



The diagram shows a vertical array of 18 elements. Each element is a three-column table row. The first column is an index (0, 1, 2, ..., 17). The second column is the key (tio, fyra, åtta, elva, ett, två, tre, sju, sex, tolv, ..., nio). The third column is a pointer value. A curly brace on the right side groups elements 0 through 7. Arrows point from the pointer values in rows 1, 2, 5, 6, 7, and 11 back to the start of the group, indicating they are indices into the same array. Row 3 has a pointer to row 0, row 4 to row 0, and row 17 to row 1.

Figure: Chained scatter table.

Since the pointers point to other elements in the array, they are implemented as integer-valued array subscripts. Since valid array subscripts start from the value zero, the *null* pointer must be represented not as zero, but by an integer value that is outside the array bounds (say -1).

To find an item in a chained scatter table, we begin by hashing that item to determine the location from which to begin the search. For example, to find the string "elva", which hashes to the value $\frac{044556741_8}{|044556741_8| \text{ mod } 16 = 1_8}$, we begin the search in array location $\frac{044556741_8}{|044556741_8| \text{ mod } 16 = 1_8}$. The item at that location is "fyra", which does not match. So we follow the pointer in location 1_8 to location 2_8 . The item there, "åtta", does not match either. We follow the pointer again, this time to location

³⁸ where we ultimately find the string we are looking for.

Comparing Figures □ and □, we see that the chained scatter table has embedded within it the linked lists which appear to be the same as those in the separately chained hash table. However, the lists are not exactly identical. When using the chained scatter table, it is possible for lists to *coalesce*.

For example, when using separate chaining, the keys "tre" and "sju" appear in a separate list from the key "tolv". This is because both "tre" and "sju" hash to position ⁵⁸, whereas "tolv" hashes to position ⁶⁸. The same keys appear together in a single list starting at position ⁵⁸ in the chained scatter table. The two lists have *coalesced*.

- [Implementation](#)
 - [init , len and purge Methods](#)
 - [Inserting and Finding an Item](#)
 - [Removing Items](#)
 - [Worst-Case Running Time](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementation

The ChainedScatterTable class is introduced in Program □. This class extends the abstract HashTable class introduced in Program □. The scatter table is implemented as an array of Entry class instances. The Entry class is a *nested class* defined within the ChainedScatterTable class.

```
1  class ChainedScatterTable(HashTable):
2
3      NULL = -1
4
5      class Entry(object):
6
7          def __init__(self, obj, next):
8              super(ChainedScatterTable.Entry, self).__init__()
9              self._obj = obj
10             self._next = next
11
12     # ...
```

Program: ChainedScatterTable.Entry class `__init__` method.

Each Entry instance has two instance attributes--`_obj` and `_next`. The former refers to an object. The latter indicates the position in the array of the next element of a chain. The value of the constant `NONE` will be used instead of zero to mark the end of a chain. The value zero is not used to mark the end of a chain because zero is a valid array subscript.

__init__, __len__ and purge Methods

Program □ defines the `__init__`, `__len__` and `purge` methods of the `ChainedScatterTable` class. In addition to `self`, the `__init__` method takes a single argument which specifies the size of scatter table desired. It creates an array of the desired length and initializes each element of the array by assigning to it an new `Entry` instance. Consequently, the running time for the `ChainedScatterTable __init__` method is $O(M)$ where M is the size of the scatter table.

```
 1  class ChainedScatterTable(HashTable):
 2
 3      def __init__(self, length):
 4          super(ChainedScatterTable, self).__init__()
 5          self._array = Array(length)
 6          for i in xrange(len(self._array)):
 7              self._array[i] = self.Entry(None, self.NULL)
 8
 9      def __len__(self):
10          return len(self._array)
11
12      def purge(self):
13          for i in len(self):
14              self._array[i] = self.Entry(None, self.NULL)
15          self._count = 0
16
17      # ...
```

Program: `ChainedScatterTable` class `__init__`, `__len__` and `purge` methods.

The `__len__` method returns the length of the `_array` instance attribute. Clearly, its running time is $O(1)$.

The `purge` method empties the scatter table by assigning null values to each entry in the table. the time required to purge the scatter table is $O(M)$, where M is the length of the table.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Inserting and Finding an Item

Program □ gives the code for the `insert` and `find` methods of the `ChainedScatterTable` class. To insert an item into a chained scatter table we need to find an unused array location in which to put the item. We first hash the item to determine the ``natural'' location for that item. If the natural location is unused, we store the item there and we are done.

```
 1  class ChainedScatterTable(HashTable):
 2
 3      def insert(self, obj):
 4          if self._count == len(self):
 5              raise ContainerFull
 6          probe = self.h(obj)
 7          if self._array[probe]._obj is not None:
 8              while self._array[probe]._next != self.NULL:
 9                  probe = self._array[probe]._next
10          tail = probe
11          probe = (probe + 1) % len(self)
12          while self._array[probe]._obj is not None:
13              probe = (probe + 1) % len(self)
14          self._array[tail]._next = probe
15          self._array[probe] = self.Entry(obj, self.NULL)
16          self._count += 1
17
18      def find(self, obj):
19          probe = self.h(obj)
20          while probe != self.NULL:
21              if obj == self._array[probe]._obj:
22                  return self._array[probe]._obj
23              probe = self._array[probe]._next
24          return None
25
26      # ...
```

Program: `ChainedScatterTable` class `insert` and `find` methods.

However, if the natural position for an item is occupied, then a collision has occurred and an alternate location in which to store that item must be found. When a collision occurs it must be the case that there is a chain emanating from

the natural position for the item. The insertion algorithm given always adds items at the end of the chain. Therefore, after a collision has been detected, the end of the chain is found (lines 8-9).

After the end of the chain is found, an unused array position in which to store the item must be found. This is done by a simple, linear search starting from the array position immediately following the end of the chain (lines 10-13). Once an unused position is found, it is linked to the end of the chain (line 14), and the item is stored in the unused position (line 15).

The worst case running time for insertion occurs when the scatter table has only one unused entry. That is, when the number of items in the table is $n=M-1$, where M is the table size. In the worst case, all of the used array elements are linked into a single chain of length $M-1$ and the item to be inserted hashes to the head of the chain. In this case, it takes $O(M)$ to find the end of the chain. In the worst case, the end of the chain immediately follows the unused array position.

Consequently, the linear search for the unused position is also $O(M)$. Once an unused position has been found, the actual insertion can be done in constant time. Therefore, the running time of the insertion operation is $T(\text{hash}) + O(M)$ in the worst case.

Program □ also gives the code for the `find` method which is used to locate a given object in the scatter table. The algorithm is straightforward. The item is hashed to find its natural location in the table. If the item is not found in the natural location but a chain emanates from that location, the chain is followed to determine if that item appears anywhere in the chain.

The worst-case running time occurs when the item for which we are looking is not in the table, the table is completely full, and all of the entries are linked together into a single linked list. In this case, the running time of the `find` algorithm is $T(\text{hash}) + M T(\text{eq}) + O(M)$.



Removing Items

Removing items from a chained scatter table is more complicated than putting them into the table. The goal when removing an item is to have the scatter table end up exactly as it would have appeared had that item never been inserted in the first place. Therefore, when an item is removed from the middle of a chain, items which follow it in the chain have to be moved up to fill in the hole. However, the moving-up operation is complicated by the fact that several chains may have coalesced.

Program □ gives an implementation of the `withdraw` method of the `ChainedScatterTable` class. The algorithm begins by checking that the table is not empty (lines 4-5). To remove an item, we first have to find it. This is what the loop on lines 6-8 does. If the item to be deleted is not in the table, when this loop terminates the variable `i` has the value `NULL` and an exception is thrown (lines 9-10). Otherwise, the item at position `i` in the table is to be removed.

```
1 class ChainedScatterTable(HashTable):
2
3     def withdraw(self, obj):
4         if self._count == 0:
5             raise ContainerEmpty
6         i = self.h(obj)
7         while i != self.NULL and self._array[i]._obj is not obj:
8             i = self._array[i]._next
9         if i == self.NULL:
10             raise KeyError
11         while True:
12             j = self._array[i]._next
13             while j != self.NULL:
14                 h = self.h(self._array[j]._obj)
15                 contained = False
16                 k = self._array[i]._next
17                 while k != self._array[j]._next \
18                     and not contained:
19                     if k == h:
20                         contained = True
21                     k = self._array[k]._next
22                 if not contained:
23                     break
24                 j = self._array[j]._next
25             if j == self.NULL:
26                 break
27             self._array[i]._obj = self._array[j]._obj
28             i = j
29             self._array[i] = self.Entry(None, self.NULL)
30             j = (i + len(self) - 1) % len(self)
31             while j != i:
32                 if self._array[j]._next == i:
33                     self._array[j]._next = self.NULL
34                     break
35                 j = (j + len(self) - 1) % len(self)
36             self._count -= 1
37
38     # ...
```

Program: ChainedScatterTable class withdraw method.

The purpose of the loop on lines 11-28 is to fill in the hole in the chain which results when the item at position i is removed by moving up items which follow it in the chain. What we need to do is to find the next item which follows the item at i that is safe to move into position i . The loop on lines 13-24 searches the rest of the chain following the item at i to find an item which can be safely

moved.

Figure □ illustrates the basic idea. The figure shows a chained scatter table of length ten that contains integer-valued keys. There is a single chain as shown in the figure. However, notice that the values in the chain are not all equal modulo 10. In fact, this chain must have resulted from the coalescing of three chains-- one which begins in position 1, one which begins in position 2, and one which begins in position 5.

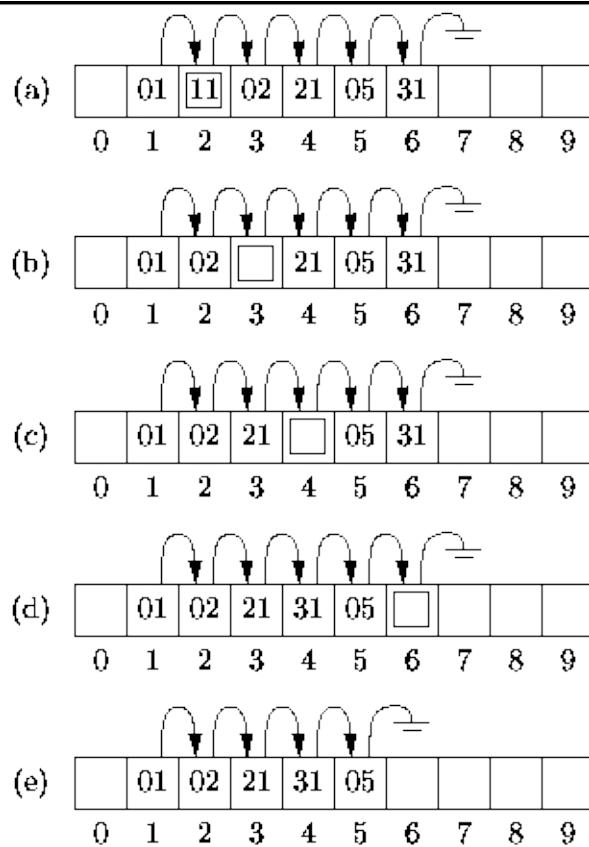


Figure: Removing items from a chained scatter table.

Suppose we wish to remove item 11 which is in position 2, which is indicated by the box in Figure □ (a). To delete it, we must follow the chain to find the next item that can be moved safely up to position 2. Item 02 which follows 11 and can be moved safely up to position 2 because that is the location to which it hashes. Moving item 02 up moves the hole down the list to position 3 (Figure □ (b)). Again we follow the chain to find that item 21 can be moved safely up giving rise to the situation in Figure □ (c).

Now we have a case where an item cannot be moved. Item 05 is the next candidate to be moved. However, it is in position 5 which is the position to which it hashes. If we were to move it up, then it would no longer be in the chain which emanates from position 5. In effect, the item would no longer be accessible! Therefore, it cannot be moved safely. Instead, we must move item 31 ahead of item 5 as shown in Figure □ (d). Eventually, the hole propagates to the end of the chain, where it can be deleted easily (Figure □ (e)).

The loop on lines 13-24 of Program □ finds the position j of an item which can be safely moved to position i . The algorithm makes use of the following fact: An item can be safely moved up *only if it does not hash to a position which appears in the linked list between i and j* . This is what the code on lines 15-21 tests.

When execution reaches line 25, either we have found an item which can be safely moved, or there does not exist such an item. If an item is found, it is moved up (line 27) and we repeat the whole process again. On the other hand, if there are no more items to be moved up, then the process is finished and the main loop (lines 11-28) terminates.

The statement on line 29 does the actual deed of removing the data from the position which i which by now is at the end of the chain. The final task to be done is to remove the pointer to position i , since there is no longer any data at that position. That is the job of the loop on lines 31-35.



[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Worst-Case Running Time

Computing a tight bound on the worst-case running time of Program □ is tricky. Assuming the item to be removed is actually in the table, then the time required to find the item (lines 6-8) is

$$T_{\langle \text{hash} \rangle} + M T_{\langle \text{--eq--} \rangle} + O(M)$$

in the worst case.

The worst-case running time of the main loop occurs when the table is full, there is only one chain, and no items can be safely moved up in the chain. In this case, the running time of the main loop (lines 11-28) is

$$\left(\frac{(M-1)M}{2} \right) T_{\langle \text{hash} \rangle} + O(M^2).$$

Finally, the worst case running time of the last loop (lines 31-35) is $O(M)$.

Therefore, the worst-case running time of the `withdraw` method for chained scatter tables is

$$\left(1 + \frac{(M-1)M}{2} \right) T_{\langle \text{hash} \rangle} + M T_{\langle \text{--eq--} \rangle} + O(M^2).$$

Clearly we don't want to be removing items from a chained scatter table very often!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Average Case Analysis

The previous section has shown that the worst-case running time to insert or to find an object into a chained scatter table is $O(M)$. The average case analysis of chained scatter tables is complicated by the fact that lists coalesce. However, if we assume that chains never coalesce, then the chains which appear in a chained scatter table for a given set of items are identical to those which appear in a separately chained hash table for the same set of items.

Unfortunately we cannot assume that lists do not coalesce--they do! We therefore expect that the average list will be longer than λ and that the running times are correspondingly slower. Knuth has shown that the average number of probes in an unsuccessful search is

$$U(\lambda) \approx 1 + \frac{1}{4}(e^{2\lambda} - 1 - 2\lambda),$$

and the average number of probes in a successful search is approximately

$$S(\lambda) \approx 1 + \frac{1}{8\lambda}(e^{2\lambda} - 1 - 2\lambda) + \frac{\lambda}{4},$$

where λ is the load factor[29]. The precise functional form of $U(\lambda)$ and $S(\lambda)$ is not so important here. What is important is that when $\lambda = 1$, i.e., when the table is full, $U(1) \approx 2.1$ and $S(1) \approx 1.8$. Regardless of the size of the table, an unsuccessful search requires just over two probes on average, and a successful search requires just under two probes on average!

Consequently, the average running time for insertion is

$$\mathcal{T}\langle_{\text{hash}}\rangle + O(U(\lambda)) = \mathcal{T}\langle_{\text{hash}}\rangle + O(1),$$

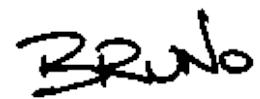
since the insertion is always done in first empty position found. Similarly, the running time for an unsuccessful search is

$$\mathcal{T}\langle_{\text{hash}}\rangle + U(\lambda)\mathcal{T}\langle_{\text{eq}}\rangle + O(U(\lambda)),$$

and for a successful search its

$$\mathcal{T}\langle_{\text{hash}}\rangle + S(\lambda)\mathcal{T}\langle_{\text{eq}}\rangle + O(S(\lambda)).$$

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Scatter Table using Open Addressing

An alternative method of dealing with collisions which entirely does away with the need for links and chaining is called *open addressing*. The basic idea is to define a *probe sequence* for every key which, when followed, always leads to the key in question.

The probe sequence is essentially a sequence of functions

$$\{h_0, h_1, \dots, h_{M-1}\},$$

where h_i is a hash function, $h_i : K \mapsto \{0, 1, \dots, M - 1\}$. To insert item x into the scatter table, we examine array locations $h_0(x), h_1(x), \dots$, until we find an empty cell. Similarly, to find item x in the scatter table we examine the same sequence of locations in the same order.

The most common probe sequences are of the form

$$h_i(x) = (h(x) + c(i)) \bmod M,$$

where $i = 0, 1, \dots, M - 1$. The function $h(x)$ is the same hash function that we have seen before. That is, the function h maps keys into integers in the range from zero to $M-1$.

The function $c(i)$ represents the collision resolution strategy. It is required to have the following two properties:

Property 1

$c(0)=0$. This ensures that the first probe in the sequence is

$$h_0(x) = (h(x) + 0) \bmod M = h(x).$$

Property 2

The set of values

$$\{c(0) \bmod M, c(1) \bmod M, c(2) \bmod M, \dots, c(M - 1) \bmod M\}$$

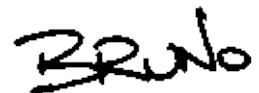
must contain every integer between 0 and $M-1$. This second property

ensures that the probe sequence eventually probes *every possible array position*.

- [Linear Probing](#)
 - [Quadratic Probing](#)
 - [Double Hashing](#)
 - [Implementation](#)
 - [Average Case Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Linear Probing

The simplest collision resolution strategy in open addressing is called *linear probing*. In linear probing, the function $c(i)$ is a linear function in i . That is, it is of the form

$$c(i) = \alpha i + \beta.$$

Property 1 requires that $c(0)=0$. Therefore, β must be zero.

In order for $c(i) = \alpha i$ to satisfy Property 2, α and M must be relatively prime. If we know the M will always be a prime number, then any α will do. On the other hand, if we cannot be certain that M is prime, then α must be one. Therefore, linear probing sequence that is usually used is

$$h_i = (h(x) + i) \bmod M,$$

for $i = 0, 1, 2, \dots, M - 1$.

Figure  illustrates an example of a scatter table using open addressing together with linear probing. For example, consider the string "åtta". This string hashes to array position ¹⁸. The corresponding linear probing sequence begins at position ¹⁸ and goes on to positions ²⁸, ³⁸, In this case, the search for the string "åtta" succeeds after three probes.

0	tio	OCCUPIED
1	fyra	OCCUPIED
2	åtta	OCCUPIED
3	elva	OCCUPIED
4	ett	OCCUPIED
5	två	OCCUPIED
6	tre	OCCUPIED
7	sju	OCCUPIED
010	sex	OCCUPIED
011	tolv	OCCUPIED
012		EMPTY
013		EMPTY
014		EMPTY
015	fem	OCCUPIED
016		EMPTY
017	nio	OCCUPIED

Figure: Scatter table using open addressing and linear probing.

To insert an item x into the scatter table, an empty cell is found by following the same probe sequence that would be used in a search for item x . Thus, linear probing finds an empty cell by doing a linear search beginning from array position $h(x)$.

An unfortunate characteristic of linear probing arises from the fact that as the table fills, clusters of consecutive cells form and the time required for a search increases with the size of the cluster. Furthermore, when we attempt to insert an item in the table at a position which is already occupied, that item is ultimately inserted at the end of the cluster--thereby increasing its length. This by itself is not inherently a bad thing. After all, when using the chained approach, every insertion increase the length of some chain by one. However, whenever an insertion is made between two clusters that are separated by one unoccupied position, the two clusters become one, thereby potentially increasing the cluster length by an amount much greater than one--a bad thing! This phenomenon is called *primary clustering* .

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Quadratic Probing

An alternative to linear probing that addresses the primary clustering problem is called *quadratic probing*. In quadratic probing, the function $c(i)$ is a quadratic function in i . \diamond The general quadratic has the form

$$c(i) = \alpha i^2 + \beta i + \gamma.$$

However, quadratic probing is usually done using $c(i) = i^2$.

Clearly, $c(i) = i^2$ satisfies property 1. What is not so clear is whether it satisfies property 2. In fact, in general it does not. The following theorem gives the conditions under which quadratic probing works:

Theorem When quadratic probing is used in a table of size M , where M is a prime number, the first $\lfloor M/2 \rfloor$ probes are distinct.

extbf{Proof} (By contradiction). Let us assume that the theorem is false. Then there exist two distinct values i and j such that $0 \leq i < j < \lfloor M/2 \rfloor$, that probe exactly the same position. Thus,

$$\begin{aligned} h_i(x) = h_j(x) &\Rightarrow h(x) + c(i) = h(x) + c(j) \pmod{M} \\ &\Rightarrow h(x) + i^2 = h(x) + j^2 \pmod{M} \\ &\Rightarrow i^2 = j^2 \pmod{M} \\ &\Rightarrow i^2 - j^2 = 0 \pmod{M} \\ &\Rightarrow (i - j)(i + j) = 0 \pmod{M} \end{aligned}$$

Since M is a prime number, the only way that the product $(i-j)(i+j)$ can be zero modulo M is for either $i-j$ to be zero or $i+j$ to be zero modulo M . Since i and j are distinct, $i - j \neq 0$. Furthermore, since both i and j are less than $\lfloor M/2 \rfloor$, the sum $i+j$ is less than M . Consequently, the sum cannot be zero. We have successfully argued an absurdity--if the theorem is false one of two quantities must be zero, neither of which can possibly be zero. Therefore, the original assumption is not correct and the theorem is true.

Applying Theorem \square we get that quadratic probing works as long as the table size is prime and there are fewer than $n=M/2$ items in the table. In terms of the

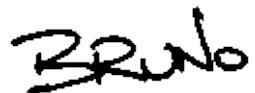
load factor $\lambda = n/M$, this occurs when $\lambda < \frac{1}{2}$.

Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search:

$$\begin{aligned} h_0(x) &= (h(x) + 0 \bmod M) \\ h_1(x) &= (h(x) + 1 \bmod M) \\ h_2(x) &= (h(x) + 4 \bmod M) \\ h_3(x) &= (h(x) + 9 \bmod M) \\ &\vdots \end{aligned}$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Double Hashing

While quadratic probing does indeed eliminate the primary clustering problem, it places a restriction on the number of items that can be put in the table--the table must be less than half full. *Double Hashing* is yet another method of generating a probing sequence. It requires two distinct hash functions,

$$h : K \rightarrow \{0, 1, \dots, M - 1\}, \\ h' : K \rightarrow \{1, 2, \dots, M - 1\}.$$

The probing sequence is then computed as follows

$$h_i(x) = (h(x) + ih'(x)) \bmod M.$$

That is, the scatter tables is searched as follows:

$$\begin{aligned} h_0 &= (h(x) + 0 \times h'(x)) \bmod M \\ h_1 &= (h(x) + 1 \times h'(x)) \bmod M \\ h_2 &= (h(x) + 2 \times h'(x)) \bmod M \\ h_3 &= (h(x) + 3 \times h'(x)) \bmod M \\ &\vdots \end{aligned}$$

Since the collision resolution function is $c(i)=ih'(x)$, the probe sequence depends on the key as follows: If $h'(x)=1$, then the probing sequence for the key x is the same as linear probing. If $h'(x)=2$, the probing sequence examines every other array position. This works as long as M is not even.

Clearly since $c(0)=0$, the double hashing method satisfies property 1. Furthermore, property 2 is satisfied as long as $h'(x)$ and M are relatively prime. Since $h'(x)$ can take on any value between 1 and $M-1$, M must be a prime number.

But what is a suitable choice for the function h' ? Recall that h is defined as the composition of two functions, $h = g \circ f$ where $g(x) = x \bmod M$. We can define h' as the composition $g' \circ f$, where

$$g'(x) = 1 + (x \bmod (M - 1)). \quad (8.6)$$

Double hashing reduces the occurrence of primary clustering since it only does a linear search if $h'(x)$ hashes to the value 1. For a good hash function, this should only happen with probability $1/(M-1)$. However, for double hashing to work at all, the size of the scatter table, M , must be a prime number. Table □ summarizes the characteristics of the various open addressing probing sequences.

Table: Characteristics of the open addressing probing sequences.

probing sequence	primary clustering capacity	limit	size restriction
linear probing	yes	none	none
quadratic probing	no	$\lambda < \frac{1}{2}$	M must be prime
double hashing	no	none	M must be prime

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Implementation

This section describes an implementation of a scatter table using open addressing with linear probing. Program □ introduces the `OpenScatterTable` class. The `OpenScatterTable` class extends the abstract `HashTable` class introduced in Program □. The scatter table is implemented as an array of elements of the nested class `Entry`. Each `Entry` instance has two instance attributes--`_obj` and `_state`. The former is refers to an object. The latter represents the state of the entry which is either `EMPTY`, `OCCUPIED` or `DELETED`.

```
1  class OpenScatterTable(HashTable):
2
3      EMPTY = 0
4      OCCUPIED = 1
5      DELETED = 2
6
7      class Entry(object):
8
9          def __init__(self, state, obj):
10              super(OpenScatterTable.Entry, self).__init__()
11              self._state = state
12              self._obj = obj
13
14      # ...
```

Program: `OpenScatterTable.Entry` class `__init__` method.

Initially, all entries are empty. When an object recorded in an entry, the state of that entry is changed to `OCCUPIED`. The purpose of the third state, `DELETED`, will be discussed in conjunction with the `withdraw` method below.

-
- [__init__ , len and purge Methods](#)
 - [Inserting Items](#)
 - [Finding Items](#)
 - [Removing Items](#)
-

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

__init__, __len__ and purge Methods

Program □ defines the `__init__`, `__len__` and `purge` methods of the `OpenScatterTable` class. In addition to `self`, the `OpenScatterTable` `__init__` method takes a single argument which specifies the size of scatter table desired. It creates an array of the desired length and initializes each element of the array by assigning to it an new `Entry` instance. Consequently, the running time for the `OpenScatterTable` `__init__` method is $O(M)$ where M is the size of the scatter table.

```
 1  class OpenScatterTable(HashTable):
 2
 3      def __init__(self, length):
 4          super(OpenScatterTable, self).__init__()
 5          self._array = Array(length)
 6          for i in xrange(len(self._array)):
 7              self._array[i] = self.Entry(self.EMPTY, None)
 8
 9      def __len__(self):
10          return len(self._array)
11
12      def purge(self):
13          for i in xrange(len(self._array)):
14              self._array[i] = self.Entry(self.EMPTY, None)
15          self._count = 0
16
17      # ...
```

Program: `OpenScatterTable` class `__init__`, `__len__` and `purge` methods.

The `__len__` method returns the length of the `_array` instance attribute. Clearly, its running time is $O(1)$.

The `purge` method empties the scatter table by nulling out all the `Entries` in the array. The time required to purge the scatter table is $O(M)$, where M is the length of the table.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Inserting Items

The method for inserting an item into a scatter table using open addressing is actually quite simple--find an unoccupied array location and then put the item in that location. To find an unoccupied array element, the array is probed according to a probing sequence. In this case, the probing sequence is linear probing. Program □ defines the methods needed to insert an item into the scatter table.

```
 1 class OpenScatterTable(HashTable):
 2
 3     def c(self, i):
 4         return i
 5
 6     def findUnoccupied(self, obj):
 7         hash = self.h(obj)
 8         for i in xrange(self._count + 1):
 9             probe = (hash + self.c(i)) % len(self)
10             if self._array[probe]._state != self.OCCUPIED:
11                 return probe
12             raise ContainerFull
13
14     def insert(self, obj):
15         if self._count == len(self):
16             raise ContainerFull
17         offset = self.findUnoccupied(obj)
18         self._array[offset] = self.Entry(self.OCCUPIED, obj)
19         self._count += 1
20
21     # ...
```

Program: OpenScatterTable class c, findUnoccupied and insert methods.

The method `c` defines the probing sequence. As it turns out, the implementation required for a linear probing sequence is trivial. The method `c` is the identity function.

The method `findUnoccupied` is to locate an unoccupied array position. The `findUnoccupied` method probes the array according the probing sequence determined by the `c` method. At most $n+1$ probes are made, where $n = \text{count}$ is

the number of items in the scatter table. When using linear probing it is always possible to find an unoccupied cell in this many probes as long as the table is not full. Notice also that we do not search for an EMPTY cell. Instead, the search terminates when a cell is found, the state of which is not OCCUPIED, that is, EMPTY or DELETED. The reason for this subtlety has to do with the way items may be removed from the table. The `findUnoccupied` method returns a value between 0 and $M-1$, where M is the length of the scatter table, if an unoccupied location is found. Otherwise, it throws an exception that indicates that the table is full.

In addition to `self`, the `insert` method takes an object and puts that object into the scatter table. It does so by calling `findUnoccupied` to determine the location of an unoccupied entry in which to put the object. The state of the unoccupied entry is set to OCCUPIED and the object is saved in the entry.

The running time of the `insert` method is determined by that of `findUnoccupied`. The worst case running time of `findUnoccupied` is $O(n)$, where n is the number of items in the scatter table. Therefore, the running time of `insert` is $T_{\text{hash}} + O(n)$.



Finding Items

The `find` and `findMatch` methods of the `OpenScatterTable` class are defined in Program □. In addition to `self`, the `findMatch` method takes an object and searches the scatter table for an object which matches the given one.

```
 1  class OpenScatterTable(HashTable):
 2
 3      def findMatch(self, obj):
 4          hash = self.h(obj)
 5          for i in xrange(len(self._array)):
 6              probe = (hash + self.c(i)) % len(self)
 7              if self._array[probe]._state == self.EMPTY:
 8                  break
 9              if self._array[probe]._state == self.OCCUPIED and \
10                  self._array[probe]._obj == obj:
11                  return probe
12          return -1
13
14      def find(self, obj):
15          offset = self.findMatch(obj)
16          if offset >= 0:
17              return self._array[offset]._obj
18          else:
19              return None
20
21      # ...
```

Program: `OpenScatterTable` class `findMatch` and `find` methods.

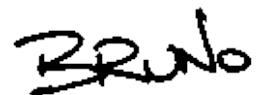
`findMatch` follows the same probing sequence used by the `insert` method. Therefore, if there is a matching object in the scatter table, `findMatch` will make exactly the same number of probes to locate the object as were made to put the object into the table in the first place. The `findMatch` method makes at most M probes, where M is the size of the scatter table. However, note that the loop immediately terminates should it encounter an `EMPTY` location. This is because if the target has not been found by the time an empty cell is encountered, then the target is not in the table. Notice also that the comparison is only attempted for entries which are marked `OCCUPIED`. Any locations marked `DELETED` are not

examined during the search but they do not terminate the search either.

The running time of the `find` method is determined by that of `findMatch`. In the worst case `findMatch` makes n comparisons, where n is the number of items in the table. Therefore, the running time of `find` is $T_{\langle \text{hash} \rangle} + nT_{\langle \text{eq} \rangle} + O(M)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Removing Items

Removing items from a scatter table using open addressing has to be done with some care. The naïve approach would be to locate the item to be removed and then change the state of its location to `EMPTY`. However, that approach does not work! Recall that the `findMatch` method which is used to locate an item stops its search when it encounters an `EMPTY` cell. Therefore, if we change the state of a cell in the middle of a cluster to `EMPTY`, all subsequent searches in that cluster will stop at the empty cell. As a result, subsequent searches for an object may fail even when the target is still in the table!

One way to deal with this is to make use of the third state, `DELETED`. Instead of marking a location `EMPTY`, we mark it `DELETED` when an item is deleted. Recall that the `findMatch` method was written in such a way that it continues past deleted cells in its search. Also, the `findUnoccupied` method was written to stop its search when it encounters either an `EMPTY` or a `DELETED` location. Consequently, the positions marked `DELETED` are available for reuse when insertion is done.

Program □ gives the implementation of the `withdraw`. In addition to `self`, the `withdraw` method takes an object and removes that object from the scatter table. It does so by first locating the specific object instance using `findInstance` and then marking the location `DELETED`. The implementation of `findInstance` has been elided. It is simply a trivial variation of the `findMatch` method.

```
 1 class OpenScatterTable(HashTable):
 2
 3     def withdraw(self, obj):
 4         if self._count == 0:
 5             raise ContainerEmpty
 6         offset = self.findInstance(obj)
 7         if offset < 0:
 8             raise KeyError
 9         self._array[offset] = self.Entry(self.DELETED, None)
10         self._count -= 1
11
12     # ...
```

Program: OpenScatterTable Class withdraw method.

The running time of the `withdraw` method is determined by that of `findInstance`. In the worst case `findInstance` has to examine every array position. Therefore, the running time of `withdraw` is $T_{\langle \text{hash} \rangle} + O(M)$.

There is a very serious problem with the technique of marking locations as `DELETED`. After a large number of insertions and deletions have been done, it is very likely that there are no cells left that are marked `EMPTY`. This is because, nowhere in any of the methods (except `purge`) is a cell ever marked `EMPTY!` This has the very unfortunate consequence that an unsuccessful search, i.e., a search for an object which is not in the scatter table, is $\Omega(M)$. Recall that `findMatch` examines at most M array locations and only stops its search early when an `EMPTY` location is encountered. Since there are no more empty locations, the search must examine all M locations.

If we are using the scatter table in an application in which we know *a priori* that no items will be removed, or perhaps only a very small number of items will be removed, then the `withdraw` method given in Program □ will suffice. However, if the application is such that a significant number of withdrawals will be made, a better implementation is required.

Ideally, when removing an item the scatter table ends up exactly as it would have appeared had that item never been inserted in the first place. Note that exactly the same constraint is met by the `withdraw` method for the `ChainedScatterTable` class given in Program □. It turns out that a variation of that algorithm can be used to implement the `withdraw` method for the `OpenScatterTable` class as shown in Program □.

```

1  class OpenScatterTableV2(OpenScatterTable):
2
3      def withdraw(self, obj):
4          if self._count == 0:
5              raise ContainerEmpty
6          i = self.findInstance(obj)
7          if i < 0:
8              raise KeyError
9          while True:
10              j = (i + 1) % len(self)
11              while self._array[j]._state == self.OCCUPIED:
12                  h = self.h(self._array[j]._obj)
13                  if ((h <= i and i < j) or (i < j and j < h) or \
14                      (j < h and h <= i)):
15                      break
16              j = (j + 1) % len(self)
17              if self._array[j]._state == self.EMPTY:
18                  break
19              self._array[i] = self._array[j]
20              i = j
21              self._array[i] = self.Entry(self.EMPTY, None)
22              self._count -= 1
23
24      # ...

```

Program: OpenScatterTableV2 class withdraw method.

The algorithm begins by checking that the scatter table is not empty. Then it calls `findInstance` to determine the position i of the item to be removed. If the item to be removed is not in the scatter table `findInstance` returns -1 and an exception is thrown. Otherwise, `findInstance` falls between 0 and $M-1$, which indicates that the item was found.

In the general case, the item to be deleted falls in the middle of a cluster. Deleting it would create a hole in the middle of the cluster. What we need to do is to find another item further down in the cluster which can be moved up to fill in the hole that would be created when the item at position i is deleted. The purpose of the loop on lines 11-16 is to find the position j of an item which can be moved safely into position i . Note the implementation here implicitly assumes that a linear probing sequence is used--the `c` method is not called explicitly. An item at position j can be moved safely to position i only if the hash value of the item at position j is not cyclically contained in the interval between i and j .

If an item is found at some position j that can be moved safely, then that item is moved to position i on line 19. The effect of moving the item at position j to position i , is to move the hole from position i to position j (line 24). Therefore, another iteration of the main loop (lines 9-20) is needed to fill in the relocated hole in the cluster.

If no item can be found to fill in the hole, then it is safe to split the cluster in two. Eventually, either because no item can be found to fill in the hole or because the hole has moved to the end of the cluster, there is nothing more to do other than to delete the hole. Thus, on line 21 the entry at position i is set to `EMPTY` and the associated `obj` is set to `None`. Notice that the third state `DELETED` is not required in this implementation of `withdraw`.

If we use the `withdraw` implementation of Program □, the scatter table entries will only ever be in one of two states—`OCCUPIED` or `EMPTY`. Consequently, we can improve the bound on the worst-case for the search from $T(\text{hash}) + nT(\text{eq}) + O(M)$ to $T(\text{hash}) + nT(\text{eq}) + O(n)$, where n is the number of items in the scatter table.

Determining the running time of Program □ is a little tricky. Assuming the item to be deleted is actually in the table, the running time to find the position of that item (line 6) is $T(\text{hash}) + O(n)$, where $n = \text{count}$ is the number of item actually in the scatter table. In the worst case, the scatter table is comprised of a single cluster of n items, and we are deleting the first item of the cluster. In this case, the main loop on lines 9-20 makes a pass through the entire cluster, in the worst case moving the hole to the end of the cluster one position at a time. Thus, the running time of the main loop is $(n - 1)T(\text{hash}) + O(n)$. The remaining lines require a constant amount of additional time. Altogether, the running time for the `Withdraw` method is $nT(\text{hash}) + O(n)$ in the worst case.

Average Case Analysis

The average case analysis of open addressing is easy if we ignore the primary clustering phenomenon. Given a scatter table of size M that contains n items, we assume that each of the $\binom{M}{n}$ combinations of n occupied and $(m-n)$ empty scatter table entries is equally likely. This is the so-called *uniform hashing model*.

In this model we assume that the entries will either be occupied or empty, i.e., the **DELETED** state is not used. Suppose a search for an empty cell requires exactly i probes. Then the first $i-1$ positions probed must have been occupied and the i^{th} position probed was empty. Consider the i cells which were probed. The number of combinations in which $i-1$ of the probed cells are occupied and one is empty is $\binom{M-i}{n-i+1}$. Therefore, the probability that exactly i probes are required is

$$P_i = \frac{\binom{M-i}{n-i+1}}{\binom{M}{n}}. \quad (8.7)$$

The average number of probes required to find an empty cell in a table which has n occupied cells is $U(n)$ where

$$U(n) = \sum_{i=1}^M i P_i. \quad (8.8)$$

Using Equation 8.7 into Equation 8.8 and simplifying the result gives

$$\begin{aligned} U(n) &= \frac{M+1}{M-n+1} \\ &= \frac{1 + \frac{1}{M}}{1 - \lambda + \frac{1}{M}}, \quad \text{where } \lambda = n/M \\ &\approx \frac{1}{1-\lambda} \end{aligned} \quad (8.9)$$

This result is actually quite intuitive. The load factor, λ , is the fraction of occupied entries. Therefore, $1 - \lambda$ entries are empty so we would expect to have to probe $1/(1 - \lambda)$ entries before finding an empty one! For example, if the load factor is 0.75, a quarter of the entries are empty. Therefore, we expect to have to probe four entries before finding an empty one.

To calculate the average number of probes for a successful search we make the observation that when an item is initially inserted, we need to find an empty cell in which to place it. For example, the number of probes to find the empty position into which the i^{th} item is to be placed is $U(i)$. And this is exactly the number of probes it takes to find the i^{th} item again! Therefore, the average number of probes required for a successful search in a table which has n occupied cells is $S(n)$ where

$$S(n) = \frac{1}{n} \sum_{i=0}^{n-1} U(i). \quad (8.11)$$

Substituting Equation 8.11 in Equation 8.10 and simplifying gives

$$\begin{aligned} S(n) &= \frac{1}{n} \sum_{i=0}^n \frac{M+1}{M-i+1} \\ &= \frac{M+1}{N} (H_{M+1} - H_{M-n+1}) \\ &\approx \frac{1}{\lambda} \ln \frac{1}{1-\lambda} \end{aligned} \quad (8.12)$$

where H_k is the k^{th} harmonic number (see Section 8.1). Again, there is an easy intuitive derivation for this result. We can use a simple integral to calculate the mean number of probes for a successful search using the approximation $U(n) = 1/(1-\lambda)$ as follows

$$\begin{aligned} S(n) &= \frac{1}{n} \sum_{i=0}^n U(i) \\ &\approx \frac{1}{\lambda} \int_0^\lambda \frac{1}{1-x} dx \\ &\approx \frac{1}{\lambda} \ln \frac{1}{1-\lambda}. \end{aligned}$$

Empirical evidence has shown that the formulas derived for the *uniform hashing model* characterize the performance of scatter tables using open addressing with quadratic probing and double hashing quite well. However, they do not capture the effect of primary clustering which occurs when linear probing is used. Knuth has shown that when primary clustering is taking into account, the number of probes required to locate an empty cell is

$$U(n) = \frac{1}{2} \left(1 + \left(\frac{1}{1-\lambda} \right)^2 \right), \quad (8.13)$$

and the number of probes required for a successful search is

$$S(n) = \frac{1}{2} \left(1 + \frac{1}{1-\lambda} \right). \quad (8.14)$$

The graph in Figure □ compares the predictions of the uniform hashing model (Equations □ and □) with the formulas derived by Knuth (Equations □ and □). Clearly, while the results are qualitatively similar, the formulas are in agreement for small load factors and they diverge as the load factor increases.

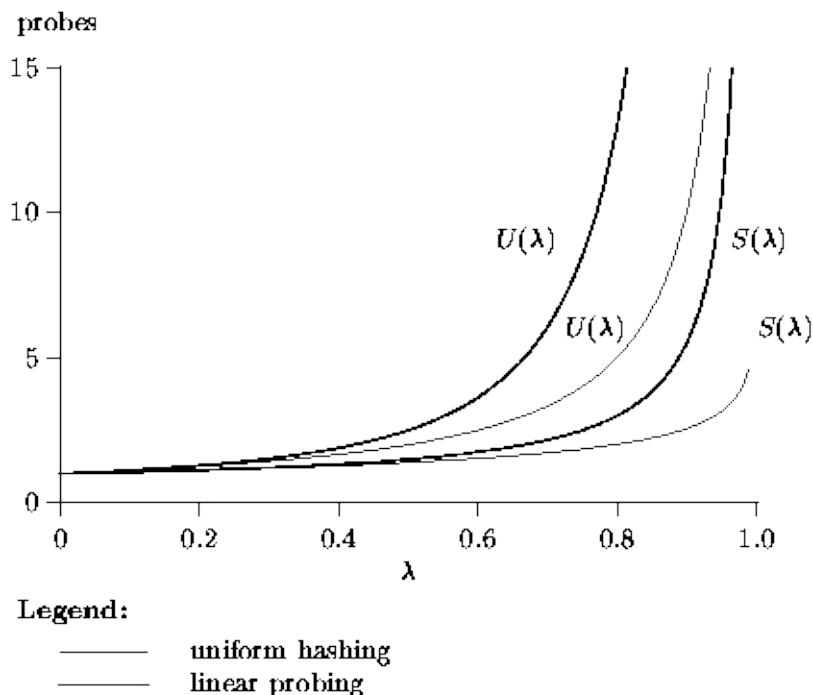


Figure: Number of probes vs. load factor for uniform hashing and linear probing.

Applications

Hash and Scatter tables have many applications. The principal characteristic of such applications is that keyed information needs to be frequently accessed and the access pattern is either unknown or known to be random. For example, hash tables are often used to implement the *symbol table* of a programming language compiler. A symbol table is used to keep track of information associated with the symbols (variable and method names) used by a programmer. In this case, the keys are character strings and each key hash associated with it some information about the symbol (e.g., type, address, value, lifetime, scope).

This section presents a simple application of hash and scatter tables. Suppose we are required to count the number of occurrences of each distinct word contained in a text file. We can do this easily using a hash or scatter table. Program □ gives the an implementation.

```
 1 class Algorithms(object):
 2
 3     class Counter(object):
 4
 5         def __init__(self, value):
 6             super(Algorithms.Counter, self).__init__()
 7             self._value = value
 8
 9         def __repr__():
10             return str(self._value)
11
12         def __iadd__(self, value):
13             self._value += value
14
15     def wordCounter(input, output):
16         table = ChainedHashTable(1031)
17         for line in input.readlines():
18             for word in line.split():
19                 assoc = table.find(Association(word))
20                 if assoc is None:
21                     table.insert(Association(
22                         word, Algorithms.Counter(1)))
23                 else:
24                     counter = assoc.value
25                     counter += 1
26             output.write(str(table) + "\n")
27     wordCounter = staticmethod(wordCounter)
```

Program: Hash/scatter table application--counting words.

The nested class Counter is used to count the number of occurrences of a word. The Counter class provides an increment method that is called to increase the value of the counter by one.

The wordCounter method does the actual work of counting the words in the input file. The local variable table refers to a ChainedHashTable that is used to keep track of the words and counts. The objects which are put into the hash table are all instances of the class Association. Each association has as its key a str class instance, and as its value a Counter class instance.

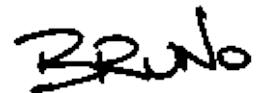
The main loop of the wordCounter method reads a line of text from the input stream. Each line of text is split into an array of words and the inner loop processes each word one at a time. For each word, a find operation is done on the hash table to determine if there is already an association for the given key. If

none is found, a new association is created and inserted into the hash table. The given word is used as the key of the new association and the value is a counter which is initialized to one. On the other hand, if there is already an association for the given word in the hash table, the corresponding counter is incremented. When the `wordCounter` method reaches the end of the input stream, it simply prints the hash table on the given output stream.

The running time of the `wordCounter` method depends on a number of factors, including the number of different keys, the frequency of occurrence of each key, and the distribution of the keys in the overall space of keys. Of course, the hash/scatter table implementation chosen has an effect as does the size of the table used. For a reasonable set of keys we expect the hash function to do a good job of spreading the keys uniformly in the table. Provided a sufficiently large table is used, the average search and insertion time is bounded by a constant. Under these ideal conditions the running time should be $O(n)$, where n is the number of words in the input file.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exercises

1. Suppose we know *a priori* that a given key is equally likely to be any integer between a and b .
 1. When is the *division method of hashing* a good choice?
 2. When is the *middle square method of hashing* a good choice?
2. Compute (by hand) the hash value obtained by Program [□](#) for the strings "ece.uw.ca" and "cs.uw.ca". **Hint:** Refer to Appendix [□](#).
3. Canadian postal codes have the format LLD DLD where L is always a letter (A-Z), D is always a digit (0-9), and is always a single space. For example, the postal code for the University of Waterloo is N2L 3G1. Devise a suitable hash function for Canadian postal codes.
4. For each type of hash table listed below, show the hash table obtained when we insert the keys

```
{"un", "deux", "trois", "quatre", "cinq", "six",
 "sept", "huit", "neuf", "dix", "onze", "douze"}.
```

in the order given into a table of size $M=16$ that is initially empty. Use the following table of hash values:

x	Hash(x) (octal)
"un"	016456
"deux"	0145446470
"trois"	016563565063
"quatre"	010440656345
"cinq"	0142505761
"six"	01625070
"sept"	0162446164
"huit"	0151645064
"neuf"	0157446446
"dix"	01455070
"onze"	0156577345
"douze"	014556647345

1. chained hash table,
2. chained scatter table,
3. open scatter table using *linear probing*,

4. open scatter table using *quadratic probing*, and
5. open scatter table using *double hashing*. (Use Equation \square as the secondary hash function).

5. For each table obtained in Exercise \square , show the result when the key "deux" is withdrawn.
6. For each table considered in Exercise \square derive an expression for the total memory space used to represent a table of size M that contains n items.
7. Consider a chained hash table of size M that contains n items. The performance of the table decreases as the load factor $\lambda = n/M$ increases. In order to keep the load factor below one, we propose to double the size of the array when $n=M$. However, in order to do so we must *rehash* all of the elements in the table. Explain why rehashing is necessary.
8. Give the sequence of M keys that fills a *chained scatter table* of size M in the *shortest* possible time. Find a tight, asymptotic bound on the minimum running time taken to fill the table.
9. Give the sequence of M keys that fills a *chained scatter table* of size M in the *longest* possible time. Find a tight, asymptotic bound on the minimum running time taken to fill the table.
10. Consider the chained hash table introduced shown in Program \square -Program \square
 - .
 - 1. Rewrite the `insert` method so that it doubles the length of the array when $\lambda = 1$.
 - 2. Rewrite the `withdraw` method so that it halves the length of the array when $\lambda = \frac{1}{2}$.
 - 3. Show that the *average* time for both insert and withdraw operations is still $O(1)$.
11. Consider two sets of integers, $S = \{s_1, s_2, \dots, s_m\}$ and $T = \{t_1, t_2, \dots, t_n\}$.
 1. Devise an algorithm that uses a hash table to test whether S is a subset of T . What is the average running time of your algorithm?
 2. Two sets are *equivalent* if and only if both $S \subseteq T$ and $T \subseteq S$. Show that we can test if two sets of integers are equivalent in $O(m+n)$ time (on average).
12. (This question should be attempted *after* reading Chapter \square). Rather than use an array of linked lists, suppose we implement a hash table with an array of *binary search trees*.
 1. What are the worst-case running times for `insert`, `find`, and

`withdraw`.

2. What are the average running times for `insert`, `find`, and `withdraw`.
13. (This question should be attempted *after* reading Section □). Consider a scatter table with open addressing. Devise a probe sequence of the form

$$h_i(x) = (h(x) + c(i)) \bmod M,$$

where $c(i)$ is a *full-period pseudo random number generator*. Why is such a sequence likely to be better than either linear probing or quadratic probing?

Projects

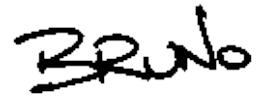
1. Complete the implementation of the `ChainedHashTable` class declared in Program □-Program □ by providing suitable definitions for the following operations: `accept`, `_compareTo`, `__contains__`, and `__iter__`. Write a test program and test your implementation.
2. Complete the implementation of the `ChainedScatterTable` class declared in Program □ by providing suitable definitions for the following operations: `accept`, `_compareTo`, `__contains__`, `getIsFull`, `getIsEmpty`, and `__iter__`. Write a test program and test your implementation.
3. Complete the implementation of the `OpenScatterTable` class declared in Program □ by providing suitable definitions for the following methods: `accept`, `_compareTo`, `__contains__`, `getIsFull`, `getIsEmpty`, and `__iter__`. Write a test program and test your implementation.
4. The `withdraw` method defined in Program □ has been written under the assumption that linear probing is used. Therefore, it does not call explicitly the collision resolution method c. Rewrite the `withdraw` method so that it works correctly regardless of the collision resolution strategy used.
5. Consider an application that has the following profile: First, n symbols (character strings) are read in. As each symbol is read, it is assigned an ordinal number from 1 to n . Then, a large number of operations are performed. In each operation we are given either a symbol or a number and we need to determine its mate. Design, implement and test a data structure that provides both mappings in $O(1)$ time.
6. Spelling checkers are often implemented using hashing. However, the space required to store all the words in a complete dictionary is usually prohibitive. An alternative solution is to use a very large array of bits. The array is initialized as follows: First, all the bits are set to zero. Then for each word w in the dictionary, we set bit $h(w)$ to one, where $h(\cdot)$ is a suitable hash function.

To check the spelling in a given document, we hash the words in the document one-by-one and examine the corresponding bit of the array. If the bit is a zero, the word does not appear in the dictionary and we conclude that it is misspelled. Note if the bit is a one, the word may still be misspelled, but we cannot tell.

Design and implement a spelling checker. **Hint:** Use the SetAsBitVector class given in Chapter □.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Trees

In this chapter we consider one of the most important non-linear information structures--*trees*. A tree is often used to represent a *hierarchy*. This is because the relationships between the items in the hierarchy suggest the branches of a botanical tree.

For example, a tree-like *organization chart* is often used to represent the lines of responsibility in a business as shown in Figure □. The president of the company is shown at the top of the tree and the vice-presidents are indicated below him. Under the vice-presidents we find the managers and below the managers the rest of the clerks. Each clerk reports to a manager, each manager reports to a vice-president, and each vice-president reports to the president.

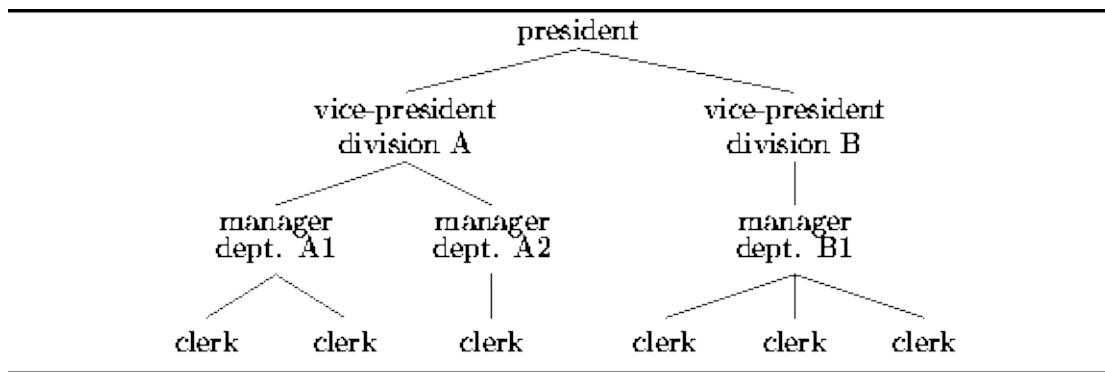


Figure: Representing a hierarchy using a tree.

It just takes a little imagination to see the tree in Figure □. Of course, the tree is upside-down. However, this is the usual way the data structure is drawn. The president is called the *root* of the tree and the clerks are the *leaves*.

A tree is extremely useful for certain kinds of computations. For example, suppose we wish to determine the total salaries paid to employees by division or by department. The total of the salaries in division A can be found by computing the sum of the salaries paid in departments A1 and A2 plus the salary of the vice-president of division A. Similarly, the total of the salaries paid in department A1 is the sum of the salaries of the manager of department A1 and of

the two clerks below him.

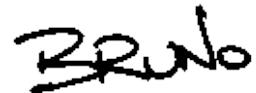
Clearly, in order to compute all the totals, it is necessary to consider the salary of every employee. Therefore, an implementation of this computation must *visit* all the employees in the tree. An algorithm that systematically *visits* all the items in a tree is called a *tree traversal*.

In this chapter we consider several different kinds of trees as well as several different tree traversal algorithms. In addition, we show how trees can be used to represent arithmetic expressions and how we can evaluate an arithmetic expression by doing a tree traversal.

- [Basics](#)
 - [N-ary Trees](#)
 - [Binary Trees](#)
 - [Tree Traversals](#)
 - [Expression Trees](#)
 - [Implementing Trees](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

[Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.](#)



Basics

The following is a mathematical definition of a tree:

Definition (Tree) A tree T is a finite, non-empty set of nodes ,

$$T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_n,$$

with the following properties:

1. A designated node of the set, r , is called the *root* of the tree; and
2. The remaining nodes are partitioned into $n \geq 0$ subsets, T_1, T_2, \dots, T_n , each of which is a tree.

For convenience, we shall use the notation $T = \{r, T_1, T_2, \dots, T_n\}$ to denote the tree T .

Notice that Definition \square is *recursive*--a tree is defined in terms of itself! Fortunately, we do not have a problem with infinite recursion because every tree has a *finite* number of nodes and because in the base case a tree has $n=0$ subtrees.

It follows from Definition \square that the minimal tree is a tree comprised of a single root node. For example $T_a = \{A\}$ is such a tree. When there is more than one node, the remaining nodes are partitioned into subtrees. For example, the $T_b = \{B, \{C\}\}$ is a tree which is comprised of the root node B and the subtree $\{C\}$. Finally, the following is also a tree

$$T_c = \{D, \{E, \{F\}\}, \{G, \{H, \{I\}\}, \{J, \{K\}, \{L\}\}, \{M\}\}\}. \quad (9.1)$$

How do T_a , T_b , and T_c resemble their arboreal namesake? The similarity becomes apparent when we consider the graphical representation of these trees shown in Figure \square . To draw such a pictorial representation of a tree, $T = \{r, T_1, T_2, \dots, T_n\}$, the following recursive procedure is used: First, we first draw the root node r . Then, we draw each of the subtrees, T_1, T_2, \dots, T_n , beside each other below the root. Finally, lines are drawn from r to the roots of each of the subtrees.

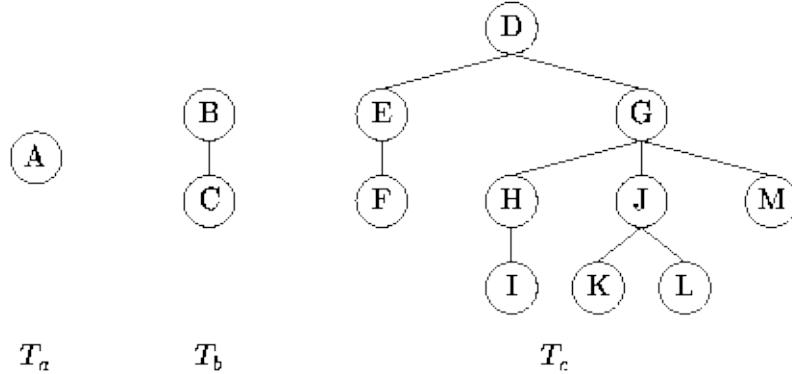


Figure: Examples of trees.

Of course, trees drawn in this fashion are upside down. Nevertheless, this is the conventional way in which tree data structures are drawn. In fact, it is understood that when we speak of ``up'' and ``down,'' we do so with respect to this pictorial representation. For example, when we move from a root to a subtree, we will say that we are moving *down* the tree.

The inverted pictorial representation of trees is probably due to the way that genealogical *lineal charts* are drawn. A *lineal chart* is a family tree that shows the descendants of some person. And it is from genealogy that much of the terminology associated with tree data structures is taken.

-
- [Terminology](#)
 - [More Terminology](#)
 - [Alternate Representations for Trees](#)
-



Terminology

Consider a tree $T = \{r, T_1, T_2, \dots, T_n\}$, $n \geq 0$, as given by Definition □.

- The *degree* of a node is the number of subtrees associated with that node. For example, the degree of tree T is n .
- A node of degree zero has no subtrees. Such a node is called a *leaf*.
- Each root r_i of subtree T_i of tree T is called a *child* of r . The term *grandchild* is defined in a similar manner.
- The root node r of tree T is the *parent* of all the roots r_i of the subtrees T_i , $1 < i \leq n$. The term *grandparent* is defined in a similar manner.
- Two roots r_i and r_j of distinct subtrees T_i and T_j of tree T are called *siblings*.

Clearly the terminology used for describing tree data structures is a curious mixture of mathematics, genealogy, and botany. There is still more terminology to be introduced, but in order to do that, we need the following definition:

Definition (Path and Path Length) Given a tree T containing the set of nodes R , a *path* in T is defined as a non-empty sequence of nodes

$$P = \{r_1, r_2, \dots, r_k\},$$

where $r_i \in R$, for $1 \leq i \leq k$ such that the i^{th} node in the sequence, r_i , is the *parent* of the $(i+1)^{\text{th}}$ node in the sequence r_{i+1} . The *length* of path P is $k-1$.

For example, consider again the tree T_r shown in Figure □. This tree contains many different paths. In fact, if you count carefully, you should find that there are exactly 29 distinct paths in tree T_r . This includes the path of length zero, $\{D\}$; the path of length one, $\{E, F\}$; and the path of length three, $\{D, G, J, K\}$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



More Terminology

Consider a tree T containing the set of nodes R as given by Definition □.

- The *level* or *depth* of a node $r_i \in R$ in a tree T is the length of the unique path in T from its root r to the node r_i . For example, the root of T is at level zero and the roots of the subtrees are at level one.
- The *height of a node* $r_i \in R$ in a tree T is the length of the longest path from node r_i to a leaf. Therefore, the leaves are all at height zero.
- The *height of a tree* T is the height of its root node r .
- Consider two nodes r_i and r_j in a tree T . The node r_i is an *ancestor* of the node r_j if there exists a path in T from r_i to r_j . Notice that r_i and r_j may be the same node. That is, a node is its own ancestor. However, the node r_i is a *proper ancestor* if there exists a path p in T from r_i to r_j such that the length of the path p is non-zero.
- Similarly, node r_j is a *descendant* of the node r_i if there exists a path in T from r_i to r_j . And since r_i and r_j may be the same node, a node is its own descendant. The node r_j is a *proper descendant* if there exists a path p in T from r_i to r_j such that the length of the path p is non-zero.

Alternate Representations for Trees

Figure □ shows an alternate representation of the tree T_c defined in Equation □. In this case, the tree is represented as a set of nested regions in the plane. In fact, what we have is a *Venn diagram* which corresponds to the view that a tree is a set of sets.

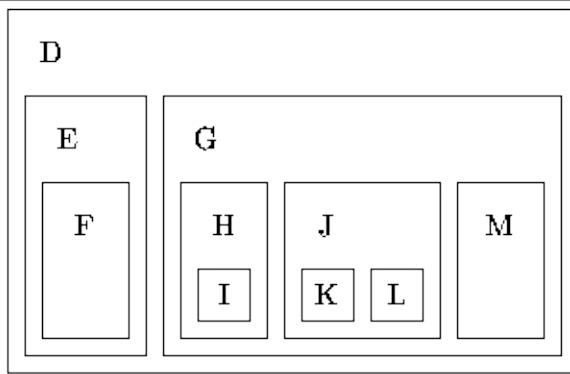


Figure: An alternate graphical representation for trees.

This hierarchical, set-within-a-set view of trees is also evoked by considering the nested structure of computer programs. For example, consider the following fragment of Python code:

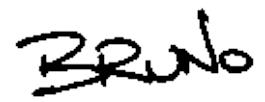
```

class D:
    class E:
        class F: pass
    class G:
        class H:
            class I: pass
        class J:
            class K: pass
            class L: pass
        class M: pass
  
```

The nesting structure of this program and the tree given in Equation □ are *isomorphic*. ♦ Therefore, it is not surprising that trees have an important role in the analysis and translation of computer programs.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

N-ary Trees

In the preceding section we considered trees in which the nodes can have arbitrary degrees. In particular, the general case allows each of the nodes of a tree to have a different degree. In this section we consider a variation in which all of the nodes of the tree are required to have exactly the same degree.

Unfortunately, simply adding to Definition □ the additional requirement that all of the nodes of the tree have the same degree does not work. It is not possible to construct a tree which has a finite number of nodes all of which have the same degree N in any case except the trivial case of $N=0$. In order to make it work, we need to introduce the notion of an empty tree as follows:

Definition (N -ary Tree) An N -ary tree T is a finite set of *nodes* with the following properties:

1. Either the set is empty, $T = \emptyset$; or
2. The set consists of a root, R , and exactly N distinct N -ary trees. That is, the remaining nodes are partitioned into $N \geq 0$ subsets, T_0, T_1, \dots, T_{N-1} , each of which is an N -ary tree such that $T = \{R, T_0, T_1, \dots, T_{N-1}\}$.

According to Definition □, an N -ary tree is either the empty tree, \emptyset , or it is a non-empty set of nodes which consists of a root and exactly N subtrees. Clearly, the empty set contains neither a root, nor any subtrees. Therefore, the degree of each node of an N -ary tree is either zero or N .

There is subtle, yet extremely important consequence of Definition □ that often goes unrecognized. The empty tree, \emptyset , is a tree. That is, it is an object of the same type as a non-empty tree. Therefore, from the perspective of object-oriented program design, an empty tree must be an instance of some object class. It is inappropriate to use the `None` reference to represent an empty tree, since the `None` reference refers to nothing at all!

The empty trees are called *external nodes* because they have no subtrees and therefore appear at the extremities of the tree. Conversely, the non-empty trees

are called *internal nodes*.

Figure □ shows the following *tertiary* ($N=3$) trees:

$$\begin{aligned} T_a &= \{A, \emptyset, \emptyset, \emptyset\}, \\ T_b &= \{B, \{C, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \\ T_c &= \{D, \{E, \{F, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \\ &\quad \{G, \{H, \{I, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \{J, \{K, \emptyset, \emptyset, \emptyset\}, \{L, \emptyset, \emptyset, \emptyset\}, \emptyset\}, \\ &\quad \{M, \emptyset, \emptyset, \emptyset\}\}, \emptyset\}. \end{aligned}$$

In the figure, square boxes denote the empty trees and circles denote non-empty nodes. Except for the empty trees, the tertiary trees shown in the figure contain the same sets of nodes as the corresponding trees shown in Figure □.

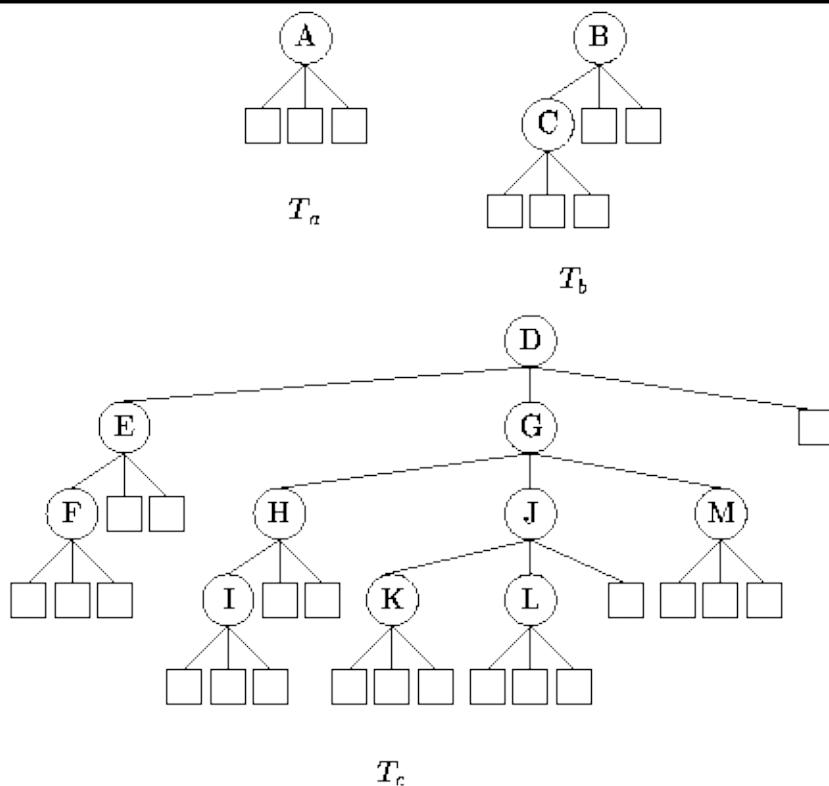


Figure: Examples of N -ary trees.

Definitions □ and □ both define trees in terms of sets. In mathematics, elements of a set are normally unordered. Therefore, we might conclude that the relative ordering of the subtrees is not important. However, most practical implementations of trees define an implicit ordering of the subtrees. Consequently, it is usual to assume that the subtrees are ordered. As a result, the

two tertiary trees, $T_1 = \{x, \{y, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}$ and $T_2 = \{x, \emptyset, \{y, \emptyset, \emptyset, \emptyset\}, \emptyset\}$, are considered to be distinct unequal trees. Trees in which the subtrees are ordered are called *ordered trees*. On the other hand, trees in which the order does not matter are called *oriented trees*. In this book, we shall assume that all trees are ordered unless otherwise specified.

Figure \square suggests that every N -ary tree contains a significant number of external nodes. The following theorem tells us precisely how many external nodes we can expect:

Theorem An N -ary tree with $n \geq 0$ internal nodes contains $(N-1)n+1$ external nodes.

extbfProof Let the number of external nodes be l . Since every node except the root (empty or not) has a parent, there must be $(n+l-1)/N$ parents in the tree since every parent has N children. Therefore, $n=(n+l-1)/N$. Rearranging this gives $l=(N-1)n+1$.

Since the external nodes have no subtrees, it is tempting to consider them to be the leaves of the tree. However, in the context of N -ary trees, it is customary to define a *leaf node* as an internal node which has only external subtrees. According to this definition, the trees shown in Figure \square have exactly the same sets of leaves as the corresponding general trees shown in Figure \square .

Furthermore, since height is defined with respect to the leaves, by having the leaves the same for both kinds of trees, the heights are also the same. The following theorem tells us something about the maximum size of a tree of a given height h :

Theorem Consider an N -ary tree T of height $h \geq 0$. The maximum number of internal nodes in T is given by

$$\frac{N^{h+1} - 1}{N - 1}.$$

extbfProof (By induction).

Base Case Consider an N -ary tree of height zero. It consists of exactly one internal node and N empty subtrees. Clearly the theorem holds for $h=0$ since

$$\left. \frac{N^{h+1} - 1}{N - 1} \right|_{h=0} = 1.$$

Inductive Hypothesis Suppose the theorem holds for $h = 0, 1, 2, \dots, k$, for some $k \geq 0$. Consider a tree of height $k+1$. Such a tree consists of a root and N subtrees each of which contains at most $\frac{(N^{k+1} - 1)/(N - 1)}{N}$ nodes. Therefore, altogether the number of nodes is at most

$$N \left(\frac{N^{k+1} - 1}{N - 1} \right) + 1 = \frac{N^{k+2} - 1}{N - 1}. \quad (9.2)$$

That is, the theorem holds for $k+1$. Therefore, by induction on k , the theorem is true for all values of h .

An interesting consequence of Theorems \square and \square is that the maximum number of external nodes in an N -ary tree of height h is given by

$$(N - 1) \left(\frac{N^{h+1} - 1}{N - 1} \right) + 1 = N^{h+1}.$$

The final theorem of this section addresses the maximum number of *leaves* in an N -ary tree of height h :

Theorem Consider an N -ary tree T of height $h \geq 0$. The maximum number of leaf nodes in T is N^h .

extbfProof (By induction).

Base Case Consider an N -ary tree of height zero. It consists of exactly one internal node which has N empty subtrees. Therefore, the one node is a leaf. Clearly the theorem holds for $h=0$ since $N^0 = 1$.

Inductive Hypothesis Suppose the theorem holds for $h = 0, 1, 2, \dots, k$, for some $k \geq 0$. Consider a tree of height $k+1$. Such a tree consists of a root and N subtrees each of which contains at most N^k leaf nodes. Therefore, altogether the number of leaves is at most $N \times N^k = N^{k+1}$. That is, the theorem holds for $k+1$. Therefore, by induction on k , the theorem is true for all values of h .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Binary Trees

In this section we consider an extremely important and useful category of tree structure--*binary trees*. A binary tree is an N -ary tree for which N is two. Since a binary tree is an N -ary tree, all of the results derived in the preceding section apply to binary trees. However, binary trees have some interesting characteristics that arise from the restriction that N is two. For example, there is an interesting relationship between binary trees and the binary number system. Binary trees are also very useful for the representation of mathematical expressions involving the binary operations such as addition and multiplication.

Binary trees are defined as follows:

Definition (Binary Tree) A *binary tree* T is a finite set of *nodes* with the following properties:

1. Either the set is empty, $T = \emptyset$; or
2. The set consists of a root, r , and exactly two distinct binary trees T_L and T_R , $T = \{r, T_L, T_R\}$.

The tree T_L is called the *left subtree* of T , and the tree T_R is called the *right subtree* of T .

Binary trees are almost always considered to be *ordered trees*. Therefore, the two subtrees T_L and T_R are called the *left* and *right* subtrees, respectively. Consider the two binary trees shown in Figure □. Both trees have a root with a single non-empty subtree. However, in one case it is the left subtree which is non-empty; in the other case it is the right subtree that is non-empty. Since the order of the subtrees matters, the two binary trees shown in Figure □ are different.

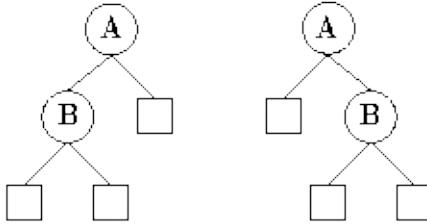


Figure: Two distinct binary trees.

We can determine some of the characteristics of binary trees from the theorems given in the preceding section by letting $N=2$. For example, Theorem \square tells us that an binary tree with $n \geq 0$ internal nodes contains $n+1$ external nodes. This result is true regardless of the shape of the tree. Consequently, we expect that the storage overhead of associated with the empty trees will be $O(n)$.

From Theorem \square we learn that a binary tree of height $h \geq 0$ has at most $2^{h+1} - 1$ internal nodes. Conversely, the height of a binary tree with n internal nodes is at least $\lceil \log_2 n + 1 \rceil - 1$. That is, the height of a binary tree with n nodes is $\Omega(\log n)$.

Finally, according to Theorem \square , a binary tree of height $h \geq 0$ has at most 2^h leaves. Conversely, the height of a binary tree with l leaves is at least $\lceil \log_2 l \rceil$. Thus, the height of a binary tree with l leaves is $\Omega(\log l)$

Tree Traversals

There are many different applications of trees. As a result, there are many different algorithms for manipulating them. However, many of the different tree algorithms have in common the characteristic that they systematically visit all the nodes in the tree. That is, the algorithm walks through the tree data structure and performs some computation at each node in the tree. This process of walking through the tree is called a *tree traversal*.

There are essentially two different methods in which to visit systematically all the nodes of a tree--*depth-first traversal* and *breadth-first traversal*. Certain depth-first traversal methods occur frequently enough that they are given names of their own: *preorder traversal*, *inorder traversal* and *postorder traversal*.

The discussion that follows uses the tree in Figure □ as an example. The tree shown in the figure is a general tree in the sense of Definition □:

$$T = \{A, \{B, \{C\}\}, \{D, \{E, \{F\}, \{G\}\}, \{H, \{I\}\}\}\} \quad (9.3)$$

However, we can also consider the tree in Figure □ to be an N -ary tree (specifically, a binary tree if we assume the existence of empty trees at the appropriate positions:

$$T = \{A, \{B, \emptyset, \{C, \emptyset, \emptyset\}\}, \{D, \{E, \{F, \emptyset, \emptyset\}, \{G, \emptyset, \emptyset\}\}, \{H, \{I, \emptyset, \emptyset\}, \emptyset\}\}\}$$

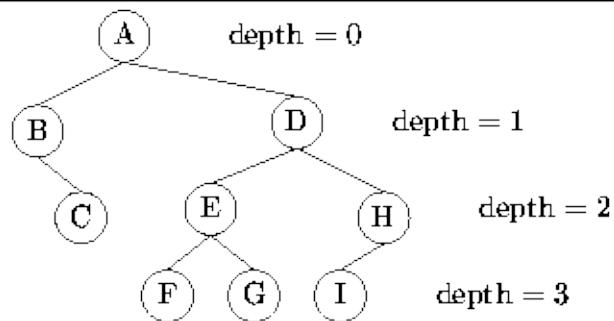
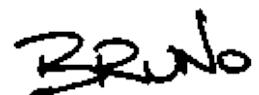


Figure: Sample tree.

-
- [Preorder Traversal](#)
 - [Postorder Traversal](#)
 - [Inorder Traversal](#)
 - [Breadth-First Traversal](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Preorder Traversal

The first depth-first traversal method we consider is called *preorder traversal* . Preorder traversal is defined recursively as follows. To do a preorder traversal of a general tree:

1. Visit the root first; and then
2. do a preorder traversal each of the subtrees of the root one-by-one in the order given.

Preorder traversal gets its name from the fact that it visits the root first. In the case of a binary tree, the algorithm becomes:

1. Visit the root first; and then
2. traverse the left subtree; and then
3. traverse the right subtree.

For example, a preorder traversal of the tree shown in Figure □ visits the nodes in the following order:

A, B, C, D, E, F, G, H, I.

Notice that the preorder traversal visits the nodes of the tree in precisely the same order in which they are written in Equation □. A preorder traversal is often done when it is necessary to print a textual representation of a tree.

Postorder Traversal

The second depth-first traversal method we consider is *postorder traversal*. In contrast with preorder traversal, which visits the root first, postorder traversal visits the root last. To do a postorder traversal of a general tree:

1. Do a postorder traversal each of the subtrees of the root one-by-one in the order given; and then
2. visit the root.

To do a postorder traversal of a binary tree

1. Traverse the left subtree; and then
2. traverse the right subtree; and then
3. visit the root.

A postorder traversal of the tree shown in Figure □ visits the nodes in the following order:

C, B, F, G, E, I, H, D, A.

Inorder Traversal

The third depth-first traversal method is *inorder traversal*. Inorder traversal only makes sense for binary trees. Whereas preorder traversal visits the root first and postorder traversal visits the root last, inorder traversal visits the root *in between* visiting the left and right subtrees:

1. Traverse the left subtree; and then
2. visit the root; and then
3. traverse the right subtree.

An inorder traversal of the tree shown in Figure □ visits the nodes in the following order:

B, C, A, F, E, G, D, I, H.



Breadth-First Traversal

Whereas the depth-first traversals are defined recursively, *breadth-first traversal* is best understood as a non-recursive traversal. The breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. Breadth-first traversal first visits all the nodes at depth zero (i.e., the root), then all the nodes at depth one, and so on. At each depth the nodes are visited from left to right.

A breadth-first traversal of the tree shown in Figure □ visits the nodes in the following order:

A, B, D, C, E, H, F, G, I.

Expression Trees

Algebraic expressions such as

$$a/b + (c - d)e \quad (9.4)$$

have an inherent tree-like structure. For example, Figure □ is a representation of the expression in Equation □. This kind of tree is called an *expression tree* .

The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (a, b, c, d , and e). The non-terminal nodes of an expression tree are the operators ($+$, $-$, \times , and \div). Notice that the parentheses which appear in Equation □ do not appear in the tree. Nevertheless, the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.

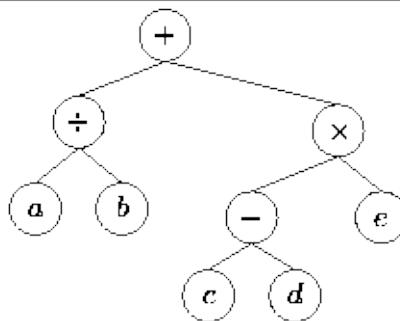


Figure: Tree representing the expression $a/b + (c-d)e$.

The common algebraic operators are either unary or binary. For example, addition, subtraction, multiplication, and division are all binary operations and negation is a unary operation. Therefore, the non-terminal nodes of the corresponding expression trees have either one or two non-empty subtrees. That is, expression trees are usually binary trees.

What can we do with an expression tree? Perhaps the simplest thing to do is to print the expression represented by the tree. Notice that an inorder traversal of the tree in Figure □ visits the nodes in the order

$a, \div, b, +, c, -, d, e.$

Except for the missing parentheses, this is precisely the order in which the symbols appear in Equation □!

This suggests that an *inorder* traversal should be used to print the expression. Consider an inorder traversal which, when it encounters a terminal node simply prints it out; and when it encounters a non-terminal node, does the following:

1. Print a left parenthesis; and then
2. traverse the left subtree; and then
3. print the root; and then
4. traverse the right subtree; and then
5. print a right parenthesis.

Applying this procedure to the tree given in Figure □ we get

$$((a \div b) + ((c - d) \times e)), \quad (9.5)$$

which, despite the redundant parentheses, represents exactly the same expression as Equation □.

-
- [Infix Notation](#)
 - [Prefix Notation](#)
 - [Postfix Notation](#)
-

Infix Notation

The algebraic expression in Equation □ is written in the usual way such mathematical expressions are written. The notation used is called *infix notation* because each operator appears *in between* its operands. As we have seen, there is a natural relationship between infix notation and inorder traversal.

Infix notation is only possible for binary operations such as addition, subtraction, multiplication, and division. Writing an operator in between its operands is possible only when it has exactly two operands. In Chapter □ we saw two alternative notations for algebraic expressions--*prefix* and *postfix*.

Prefix Notation

In prefix notation the operator is written before its operands. Therefore, in order to print the prefix expression from an expression tree, preorder traversal is done. That is, at every non-terminal node we do the following:

1. Print the root; and then
2. print a left parenthesis; and then
3. traverse the left subtree; and then
4. print a comma; and then
5. traverse the right subtree; and then
6. print a right parenthesis.

If we use this procedure to print the tree given in Figure □ we get the prefix expression

$$+ (\div(a, b), \times(-(c, d), e)). \quad (9.6)$$

While this notation may appear unfamiliar at first, consider the result obtained when we spell out the names of the operators:

```
__add__(__div__(a, b), __mul__(__sub__(c, d), e))
```

This is precisely the notation used in a Python program to invoke user defined methods `__add__`, `__sub__`, `__mul__` and `__div__`.



Postfix Notation

Since inorder traversal produces an infix expression and preorder traversal produces a prefix expression, it should not come as a surprise that postorder traversal produces a postfix expression. In a postfix expression, an operator always follows its operands. The beauty of postfix (and prefix) expressions is that parentheses are not necessary.

A simple postorder traversal of the tree in Figure 9.1 gives the postfix expression

$$a\ b\ \div\ c\ d\ -\ e\ \times\ +. \quad (9.7)$$

In Section 9.1 we saw that a postfix expression is easily evaluated using a stack. So, given an expression tree, we can evaluate the expression by doing a postorder traversal to create the postfix expression and then using the algorithm given in Section 9.1 to evaluate the expression.

In fact, it is not really necessary to first create the postfix expression before computing its value. The expression can be evaluated by making use of an *evaluation stack* during the course of the traversal as follows: When a terminal node is visited, its value is pushed onto the stack. When a non-terminal node is visited, two values are popped from the stack, the operation specified by the node is performed on those values, and the result is pushed back onto the evaluation stack. When the traversal terminates, there will be one result in the evaluation stack and that result is the value of the expression.

Finally, we can take this one step further. Instead of actually evaluating the expression, the code to compute the value of the expression is emitted. Again, a postorder traversal is done. However, now instead of performing the computation as each node is visited, the code needed to perform the evaluation is emitted. This is precisely what a compiler does when it compiles an expression such as Equation 9.7 for execution.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Implementing Trees

In this section we consider the implementation of trees including general trees, N -ary trees, and binary trees. The implementations presented have been developed in the context of the abstract data type framework presented in Chapter 1. That is, the various types of trees are viewed as classes of *containers* as shown in Figure 1.

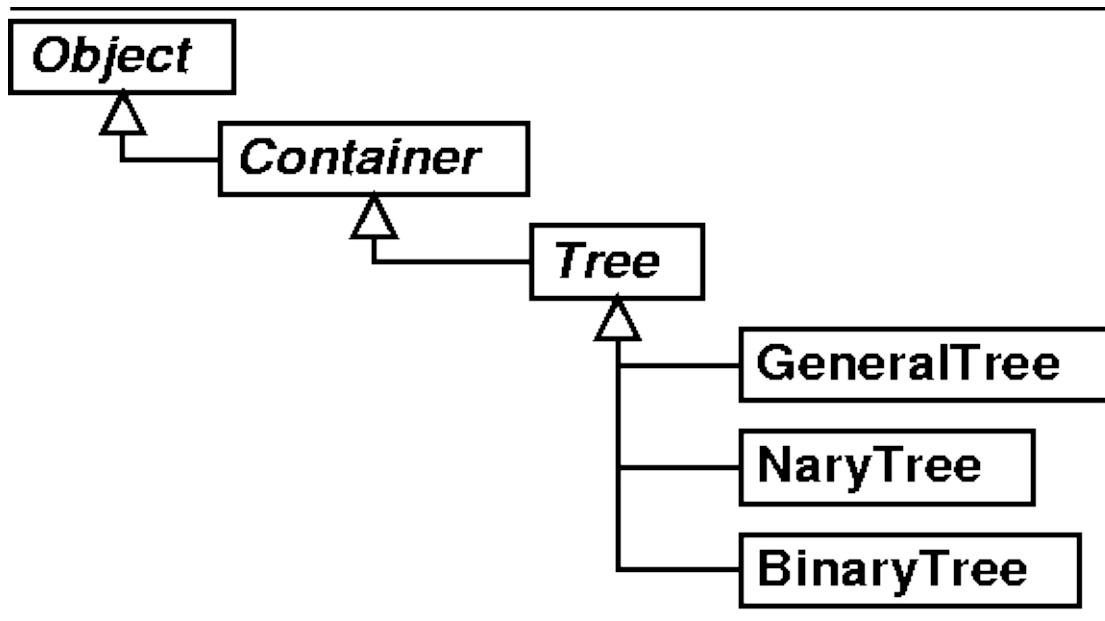


Figure: Object class hierarchy

Program 1 introduces the **Tree** class. The abstract **Tree** class extends the abstract **Container** class introduced in Program 1.

```
 1 class Tree(Container):
 2
 3     def getKey(self):
 4         pass
 5     getKey = abstractmethod(getKey)
 6
 7     key = property(
 8         fget = lambda self: self.getKey())
 9
10     def getSubtree(self, i):
11         pass
12     getSubtree = abstractmethod(getSubtree)
13
14     def getIsLeaf(self):
15         pass
16     getIsLeaf = abstractmethod(getIsLeaf)
17
18     isLeaf = property(
19         fget = lambda self: self.getIsLeaf())
20
21     def getDegree(self):
22         pass
23     getDegree = abstractmethod(getDegree)
24
25     degree = property(
26         fget = lambda self: self.getDegree())
27
28     def getHeight(self):
29         pass
30     getHeight = abstractmethod(getHeight)
31
32     height = property(
33         fget = lambda self: self.getHeight())
34
35     # ...
```

Program: Abstract Tree class.

The abstract Tree class adds the following operations to those inherited from the Container class:

key

This property accesses the object contained in the root node of a tree.

getSubtree

This method returns the i^{th} subtree of the given tree.

isEmpty

This bool-valued property is true if the root of the tree is an empty tree, i.e., an external node.

isLeaf

This bool-valued property is true if the root of the tree is a leaf node.

degree

This property accesses the degree of the root node of the tree. By definition, the degree of an external node is zero.

height

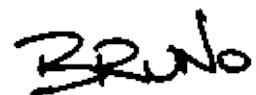
This property accesses the height of the tree. By definition, the height of an empty tree is -1.

depthFirstTraversal and breadthFirstTraversal

- [Tree Traversals](#)
 - [Tree Iterators](#)
 - [General Trees](#)
 - [N-ary Trees](#)
 - [Binary Trees](#)
 - [Binary Tree Traversals](#)
 - [Comparing Trees](#)
 - [Applications](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



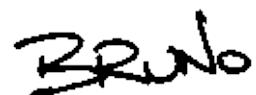
Tree Traversals

The abstract `Tree` class also provides default implementations for two methods called `depthFirstTraversal` and `breadthFirstTraversal`. These methods are like the `accept` method of the container class (see Section □). Both of these methods perform a traversal. That is, all the nodes of the tree are visited systematically. Both of these implementations access abstract properties, such as `key`, and call abstract methods, such as `getSubtree`. In effect, they are *abstract algorithms*. An abstract algorithm describes behavior in the absence of implementation!

- [Depth-First Traversal](#)
 - [Preorder, Inorder, and Postorder Traversals](#)
 - [Breadth-First Traversal](#)
 - [accept Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Depth-First Traversal

Program □ defines the `depthFirstTraversal` method of the abstract `Tree` class. In addition to `self`, the traversal method takes one argument--any object that is an instance of a subclass of the `PrePostVisitor` class defined in Program □.

```
 1  class Tree(Container):
 2
 3      def depthFirstTraversal(self, visitor):
 4          assert isinstance(visitor, PrePostVisitor)
 5          if not self.isEmpty and not visitor.isDone:
 6              visitor.preVisit(self.key)
 7              for i in xrange(self.degree):
 8                  self.getSubtree(i).depthFirstTraversal(visitor)
 9              visitor.postVisit(self.key)
10
11      # ...
```

Program: Abstract Tree class `depthFirstTraversal` method.

A `PrePostVisitor` is a visitor with three methods, `preVisit`, `inVisit`, `postVisit`, and the propderty `isDone`. During a depth-first traversal, the `preVisit` and `postVisit` methods are each called once for every node in the tree. (The `inVisit` method is provided for binary trees and is discussed in Section □).

```
1 class PrePostVisitor(Visitor):
2
3     def __init__(self):
4         super(PrePostVisitor, self).__init__()
5
6     def preVisit(self, obj):
7         pass
8
9     def inVisit(self, obj):
10        pass
11
12     def postVisit(self, obj):
13        pass
14
15     visit = inVisit
```

Program: PrePostVisitor class.

The depth-first traversal method first calls the `preVisit` method with the object in the root node. Then, it calls recursively the `depthFirstTraversal` method for each subtree of the given node. After all the subtrees have been visited, the `postVisit` method is called. Assuming that the `isEmpty`, `key`, and `getSubtree` operations all run in constant time, the total running time of the `depthFirstTraversal` method is

$$n(\mathcal{T}(\text{preVisit}) + \mathcal{T}(\text{postVisit})) + O(n),$$

where n is the number of nodes in the tree, $\mathcal{T}(\text{preVisit})$ is the running time of `preVisit`, and $\mathcal{T}(\text{postVisit})$ is the running time of `postVisit`.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Preorder, Inorder, and Postorder Traversals

Preorder, inorder, and postorder traversals are special cases of the more general depth-first traversal described in the preceding section. Rather than implement each of these traversals directly, we make use of a design pattern called *adapter*, which allows the single method to provide all the needed functionality.

Suppose we have an instance of the `PrintingVisitor` class (see Section □). The `PrintingVisitor` class extends the `Visitor` class. However, we cannot pass a `PrintingVisitor` instance to the `DepthFirstTraversal` method shown in Program □ because it expects an object that extends the `PrePostVisitor` class.

The problem is that the protocol implemented by the `PrintingVisitor` does not match the protocol expected by the `DepthFirstTraversal` method. The solution to this problem is to use an adapter. An *adapter* converts the protocol provided by one class to the protocol required by another. For example, if we want a preorder traversal, then the call to the `previsit` (made by `depthFirstTraversal`) should be mapped to the `visit` method (provided by the `PrintingVisitor`). Similarly, a postorder traversal is obtained by mapping `postVisit` to `visit`.

Programs □, □ and □ define three adapter classes--`PreOrder`, `InOrder`, and `PostOrder`. All three classes are similar: They all extend the `PrePostVisitor` class defined in Program □; all have a single `instance` attribute that refers to a `Visitor`; and all have a `__init__` method that takes a `Visitor`.

```
1 class PreOrder(PrePostVisitor):
2
3     def __init__(self, visitor):
4         super(PreOrder, self).__init__()
5         self._visitor = visitor
6
7     def preVisit(self, obj):
8         self._visitor.visit(obj)
9
10    def getIsDone(self):
11        return self._visitor.isDone
```

Program: PreOrder class.

```
1 class InOrder(PrePostVisitor):
2
3     def __init__(self, visitor):
4         super(InOrder, self).__init__()
5         self._visitor = visitor
6
7     def inVisit(self, obj):
8         self._visitor.visit(obj)
9
10    def getIsDone(self):
11        return self._visitor.isDone
```

Program: InOrder class.

```
1 class PostOrder(PrePostVisitor):
2
3     def __init__(self, visitor):
4         super(PostOrder, self).__init__()
5         self._visitor = visitor
6
7     def postVisit(self, obj):
8         self._visitor.visit(obj)
9
10    def getIsDone(self):
11        return self._visitor.isDone
```

Program: PostOrder class.

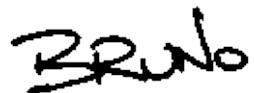
Each class provides a different interface mapping. For example, the `preVisit` method of the `PreOrder` class simply calls the `visit` method on the `_visitor` instance attribute. Notice that the adapter provides no functionality of its own--it simply forwards method calls to the `_visitor` instance as required.

The following code fragment illustrates how these adapters are used:

```
v = PrintingVisitor()
t = SomeTree()
# ...
t.depthFirstTraversal(preOrder(v))
t.depthFirstTraversal(inOrder(v))
t.depthFirstTraversal(postOrder(v))
```

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Breadth-First Traversal

Program □ defines the `breadthFirstTraversal` method of the abstract `Tree` class. As defined in Section □, a breadth-first traversal of a tree visits the nodes in the order of their depth in the tree and at each level the nodes are visited from left to right.

```
 1  class Tree(Container):
 2
 3      def breadthFirstTraversal(self, visitor):
 4          assert isinstance(visitor, Visitor)
 5          queue = QueueAsLinkedList()
 6          if not self.isEmpty:
 7              queue.enqueue(self)
 8          while not queue.isEmpty and not visitor.isDone:
 9              head = queue.dequeue()
10              visitor.visit(head.key)
11              for i in xrange(head.degree):
12                  child = head.getSubtree(i)
13                  if not child.isEmpty:
14                      queue.enqueue(child)
15
16      # ...
```

Program: Abstract Tree class `breadthFirstTraversal` method.

We have already seen in Section □ a non-recursive breadth-first traversal algorithm for N -ary trees. This algorithm makes use of a queue as follows. Initially, the root node of the given tree is enqueued, provided it is not the empty tree. Then, the following steps are repeated until the queue is empty:

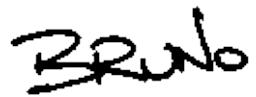
1. Remove the node at the head of the queue and call it `head`.
2. Visit the object contained in `head`.
3. Enqueue in order each non-empty subtree of `head`.

Notice that empty trees are never put into the queue. Furthermore, it should be obvious that each node of the tree is enqueued exactly once. Therefore, it is also dequeue exactly once. Consequently, the running time for the breadth-first

traversal is $nT\langle \text{visit} \rangle + O(n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





accept Method

The abstract Tree class replaces the functionality provided by the single method accept with two different kinds of traversal. Whereas the accept method is allowed to visit the nodes of a tree in any order, the tree traversals visit the nodes in two different, but well-defined orders. Consequently, we have chosen to provide a default implementation of the accept method which does a preorder traversal.

Program □ shows the implementation of the accept method of the abstract Tree class. This method uses the preOrder adapter to pass on a given visitor to the depthFirstTraversal method.

```
1 class Tree(Container):
2
3     def accept(self, visitor):
4         assert isinstance(visitor, Visitor)
5         self.depthFirstTraversal(PreOrder(visitor))
6
7     # ...
```

Program: Abstract Tree class accept method.

Tree Iterators

This section describes the implementation of an iterator which can be used to step through the contents of any tree instance. For example, suppose we have declared a variable `tree` which refers to a `BinaryTree`. Then we can view the tree instance as a container and print its contents as follows:

```
tree = BinaryTree()  
# ...  
it = iter(tree)  
while True:  
    try:  
        print it.next()  
    except StopIteration:  
        break
```

The loop above can also be written using a Python `for` loop like this:

```
for obj in tree:  
    print obj
```

Every concrete class that extends the abstract `Container` class must provide an `__iter__` method. This method returns an instance of a class that extends the abstract `Iterator` class defined in Section □. The iterator can then be used to systematically visit the contents of the associated container.

We have already seen that when we systematically visit the nodes of a tree, we are doing a tree traversal. Therefore, the implementation of the iterator must also do a tree traversal. However, there is a catch. A recursive tree traversal method such as `depthFirstTraversal` keeps track of where it is *implicitly* using the Python virtual machine stack. However, when we implement an iterator we must keep track of the state of the traversal *explicitly*. This section presents an iterator implementation which does a preorder traversal of the tree and keeps track of the current state of the traversal using a stack from Chapter □.

Program □ defines the nested class `Tree.Iterator`. The `Tree.Iterator` class extends the abstract `Iterator` class defined in Section □. The `Iterator` contains two instance attributes--`_container` and `_stack`. As shown in Program □, the `__iter__` method of the abstract `Tree` class returns a new instance of the `Tree.Iterator` class each time it is called.

```
1 class Tree(Container):
2
3     class Iterator(Iterator):
4
5         def __init__(self, tree):
6             super(Tree.Iterator, self).__init__(tree)
7             self._stack = StackAsLinkedList()
8             if not tree.isEmpty:
9                 self._stack.push(tree)
10
11         def next(self):
12             if self._stack.isEmpty:
13                 raise StopIteration
14             top = self._stack.pop()
15             i = top.degree - 1
16             while i >= 0:
17                 subtree = top.getSubtree(i)
18                 if not subtree.isEmpty:
19                     self._stack.push(subtree)
20                 i -= 1
21             return top.key
22
23     def __iter__(self):
24         return Tree.Iterator(self)
25
26     # ...
```

Program: Abstract Tree class `__iter__` method and the `Tree.Iterator` class.

-
- [__init__ Method](#)
 - [next Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



`__init__` Method

The code for the `Tree.Iterator __init__` method is given in Program □. Notice that it uses an instance of the `StackAsLinkedList` class. (The linked-list implementation of stacks is described in Section □). An empty stack can be created in constant time and the push operation can be performed in constant time. Therefore, the running time of the `__init__` method is $O(1)$.



next Method

Program [□](#) also defines the `next` method of the `Tree.Iterator` class. The iterator uses a stack to keep track nodes in the tree to be enumerated. As long as the stack is not empty, the `next` method returns the key of the tree node at the top of the stack. The `next` method pops the top tree from the stack and then pushes its subtrees onto the stack (provided that they are not empty). Notice the order is important here. In a preorder traversal, the first subtree of a node is traversed before the second subtree. Therefore, the second subtree should appear in the stack *below* the first subtree. That is why the subtrees are pushed in reverse order. The running time for `next` is $O(d)$ where d is the degree of the tree node at found at the top of the stack.

General Trees

This section outlines an implementation of general trees in the sense of Definition □. The salient features of the definition are first, that the nodes of a general tree have arbitrary degrees; and second, that there is no such thing as an empty tree.

The recursive nature of Definition □ has important implications when considering the implementation of such trees as containers. In effect, since a tree contains zero or more subtrees, when implemented as a container, we get a container which contains other containers!

Figure □ shows the approach we have chosen for implementing general trees. This figure shows how the general tree T_c in Figure □ can be stored in memory. The basic idea is that each node has associated with it a linked list of the subtrees of that node. A linked list is used because there is no *a priori* restriction on its length. This allows each node to have an arbitrary degree. Furthermore, since there are no empty trees, we need not worry about representing them. An important consequence of this is that the implementation never makes use of the None reference!

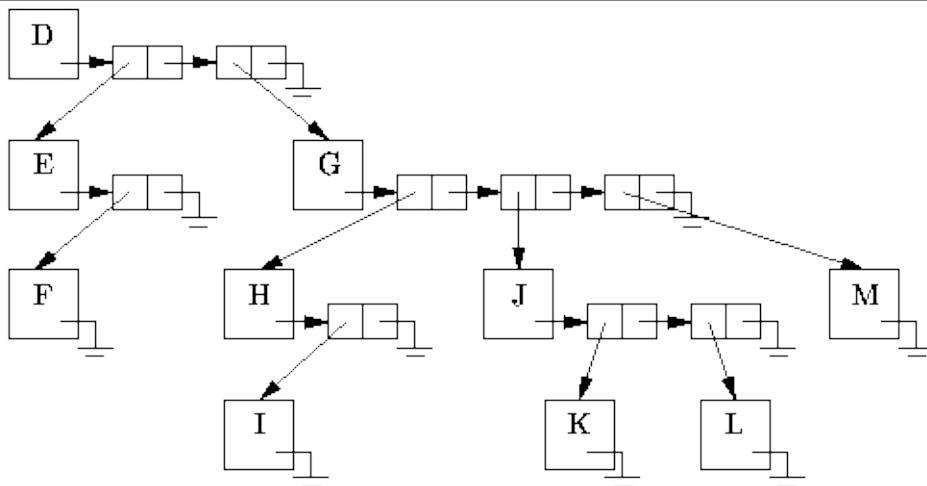


Figure: Representing general trees using linked lists.

Program □ introduces the `GeneralTree` class which is used to represent general

trees as specified by Definition □. The GeneralTree class extends the abstract Tree class introduced in Program □.

```
1  class GeneralTree(Tree):
2
3      def __init__(self, key):
4          super(GeneralTree, self).__init__()
5          self._key = key
6          self._degree = 0
7          self._list = LinkedList()
8
9      def purge(self):
10         self._list.purge()
11         self._degree = 0
12     # ...
```

Program: GeneralTree class __init__ and purge methods.

- [Instance Attributes](#)
 - [__init__ and Purge Methods](#)
 - [getKey and GetSubtree Methods](#)
 - [attachSubtree and detachSubtree Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Instance Attributes

The `GeneralTree` class defines three instance attributes--`_key`, `_degree`, and `_list`. The first, `_key`, represents the root node of the tree. The second, an integer `_degree`, records the degree of the root node of the tree. The third, `_list`, is an instance of the `LinkedList` class defined in Chapter □. It is used to contain the subtrees of the given tree.



__init__ and Purge Methods

Program □ defines the GeneralTree __init__ and Purge methods. According to Definition □, a general tree must contain at least one node--an empty tree is not allowed. Therefore, in addition to self, the __init__ method takes one argument. The __init__ method initializes the instance attributes as follows: The _key instance attribute is assigned the argument; the _degree instance attribute is set to zero; and, the _list instance attribute is assigned an empty linked list. The running time of the __init__ method is clearly O(1).

The purge method of a container normally empties the container. In this case, the container is a general tree which is not allowed to be empty. Thus, the purge method shown in Program □ discards the subtrees of the tree, but it does not discard the root. The running time of the purge method is clearly O(1).



getKey and GetSubtree Methods

Program □ defines the various GeneralTree class methods for manipulating general trees. The getKey method returns the object contained by the root node of the tree. Clearly, its running time is $O(1)$.

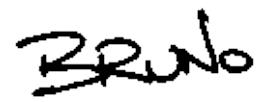
```
 1  class GeneralTree(Tree):
 2
 3      def getKey(self):
 4          return self._key
 5
 6      def getSubtree(self, i):
 7          if i < 0 or i >= self._degree:
 8              raise IndexError
 9          ptr = self._list.head
10          for j in xrange(i):
11              ptr = ptr.next
12          return ptr.datum
13
14      def attachSubtree(self, t):
15          self._list.append(t)
16          self._degree += 1
17
18      def detachSubtree(self, t):
19          self._list.extract(t)
20          self._degree -= 1
21          return t
22      # ...
```

Program: GeneralTree class getKey, getSubtree, attachSubtree and detachSubtree methods.

The getSubtree method takes as its argument an int, i , which must be between 0 and $\frac{\text{degree} - 1}{d}$, where degree is the degree of the root node of the tree. It returns the i^{th} subtree of the given tree. The getSubtree method simply takes i steps down the linked list and returns the appropriate subtree. Assuming that i is valid, the worst case running time for getSubtree is $O(d)$, where $d = \text{degree}$ is the degree of the root node of the tree.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



attachSubtree and detachSubtree Methods

Program [□](#) also defines two methods for manipulating the subtrees of a general tree. The purpose of the `attachSubtree` method is to add the specified subtree to the root of a given tree. This method takes as its argument a `GeneralTree` instance which is to be attached. The `attachSubtree` method simply appends to the linked list a pointer to the tree to be attached and then adds one to the `_degree` variable. The running time for `attachSubtree` is $O(1)$.

Similarly, the `detachSubtree` method removes the specified subtree from the given tree. This method takes as its argument the `GeneralTree` instance which is to be removed. It removes the appropriate item from the linked list and then subtracts one from the `_degree` variable. The running time for `detachSubtree` is $O(d)$ in the worst case, where $d = \text{degree}$.

N-ary Trees

We now turn to the implementation of N -ary trees as given by Definition □. According to this definition, an N -ary tree is either an empty tree or it is a tree comprised of a root and exactly N subtrees. The implementation follows the design pattern established in the preceding section. Specifically, we view an N -ary tree as a container.

Figure □ illustrates the way in which N -ary trees can be represented. The figure gives the representation of the tertiary ($N=3$) tree

$$\{A, \{B, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}.$$

The basic idea is that each node has associated with it an array of length N of pointers to the subtrees of that node. An array is used because we assume that the *arity* of the tree, N , is known *a priori*.

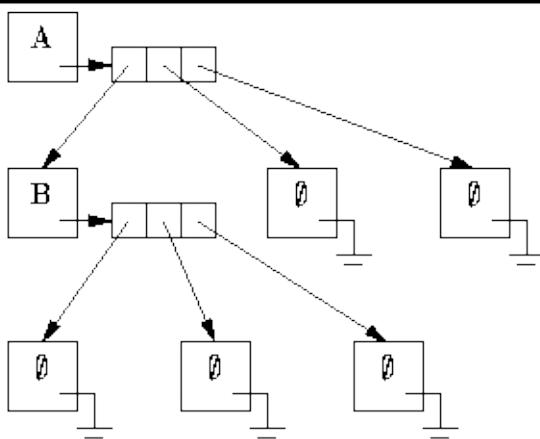


Figure: Representing N -ary trees using pointer arrays.

Notice that we explicitly represent the empty trees. That is, a separate object instance is allocated for each empty tree. Of course, an empty tree contains neither root nor subtrees.

Program □ introduces the the `NaryTree` class which represents N -ary trees as specified by Definition □. The `NaryTree` class extends the abstract `Tree` class introduced in Program □.

```
1 class NaryTree(Tree):
2
3     def __init__(self, *args):
4         super(NaryTree, self).__init__()
5         if len(args) == 1:
6             self._degree = args[0]
7             self._key = None
8             self._subtree = None
9         elif len(args) == 2:
10             self._degree = args[0]
11             self._key = args[1]
12             self._subtree = Array(self._degree)
13             for i in xrange(self._degree):
14                 self._subtree[i] = NaryTree(self._degree)
15
16     def purge(self):
17         self._key = None
18         self._subtree = None
19
20     # ...
```

Program: NaryTree class `__init__` and `purge` methods.

-
- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [getIsEmpty Method](#)
 - [getKey, AttachKey and DetachKey Methods](#)
 - [getSubtree, attachSubtree and detachSubtree Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Instance Attributes

The implementation the NaryTree class is very similar to that of the GeneralTree class. The NaryTree class definition also comprises three instance attributes--_key, _degree, and _subtree. The first, _key, represents the root node of the tree. The second, an integer _degree, records the degree of the root node of the tree. The third, _subtree, is an array of NaryTrees. This array contains the subtrees of the given tree.



`__init__` and `purge` Methods

The `__init__` method for the `NaryTree` class is given in Program □. In addition to `self`, the `__init__` method takes either one or two arguments. The first argument always an `int` that specifies the degree of the tree. If a second argument is not provided, the `__init__` method creates an empty tree. It does so by setting the `_key` instance attribute to `None`, and by setting the `_subtree` array to `None`. In this case the running time of the `__init__` method is $O(1)$.

When a second argument is provided it is taken to be an object to be stored in the tree. In this case, the `__init__` method creates a non-empty tree in which the specified object occupies the root node. According to Definition □, every internal node in an N -ary tree must have exactly N subtrees. Therefore, the `__init__` method creates and attaches N empty subtrees to the root node. In this case, the running time of the `__init__` method is $O(N)$, since N empty subtrees are created and constructed and constructing an empty N -ary tree takes $O(1)$ time.



getIsEmpty Method

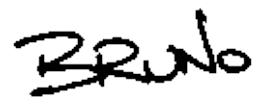
The `getIsEmpty` method indicates whether a given N -ary tree is the empty tree. The implementation of this method is given in Program □. In this implementation, the `key` instance attribute is `None` if the tree is the empty tree. Therefore, `getIsEmpty` method simply tests the `key` instance attribute. Clearly, this is a constant time operation.

```
 1  class NaryTree(Tree):
 2
 3      def getIsEmpty(self):
 4          return self._key is None
 5
 6      def getKey(self):
 7          if self.isEmpty:
 8              raise StateError
 9          return self._key
10
11      def attachKey(self, obj):
12          if not self.isEmpty:
13              raise StateError
14          self._key = obj
15          self._subtree = Array(self._degree)
16          for i in xrange(self._degree):
17              self._subtree[i] = NaryTree(self._degree)
18
19      def detachKey(self):
20          if not isLeaf:
21              raise StateError
22          result = self._key
23          self._key = None
24          self._subtree = None
25          return result
26
27      # ...
```

Program: `NaryTree` class `getKey`, `getIsEmpty`, `attachKey` and `detachKey` methods.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

getKey, AttachKey and DetachKey Methods

Program □ also defines three operations for manipulating the root of an N -ary tree. The `getKey` method returns the object contained in the root node of the tree. Clearly, this operation is not defined for the empty tree. If the tree is not empty, the running time of this method is $O(1)$.

The purpose of `attachKey` is to insert the specified object into a given N -ary tree at the root node. This operation is only defined for an empty tree. The `attachKey` method takes as its argument an object to be inserted in the root node and assigns that object to the `_key` instance attribute. Since the node is no longer empty, it must have exactly N subtrees. Therefore, N new empty subtrees are created and attached to the node. The running time is $O(N)$ since N subtrees are created, and the running time of the `__init__` method for an empty N -ary tree takes $O(1)$.

Finally, `detachKey` is used to remove the object from the root of a tree. In order that the tree which remains still conforms to Definition □, it is only permissible to remove the root from a leaf node. And upon removal, the leaf node becomes an empty tree. The implementation given in Program □ throws an exception if an attempt is made to remove the root from a non-leaf node. Otherwise, the node is a leaf which means that its N subtrees are all empty. When the root is detached, the array of subtrees is also discarded. The running time of this method is clearly $O(1)$.



getSubtree, attachSubtree and detachSubtree Methods

Program 1 defines the three methods for manipulating the subtrees of an N -ary tree. The `getSubtree` method takes as its argument an `int`, i , which must be between 0 and $N-1$. It returns the i^{th} subtree of the given tree. Note that this operation is only defined for a non-empty N -ary tree. Given that the tree is not empty, the running time is $O(1)$.

```
1  class NaryTree(Tree):
2
3      def getSubtree(self, i):
4          if self.isEmpty:
5              raise StateError
6          return self._subtree[i]
7
8      def attachSubtree(self, i, t):
9          if self.isEmpty or not self._subtree[i].isEmpty:
10             raise StateError
11             self._subtree[i] = t
12
13     def detachSubtree(self, i):
14         if self.isEmpty:
15             raise StateError
16             result = self._subtree[i]
17             self._subtree[i] = NaryTree(degree)
18             return result
19
20     # ...
```

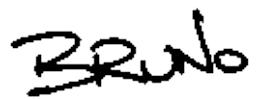
Program: `NaryTree` class `getSubtree`, `attachSubtree` and `detachSubtree` methods.

In addition to `self`, the `attachSubtree` method takes two arguments. The first is an integer i between 0 and $N-1$. The second is an `NaryTree` instance. The purpose of this method is to make the N -ary tree specified by the second argument become the i^{th} subtree of the given tree. It is only possible to attach a subtree to a non-empty node and it is only possible to attach a subtree in a place occupied by an empty subtree. If none of the exceptions are thrown, the running time of this method is simply $O(1)$.

in addition to `self`, the `detachSubtree` method takes a single argument `i` which is an integer between 0 and $N-1$. This method removes the i^{th} subtree from a given N -ary tree and returns that subtree. Of course, it is only possible to remove a subtree from a non-empty tree. Since every non-empty node must have N subtrees, when a subtree is removed it is replaced by an empty tree. Clearly, the running time is $O(1)$ if we assume that no exceptions are thrown.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Binary Trees

This section presents an implementation of binary trees in the sense of Definition □. A binary tree is essentially a N -ary tree where $N=2$. Therefore, it is possible to implement binary trees using the `NaryTree` class presented in the preceding section. However, because the `NaryTree` class implementation is a general implementation which can accommodate any value of N , it is somewhat less efficient in both time and space than an implementation which is designed specifically for the case $N=2$. Since binary trees occur quite frequently in practice, it is important to have a good implementation.

Another consequence of restricting N to two is that we can talk of the left and right subtrees of a tree. Consequently the interface provided by a binary tree class is quite different from the general interface provided by an N -ary tree class.

Figure □ shows how the binary tree given in Figure □ is represented. The basic idea is that each node of the tree contains two instance attributes that refer to the subtrees of that node. Just as we did for N -ary trees, we represent explicitly the empty trees. Since an empty tree node contains neither root nor subtrees it is represented by a structure in which all the instance attributes are `None`.

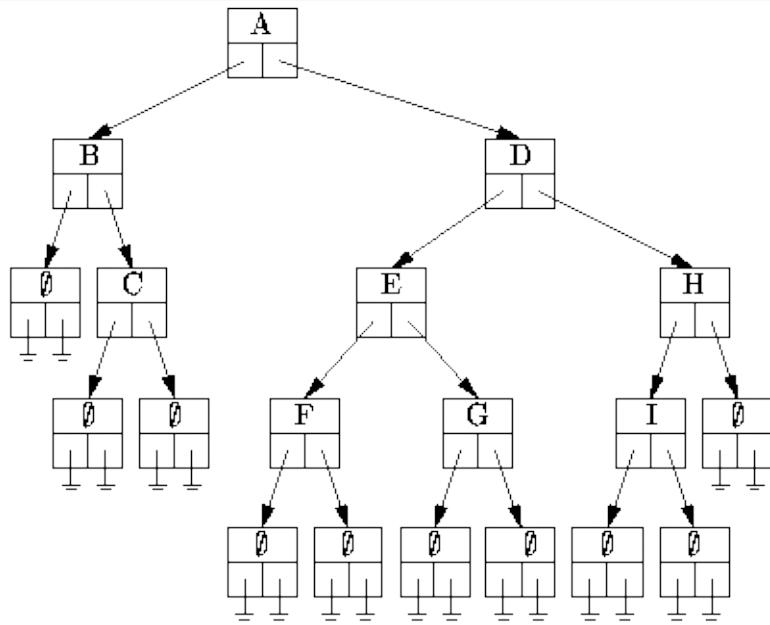


Figure: Representing binary trees.

The `BinaryTree` class is introduced in Program □. The `BinaryTree` class extends the abstract `Tree` class introduced in Program □.

```
1 class BinaryTree(Tree):
2
3     def __init__(self, *args):
4         super(BinaryTree, self).__init__()
5         if len(args) == 0:
6             self._key = None
7             self._left = None
8             self._right = None
9         elif len(args) == 1:
10             self._key = args[0]
11             self._left = BinaryTree()
12             self._right = BinaryTree()
13         elif len(args) == 3:
14             self._key = args[0]
15             self._left = args[1]
16             self._right = args[2]
17         else:
18             raise ValueError
19
20     def purge(self):
21         self._key = None
22         self._left = None
23         self._right = None
24
25     # ...
```

Program: BinaryTree class `__init__` and `purge` methods.

-
- [Instance Attributes](#)
 - [__init__ method](#)
 - [purge Method](#)
 - [right and left Properties](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Instance Attributes

The `BinaryTree` class has three instance attributes--`_key`, `_left`, and `_right`. The first, `_key`, represents the root node of the tree. The latter two, represent the left and right subtrees of the given tree. All three instance attributes are `None` if the node represents the empty tree. Otherwise, the tree must have a root and two subtrees. Consequently, all three instance attributes are non-`None` in a non-empty node.



__init__ method

Program `BT` defines the `__init__` method for the `BinaryTree` class. In addition to `self`, the `__init__` method takes either zero, one, or exactly three arguments. When no arguments are provided, the `__init__` method creates an empty binary tree. It simply sets all three instance attributes to `None`.

When one argument is provided, the third `__init__` method creates a binary tree with the specified object as its root. Since every binary tree has exactly two subtrees, the `__init__` method creates two empty subtrees and assigns them to the `_left` and `_right` instance attributes.

Finally, when exactly three arguments are provided, the `__init__` method assigns each of them to the corresponding instance attribute. In all cases, the running time of the `__init__` method is $O(1)$.

purge Method

The Purge method for the `BinaryTree` class is defined in Program □. The purpose of the Purge method is to make the tree empty. It does this by assigning `None` to all the instance attributes. Clearly, the running time of the `purge` method is $O(1)$.

right and left Properties

Program □ defines the `left` and `right` properties. These properties are associated with the methods `getLeft` and `getRight` which return the left and right subtrees of the given tree, respectively.

```
 1 class BinaryTree(Tree):
 2
 3     def getLeft(self):
 4         if self.isEmpty:
 5             raise StateError
 6         return self._left
 7
 8     left = property(
 9         fget = lambda self: self.getLeft())
10
11     def getRight(self):
12         if self.isEmpty:
13             raise StateError
14         return self._right
15
16     right = property(
17         fget = lambda self: self.getRight())
18
19     # ...
```

Program: `BinaryTree` `left` and `right` properties

Binary Tree Traversals

Program □ defines the `depthFirstTraversal` method of the `BinaryTree` class. This method supports all three tree traversal methods--preorder, inorder, and postorder. The implementation follows directly from the definitions given in Section □. The traversal is implemented using recursion. That is, the method calls itself recursively to visit the subtrees of the given node. Note that the recursion terminates properly when an empty tree is encountered since the method does nothing in that case.

```
1  class BinaryTree(Tree):
2
3      def depthFirstTraversal(self, visitor):
4          assert isinstance(visitor, PrePostVisitor)
5          if not self.isEmpty:
6              visitor.preVisit(self.key)
7              self.left.depthFirstTraversal(visitor)
8              visitor.inVisit(self.key)
9              self.right.depthFirstTraversal(visitor)
10             visitor.postVisit(self.key)
11
12     # ...
```

Program: `BinaryTree` class `DepthFirstTraversal` method.

In addition to `self`, the traversal method takes as its argument any instance of a class that extends the `prePostVisitor` interface defined in Program □. As each node is ``visited'' during the course of the traversal, the `previsit`, `inVisit`, and `postVisit` methods of the visitor are invoked on the object contained in that node.

Comparing Trees

A problem which is relatively easy to solve is determining if two trees are equivalent. Two trees are *equivalent* if they both have the same topology and if the objects contained in corresponding nodes are equal. Clearly, two empty trees are equivalent. Consider two non-empty binary trees $T_A = \{R_A, T_{AL}, T_{AR}\}$ and $T_B = \{R_B, T_{BL}, T_{BR}\}$. Equivalence of trees is given by

$$T_A \equiv T_B \iff R_A = R_B \wedge T_{AL} \equiv T_{BL} \wedge T_{AR} \equiv T_{BR}.$$

A simple, recursive algorithm suffices to test the equivalence of trees.

Since the `BinaryTree` class is ultimately derived from the `Object` class introduced in Program \square , it must provide a `_compareTo` method to compare binary trees. Recall that the `_compareTo` method is called from the `__cmp__` method of the `Object` class. To compare two objects, say `obj1` and `obj2`, the `_compareTo` method is called like this:

```
obj1._compareTo(obj2)
```

It returns a negative number if `obj1 < obj2`, a positive number if `obj1 > obj2`, and zero if `obj1 == obj2`.

So what we need is to define a *total order* relation on binary trees. Fortunately, it is possible to define such a relation for binary trees provided that the objects contained in the nodes of the trees are drawn from a totally ordered set.

Theorem Consider two binary trees T_A and T_B and the relation $<$ given by

$$\begin{aligned} T_A < T_B \iff & T_B \neq \emptyset \wedge (T_A = \emptyset \vee \\ & T_A \neq \emptyset \wedge (R_A < R_B \vee \\ & R_A = R_B \wedge (T_{AL} < T_{BL} \vee \\ & T_{AL} = T_{BL} \wedge T_{AR} < T_{BR}))) \end{aligned}$$

where T_A is either \emptyset or $T_A = \{R, T_{AL}, T_{AR}\}$ and T_B is $T_B = \{R, T_{BL}, T_{BR}\}$. The relation $<$ is a total order.

The proof of Theorem \square is straightforward albeit tedious. Essentially we need to

show the following:

- For any two distinct trees T_A and T_B , such that $T_A \neq T_B$, either $T_A < T_B$ or $T_B < T_A$.
- For any three distinct trees T_A , T_B , and T_C , if $T_A < T_B$ and $T_B < T_C$ then $T_A < T_C$.

The details of the proof are left as an exercise for the reader (Exercise \square).

Program \square gives an implementation of the `_compareTo` method for the `BinaryTree` class. This implementation is based on the total order relation $<$ defined in Theorem \square .

```
1  class BinaryTree(Tree):
2
3      def _compareTo(self, bt):
4          assert isinstance(self, bt.__class__)
5          if self.isEmpty():
6              if bt.isEmpty():
7                  return 0
8              else:
9                  return -1
10             elif bt.isEmpty():
11                 return 1
12             else:
13                 result = cmp(self._key, bt._key)
14                 if result == 0:
15                     result = cmp(self._left, bt._left)
16                     if result == 0:
17                         result = cmp(self._right, bt._right)
18                     return result
19
20     # ...
```

Program: `BinaryTree` class `CompareTo` method.

The `_compareTo` method compares the two binary trees `self` and `bt`. If they are both empty trees, `_compareTo` returns zero. If `self` is empty and `bt` is not, `_compareTo` returns -1; and if `bt` is empty and `self` is not, it returns 1.

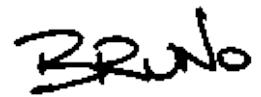
Otherwise, both trees are non-empty. In this case, `_compareTo` first compares their respective roots. If the roots are equal, then the left subtrees are compared.

Then, if the roots and the left subtrees are equal, the right subtrees are compared.

Clearly the worst-case running occurs when comparing identical trees. Suppose there are exactly n nodes in each tree. Then, the running time of the `_compareTo` method is $n\mathcal{T}_{\text{cmp}} + O(n)$, where \mathcal{T}_{cmp} is the time needed to compare the objects contained in the nodes of the trees.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Applications

Section □ shows how a stack can be used to compute the value of a postfix expression such as

$$a \ b \div \ c \ d - \ e \times +. \quad (9.8)$$

Suppose instead of evaluating the expression we are interested in constructing the corresponding expression tree. Once we have an expression tree, we can use the methods described in Section □ to print out the expression in prefix or infix notation. Thus, we have a means for translating expressions from one notation to another.

It turns out that an expression tree can be constructed from the postfix expression relatively easily. The algorithm to do this is a modified version of the algorithm for evaluating the expression. The symbols in the postfix expression are processed from left to right as follows:

1. If the next symbol in the expression is an operand, a tree comprised of a single node labeled with that operand is pushed onto the stack.
2. If the next symbol in the expression is a binary operator, the top two trees in the stack correspond to its operands. Two trees are popped from the stack and a new tree is created which has the operator as its root and the two trees corresponding to the operands as its subtrees. Then the new tree is pushed onto the stack.

After all the symbols of the expression have been processed in this fashion, the stack will contain a single tree which is the desired expression tree. Figure □ illustrates the use of a stack to construct the expression tree from the postfix expression given in Equation □.

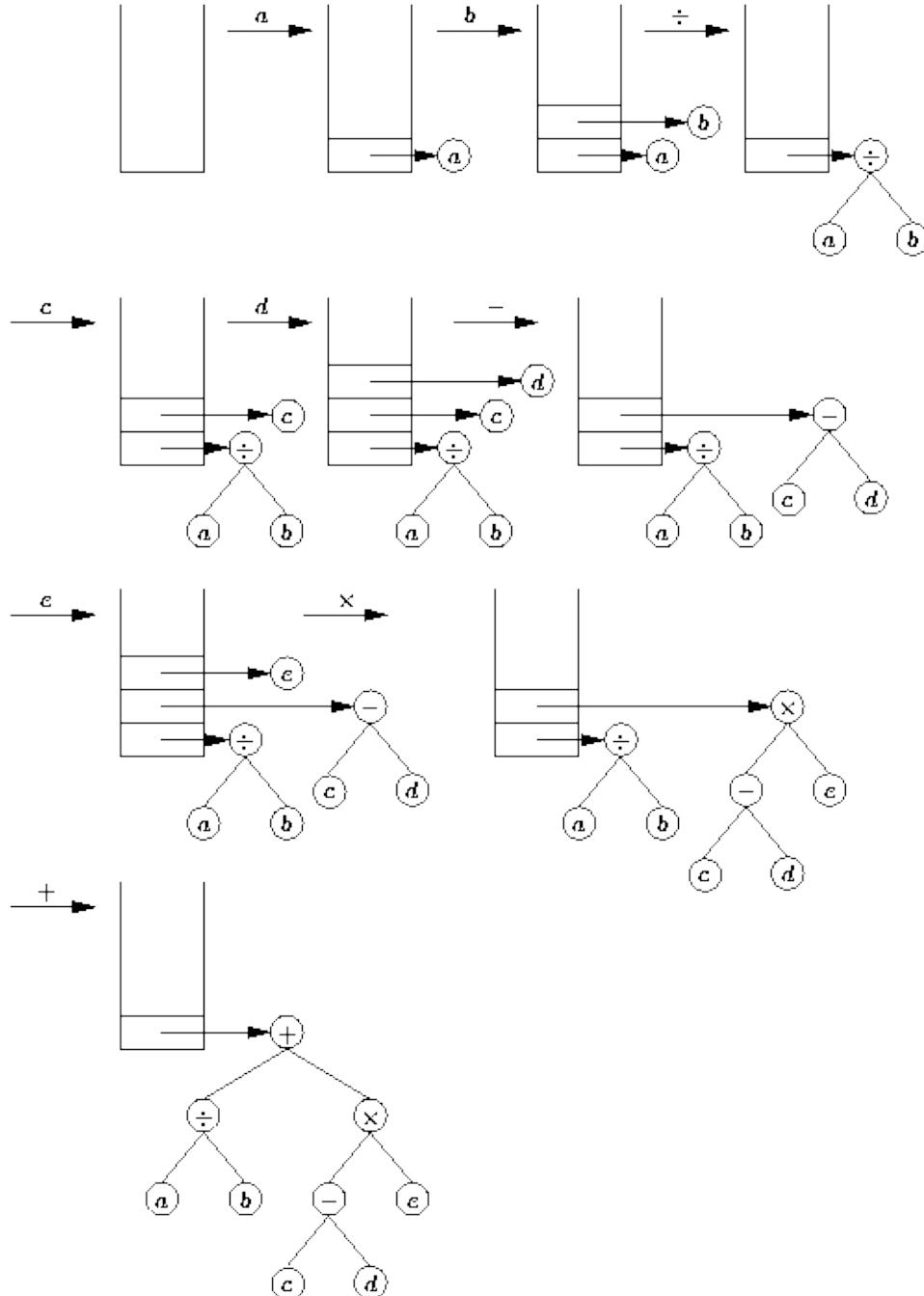


Figure: Postfix to infix conversion using a stack of trees.

- [Implementation](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Implementation

Program □ introduces the `ExpressionTree` class. This class provides a static method, called `parsePostfix`, which translates a postfix expression to an infix expression using the method described above. This method reads an expression from the input stream one word at a time. The expression is assumed to be a syntactically valid postfix expression comprised of numbers, variables, and the binary operators +, -, *, and /, each separated by blank space.

```
1  class ExpressionTree(BinaryTree):
2
3      def __init__(self, word):
4          super(ExpressionTree, self).__init__(word)
5
6      def parsePostfix(input):
7          stack = StackAsLinkedList()
8
9          for line in input.readlines():
10             for word in line.split():
11                 if word == "+" or word == "-":
12                     or word == "*" or word == "/":
13                     result = ExpressionTree(word)
14                     result.attachRight(stack.pop())
15                     result.attachLeft(stack.pop())
16                     stack.push(result)
17                 else:
18                     stack.push(ExpressionTree(word))
19             return stack.pop()
20     parsePostfix = staticmethod(parsePostfix)
21
22     # ...
```

Program: Binary tree application--postfix to infix conversion.

Since only binary operators are allowed, the resulting expression tree is a binary tree. Consequently, the `ExpressionTree` class extends the `BinaryTree` class introduced in Program □.

The main program loop, lines 9-18, reads words from the input stream one at a

time. If an operator is found, a new tree is created with the operator as its root (line 13). Next, two trees are popped from the stack and attached to the new tree which is then pushed onto the stack (lines 14-15). Otherwise, the input word is taken to be an argument. A new expression tree with that argument at its root is created and pushed onto the stack (line 18).

When the `parsePostfix` method encounters the end-of-file, its main loop terminates. The resulting expression tree is popped from the stack and returned from the `parsePostfix` method (line 19).

Program □ defines the `__str__` method for the `ExpressionTree` class. This method can be used to print out the expression represented by the tree.

```
 1  class ExpressionTree(BinaryTree):
 2
 3      class InfixVisitor(PrePostVisitor):
 4
 5          def __init__(self):
 6              self._s = ""
 7
 8          def preVisit(self, obj):
 9              self._s = self._s + "("
10
11          def inVisit(self, obj):
12              self._s = self._s + str(obj)
13
14          def postVisit(self, obj):
15              self._s = self._s + ")"
16
17          def __str__(self):
18              return self._s
19
20      def __str__(self):
21          visitor = self.InfixVisitor()
22          self.depthFirstTraversal(visitor)
23          return str(visitor)
24
25      # ...
```

Program: Binary tree application--printing infix expressions.

The `__str__` method constructs a string that represents the expression using an `InfixVisitor` which does a depth-first traversal and accumulates its result in string like this: At each non-terminal node of the expression tree, the depth-first

traversal first calls `previsit`, which appends a left parenthesis to the string. In between the traversals of the left and right subtrees, the `inVisit` method is called, which appends a textual representation of the object contained within the node to the string. Finally, after traversing the right subtree, `PostVisit` appends a right parenthesis to the string. Given the input `ab/cd-e*+`, the program constructs the expression tree as shown in Figure □, and then forms the infix expression

`((a)/(b))+(((c)-(d))*(e))).`

The running time of the `parsePostfix` method depends upon the number of symbols in the input. The running time for one iteration the main loop is $O(1)$. Therefore, the time required to construct the expression tree given n input symbols is $O(n)$. The `depthFirstTraversal` method visits each node of the expression tree exactly once and a constant amount of work is required to print a node. As a result, printing the infix expression is also $O(n)$ where n is the number of input symbols.

The output expression contains all of the input symbols plus the parentheses added by the `__str__` method. It can be shown that a valid postfix expression that contains n symbols, always has $(n-1)/2$ binary operators and $(n+1)/2$ operands (Exercise □). Hence, the expression tree contains $(n-1)/2$ non-terminal nodes and since a pair of parentheses is added for each non-terminal node in the expression tree, the output string contains $2n-1=O(n)$ symbols altogether. Therefore, the overall running time needed to translate a postfix expression comprised of n symbols to an infix expression is $O(n)$.

Exercises

1. For each tree shown in Figure □ show the order in which the nodes are visited during the following tree traversals:
 1. preorder traversal,
 2. inorder traversal (if defined),
 3. postorder traversal, and
 4. breadth-first traversal.

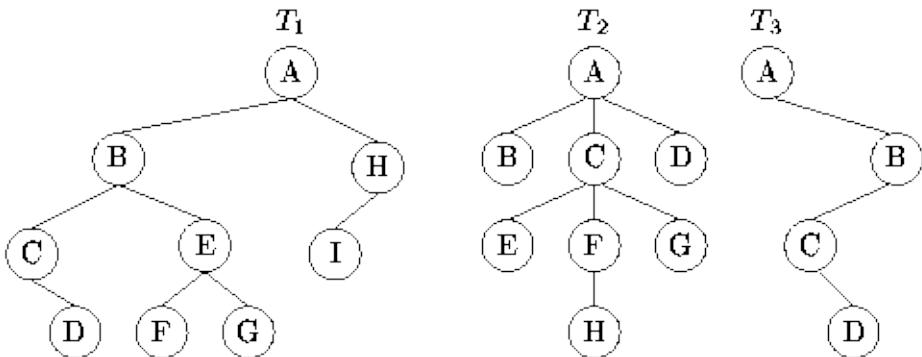


Figure: Sample trees for Exercise □.

2. Write a visitor that prints the nodes of a general tree in the format of Equation □.
3. Derive an expression for the total space needed to represent a tree of n internal nodes using each of the following classes:
 1. GeneralTree introduced in Program □,
 2. NaryTree introduced in Program □, and
 3. BinaryTree introduced in Program □.
4. A full node in a binary tree is a node with two non-empty subtrees. Let l be the number of leaf nodes in a binary tree. Show that the number of full nodes is $l-1$.
5. The generic `depthFirstTraversal` method defined in Program □ is a recursive method. Write a non-recursive depth-first traversal method that has exactly the same effect as the recursive version.
6. Program □ defines a visitor that prints using *infix* notation the expression represented by an expression tree. Write a visitor that prints the same expression in *prefix* notation with the following format:

$+(/(a,b),*(-(c,d),e)).$

7. Repeat Exercise \square , but this time write a visitor that the expression in *postfix* notation with the following format:

$ab/cd-e**.$

8. The visitor defined in Program \square prints many redundant parentheses because it does not take into consideration the precedence of the operators. Rewrite the visitor so that it prints

$a/b+(c-d)*e$

rather than

$((((a)/(b))+(((c)-(d))*(e))).$

9. Consider postfix expressions involving only binary operators. Show that if such an expression contains n symbols, it always has $(n-1)/2$ operators and $(n+1)/2$ operands.
10. Prove Theorem \square .
11. Generalize Theorem \square so that it applies to N -ary trees.
12. Consider two binary trees, $T_A = \{R_A, T_{AL}, T_{AR}\}$ and $T_B = \{R_B, T_{BL}, T_{BR}\}$ and the relation \simeq given by

$$\begin{aligned} T_A \simeq T_B \iff & (T_A = \emptyset \wedge T_B = \emptyset) \vee \\ & ((T_A \neq \emptyset \wedge T_B \neq \emptyset) \wedge \\ & ((T_{AL} \simeq T_{BL} \wedge T_{AR} \simeq T_{BR}) \vee \\ & (T_{AL} \simeq T_{BR} \wedge T_{AR} \simeq T_{BL}))). \end{aligned}$$

If $T_A \simeq T_B$, the trees are said to be *isomorphic*. Devise an algorithm to test whether two binary trees are isomorphic. What is the running time of your algorithm?

Projects

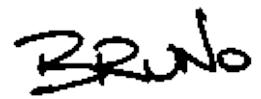
1. Devise an algorithm to compute the height of a tree. Write an implementation of your algorithm as the `getHeight` method of the abstract `Tree` class introduced in Program □.
2. Devise an algorithm to count the number of internal nodes in a tree. Write an implementation of your algorithm as the `getCount` method of the `AbstractTree` class introduced in Program □.
3. Devise an algorithm to count the number of leaves in a tree. Write an implementation of your algorithm as a method of the `AbstractTree` class introduced in Program □.
4. Devise an abstract (generic) algorithm to compare trees. (See Exercise □). Write an implementation of your algorithm as the `_compareTo` method of the abstract `Tree` class introduced in Program □.
5. The `Iterator` class introduced in Program □ does a *preorder* traversal of a tree.
 1. Write an iterator class that does a *postorder* traversal.
 2. Write an iterator class that does a *breadth-first* traversal.
 3. Write an iterator class that does an *inorder* traversal. (In this case, assume that the tree is a `BinaryTree`).
6. Complete the `GeneralTree` class introduced in Program □ by providing suitable definitions for the following operations: `getIsEmpty`, `getIsLeaf`, `getDegree`, and `_compareTo`. Write a test program and test your implementation.
7. Complete the `NaryTree` class introduced in Program □ by providing suitable definitions for the following operations: `getIsLeaf`, `getDegree`, and `_compareTo`. Write a test program and test your implementation.
8. Complete the `BinaryTree` class introduced in Program □ by providing suitable definitions for the following operations: `getIsEmpty`, `getIsLeaf`, `getDegree`, `getKey`, `attachKey`, `detachKey`, `attachLeft`, `detachLeft`, `attachRight`, `detachRight`, and `getSubtree`. Write a test program and test your implementation.
9. Write a visitor that draws a picture of a tree on the screen.
10. Design and implement an algorithm that constructs an expression tree from an *infix* expression such as

$$a/b + (c - d) * e.$$

Hint: See Project □.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Search Trees

In the preceding chapter we consider trees in which the relative positions of the nodes in the tree are unconstrained. In other words, a given item may appear anywhere in the tree. Clearly, this allows us complete flexibility in the kind of tree that we may construct. And depending on the application, this may be precisely what we need. However, if we lose track of an item, in order to find it again it may be necessary to do a complete traversal of the tree (in the worst case).

In this chapter we consider trees that are designed to support efficient search operations. In order to make it easier to search, we constrain the relative positions of the items in the tree. In addition, we show that by constraining the *shape* of the tree as well as the relative positions of the items in the tree, search operations can be made even more efficient.

- [Basics](#)
 - [Searching a Search Tree](#)
 - [Average Case Analysis](#)
 - [Implementing Search Trees](#)
 - [AVL Search Trees](#)
 - [M-Way Search Trees](#)
 - [B-Trees](#)
 - [Applications](#)
 - [Exercises](#)
 - [Projects](#)
-

Basics

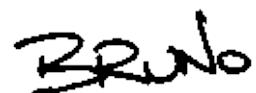
A tree which supports efficient search, insertion, and withdrawal operations is called a *search tree*. In this context the tree is used to store a finite set of keys drawn from a totally ordered set of keys K . Each node of the tree contains one or more keys and all the keys in the tree are unique, i.e., no duplicate keys are permitted.

What makes a tree into a search tree is that the keys do not appear in arbitrary nodes of the tree. Instead, there is a *data ordering criterion* which determines where a given key may appear in the tree in relation to the other keys in that tree. The following sections present two related types of search trees, M -way search trees and binary search trees.

- [M-Way Search Trees](#)
 - [Binary Search Trees](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



M-Way Search Trees

Definition (M-way Search Tree) An *M-way search tree* T is a finite set of keys. Either the set is empty, $T = \emptyset$; or the set consists of n *M-way subtrees* T_0, T_1, \dots, T_{n-1} , and $n-1$ keys, k_1, k_2, \dots, k_{n-1} ,

$$T = \{T_0, k_1, T_1, k_2, T_2, \dots, k_{n-1}, T_{n-1}\},$$

where $2 \leq n \leq M$, such that the keys and nodes satisfy the following *data ordering properties* :

1. The keys in each node are distinct and ordered, i.e., $k_i < k_{i+1}$ for $1 \leq i \leq n - 1$.
2. All the keys contained in subtree T_{i-1} are less than k_i . The tree T_{i-1} is called the *left subtree* with respect to the key k_i .
3. All the keys contained in subtree T_i are greater than k_i . The tree T_{i+1} is called the *right subtree* with respect to the key k_i .

Figure  gives an example of an *M-way search tree* for $M=4$. In this case, each of the non-empty nodes of the tree has between one and three keys and at most four subtrees. All the keys in the tree satisfy the data ordering properties.

Specifically, the keys in each node are ordered and for each key in the tree, all the keys in the left subtree with respect to the given key are less than the given key, and all the keys in the right subtree with respect to the given key are larger than the given key. Finally, it is important to note that the topology of the tree is not determined by the particular set of keys it contains.

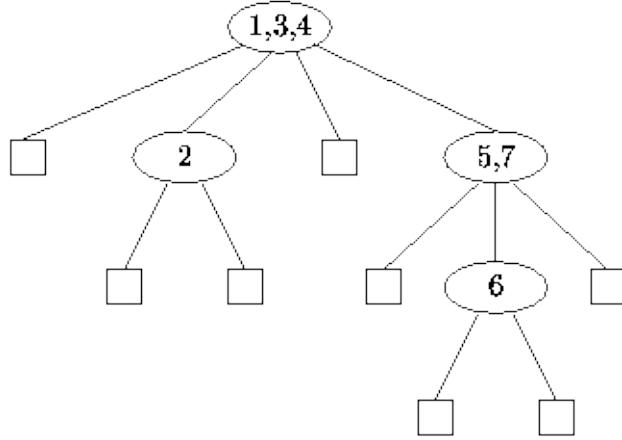


Figure: An M -way search tree.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Binary Search Trees

Just as the binary tree is an important category of N -ary trees, the *binary search tree* is an important category of M -way search trees.

Definition (Binary Search Tree) A *binary search tree* T is a finite set of keys. Either the set is empty, $T = \emptyset$; or the set consists of a root r and exactly two binary search trees T_L and T_R , $T = \{r, T_L, T_R\}$, such that the following properties are satisfied:

1. All the keys contained in left subtree, T_L , are less than r .
2. All the keys contained in the right subtree, T_R , are greater than r .

Figure □ shows an example of a binary search tree. In this case, since the nodes of the tree carry alphabetic rather than numeric keys, the ordering of the keys is alphabetic. That is, all the keys in the left subtree of a given node precede alphabetically the root of the that node, and all the keys in the right subtree of a given node follow alphabetically the root of that node. The empty trees are shown explicitly as boxes in Figure □. However, in order to simplify the graphical representation, the empty trees are often omitted from the diagrams.

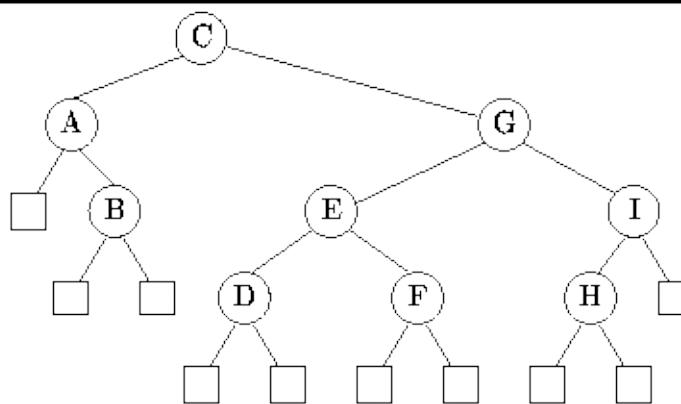


Figure: A binary search tree.

Bruno

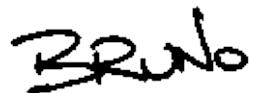
Searching a Search Tree

The main advantage of a search tree is that the data ordering criterion ensures that it is not necessary to do a complete tree traversal in order to locate a given item. Since search trees are defined recursively, it is easy to define a recursive search method.

- [Searching an M-way Tree](#)
 - [Searching a Binary Tree](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Searching an *M*-way Tree

Consider the search for a particular item, say x , in an M -way search tree. The search always begins at the root. If the tree is empty, the search fails. Otherwise, the keys contained in the root node are examined to determine if the object of the search is present. If it is, the search terminates successfully. If it is not, there are three possibilities: Either the object of the search, x , is less than k_1 , in which case subtree T_0 is searched; or x is greater than k_{n-1} , in which case subtree T_{n-1} is searched; or there exists an i such that $1 \leq i < n - 1$ for which $k_i < x < k_{i+1}$, in which case subtree T_i is searched.

Notice that when x is not found in a given node, only one of the n subtrees of that node is searched. Therefore, a complete tree traversal is not required. A successful search begins at the root and traces a downward path in the tree, which terminates at the node containing the object of the search. Clearly, the running time of a successful search is determined by the *depth* in the tree of object of the search.

When the object of the search is not in the search tree, the search method described above traces a downward path from the root which terminates when an empty subtree is encountered. In the worst case, the search path passes through the deepest leaf node. Therefore, the worst-case running time for an unsuccessful search is determined by the *height* of the search tree.



Searching a Binary Tree

The search method described above applies directly to binary search trees. As above, the search begins at the root node of the tree. If the object of the search, x , matches the root r , the search terminates successfully. If it does not, then if x is less than r , the left subtree is searched; otherwise x must be greater than r , in which case the right subtree is searched.

Figure □ shows two binary search trees. The tree T_a is an example of a particularly bad search tree because it is not really very tree-like at all. In fact, it is topologically isomorphic with a linear, linked list. In the worst case, a tree which contains n items has height $O(n)$. Therefore, in the worst case an unsuccessful search must visit $O(n)$ internal nodes.

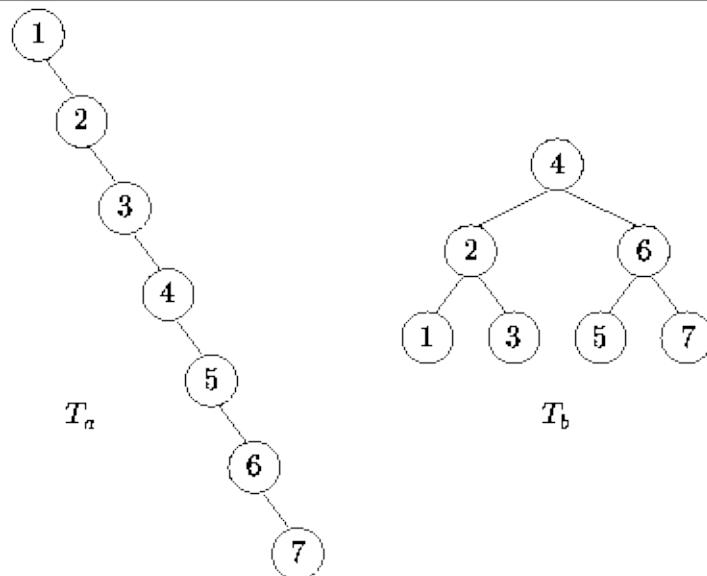


Figure: Examples of search trees.

On the other hand, tree T_b in Figure □ is an example of a particularly good binary search tree. This tree is an instance of a *perfect binary tree*.

Definition (Perfect Binary Tree) A *perfect binary tree* of height $h \geq 0$ is a

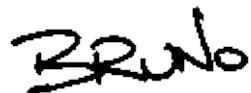
binary tree $T = \{r, T_L, T_R\}$ with the following properties:

1. If $h=0$, then $T_L = \emptyset$ and $T_R = \emptyset$.
2. Otherwise, $h>0$, in which case both T_L and T_R are both perfect binary trees of height $h-1$.

It is fairly easy to show that a perfect binary tree of height h has exactly $2^{h+1} - 1$ internal nodes. Conversely, the height of a perfect binary tree with n internal nodes is $\log_2(n + 1)$. If we have a search tree that has the shape of a perfect binary tree, then every unsuccessful search visits exactly $h+1$ internal nodes, where $h = \log_2(n + 1)$. Thus, the worst case for unsuccessful search in a perfect tree is $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

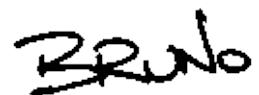


Average Case Analysis

- [Successful Search](#)
 - [Solving The Recurrence-Telescoping](#)
 - [Unsuccessful Search](#)
 - [Traversing a Search Tree](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Successful Search

When a search is successful, exactly $d+1$ internal nodes are visited, where d is the depth in the tree of object of the search. For example, if the object of the search is at the root which has depth zero, the search visits just one node--the root itself. Similarly, if the object of the search is at depth one, two nodes are visited, and so on. We shall assume that it is equally likely for the object of the search to appear in any node of the search tree. In that case, the *average* number of nodes visited during a successful search is $\bar{d} + 1$, where \bar{d} is the average of the depths of the nodes in a given tree. That is, given a binary search tree with $n > 0$ nodes,

$$\bar{d} = \frac{1}{n} \sum_{i=1}^n d_i,$$

where d_i is the depth of the i^{th} node of the tree.

The quantity $\sum_{i=1}^n d_i$ is called the *internal path length*. The internal path length of a tree is simply the sum of the depths (levels) of all the internal nodes in the tree. Clearly, the average depth of an internal node is equal to the internal path length divided by n , the number of nodes in the tree.

Unfortunately, for any given number of nodes n , there are many different possible search trees. Furthermore, the internal path lengths of the various possibilities are not equal. Therefore, to compute the average depth of a node in a tree with n nodes, we must consider all possible trees with n nodes. In the absence of any contrary information, we shall assume that all trees having n nodes are equiprobable and then compute the average depth of a node in the average tree containing n nodes.

Let $I(n)$ be the average internal path length of a tree containing n nodes. Consider first the case of $n=1$. Clearly, there is only one binary tree that contains one node--the tree of height zero. Therefore, $I(1)=0$.

Now consider an arbitrary tree, $T_n(l)$, having $n \geq 1$ internal nodes altogether, l of which are found in its left subtree, where $0 \leq l < n$. Such a tree consists of a root,

the left subtree with l internal nodes and and a right subtree with $n-l-1$ internal nodes. The average internal path length for such a tree is the sum of the average internal path length of the left subtree, $I(l)$, plus that of the right subtree, $I(n-l-1)$, plus $n-1$ because the nodes in the two subtrees are one level lower in $T_n(l)$.

In order to determine the average internal path length for a tree with n nodes, we must compute the average of the internal path lengths of the trees $T_n(l)$ averaged over all possible sizes, l , of the (left) subtree, $0 \leq l < n$.

To do this we consider an ordered set of n distinct keys, $k_0 < k_1 < \dots < k_{n-1}$. If we select the i^{th} key, k_i , to be the root of a binary search tree, then there are l keys, k_0, k_1, \dots, k_{i-1} , in its left subtree and $n-l-1$ keys, $k_{i+1}, k_{i+2}, \dots, k_{n-1}$ in its right subtree.

If we assume that it is equally likely for any of the n keys to be selected as the root, then all the subtree sizes in the range $0 \leq l < n$ are equally likely. Therefore, the average internal path length for a tree with $n \geq 1$ nodes is

$$\begin{aligned} I(n) &= \frac{1}{n} \sum_{i=0}^{n-1} (I(i) + I(n-i-1) + n-1), \quad n > 1 \\ &= \frac{2}{n} \sum_{i=0}^{n-1} I(i) + n - 1. \end{aligned}$$

Thus, in order to determine $I(n)$ we need to solve the recurrence

$$I(n) = \begin{cases} 0 & n = 1, \\ \frac{2}{n} \sum_{i=0}^{n-1} I(i) + n - 1 & n > 1. \end{cases} \quad (10.1)$$

To solve this recurrence we consider the case $n > 1$ and then multiply Equation \square by n to get

$$nI(n) = 2 \sum_{i=0}^{n-1} I(i) + n^2 - n. \quad (10.2)$$

Since this equation is valid for any $n > 1$, by substituting $n-1$ for n we can also write

$$(n-1)I(n-1) = 2 \sum_{i=0}^{n-2} I(i) + n^2 - 3n + 2, \quad (10.3)$$

which is valid for $n > 2$. Subtracting Equation 10.3 from Equation 10.2 gives

$$nI(n) - (n-1)I(n-1) = 2I(n-1) + 2n - 2,$$

which can be rewritten as

$$I(n) = \frac{(n+1)I(n-1) + 2n - 2}{n}. \quad (10.4)$$

Thus, we have shown the solution to the recurrence in Equation 10.2 is the same as the solution of the recurrence

$$I(n) = \begin{cases} 0 & n = 1, \\ 1 & n = 2, \\ ((n+1)I(n-1) + 2n - 2)/n & n > 2. \end{cases} \quad (10.5)$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Solving The Recurrence-Telescoping

This section presents a technique for solving recurrence relations such as Equation □ called *telescoping*. The basic idea is this: We rewrite the recurrence formula so that a similar functional form appears on both sides of the equal sign. For example, in this case, we consider $n > 2$ and divide both sides of Equation □ by $n+1$ to get

$$\frac{I(n)}{n+1} = \frac{I(n-1)}{n} + \frac{2}{n} - \frac{4}{n(n+1)}.$$

Since this equation is valid for any $n > 2$, we can write the following series of equations:

$$\frac{I(n)}{n+1} = \frac{I(n-1)}{n} + \frac{2}{n} - \frac{4}{n(n+1)}, \quad n > 2 \quad (10.6)$$

$$\frac{I(n-1)}{n} = \frac{I(n-2)}{n-1} + \frac{2}{n-1} - \frac{4}{(n-1)n}, \quad n-1 > 2$$

$$\frac{I(n-2)}{n-1} = \frac{I(n-3)}{n-2} + \frac{2}{n-2} - \frac{4}{(n-2)(n-1)}, \quad n-2 > 2$$

⋮

$$\frac{I(n-k)}{n-k+1} = \frac{I(n-k-1)}{n-k} + \frac{2}{n-k} - \frac{4}{(n-k)(n-k+1)}, \quad n-k > 2$$

⋮

$$\frac{I(3)}{4} = \frac{I(2)}{3} + \frac{2}{3} - \frac{4}{3 \cdot 4} \quad (10.7)$$

Each subsequent equation in this series is obtained by substituting $n-1$ for n in the preceding equation. In principle, we repeat this substitution until we get an expression on the right-hand-side involving the base case. In this example, we stop at $n-k-1=2$.

Because Equation □ has a similar functional form on both sides of the equal sign, when we add Equation □ through Equation □ together, most of the terms cancel leaving

$$\begin{aligned}
\frac{I(n)}{n+1} &= \frac{I(2)}{3} + 2 \sum_{i=3}^n \frac{1}{i} - 4 \sum_{i=3}^n \frac{1}{i(i+1)}, \quad n > 2 \\
&= 2 \sum_{i=1}^n \frac{1}{i} - 4 \sum_{i=1}^n \frac{1}{i(i+1)} \\
&= 2H_n - 4n/(n+1),
\end{aligned}$$

where H_n is the n^{th} harmonic number. In Section □ it is shown that $H_n \approx \ln n + \gamma$, where $\gamma \approx 0.577215$ is called *Euler's constant*. Thus, we get that the average internal path length of the average binary search tree with n internal nodes is

$$\begin{aligned}
I(n) &= 2(n+1)H_n - 4n \\
&\approx 2(n+1)(\ln n + \gamma) - 4n.
\end{aligned}$$

Finally, we get to the point: The average depth of a node in the average binary search tree with n nodes is

$$\begin{aligned}
\bar{d} &= I(n)/n \\
&= 2 \left(\frac{n+1}{n} \right) H_n - 4 \\
&\approx 2 \left(\frac{n+1}{n} \right) (\ln n + \gamma) - 4 \\
&= O(\log n).
\end{aligned}$$

Unsuccessful Search

All successful searches terminate when the object of the search is found. Therefore, all successful searches terminate at an internal node. In contrast, all unsuccessful searches terminate at an external node. In terms of the binary tree shown in Figure □, a successful search terminates in one of the nodes which are drawn as a circles and an unsuccessful search terminates in one of the boxes.

The preceding analysis shows that the average number of nodes visited during a successful search depends on the *internal path length*, which is simply the sum of the depths of all the internal nodes. Similarly, the average number of nodes visited during an unsuccessful search depends on the *external path length*, which is the sum of the depths of all the external nodes. Fortunately, there is a simple relationship between the internal path length and the external path length of a binary tree.

Theorem Consider a binary tree T with n internal nodes and an internal path length of I . The external path length of T is given by

$$E = I + 2n.$$

In other words, Theorem □ says that the *difference* between the internal path length and the external path length of a binary tree with n internal nodes is $E - I = 2n$.

extbfProof (By induction).

Base Case Consider a binary tree with one internal node and internal path length of zero. Such a tree has exactly two empty subtrees immediately below the root and its external path length is two. Therefore, the theorem holds for $n=1$.

Inductive Hypothesis Assume that the theorem holds for $n = 1, 2, 3, \dots, k$ for some $k \geq 1$. Consider an arbitrary tree, T_k , that has k internal nodes. According to Theorem □, T_k has $k+1$ external nodes. Let I_k and E_k be the internal and external path length of T_k , respectively. According to the inductive hypothesis, $E_k - I_k = 2k$.

Consider what happens when we create a new tree T_{k+1} by removing an external node from T_k and replacing it with an internal node that has two empty subtrees. Clearly, the resulting tree has $k+1$ internal nodes. Furthermore, suppose the external node we remove is at depth d . Then the internal path length of T_{k+1} is $I_{k+1} = I_k + d$ and the external path length of T_{k+1} is $E_{k+1} = E_k - d + 2(d+1) = E_k + d + 2$.

The difference between the internal path length and the external path length of T_{k+1} is

$$\begin{aligned} E_{k+1} - I_{k+1} &= (E_k + d + 2) - (I_k + d) \\ &= E_k - I_k + 2 \\ &= 2(k+1). \end{aligned}$$

Therefore, by induction on k , the difference between the internal path length and the external path length of a binary tree with n internal nodes is $2n$ for all $n \geq 1$.

Since the difference between the internal and external path lengths of any tree with n internal nodes is $2n$, then we can say the same thing about the *average* internal and external path lengths averaged over all search trees. Therefore, $E(n)$, the average external path length of a binary search tree is given by

$$\begin{aligned} E(n) &= I(n) + 2n \\ &= 2(n+1)H_n - 2n \\ &\approx 2(n+1)(\ln n + \gamma) - 2n. \end{aligned}$$

A binary search tree with internal n nodes has $n+1$ external nodes. Thus, the average depth of an external node of a binary search tree with n internal nodes, \bar{e} , is given by

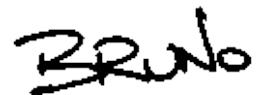
$$\begin{aligned} \bar{e} &= E(n)/(n+1) \\ &= 2H_n - 2n/(n+1) \\ &\approx 2(\ln n + \gamma) - 2n/(n+1) \\ &= O(\log n). \end{aligned}$$

These very nice results are the *raison d'être* for binary search trees. What they say is that the average number of nodes visited during either a successful or an unsuccessful search in the average binary search tree having n nodes is $O(\log n)$.

We must remember, however, that these results are premised on the assumption that all possible search trees of n nodes are equiprobable. It is important to be aware that in practice this may not always be the case.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Traversing a Search Tree

In Section □, the inorder traversal of a binary tree is defined as follows:

1. Traverse the left subtree; and then
2. visit the root; and then
3. traverse the right subtree.

It should not come as a surprise that when an *inorder traversal* of a binary search tree is done, the nodes of the tree are visited *in order*!

In an inorder traversal the root of the tree is visited after the entire left subtree has been traversed and in a binary search tree everything in the left subtree is less than the root. Therefore, the root is visited only after all the keys less than the root have been visited.

Similarly, in an inorder traversal the root is visited before the right subtree is traversed and everything in the right subtree is greater than the root. Hence, the root is visited before all the keys greater than the root are visited. Therefore, by induction, the keys in the search tree are visited in order.

Inorder traversal is not defined for arbitrary N -ary trees--it is only defined for the case of $N=2$. Essentially this is because the nodes of N -ary trees contain only a single key. On the other hand, if a node of an M -way search tree has n subtrees, then it must contain $n-1$ keys, such that $2 < n \leq M$. Therefore, we can define *inorder traversal of an M -way tree* as follows:

To traverse a node of an M -way tree having n subtrees,

Traverse T_0 ; and then
visit k_1 ; and then
traverse T_1 ; and then
visit k_2 ; and then
traverse T_2 ; and then
⋮

2n-2.

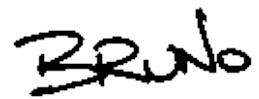
visit k_{n-1} ; and then

2n-1.

traverse T_{n-1} .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementing Search Trees

Since search trees are designed to support efficient searching, it is only appropriate that they extend the `SearchableContainer` class. Recall from Section □ that the searchable container class includes the operations `__contains__`, `find`, `insert`, and `withdraw`.

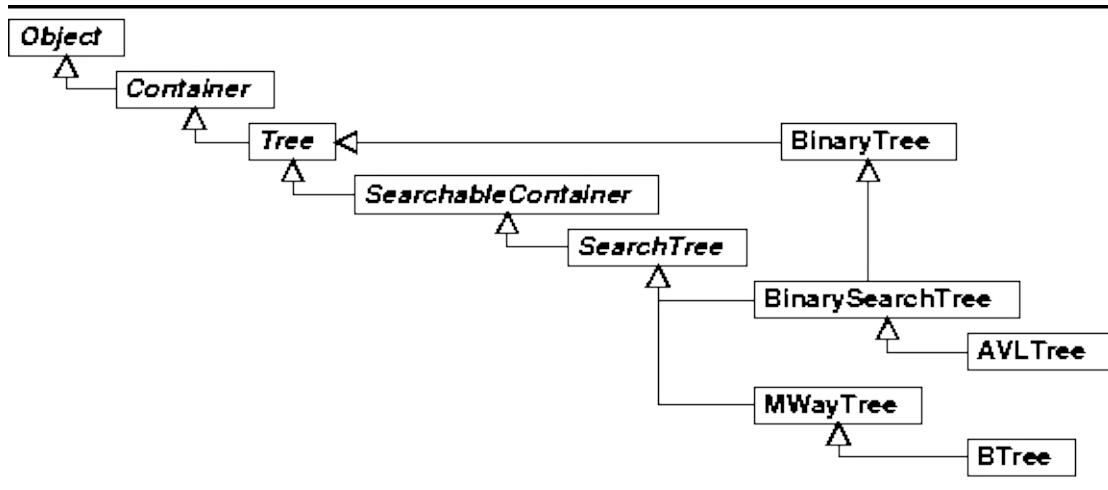


Figure: Object class hierarchy

Program □ defines the `SearchTree` class. The abstract `SearchTree` class extends the abstract `Tree` class introduced in Program □ as well as the and the abstract `SearchableContainer` class defined in Program □.

```
 1 class SearchTree(Tree, SearchableContainer):
 2
 3     def __init__(self):
 4         super(SearchTree, self).__init__()
 5
 6     def getMin(self):
 7         pass
 8     getMin = abstractmethod(getMin)
 9
10     min = property(
11         fget = lambda self: self.getMin())
12
13     def getMax(self):
14         pass
15     getMax = abstractmethod(getMax)
16
17     max = property(
18         fget = lambda self: self.getMax())
```

Program: Abstract SearchTree class min and max properties.

In addition, two properties are defined--min and max. The min property uses the getMin method to return the object contained in the search tree having the smallest key. Similarly, the max uses the getMax method to returns the contained object having the largest key.

-
- [Binary Search Trees](#)
 - [Inserting Items in a Binary Search Tree](#)
 - [Removing Items from a Binary Search Tree](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Binary Search Trees

The class `BinarySearchTree` introduced in Program □ represents binary search trees. Since binary trees and binary search trees are topologically similar, the `BinarySearchTree` class extends the `BinaryTree` introduced in Program □. In addition, because it represents search trees, the `BinarySearchTree` class extends the abstract `SearchTree` class defined in Program □.

```
1  class BinarySearchTree(BinaryTree, SearchTree):
2
3      def __init__(self):
4          super(BinarySearchTree, self).__init__()
5
6      # ...
```

Program: `BinarySearchTree` class `__init__` method.

-
- [Instance Attributes](#)
 - [find Method](#)
 - [getMin Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Instance Attributes

The `BinarySearchTree` class inherits the three instance attributes `_key`, `_left`, and `_right` from the `BinaryTree` class. The first refers to any object instance, and the latter two are `BinaryTree` instances which are the subtrees of the given tree. All three instance attributes are `None` if the node represents the empty tree. Otherwise, the tree must have a root and two subtrees. Therefore, all three instance attributes are non-`None` in an internal node.



find Method

Program □ gives the code for the `find` method of the `BinarySearchTree` class. In addition to `self`, the `find` method takes as its argument an object. The purpose of the method is to search the tree for an object in the tree that matches the argument. If a match is found, `find` returns the matching object. Otherwise, `find` returns `None`.

```

1  class BinarySearchTree(BinaryTree, SearchTree):
2
3      def find(self, obj):
4          if self.isEmpty:
5              return None
6          diff = cmp(obj, self._key)
7          if diff == 0:
8              return self._key
9          elif diff < 0:
10              return self._left.find(obj)
11          elif diff > 0:
12              return self._right.find(obj)
13
14      def getMin(self):
15          if self.isEmpty:
16              return None
17          elif self._left.isEmpty:
18              return self._key
19          else:
20              return self._left.min
21
22      # ...

```

Program: `BinarySearchTree` class `find` and `getMin` methods.

The recursive `find` method starts its search at the root and descends one level in the tree for each recursive call. At each level at most one object comparison is made (line 6). The worst case running time for a search is

$$nT(\text{cmp}) + O(n),$$

where $\mathcal{T}^{\langle \text{comp} \rangle}$ is the time to compare two objects and n is the number of internal nodes in the tree. The same asymptotic running time applies for both successful and unsuccessful searches.

The average running time for a successful search is $(\bar{d} + 1)\mathcal{T}^{\langle \text{comp} \rangle} + O(\bar{d})$, where $\bar{d} = 2(n + 1)H_n/n - 4$ is the average depth of an internal node in a binary search tree. If $\mathcal{T}^{\langle \text{comp} \rangle} = O(1)$, the average time of a successful search is $O(\log n)$.

The average running time for an unsuccessful search is $\bar{e}\mathcal{T}^{\langle \text{comp} \rangle} + O(\bar{e})$, where $\bar{e} = 2H_n - 4n/(n + 1)$ is the average depth of an external node in a binary search tree. If $\mathcal{T}^{\langle \text{comp} \rangle} = O(1)$, the average time of an unsuccessful search is $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





getMin Method

Program [□](#) also shows a recursive implementation of the `getMin` method of the `BinarySearchTree` class. It follows directly from the data ordering property of search trees that to find the node containing the smallest key in the tree, we start at the root and follow the chain of left subtrees until we get to the node that has an empty left subtree. The key in that node is the smallest in the tree. Notice that no object comparisons are necessary to identify the smallest key in the tree.

The running time analysis of the `getMin` method follows directly from that of the `find` method. The worst case running time of `getMin` is $O(n)$ and the average running time is $O(\log n)$, where n is the number of internal nodes in the tree.

Inserting Items in a Binary Search Tree

The simplest way to insert an item into a binary search tree is to pretend that the item is already in the tree and then follow the path taken by the `find` method to determine where the item would be. Assuming that the item is not already in the tree, the search will be unsuccessful and will terminate at an external, empty node. That is precisely where the item to be inserted is placed!

- [insert and attachKey Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





insert and attachKey Methods

The `insert` method of the `BinarySearchTree` class is defined in Program □. This method takes as its argument the object which is to be inserted into the binary search tree. It is assumed in this implementation that duplicate keys are not permitted. That is, all of the keys contained in the tree are unique.

```
 1  class BinarySearchTree(BinaryTree, SearchTree):
 2
 3      def insert(self, obj):
 4          if self.isEmpty:
 5              self.attachKey(obj)
 6          else:
 7              diff = cmp(obj, self._key)
 8              if diff == 0:
 9                  raise ValueError
10              elif diff < 0:
11                  self._left.insert(obj)
12              elif diff > 0:
13                  self._right.insert(obj)
14          self.balance()
15
16      def attachKey(self, obj):
17          if not self.isEmpty:
18              raise StateError
19          self._key = obj
20          self._left = BinarySearchTree()
21          self._right = BinarySearchTree()
22
23      def balance(self):
24          pass
25
26      # ...
```

Program: `BinarySearchTree` class `insert`, `attachKey` and `balance` methods.

The `insert` method behaves like the `find` method until it arrives at an external, empty node. Once the empty node has been found, it is transformed into an internal node by calling the `attachKey` method. `attachKey` works as follows: The object being inserted is assigned to the `_key` instance attribute and two new

empty binary trees are attached to the node.

Notice that after the insertion is done, the `balance` method is called. However, as shown in Program □, the `BinarySearchTree.balance` method does nothing. (Section □ describes the class `AVLTree` which is derived from the `BinarySearchTree` class and which inherits the `insert` method but overrides the `balance` operation).

The asymptotic running time of the `insert` method is the same as that of `find` for an unsuccessful search. That is, in the worst case the running time is $nT(\text{comp}) + O(n)$ and the average case running time is

$$\bar{e}T(\text{comp}) + O(\bar{e}),$$

where $\bar{e} = 2H_n - 2n/(n+1)$ is the average depth of an external node in a binary search tree with n internal nodes. When $T(\text{comp}) = O(1)$, the worst case running time is $O(n)$ and the average case is $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Removing Items from a Binary Search Tree

When removing an item from a search tree, it is imperative that the tree that remains satisfies the data ordering criterion. If the item to be removed is in a leaf node, then it is fairly easy to remove that item from the tree since doing so does not disturb the relative order of any of the other items in the tree.

For example, consider the binary search tree shown in Figure □ (a). Suppose we wish to remove the node labeled 4. Since node 4 is a leaf, its subtrees are empty. When we remove it from the tree, the tree remains a valid search tree as shown in Figure □ (b).

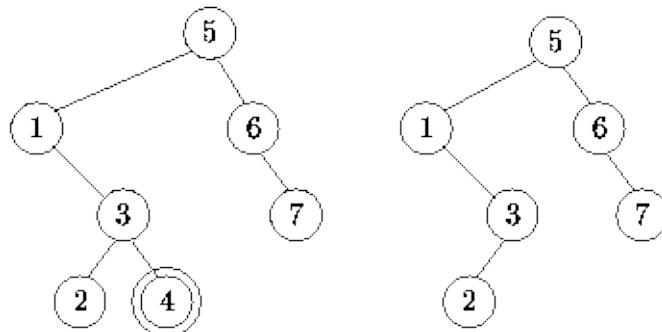


Figure: Removing a leaf node from a binary search tree.

To remove a non-leaf node, we move it down in the tree until it becomes a leaf node since a leaf node is easily deleted. To move a node down we swap it with another node which is further down in the tree. For example, consider the search tree shown in Figure □ (a). Node 1 is not a leaf since it has an empty left subtree but a non-empty right subtree. To remove node 1, we swap it with the smallest key in its right subtree, which in this case is node 2, Figure □ (b). Since node 1 is now a leaf, it is easily deleted. Notice that the resulting tree remains a valid search tree, as shown in Figure □ (c).

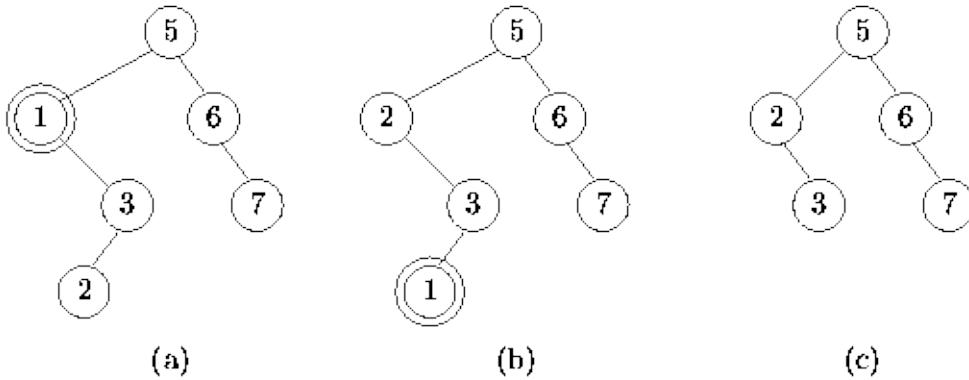


Figure: Removing a non-leaf node from a binary search tree.

To move a non-leaf node down in the tree, we either swap it with the smallest key in the right subtree or with the largest one in the left subtree. At least one such swap is always possible, since the node is a non-leaf and therefore at least one of its subtrees is non-empty. If after the swap, the node to be deleted is not a leaf, the we push it further down the tree with yet another swap. Eventually, the node must reach the bottom of the tree where it can be deleted.

-
- [withdraw Method](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



withdraw Method

Program □ gives the code for the withdraw method of the BinarySearchTree class. The withdraw method takes as its argument the object instance to be removed from the tree. The algorithm first determines the location of the object to be removed and then removes it according to the procedure described above.

```
1  class BinarySearchTree(BinaryTree, SearchTree):
2
3      def withdraw(self, obj):
4          if self.isEmpty():
5              raise KeyError
6          diff = cmp(obj, self._key)
7          if diff == 0:
8              if not self._left.isEmpty():
9                  max = self._left.max
10                 self._key = max
11                 self._left.withdraw(max)
12             elif not self._right.isEmpty():
13                 min = self._right.min
14                 self._key = min
15                 self._right.withdraw(min)
16             else:
17                 self.detachKey()
18             elif diff < 0:
19                 self._left.withdraw(obj)
20             elif diff > 0:
21                 self._right.withdraw(obj)
22             self.balance()
23
24     # ...
```

Program: BinarySearchTree class withdraw method.

AVL Search Trees

The problem with binary search trees is that while the average running times for search, insertion, and withdrawal operations are all $O(\log n)$, any one operation is still $O(n)$ in the worst case. This is so because we cannot say anything in general about the shape of the tree.

For example, consider the two binary search trees shown Figure □. Both trees contain the same set of keys. The tree T_a is obtained by starting with an empty tree and inserting the keys in the following order

$$1, 2, 3, 4, 5, 6, 7.$$

The tree T_b is obtained by starting with an empty tree and inserting the keys in this order

$$4, 2, 6, 1, 3, 5, 7.$$

Clearly, T_b is a better tree search tree than T_a . In fact, since T_b is a *perfect binary tree*, its height is $\log_2(n+1) - 1$. Therefore, all three operations, search, insertion, and withdrawal, have the same worst case asymptotic running time, $O(\log n)$.

The reason that T_b is better than T_a is that it is the more *balanced* tree. If we could ensure that the search trees we construct are balanced then the worst-case running time of search, insertion, and withdrawal, could be made logarithmic rather than linear. But under what conditions is a tree *balanced*?

If we say that a binary tree is balanced if the left and right subtrees of every node have the same height, then the only trees which are balanced are the perfect binary trees. A perfect binary tree of height h has exactly $2^{h+1} - 1$ internal nodes. Therefore, it is only possible to create perfect trees with n nodes for $n = 1, 3, 7, 15, 31, 63, \dots$. Clearly, this is an unsuitable balance condition because it is not possible to create a balanced tree for every n .

What are the characteristics of a good *balance condition* ?

1. A good balance condition ensures that the height of a tree with n nodes is $O(\log n)$.
2. A good balance condition can be maintained efficiently. That is, the additional work necessary to balance the tree when an item is inserted or deleted is $O(1)$.

Adelson-Velskii and Landis[◇] were the first to propose the following balance condition and show that it has the desired characteristics.

Definition (AVL Balance Condition) An empty binary tree is *AVL balanced*. A non-empty binary tree, $T = \{r, T_L, T_R\}$, is AVL balanced if both T_L and T_R are AVL balanced and

$$|h_L - h_R| \leq 1,$$

where h_L is the height of T_L and h_R is the height of T_R .

Clearly, all perfect binary trees are AVL balanced. What is not so clear is that heights of all trees that satisfy the AVL balance condition are logarithmic in the number of internal nodes.

Theorem The height, h , of an AVL balanced tree with n internal nodes satisfies

$$\log_2(n + 1) + 1 \leq h \leq 1.440 \log(n + 2) - 0.328.$$

extbf{Proof} The lower bound follows directly from Theorem \square . It is in fact true for all binary trees regardless of whether they are AVL balanced.

To determine the upper bound, we turn the problem around and ask the question, what is the minimum number of internal nodes in an AVL balanced tree of height h ?

Let T_h represent an AVL balanced tree of height h which has the smallest possible number of internal nodes, say N_h . Clearly, T_h must have at least one subtree of height $h-1$ and that subtree must be T_{h-1} . To remain AVL balanced, the other subtree can have height $h-1$ or $h-2$. Since we want the smallest number of

internal nodes, it must be T_{h-2} . Therefore, the number of internal nodes in T_h is $N_h = N_{h-1} + N_{h-2} + 1$, where $h \geq 2$.

Clearly, T_0 contains a single internal node, so $N_0 = 1$. Similarly, T_1 contains exactly two nodes, so $N_1 = 2$. Thus, N_h is given by the recurrence

$$N_h = \begin{cases} 1 & h = 0, \\ 2 & h = 1, \\ N_{h-1} + N_{h-2} + 1 & h \geq 2. \end{cases} \quad (10.8)$$

The remarkable thing about Equation \square is its similarity with the definition of *Fibonacci numbers* (Equation \square). In fact, it can easily be shown by induction that

$$N_h \geq F_{h+2} - 1$$

for all $h \geq 0$, where F_k is the k^{th} Fibonacci number.

Base Cases

$$\begin{aligned} N_0 &= 1, & F_2 &= 1 \implies N_0 \geq F_2 - 1, \\ N_1 &= 2, & F_3 &= 2 \implies N_1 \geq F_3 - 1. \end{aligned}$$

Inductive Hypothesis Assume that $N_h \geq F_{h+2} - 1$ for $h = 0, 1, 2, \dots, k$. Then

$$\begin{aligned} N_{k+1} &= N_k + N_{k-1} + 1 \\ &\geq F_{k+2} - 1 + F_{k+1} - 1 + 1 \\ &\geq F_{k+3} - 1 \\ &\geq F_{(k+1)+2} - 1. \end{aligned}$$

Therefore, by induction on k , $N_h \geq F_{h+2} - 1$, for all $h \geq 0$.

According to Theorem \square , the Fibonacci numbers are given by

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n),$$

where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Furthermore, since $\hat{\phi} \approx -0.618$, $|\hat{\phi}^n/\sqrt{5}| < 1$.

Therefore,

$$\begin{aligned} N_h \geq F_{h+2} - 1 &\Rightarrow N_h \geq \phi^{h+2}/\sqrt{5} - 2 \\ &\Rightarrow \sqrt{5}(N_h + 2) \geq \phi^{h+2} \\ &\Rightarrow \log_\phi(\sqrt{5}(N_h + 2)) \geq h + 2 \\ &\Rightarrow h \leq \log_\phi(N_h + 2) + \log_\phi \sqrt{5} - 2 \\ &\Rightarrow h \approx 1.440 \log_2(N_h + 2) - 0.328 \end{aligned}$$

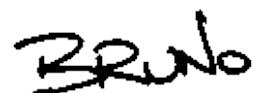
This completes the proof of the upper bound.

So, we have shown that the AVL balance condition satisfies the first criterion of a good balance condition--the height of an AVL balanced tree with n internal nodes is $\Theta(\log n)$. What remains to be shown is that the balance condition can be efficiently maintained. To see that it can, we need to look at an implementation.

- [Implementing AVL Trees](#)
 - [Inserting Items into an AVL Tree](#)
 - [Removing Items from an AVL Tree](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementing AVL Trees

Having already implemented a binary search tree class, `BinarySearchTree`, we can make use of much of the existing code to implement an AVL tree class. Program □ introduces the `AVLTree` class which extends the `BinarySearchTree` class introduced in Program □. The `AVLTree` class inherits most of its functionality from the binary tree class. In particular, it uses the inherited `insert` and `withdraw` methods! However, the inherited `balance`, `attachKey` and `detachKey` methods are overridden and a number of new methods are declared.

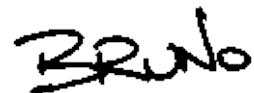
```
1  class AVLTree(BinarySearchTree):
2
3      def __init__(self):
4          super(AVLTree, self).__init__()
5          self._height = -1
6
7      # ...
```

Program: `AVLTree` class `__init__` method.

-
- [Instance Attributes](#)
 - [__init__ Method](#)
 - [adjustHeight and getHeight Methods and balanceFactor Property](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Instance Attributes

Program `□` indicates that an additional instance attribute is added in the `AVLTree` class. This turns out to be necessary because we need to be able to determine quickly, i.e., in $O(1)$ time, that the AVL balance condition is satisfied at a given node in the tree. In general, the running time required to compute the height of a tree containing n nodes is $O(n)$. Therefore, to determine whether the AVL balance condition is satisfied at a given node, it is necessary to traverse completely the subtrees of the given node. But this cannot be done in constant time.

To make it possible to verify the AVL balance condition in constant time, the instance attribute `_height` has been added. Thus, every node in an `AVLTree` keeps track of its own height. In this way it is possible for the `height` property to run in constant time--all it needs to do is to return the value of the `_height` instance attribute. And this makes it possible to test whether the AVL balanced condition satisfied at a given node in constant time.

`__init__` Method

An `__init__` method is shown in Program □. This `__init__` method creates an empty AVL tree. The `_height` instance attribute is set to the value -1, which is consistent with the empty tree. Notice that according to Definition □, the empty tree is AVL balanced. Therefore, the result is a valid AVL tree. Clearly, the running time of the `__init__` method is $O(1)$.



adjustHeight and getHeight Methods and balanceFactor Property

Program 1 defines the getHeight and adjustHeight methods as well as the balanceFactor property of the AVLTree class. The getHeight method simply returns the value of the _height instance attribute. The running time of this method is constant.

```
1  class AVLTree(BinarySearchTree):
2
3      def getHeight(self):
4          return self._height
5
6      def adjustHeight(self):
7          if self.isEmpty:
8              self._height = -1
9          else:
10              self._height = 1 + max(
11                  self._left._height, self._right._height)
12
13     def getBalanceFactor(self):
14         if self.isEmpty:
15             return 0
16         else:
17             return self._left._height - self._right._height
18
19     balanceFactor = property(
20         fget = lambda self: self.getBalanceFactor())
21
22     # ...
```

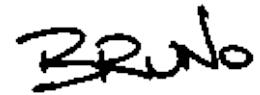
Program: AVLTree class adjustHeight and getHeight methods and balanceFactor property.

The purpose of adjustHeight is to recompute the height of a node and to update the _height instance attribute. This method must be called whenever the height of one of the subtrees changes in order to ensure the _height instance attribute is always up to date. The adjustHeight method determines the height of a node by adding one to the height of the highest subtree. The running time of this method is constant.

The `balanceFactor` property invokes the `getBalanceFactor` method to return the difference between the heights of the left and right subtrees of a given AVL tree. By Definition □, the empty node is AVL balanced. Therefore, the `balanceFactor` is zero for an empty tree. The running time of `balanceFactor` is constant.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Inserting Items into an AVL Tree

Inserting an item into an AVL tree is a two-part process. First, the item is inserted into the tree using the usual method for insertion in binary search trees. After the item has been inserted, it is necessary to check that the resulting tree is still AVL balanced and to balance the tree when it is not.

Just as in a regular binary search tree, items are inserted into AVL trees by attaching them to the leaves. To find the correct leaf we pretend that the item is already in the tree and follow the path taken by the `find` method to determine where the item should go. Assuming that the item is not already in the tree, the search is unsuccessful and terminates at an external, empty node. The item to be inserted is placed in that external node.

Inserting an item in a given external node affects potentially the heights of all of the nodes along the *access path*, i.e., the path from the root to that node. Of course, when an item is inserted in a tree, the height of the tree may increase by one. Therefore, to ensure that the resulting tree is still AVL balanced, the heights of all the nodes along the access path must be recomputed and the AVL balance condition must be checked.

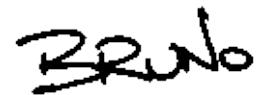
Sometimes increasing the height of a subtree does not violate the AVL balance condition. For example, consider an AVL tree $T = \{r, T_L, T_R\}$. Let h_L and h_R be the heights of T_L and T_R , respectively. Since T is an AVL tree, then $|h_L - h_R| \leq 1$. Now, suppose that $h_L = h_R + 1$. Then, if we insert an item into T_R , its height may increase by one to $h'_R = h_R + 1$. The resulting tree is still AVL balanced since $h_L - h'_R = 0$. In fact, this particular insertion actually makes the tree more balanced! Similarly if $h_L = h_R$ initially, an insertion in either subtree will not result in a violation of the balance condition at the root of T .

On the other hand, if $h_L = h_R + 1$ and an insertion of an item into the left subtree T_L increases the height of that tree to $h'_L = h_L + 1$, the AVL balance condition is no longer satisfied because $h'_L - h_R = 2$. Therefore it is necessary to change the structure of the tree to bring it back into balance.

-
- [Balancing AVL Trees](#)
 - [Single Rotations](#)
 - [Double Rotations](#)
 - [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



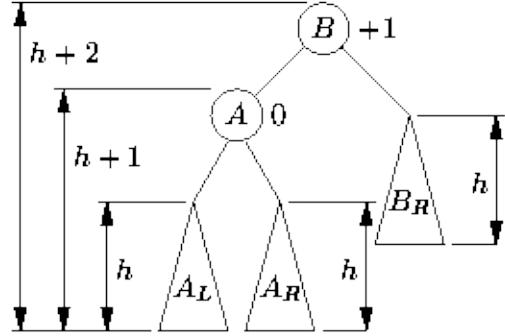
Balancing AVL Trees

When an AVL tree becomes unbalanced, it is possible to bring it back into balance by performing an operation called a *rotation*. It turns out that there are only four cases to consider and each case has its own rotation.

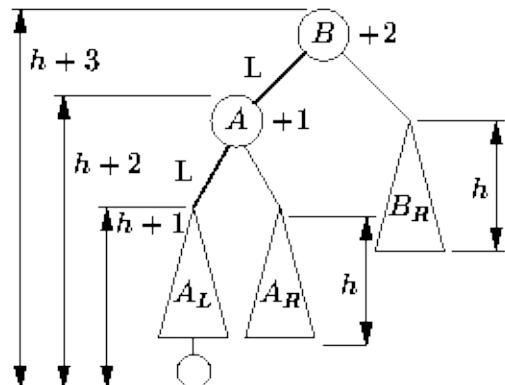


Single Rotations

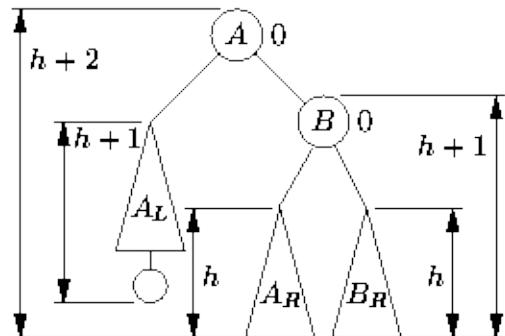
Figure □ (a) shows an AVL balanced tree. For example, the balance factor for node A is zero, since its left and right subtrees have the same height; and the balance factor of node B is +1, since its left subtree has height $h+1$ and its right subtree has height h .



(a)



(b)



(c)

Figure: Balancing an AVL tree with a single (LL) rotation.

Suppose we insert an item into A_L , the left subtree of A . The height of A_L can either increase or remain the same. In this case we assume that it increases. Then, as shown in Figure (b), the resulting tree is no longer AVL balanced. Notice where the imbalance has been manifested--node A is balanced but node B is not.

Balance can be restored by reorganizing the two nodes A and B , and the three subtrees, A_L , A_R , and B_R , as shown in Figure □(c). This is called an *LL rotation* , because the first two edges in the insertion path from node B both go to the left.

There are three important properties of the LL rotation:

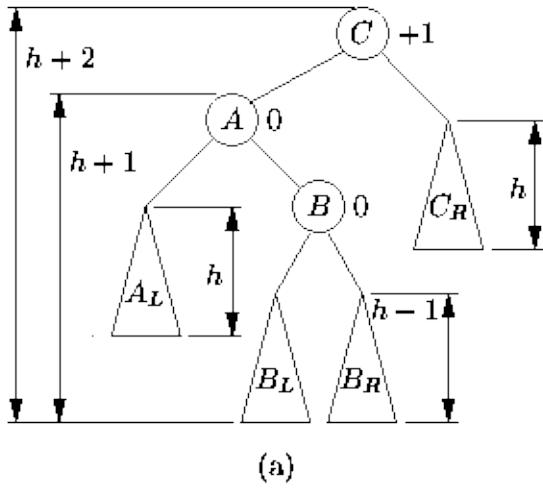
1. The rotation does not destroy the data ordering property so the result is still a valid search tree. Subtree A_L remains to the left of node A , subtree A_R remains between nodes A and B , and subtree B_R remains to the right of node B .
2. After the rotation both A and B are AVL balanced. Both nodes A and B end up with zero balance factors.
3. After the rotation, the tree has the same height it had originally. Inserting the item did not increase the overall height of the tree!

Notice, the LL rotation was called for because the root became unbalanced with a positive balance factor (i.e., its left subtree was too high) and the left subtree of the root also had a positive balance factor.

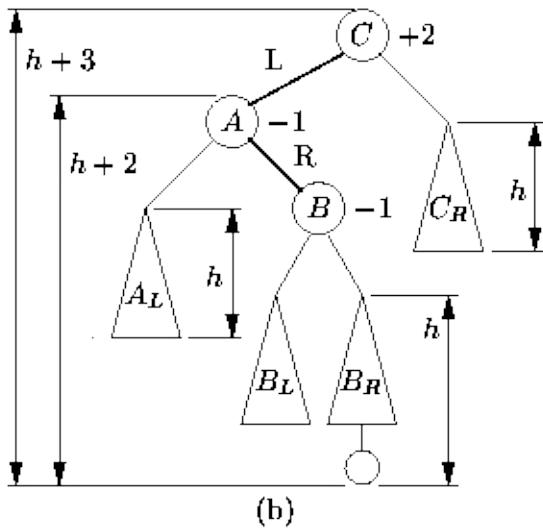
Not surprisingly, the left-right mirror image of the LL rotation is called an *RR rotation* . An RR rotation is called for when the root becomes unbalanced with a negative balance factor (i.e., its right subtree is too high) and the right subtree of the root also has a negative balance factor.

Double Rotations

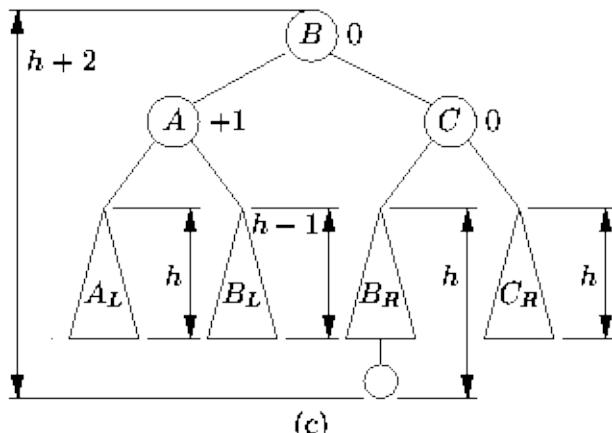
The preceding cases have dealt with access paths LL and RR. Clearly two more cases remain to be implemented. Consider the case where the root becomes unbalanced with a positive balance factor but the left subtree of the root has a negative balance factor. This situation is shown in Figure □ (b).



(a)



(b)



(c)

Figure: Balancing an AVL tree with a double (LR) rotation.

The tree can be restored by performing an RR rotation at node A , followed by an LL rotation at node C . The tree which results is shown in Figure □ (c). The LL

and RR rotations are called *single rotations*. The combination of the two single rotations is called a *double rotation* and is given the name *LR rotation* because the first two edges in the insertion path from node C both go left and then right.

Obviously, the left-right mirror image of the LR rotation is called an *RL rotation*. An RL rotation is called for when the root becomes unbalanced with a negative balance factor but the right subtree of the root has a positive balance factor. Double rotations have the same properties as the single rotations: The result is a valid, AVL-balanced search tree and the height of the result is the same as that of the initial tree.

Clearly the four rotations, LL, RR, LR, and RL, cover all the possible ways in which any one node can become unbalanced. But how many rotations are required to balance a tree when an insertion is done? The following theorem addresses this question:

Theorem When an AVL tree becomes unbalanced after an insertion, exactly one single or one double rotation is required to balance the tree.

extbfProof When an item, x , is inserted into an AVL tree, T , that item is placed in an external node of the tree. The only nodes in T whose heights may be affected by the insertion of x are those nodes which lie on the access path from the root of T to x . Therefore, the only nodes at which an imbalance can appear are those along the access path. Furthermore, when a node is inserted into a tree, either the height of the tree remains the same or the height of the tree increases by one.

Consider some node c along the access path from the root of T to x . When x is inserted, the height of c either increases by one, or remains the same. If the height of c does not change, then no rotation is necessary at c or at any node above c in the access path.

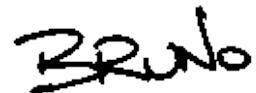
If the height of c increases then there are two possibilities: Either c remains balanced or an imbalance appears at c . If c remains balanced, then no rotation is necessary at c . However, a rotation may be needed somewhere above c along the access path.

On the other hand, if c becomes unbalanced, then a single or a double rotation must be performed at c . After the rotation is done, the height of c is the same as it was before the insertion. Therefore, no further rotation is needed above c in the access path.

Theorem \square suggests the following method for balancing an AVL tree after an insertion: Begin at the node containing the item which was just inserted and move back along the access path toward the root. For each node determine its height and check the balance condition. If the height of the current node does not increase, then the tree is AVL balanced and no further nodes need be considered. If the node has become unbalanced, a rotation is needed to balance it. After the rotation, the height of the node remains unchanged, the tree is AVL balanced and no further nodes need be considered. Otherwise, the height of the node increases by one, but no rotation is needed and we proceed to the next node on the access path.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Program □ gives the code for the `doLLRotation` method of the `AVLTree` class. This code implements the LL rotation shown in Figure □. The purpose of the `doLLRotation` method is to perform an LL rotation at the root of a given AVL tree instance.

```
 1  class AVLTree(BinarySearchTree):
 2
 3      def doLLRotation(self):
 4          if self.isEmpty:
 5              raise StateError
 6          tmp = self._right
 7          self._right = self._left
 8          self._left = self._right._left
 9          self._right._left = self._right._right
10          self._right._right = tmp
11
12          tmp = self._key
13          self._key = self._right._key
14          self._right._key = tmp
15
16          self._right.adjustHeight()
17          self.adjustHeight()
18
19          # ...
```

Program: `AVLTree` class `doLLRotation` method.

The rotation is simply a sequence of variable manipulations followed by two height adjustments. Notice the rotation is done in such a way so that the the given `AVLTree` instance remains the root of the tree. This is done so that if the tree has a parent, it is not necessary to modify the contents of the parent.

The `AVLTree` class also requires an `doRRRotation` method to implement an RR rotation. The implementation of that method follows directly from Program □. Clearly, the running time for the single rotations is $O(1)$.

Program □ gives the implementation for the `doLRRotation` method of the

AVLTree class. This double rotation is trivially implemented as a sequence of two single rotations. As above, the method for the complementary rotation is easily derived from the given code. The running time for each of the double rotation methods is also $O(1)$.

```
1  class AVLTree(BinarySearchTree):
2
3      def doLRRotation(self):
4          if self.isEmpty:
5              raise StateError
6          self._left.doRRRotation()
7          self.doLLRotation()
8
9      # ...
```

Program: AVLTree class doLRRotation method.

When an imbalance is detected, it is necessary to correct the imbalance by doing the appropriate rotation. The code given in Program □ takes care of this. The balance method tests for an imbalance using the balanceFactor property. The balance test itself takes constant time. If the node is balanced, only a constant-time height adjustment is needed.

```
1  class AVLTree(BinarySearchTree):
2
3      def balance(self):
4          self.adjustHeight()
5          if self.balanceFactor > 1:
6              if self._left.balanceFactor > 0:
7                  self.doLLRotation()
8              else:
9                  self.doLRRotation()
10             elif self.balanceFactor < -1:
11                 if self._right.balanceFactor < 0:
12                     self.doRRRotation()
13                 else:
14                     self.doRLRotation()
15
16     # ...
```

Program: AVLTree class balance method.

Otherwise, the balance method of the AVLTree class determines which of the four cases has occurred, and invokes the appropriate rotation to correct the

imbalance. To determine which case has occurred, the `balance` method calls the `balanceFactor` property at most twice. Therefore, the time for selecting the case is constant. In all only one rotation is done to correct the imbalance. Therefore, the running time of this method is $O(1)$.

The `insert` method for AVL trees is inherited from the `BinarySearchTree` class (see Program □). The `insert` method calls `attachKey` to do the actual insertion. The `attachKey` method is overridden in the `AVLTree` class as shown in Program □.

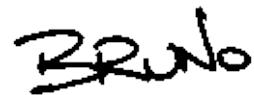
```
1  class AVLTree(BinarySearchTree):
2
3      def attachKey(self, obj):
4          if not self.isEmpty:
5              raise StateError
6          self._key = obj
7          self._left = AVLTree()
8          self._right = AVLTree()
9          self._height = 0
10
11      # ...
```

Program: `AVLTree` class `attachKey` method.

The very last thing that the `insert` method does is to call the `balance` method, which has also been overridden as shown in Program □. As a result the `insert` method adjusts the heights of the nodes along the insertion path and does a rotation when an imbalance is detected. Since the height of an AVL tree is guaranteed to be $O(\log n)$, the time for insertion is simply $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Removing Items from an AVL Tree

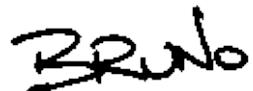
The method for removing items from an AVL tree is inherited from the `BinarySearchTree` class in the same way as AVL insertion. (See Program □). All the differences are encapsulated in the `detachKey` and `balance` methods. The `balance` method is discussed above. The `detachKey` method is defined in Program □

```
1  class AVLTree(BinarySearchTree):
2
3      def detachKey(self):
4          self._height = -1
5          return super(AVLTree, self).detachKey()
6
7      # ...
```

Program: `AVLTree` class `detachKey` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



M-Way Search Trees

As defined in Section □, an internal node of an M -way search tree contains n subtrees and $n-1$ keys, where $2 \leq n \leq M$, for some fixed value of $M \geq 2$. The preceding sections give implementations for the special case in which the fixed value of $M=2$ is assumed (binary search trees). In this section, we consider the implementation of M -way search trees for *arbitrary*, larger values of $M \gg 2$.

Why are we interested in larger values of M ? Suppose we have a very large data set--so large that we cannot get it all into the main memory of the computer at the same time. In this situation we implement the search tree in secondary storage, i.e., on disk. The unique characteristics of disk-based storage *vis-à-vis* memory-based storage make it necessary to use larger values of M in order to implement search trees efficiently.

The typical disk access time is 1-10 ms, whereas the typical main memory access time is 10-100 ns. Thus, main memory accesses are between 10000 and 1000000 times faster than typical disk accesses. Therefore to maximize performance, it is imperative that the total number of disk accesses be minimized.

In addition, disks are block-oriented devices. Data are transferred between main memory and disk in large blocks. The typical block sizes are between 512 bytes and 4096 bytes. Consequently, it makes sense to organize the data structure to take advantage of the ability to transfer entire blocks of data efficiently.

By choosing a suitably large value for M , we can arrange that one node of an M -way search tree occupies an entire disk block. If every internal node in the M -way search tree has exactly M children, we can use Theorem □ to determine the height of the tree:

$$h \geq \lceil \log_M((M-1)n + 1) \rceil - 1, \quad (10.9)$$

where n is the number of internal nodes in the search tree. A node in an M -way search tree that has M children contains exactly $M-1$ keys. Therefore, altogether there are $K=(M-1)n$ keys and Equation □ becomes $h \geq \lceil \log_M(K + 1) \rceil - 1$. Ideally

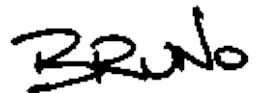
the search tree is well balanced and the inequality becomes an equality.

For example, consider a search tree which contains $K = 2\,097\,151$ keys. Suppose the size of a disk block is such that we can fit a node of size $M=128$ in it. Since each node contains at most 127 keys, at least 16513 nodes are required. In the best case, the height of the M -way search tree is only two and at most three disk accesses are required to retrieve any key! This is a significant improvement over a binary tree, the height of which is at least 20.

- [Implementing \$M\$ -Way Search Trees](#)
 - [Finding Items in an \$M\$ -Way Search Tree](#)
 - [Inserting Items into an \$M\$ -Way Search Tree](#)
 - [Removing Items from an \$M\$ -Way Search Tree](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementing M-Way Search Trees

In order to illustrate the basic ideas, this section describes an implementation of M -way search trees in main memory. According to Definition □, each internal node of an M -way search tree has n subtrees, where n is at least two and at most M . Furthermore, if a node has n subtrees, it must contain $n-1$ keys.

Figure □ shows how we can implement a single node of an M -way search tree. The idea is that we use two arrays in each node--the first holds the keys and the second contains pointers to the subtrees. Since there are at most M subtrees but only $M-1$ keys, the first element of the array of keys is not used.

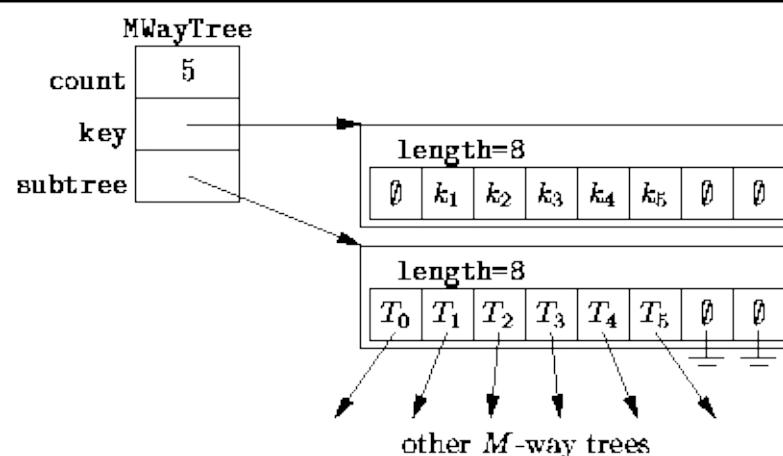


Figure: Representing a node of an M -way search tree.

-
- [Implementation](#)
 - [__init__ Method and m Property](#)
 - [Inorder Traversal](#)
-

Bruno



Implementation

Program 1 introduces the `MWayTree` class. The `MWayTree` class extends the abstract `SearchTree` class introduced in Program 1. The two instance attributes, `_key` and `_subtree`, correspond to the components of a node shown in Figure 1. (Remember, the `_count` instance attribute is inherited from the abstract `Container` base class introduced in Program 1).

```
 1  class MWayTree(SearchTree):
 2
 3      def __init__(self, m):
 4          assert m > 2
 5          super(MWayTree, self).__init__()
 6          self._key = Array(m - 1, 1)
 7          self._subtree = Array(m)
 8
 9      def getM(self):
10          return len(self._subtree)
11
12      m = property(
13          fget = lambda self: self.getM())
14
15      # ...
```

Program: `MWayTree` class `__init__` method and `m` property

The first instance attribute, `_key`, is an array used to record the keys contained in the node. The second instance attribute, `_subtree`, is an array of `MWayTree` instances which are the subtrees of the given node.

The inherited `_count` instance attribute keeps track of the number of keys contained in the node. Recall, a node which contains `_count` keys has $_{\text{count}} + 1$ subtrees. We have chosen to keep track of the number of keys of a node rather than the number of subtrees because it simplifies the coding of the algorithms by eliminating some of the special cases.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



__init__ Method and m Property

Program □ defines the `__init__` method and the `m` property of the `MWayTree` class. In addition to `self`, the `__init__` method takes a single, integer-valued argument that specifies the desired value of M . Provided M is greater than or equal to two, the `__init__` method creates two arrays, one of length $M-1$ and the other of length M . Note, a node in an M -way tree contains at most $M-1$ keys. In the implementation shown, `key[0]` is never used. Because the `__init__` method allocates arrays of length $M-1$ and M , its worst-case running time is $O(M)$.

The `m` property shown in Program □ invokes the `getM` method to return the value of M . That is, it returns the maximum number of subtrees that a node is allowed to have. This is simply given by the length of the `_subtree` array. The running time of this property is clearly $O(1)$.



Inorder Traversal

Whereas inorder traversal of an N -ary tree is *not* defined for $N > 2$, inorder traversal *is* defined for an M -way search tree: By definition, the inorder traversal of a search tree visits all the keys contained in the search tree *in order*.

Program □ is an implementation of the algorithm for depth-first traversal of an M -way search tree given in Section □. The keys contained in a given node are visited (by calling the `inVisit` method of the visitor) *in between* the appropriate subtrees of that node. That is, key k_i is visited *in between* subtrees T_{i-1} and T_i .

```

1  class MWayTree(SearchTree):
2
3      def depthFirstTraversal(self, visitor):
4          assert isinstance(visitor, PrePostVisitor)
5          if not self.isEmpty:
6              for i in xrange(self._count + 2):
7                  if i > 1:
8                      visitor.postVisit(self._key[i - 1])
9                  if i >= 1 and i <= self._count:
10                     visitor.inVisit(self._key[i])
11                  if i < self._count:
12                      visitor.preVisit(self._key[i + 1])
13                  if i <= self._count:
14                      self._subtree[i].depthFirstTraversal(visitor)
15
16      # ...

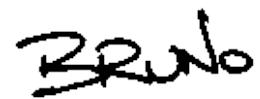
```

Program: `MWayTree` class `depthFirstTraversal` method.

In addition, the `postVisit` method is called on k_{i-1} *after* subtree T_{i-1} has been visited, and the `preVisit` method is called on k_{i+1} *before* subtree T_i is visited.

It is clear that the amount of work done at each node during the course of a depth-first traversal is proportional to the number of keys contained in that node. Therefore, the total running time for the depth-first traversal is $K(T_{\text{preVisit}} + T_{\text{inVisit}} + T_{\text{postVisit}}) + O(K)$, where K is the number of keys contained in the search tree.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

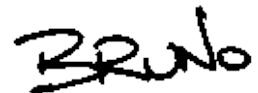
Finding Items in an *M*-Way Search Tree

Two algorithms for finding items in an *M*-way search tree are described in this section. The first is a naive implementation using linear search. The second version improves upon the first by using a binary search.

- [Linear Search](#)
 - [Binary Search](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Linear Search

Program □ gives the naive version of the `find` method of the `MWayTree` class. The `find` method takes an object and locates the item in the search tree which matches the given object.

```

1  class MWayTree(SearchTree):
2
3      def find(self, obj):
4          if self.isEmpty:
5              return None
6          i = self._count
7          while i > 0:
8              diff = cmp(obj, self._key[i])
9              if diff == 0:
10                  return self._key[i]
11              if diff > 0:
12                  break
13              i = i - 1
14          return self._subtree[i].find(obj)
15
16      # ...

```

Program: `MWayTree` class `find` method (linear search).

Consider the execution of the `find` method for a node T of a M -way search tree. Suppose the object of the search is x . Clearly, the search fails when $T = \emptyset$ (lines 4-5). In this case, `None` is returned.

Suppose $T = \{T_0, k_1, T_1, k_2, T_2, \dots, k_{n-1}, T_{n-1}\}$. The linear search on lines 6-13 considers the keys $k_{n-1}, k_{n-2}, k_{n-3}, \dots, k_1$, in that order. If a match is found, the matching object is returned immediately (lines 9-10).

Otherwise, when the main loop terminates there are three possibilities: $i=0$ and $x < k_{i+1}$; $1 \leq i \leq n-2$ and $k_i < x < k_{i+1}$; or $i=n-1$ and $k_i < x$. In all three cases, the appropriate subtree in which to continue search is T_i (line 14).

Clearly the running time of Program \square is determined by the main loop. In the worst case, the loop is executed $M-1$ times. Therefore, at each node in the search path at most $M-1$ object comparisons are done.

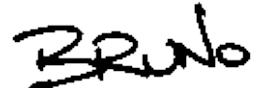
Consider an unsuccessful search in an M -way search tree. The running time of the `find` method is

$$(M - 1)(h + 1)\mathcal{T}(\text{cmp}) + O(Mh)$$

in the worst case, where h is the height of the tree and $\mathcal{T}(\text{cmp})$ is the time required to compare two objects. Clearly, the time for a successful search has the same asymptotic bound. If the tree is balanced and $\mathcal{T}(\text{cmp}) = O(1)$, then the running time of Program \square is $O(M \log_M K)$, where K is the number of keys in the tree.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Binary Search

We can improve the performance of the M -way search tree search algorithm by recognizing that since the keys are kept in a sorted array, we can do a binary search rather than a linear search. Program \square gives an alternate implementation for the `find` method of the `MWayTree` class. This method makes use of the `findIndex` method which does the actual binary search.

```

1  class MWayTree(SearchTree):
2
3      def findIndex(self, obj):
4          if self.isEmpty or obj < self._key[1]:
5              return 0
6          left = 1
7          right = self._count
8          while left < right:
9              middle = (left + right + 1) / 2
10             if obj < self._key[middle]:
11                 right = middle - 1
12             else:
13                 left = middle
14         return left
15
16     def find(self, obj):
17         if self.isEmpty:
18             return None
19         index = self.findIndex(obj)
20         if index != 0 and self._key[index] == obj:
21             return self._key[index]
22         else:
23             return self._subtree[index].find(obj)
24
25     # ...

```

Program: `MWayTree` class `findIndex` and `find` methods (binary search).

The `findIndex` method as its argument an object, say x , and returns an `int` in the range between 0 and $n-1$, where n is the number of subtrees of the given node.

The result is the largest integer i , if it exists, such that $x \geq k_i$ where k_i is the i^{th}

key. Otherwise, it returns the value 0.

`findIndex` determines its result by doing a binary search. In the worst case, $\lceil \log_2(M - 1) \rceil + 1$ iterations of the main loop (lines 8-13) are required to determine the correct index. One object comparison is done before the loop (line 4) and one comparison is done in each loop iteration (line 10). Therefore, the running time of the `findIndex` method is

$$(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}(\text{cmp}) + O(\log_2 M).$$

If $\mathcal{T}(\text{cmp}) = O(1)$, this simplifies to $O(\log M)$.

The `find` method of the `MWayTree` class does the actual search. It calls `findIndex` to determine largest integer i , if it exists, such that $x \geq k_i$ where k_i is the i^{th} key (line 19). If it turns out that $x = k_i$, then the search is finished (lines 20-21). Otherwise, `find` calls itself recursively to search subtree T_i (line 23).

Consider a search in an M -way search tree. The running time of the second version of `find` is

$$(h + 1)(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}(\text{cmp}) + O(h \log M),$$

where h is the height of the tree and regardless of whether the search is successful. If the tree is balanced and $\mathcal{T}(\text{cmp}) = O(1)$, then the running time of Program \square is simply $O((\log_2 M)(\log_M K))$, where K is the number of keys in the tree.

Inserting Items into an M -Way Search Tree

The method for inserting items in an M -way search tree follows directly from the algorithm for insertion in a binary search tree given in Section □. The added wrinkle in an M -way tree is that an internal node may contain between 1 and M -1 keys whereas an internal node in a binary tree must contain exactly one key.

Program □ gives the implementation of the `insert` method of the `MWayTree` class. In addition to `self`, this method takes as its argument the object to be inserted into the search tree.

```
 1  class MWayTree(SearchTree):
 2
 3      def insert(self, obj):
 4          if self.isEmpty:
 5              self._subtree[0] = MWayTree(self.m)
 6              self._key[1] = obj
 7              self._subtree[1] = MWayTree(self.m)
 8              self._count = 1
 9          else:
10              index = self.findIndex(obj)
11              if index != 0 and self._key[index] == obj:
12                  raise ValueError
13              if not self.isFull:
14                  i = self._count
15                  while i > index:
16                      self._key[i + 1] = self._key[i]
17                      self._subtree[i + 1] = self._subtree[i]
18                      i = i - 1
19                  self._key[index + 1] = obj
20                  self._subtree[index + 1] = MWayTree(self.m)
21                  self._count = self._count + 1
22              else:
23                  self._subtree[index].insert(obj)
24
25      # ...
```

Program: `MWayTree` class `insert` method.

The general algorithm for insertion is to search for the item to be inserted and then to insert it at the point where the search terminates. If the search terminates at an external node, that node is transformed to an internal node of the form

$\{\emptyset, x, \emptyset\}$, where x is the key just inserted (lines 5-8).

If the search terminates at an internal node, we insert the new item into the sorted list of keys at the appropriate offset. Inserting the key x in the array of keys moves all the keys larger than x and the associated subtrees to the right one position (lines 14-18). The hole in the list of subtrees is filled with an empty tree (line 19).

The preceding section gives the running time for a search in an M -way search tree as

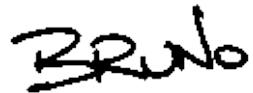
$$(h + 1)(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}_{\text{comp}} + O(h \log M),$$

where h is the height of the tree. The additional time required to insert the item into the node once the correct node has been located is $O(M)$. Therefore, the total running time for the `insert` algorithm given in Program □ is

$$(h + 1)(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}_{\text{comp}} + O(h \log M) + O(M).$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Removing Items from an M -Way Search Tree

The algorithm for removing items from an M -way search tree follows directly from the algorithm for removing items from a binary search tree given in Section □. The basic idea is that the item to be deleted is pushed down the tree from its initial position to a node from which it can be easily deleted. Clearly, items are easily deleted from leaf nodes. In addition, consider an internal node of an M -way search tree of the form

$$T = \{T_0, k_1, T_1, \dots, T_{i-1}, k_i, T_i, \dots, k_{n-1}, T_{n-1}\}.$$

If both T_{i-1} and T_i are empty trees, then the key k_i can be deleted from T by removing both k_i and T_i , say. If T_{i-1} is non-empty, k_i can be pushed down the tree by swapping it with the largest key in T_{i-1} ; and if T_i is non-empty, k_i can be pushed down the tree by swapping it with the smallest key in T_i .

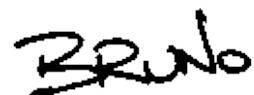
Program □ gives the code for the `withdraw` method of the `MwayTree` class. The general form of the algorithm follows that of the `withdraw` method for the `BinarySearchTree` class (Program □).

```
1 class MWayTree(SearchTree):
2
3     def withdraw(self, obj):
4         if self.isEmpty():
5             raise KeyError
6         index = self.findIndex(obj)
7         if index != 0 and self._key[index] == obj:
8             if not self._subtree[index - 1].isEmpty():
9                 max = self._subtree[index - 1].max
10                self._key[index] = max
11                self._subtree[index - 1].withdraw(max)
12            elif not self._subtree[index].isEmpty():
13                min = self._subtree[index].min
14                self._key[index] = min
15                self._subtree[index].withdraw(min)
16            else:
17                self._count = self._count - 1
18                i = index
19                while i <= self._count:
20                    self._key[i] = self._key[i + 1]
21                    self._subtree[i] = self._subtree[i + 1]
22                    i = i + 1
23                    self._key[i] = None
24                    self._subtree[i] = None
25                    if self._count == 0:
26                        self._subtree[0] = None
27                else:
28                    self._subtree[index].withdraw(obj)
29
30    # ...
```

Program: MWayTree class withdraw method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



B-Trees

Just as AVL trees are balanced binary search trees, *B-trees* are balanced M -way search trees. \diamond By imposing a *balance condition*, the shape of an AVL tree is constrained in a way which guarantees that the search, insertion, and withdrawal operations are all $O(\log n)$, where n is the number of items in the tree. The shapes of B-Trees are constrained for the same reasons and with the same effect.

Definition (B-Tree) A *B-Tree of order M* is either the empty tree or it is an M -way search tree T with the following properties:

1. The root of T has at least two subtrees and at most M subtrees.
2. All internal nodes of T (other than its root) have between $\lceil M/2 \rceil$ and M subtrees.
3. All external nodes of T are at the same level.

A B-tree of order one is clearly impossible. Hence, B-trees of order M are really only defined for $M \geq 2$. However, in practice we expect that M is large for the same reasons that motivate M -way search trees--large databases in secondary storage.

Figure \square gives an example of a B-tree of order $M=3$. By Definition \square , the root of a B-tree of order three has either two or three subtrees and the internal nodes also have either two or three subtrees. Furthermore, all the external nodes, which are shown as small boxes in Figure \square , are at the same level.

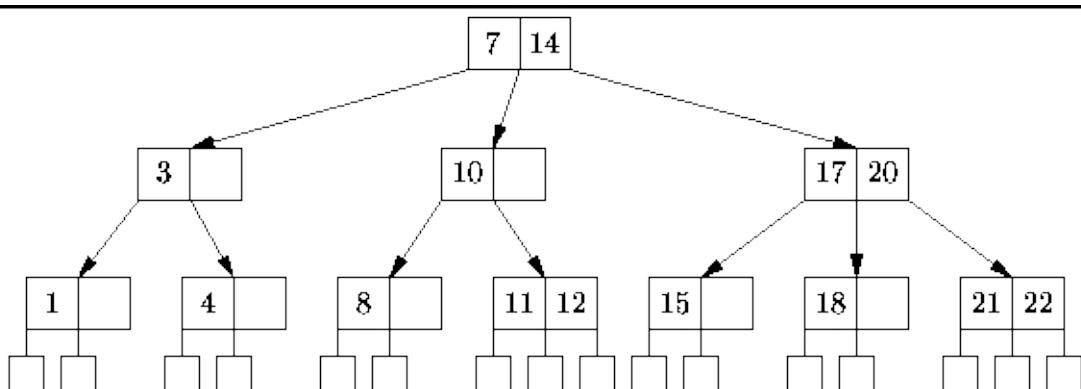


Figure: A B-tree of order 3.

It turns out that the balance conditions imposed by Definition \square are good in the same sense as the AVL balance conditions. That is, the balance condition guarantees that the height of B-trees is logarithmic in the number of keys in the tree and the time required for insertion and deletion operations remains proportional to the height of the tree even when balancing is required.

Theorem The minimum number of keys in a B-tree of order $M \geq 2$ and height $h \geq 0$ is $n_h = 2\lceil M/2 \rceil^h - 1$.

extbfProof Clearly, a B-tree of height zero contains at least one node. Consider a B-tree order M and height $h > 0$. By Definition \square , each internal node (except the root) has at least $\lceil M/2 \rceil$ subtrees. This implies the minimum number of keys contained in an internal node is $\lceil M/2 \rceil - 1$. The minimum number of keys a level zero is 1; at level one, $2(\lceil M/2 \rceil - 1)$; at level two, $2\lceil M/2 \rceil(\lceil M/2 \rceil - 1)$; at level three, $2\lceil M/2 \rceil^2(\lceil M/2 \rceil - 1)$; and so on.

Therefore the minimum number of keys in a B-tree of height $h > 0$ is given by the summation

$$\begin{aligned} n_h &= 1 + 2(\lceil M/2 \rceil - 1) \sum_{i=0}^{h-1} \lceil M/2 \rceil^i \\ &= 1 + 2(\lceil M/2 \rceil - 1) \left(\frac{\lceil M/2 \rceil^h - 1}{\lceil M/2 \rceil - 1} \right) \\ &= 2\lceil M/2 \rceil^h - 1. \end{aligned}$$

A corollary of Theorem \square is that the height, h , of a B-tree containing n keys is given by

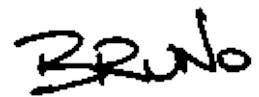
$$h \leq \log_{\lceil M/2 \rceil}((n+1)/2).$$

Thus, we have shown that a B-tree satisfies the first criterion of a good balance condition--the height of B-tree with n internal nodes is $O(\log n)$. What remains to be shown is that the balance condition can be efficiently maintained during insertion and withdrawal operations. To see that it can, we need to look at an implementation.

-
- [Implementing B-Trees](#)
 - [Inserting Items into a B-Tree](#)
 - [Removing Items from a B-Tree](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Implementing B-Trees

Having already implemented the M -way search tree class, `MWayTree`, we can make use of much the existing code to implement a B-tree class. Program □ introduces the `BTree` class which extends the class `MWayTree` class introduced in Program □ With the exception of the two methods which modify the tree, `insert` and `withdraw`, the `BTree` class inherits all its functionality from the M -way tree class. Of course, the `insert` and `withdraw` methods need to be redefined in order to ensure that every time tree is modified the tree which results is a B-tree.

```
 1  class BTree(MWayTree):
 2
 3      def __init__(self, m):
 4          super(BTree, self).__init__(m)
 5          self._parent = None
 6
 7      def attachSubtree(self, i, t):
 8          self._subtree[i] = t
 9          t._parent = self
10
11      # ...
```

Program: `BTree` class `__init__` and `attachSubtree` methods.

-
- [Instance Attributes](#)
 - [__init__ and attachSubtree Methods](#)
-

Instance Attributes

To simplify the implementation of the algorithms, a `_parent` instance attribute has been added. The `_parent` instance attribute refers to the `BTree` node which is the parent of the given node. Whereas the `_subtree` instance attribute of the `MWayTree` class allows an algorithm to move down the tree; the `_parent` instance attribute admits movement up the tree. Since the root of a tree has no parent, the `_parent` instance attribute of the root node is assigned the value `None`.



`__init__` and `attachSubtree` Methods

Program `BT` defines a `__init__` method that takes a single `int` argument M (in addition to `self`) and creates an empty B-tree of order M . By default, the `_parent` instance attribute is `null`.

The `attachSubtree` method is used to attach a new subtree to a given node. In addition to `self`, the `attachSubtree` routine takes an integer, i , and an M -way tree (which must be a B-tree instance), and makes it the i^{th} subtree of the given node. Notice that this method also modifies the `_parent` instance attribute in the attached node.

Inserting Items into a B-Tree

The algorithm for insertion into a B-Tree begins as do all the other search tree insertion algorithms: To insert item x , we begin at the root and conduct a search for it. Assuming the item is not already in the tree, the unsuccessful search will terminate at a leaf node. This is the point in the tree at which the x is inserted.

If the leaf node has fewer than $M-1$ keys in it, we simply insert the item in the leaf node and we are done. For example, consider a leaf node with $n < M$ subtrees and $n-1$ keys of the form

$$T = \{T_0, k_1, T_1, k_2, T_2, \dots, k_{n-1}, T_{n-1}\}.$$

For every new key inserted in the node, a new subtree is required too. In this case because T is a leaf, all its subtrees are empty trees. Therefore, when we insert item x , we really insert the pair of items (x, \emptyset) . Suppose the key to be inserted falls between k_i and k_{i+1} , i.e., $k_i < x < k_{i+1}$. When we insert the pair (x, \emptyset) into T we get the new leaf T' given by

$$T' = \{T_0, k_1, T_1, k_2, T_2, \dots, k_i, T_i, x, \emptyset, k_{i+1}, T_{i+1}, \dots, k_{n-1}, T_{n-1}\}.$$

What happens when the leaf is full? That is, suppose we wish to insert the pair, (x, \emptyset) into a node T which already has $M-1$ keys. Inserting the pair in its correct position gives a result of the form

$$T' = \{T_0, k_1, T_1, k_2, T_2, \dots, k_M, T_M\}.$$

However, this is not a valid node in a B-tree of order M because it has $M+1$ subtrees and M keys. The solution is to split node T in half as follows

$$\begin{aligned} T'_L &= \{T_0, k_1, T_1, \dots, k_{\lceil M/2 \rceil - 1}, T_{\lceil M/2 \rceil - 1}\} \\ T'_R &= \{T_{\lceil M/2 \rceil}, k_{\lceil M/2 \rceil + 1}, T_{\lceil M/2 \rceil + 1}, \dots, k_M, T_M\} \end{aligned}$$

Note, T'_L is a valid B-tree node because it contains $\lceil M/2 \rceil$ subtrees and $\lceil M/2 \rceil - 1$ keys. Similarly, T'_R is a valid B-tree node because it contains $\lceil (M+1)/2 \rceil$ subtrees and $\lceil (M+1)/2 \rceil - 1$ keys. Note that there is still a key left over, namely $k_{\lceil M/2 \rceil}$.

There are now two cases to consider--either T is the root or it is not. Suppose T is

not the root. Where we once had the single node T , we now have the two nodes, T'_L and T'_R , and the left-over key, $k_{\lceil M/2 \rceil}$. This situation is resolved as follows:

First, T'_L replaces T in the parent of T . Next, we take the pair $(k_{\lceil M/2 \rceil}, T'_R)$ and recursively insert it in the parent of T .

Figure \square illustrates this case for a B-tree of order three. Inserting the key 6 in the tree causes the leaf node to overflow. The leaf is split in two. The left half contains key 5; and the right, key 7; and key 6 is left over. The two halves are re-attached to the parent in the appropriate place with the left-over key between them.

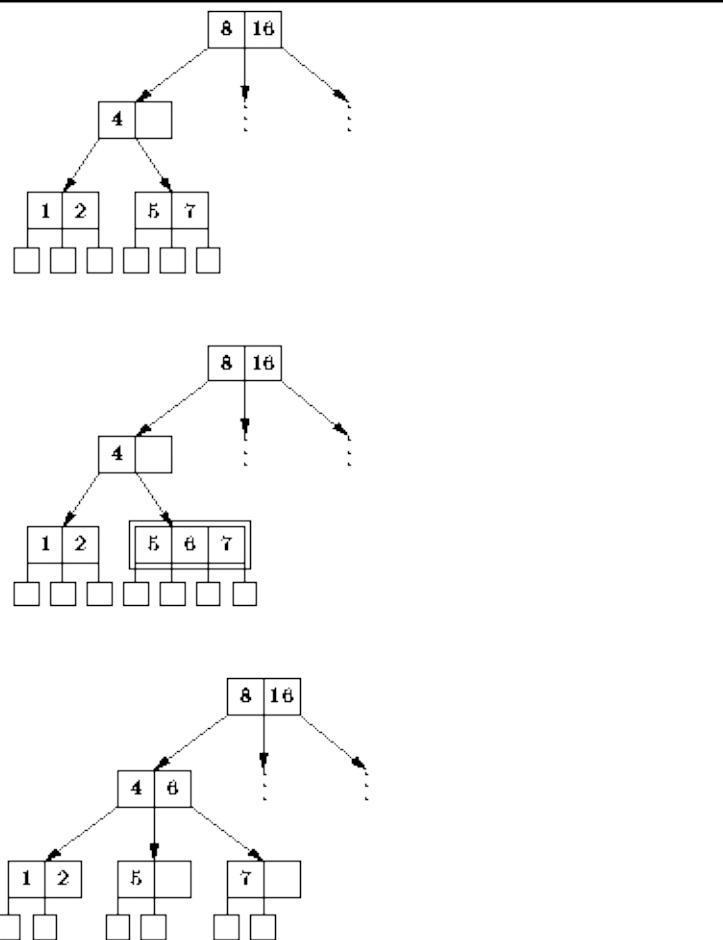


Figure: Inserting items into a B-tree (insert 6).

If the parent node fills up, then it too is split and the two new nodes are inserted in the grandparent. This process may continue all the way up the tree to the root. What do we do when the root fills up? When the root fills, it is also split.

However, since there is no parent into which to insert the two new children, a new root is inserted above the old root. The new root will contain exactly two subtrees and one key, as allowed by Definition □.

Figure □ illustrates this case for a B-tree of order three. Inserting the key 3 in the tree causes the leaf node to overflow. Splitting the leaf and reattaching it causes the parent to overflow. Similarly, splitting the parent and reattaching it causes the grandparent to overflow but the grandparent is the root. The root is split and a new root is added above it.

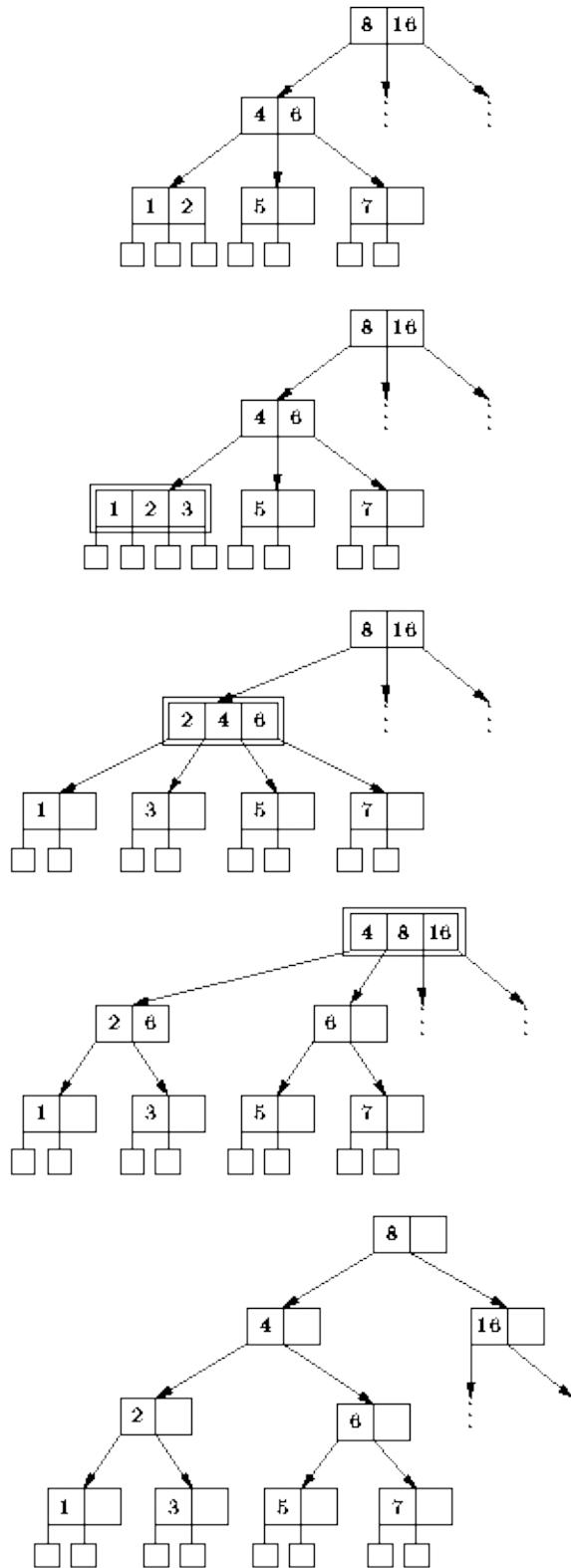


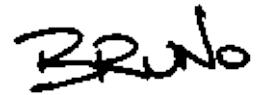
Figure: Inserting items into a B-tree (insert 3).

Notice that the height of the B-tree only increases when the root node splits. Furthermore, when the root node splits, the two halves are both attached under the new root. Therefore, the external nodes all remain at the same depth, as required by Definition □.

- [Implementation](#)
 - [Running Time Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Insertion in a B-tree is a two-pass process. The first pass moves down the tree from the root in order to locate the leaf in which the insertion is to begin. This part of the algorithm is quite similar to the `find` method given in Program □. The second pass moves from the bottom of the tree back up to the root, splitting nodes and inserting them further up the tree as needed. Program □ gives the code for the first (downward) pass (`insert` method) and the Program □ gives the code for the second (upward) pass (`insertUp` method).

```

1  class BTTree(MWayTree):
2
3      def insert(self, obj):
4          if self.isEmpty():
5              if self._parent is None:
6                  self.attachSubtree(0, BTTree(self.m))
7                  self._key[1] = obj
8                  self.attachSubtree(1, BTTree(self.m))
9                  self._count = 1
10             else:
11                 self._parent.insertUp(obj, BTTree(self.m))
12             else:
13                 index = self.findIndex(obj)
14                 if index != 0 and self._key == obj:
15                     raise KeyError
16                     self._subtree[index].insert(obj)
17
18     # ...

```

Program: `BTTree` class `insert` method.

In the implementation shown, the downward pass starts at the root node and descends the tree until it arrives at an external node. If the external node has no parent, it must be the root and, therefore, the tree is empty. In this case, the root becomes an internal node containing a single key and two empty subtrees (lines 6-9). Otherwise, we have arrived at an external node in a non-empty tree and the second pass begins by calling `insertUp` to insert the pair (x, \emptyset) in the parent (line 11).

The upward pass of the insertion algorithm is done by the recursive `insertUp` method shown in Program □. The `insertUp` method takes two arguments. The first, `obj`, is an object and the second, `child`, is a `BTree`. It is assumed that all the keys in `child` are strictly greater than `obj`.

```

1  class BTree(MWayTree):
2
3      def insertUp(self, obj, child):
4          index = self.findIndex(obj)
5          if not self.isFull:
6              self.insertPair(index + 1, obj, child)
7              self._count = self._count + 1
8          else:
9              (extraKey, extraTree) = self.insertPair(
10                  index + 1, obj, child)
11              if self._parent is None:
12                  left = BTree(self.m)
13                  right = BTree(self.m)
14                  left.attachLeftHalfOf(self)
15                  right.attachRightHalfOf(self)
16                  right.insertUp(extraKey, extraTree)
17                  self.attachSubtree(0, left)
18                  self._key[1] = self._key[(self.m + 1)/2]
19                  self.attachSubtree(1, right)
20                  self._count = 1
21              else:
22                  self._count = (self.m + 1)/2 - 1
23                  right = BTree(self.m)
24                  right.attachRightHalfOf(self)
25                  right.insertUp(extraKey, extraTree)
26                  self._parent.insertUp(
27                      self._key[(self.m + 1)/2], right)
28
29      # ...

```

Program: `BTree` class `insertUp` method.

The `insertUp` method calls `findIndex` to determine the position in the array of keys at which pair (`obj,child`) should be inserted (line 8). If this node is not full (line 5), the `insertPair` method is called to insert the given key and tree at the specified position in the `_key` and `_subtree` arrays, respectively (line 6).

In the event that the node is full, the `insertPair` method returns the key the subtree that fall off the right end of the array. These are assigned to `extraKey`

and extraSubtree (line 9-10).

The node has now overflowed and it is necessary to balance the B-tree. If the node overflows and it is the root (line 11), then two new B-trees, left and right are created (lines 12-13). The first $\lceil \frac{M}{2} \rceil - 1$ keys and $\lceil \frac{M}{2} \rceil$ subtrees of the given node are moved to the left tree by the attachLeftHalfOf method (line 14); and the last $\lceil \frac{(M+1)}{2} \rceil - 2$ keys and $\lceil \frac{(M+1)}{2} - 1 \rceil$ subtrees of the given node are moved to the right tree by the attachRightHalfOf method (line 15). Then, the pair (extraKey,extraTree) is inserted into the right tree (line 16).

The left-over key is the one in the middle of the array, i.e., $k_{\lceil \frac{M}{2} \rceil}$. Finally, the root node is modified so that it contains the two new subtrees and the single left-over key (lines 17-19).

If the node overflows and it is not the root, then one new B-tree is created, right (line 23). The last $\lceil \frac{(M+1)}{2} \rceil - 2$ keys and $\lceil \frac{(M+1)}{2} - 1 \rceil$ subtrees of the given node are moved to the left tree by the attachRightHalfOf method (line 24) and the pair (extraKey,extraTree) is inserted in the right tree (line 25). The first $\lceil \frac{M}{2} \rceil - 1$ keys and $\lceil \frac{M}{2} \rceil$ subtrees of the given node remain attached to it.

Finally, the insertUp method calls itself recursively to insert the left-over key, $k_{\lceil \frac{M}{2} \rceil}$, and the new B-tree, right, into the parent of this (line 26). This is the place where the _parent instance attribute is needed!

Running Time Analysis

The running time of the downward pass of the insertion algorithm is identical to that of an unsuccessful search (assuming the item to be inserted is not already in the tree). That is, for a B-tree of height h , the worst-case running time of the downward pass is

$$(h + 1)(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}_{\text{comp}} + O(h \log M).$$

The second pass of the insertion algorithm does the insertion and balances the tree if necessary. In the worst case, all of the nodes in the insertion path up to the root need to be balanced. Each time the `insertUp` method is invoked, it calls `findIndex` which has running time $(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}_{\text{comp}} + O(\log M)$ in the worst case. The additional time required to balance a node is $O(M)$. Therefore, the worst-case running time of the upward pass is

$$(h + 1)(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}_{\text{comp}} + O(hM).$$

Therefore, the total running time for insertion is

$$2(h + 1)(\lceil \log_2(M - 1) \rceil + 2)\mathcal{T}_{\text{comp}} + O(hM).$$

According to Theorem \square , the height of a B-tree is $h \leq \log_{\lceil M/2 \rceil}((n + 1)/2)$, where n is the number of keys in the B-tree. If we assume that two keys can be compared in constant time, i.e., $\mathcal{T}_{\text{comp}} = O(1)$, then the running time for insertion in a B-tree is simply $O(M \log n)$.

Removing Items from a B-Tree

The algorithm for removing items from a B-tree is similar to the algorithm for removing item from an AVL tree. That is, once the item to be removed has been found, it is pushed down the tree to a leaf node where it can be easily deleted. When an item is deleted from a node it is possible that the number of keys remaining is less than $\lceil M/2 \rceil - 1$. In this case, balancing is necessary.

The algorithm of balancing after deletion is like the balancing after insertion in that it progresses from the leaf node up the tree toward the root. Given a node T which has $\lceil M/2 \rceil - 2$ keys, there are four cases to consider.

In the first case, T is the root. If no keys remain, T becomes the empty tree. Otherwise, no balancing is needed because the root is permitted to have as few as two subtrees and one key. For the remaining cases T is not the root.

In the second case T has $\lceil M/2 \rceil - 2$ keys and it also has a sibling immediately on the left with at least $\lceil M/2 \rceil$ keys. The tree can be balanced by doing an LL rotation as shown in Figure □. Notice that after the rotation, both siblings have at least $\lceil M/2 \rceil - 1$ keys. Furthermore, the heights of the siblings remain unchanged. Therefore, the resulting tree is a valid B-tree.

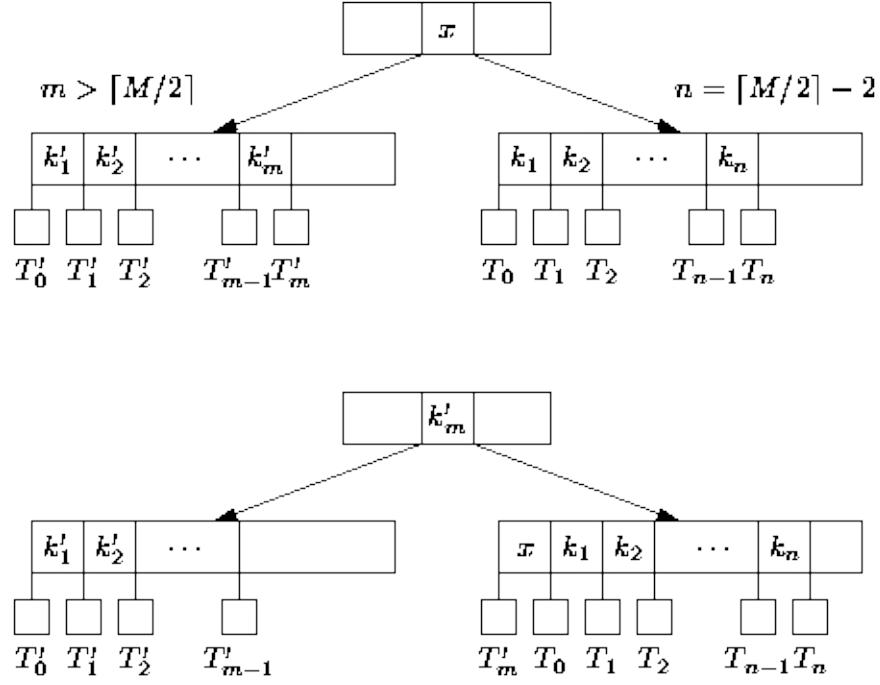


Figure: LL rotation in a B-tree.

The third case is the left-right mirror of the second case. That is, T has $\lceil M/2 \rceil - 2$ keys and it also has a sibling immediately on the right with at least $\lceil M/2 \rceil$ keys. In this case, the tree can be balanced by doing an RR rotation .

In the fourth and final case, T has $\lceil M/2 \rceil - 2$ keys, and its immediate sibling(s) have $\lceil M/2 \rceil - 1$ keys. In this case, the sibling(s) cannot give-up a key in a rotation because they already have the minimum number of keys. The solution is to merge T with one of its siblings as shown in Figure □.

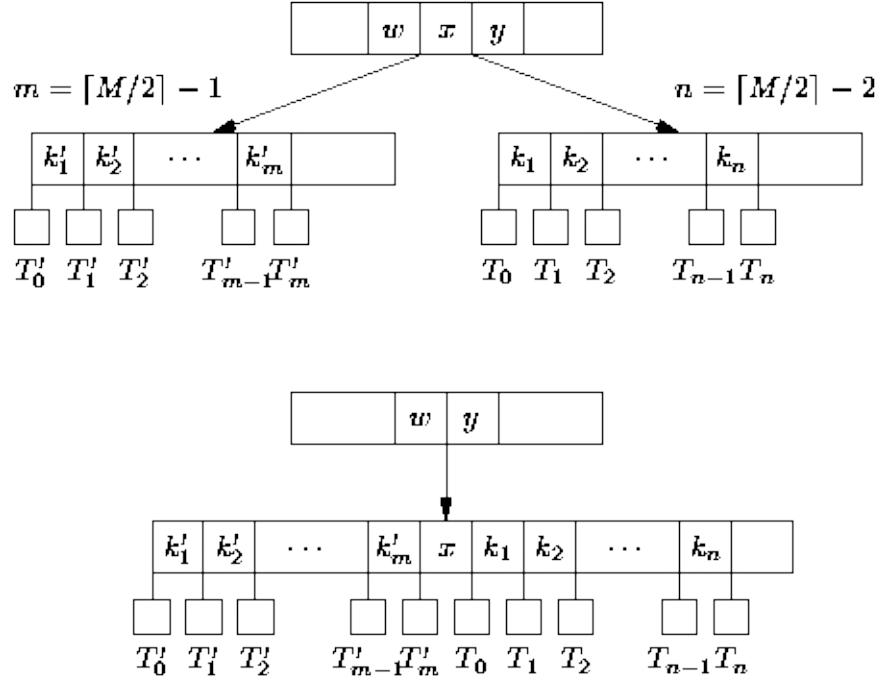


Figure: Merging nodes in a B-tree.

The merged node contains $\lceil M/2 \rceil - 2$ keys from T , $\lceil M/2 \rceil - 1$ keys from the sibling, and one key from the parent (the key x in Figure □). The resulting node contains $2\lceil M/2 \rceil - 2$ keys altogether, which is $M-2$ if M is even and $M-1$ if M is odd. Either way, the resulting node contains no more than $M-1$ keys and is a valid B-tree node. Notice that in this case a key has been removed from the parent of T . Therefore, it may be necessary to balance the parent. Balancing the parent may necessitate balancing the grandparent, and so on, up the tree to the root.

Applications

There are many applications for search trees. The principal characteristic of such applications is that a database of keyed information needs to be frequently accessed and the access pattern is either unknown or known to be random. For example, *dictionaries* are often implemented using search trees. A dictionary is essentially a container that contains ordered key/value pairs. The keys are words in a source language and, depending on the application, the values may be the definitions of the words or the translation of the word in a target language.

This section presents a simple application of search trees. Suppose we are required to translate the words in an input file one-by-one from some source language to another target language. In this example, the translation is done one word at a time. That is, no natural language syntactic or semantic processing is done.

In order to implement the translator we assume that there exists a text file, which contains pairs of words. The first element of the pair is a word in the source language and the second element is a word in the target language. To translate a text, we first read the words and the associated translations and build a search tree. The translation is created one word at a time by looking up each word in the text.

Program □ gives an implementation of the translator. The `translate` method uses a search tree to hold the pairs of words. In this case, an AVL tree is used. However, this implementation works with all the search tree types described in this chapter (e.g., `BinarySearchTree`, `AVLTree`, `MWayTree`, and `BTree`).

```
 1 class Algorithms(object):
 2
 3     def translate(dictionary, input, output):
 4         searchTree = AVLTree()
 5         for line in dictionary.readlines():
 6             words = line.split()
 7             assert len(words) == 2
 8             searchTree.insert(Association(words[0], words[1]))
 9         for line in input.readlines():
10             for word in line.split():
11                 assoc = searchTree.find(Association(word))
12                 if assoc is None:
13                     output.write(word + " ")
14                 else:
15                     output.write(assoc.value + " ")
16             output.write("\n")
17     translate = staticmethod(translate)
```

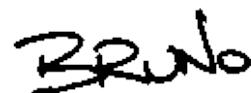
Program: Application of search trees--word translation.

The `translate` method reads pairs of strings from the input stream (lines 5-8). The `Association` class defined in Section □ is used to contain the key/value pairs. A new instance is created for each key/value pair which is then inserted into the search tree (line 9). The process of building the search tree terminates when the end-of-file is encountered.

During the translation phase, the `translate` method reads words one at a time from the input stream and writes the translation of each word on the output stream. Each word is looked up as it is read (lines 10-12). If no key matches the given word, the word is printed followed by a question mark (lines 13-14). Otherwise, the value associated with the matching key is printed (line 16).

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exercises

1. For each of the following key sequences determine the binary search tree obtained when the keys are inserted one-by-one in the order given into an initially empty tree:
 1. 1, 2, 3, 4, 5, 6, 7.
 2. 4, 2, 1, 3, 6, 5, 7.
 3. 1, 6, 7, 2, 4, 3, 5.
2. For each of the binary search trees obtained in Exercise □ determine the tree obtained when the root is withdrawn.
3. Repeat Exercises □ and □ for AVL trees.
4. Derive an expression for the total space needed to represent a tree of n internal nodes using each of the following classes:
 1. `BinarySearchTree` introduced in Program □,
 2. `AVLTree` introduced in Program □,
 3. `MWayTree` introduced in Program □, and
 4. `BTree` introduced in Program □.

Hint: For the `MWayTree` and `BTree` assume that the tree contains are k keys, where $k \geq n$.

5. To delete a non-leaf node from a binary search tree, we swap it either with the smallest key its right subtree or with the largest key in its left subtree and then recursively delete it from the subtree. In a tree of n nodes, what its the maximum number of swaps needed to delete a key?
6. Devise an algorithm to compute the internal path length of a tree. What is the running time of your algorithm?
7. Devise an algorithm to compute the external path length of a tree. What is the running time of your algorithm?
8. Suppose that you are given a sorted sequence of n keys, $k_0 \leq k_1 \leq \dots \leq k_{n-1}$, to be inserted into a binary search tree.
 1. What is the minimum height of a binary tree that contains n nodes.
 2. Devise an algorithm to insert the given keys into a binary search tree so that the height of the resulting tree is minimized.
 3. What is the running time of your algorithm?
9. Devise an algorithm to construct an AVL tree of a given height h that contains the minimum number of nodes. The tree should contain the keys $1, 2, 3, \dots, N_h$, where N_h is given by Equation □.

10. Consider what happens when we insert the keys $1, 2, 3, \dots, 2^{h+1} - 1$ one-by-one in the order given into an initially empty AVL tree for $h \geq 0$. Prove that the result is always a perfect tree of height h .
11. The `find` method defined in Program [□](#) is recursive. Write a non-recursive method to find a given item in a binary search tree.
12. Repeat Exercise [□](#) for the `min` method defined in Program [□](#).
13. Devise an algorithm to select the k^{th} key in a binary search tree. For example, given a tree with n nodes, $k=0$ selects the smallest key, $k=n-1$ selects the largest key, and $k = \lceil n/2 \rceil - 1$ selects the median key.
14. Devise an algorithm to test whether a given binary search tree is AVL balanced. What is the running time of your algorithm?
15. Devise an algorithm that takes two values, a and b such that $a \leq b$, and which visits all the keys x in a binary search tree such that $a \leq x \leq b$. The running time of your algorithm should be $O(N + \log n)$, where N is the number of keys visited and n is the number of keys in the tree.
16. Devise an algorithm to merge the contents of two binary search trees into one. What is the running time of your algorithm?
17. (This question should be attempted *after* reading Chapter [□](#)). Prove that a *complete binary tree* (Definition [□](#)) is AVL balanced.
18. Do Exercise [□](#).
19. For each of the following key sequences determine the 3-way search tree obtained when the keys are inserted one-by-one in the order given into an initially empty tree:
1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
 2. 3, 1, 4, 5, 9, 2, 6, 8, 7, 0.
 3. 2, 7, 1, 8, 4, 5, 9, 0, 3, 6.
20. Repeat Exercise [□](#) for B-trees of order 3.

Projects

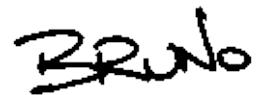
1. Complete the implementation of the `BinarySearchTree` class introduced in Program □ by providing suitable definitions for the following operations: `__contains__` and `getMax`. You must also have a complete implementation of the base class `BinaryTree`. (See Project □). Write a test program and test your implementation.
2. Complete the implementation of the `AVLTree` class introduced in Program □ by providing suitable definitions for the following operations: `doRRRotation`, and `doLRLRotation`. You must also have a complete implementation of the base class `BinarySearchTree`. (See Project □). Write a test program and test your implementation.
3. Complete the implementation of the `MWayTree` class introduced in Program □ by providing suitable definitions for the following operations: `purge`, `getCount`, `getIsEmpty`, `getIsLeaf`, `getDegree`, `getKey`, `getSubtree`, `__contains__`, `getMin`, `getMax`, `breadthFirstTraversal`, and `__iter__`. Write a test program and test your implementation.
4. Complete the implementation of the `BTree` class introduced in Program □ by providing suitable definitions for the following methods: `insertKey`, `insertSubtree`, `attachLeftHalfOf`, `attachRightHalfOf`, and `withdraw`. You must also have a complete implementation of the base class `MWayTree`. (See Project □). Write a test program and test your implementation.
5. The binary search tree `withdraw` method shown in Program □ is biased in the following way: If the key to be deleted is in a non-leaf node with two non-empty subtrees, the key is swapped with the maximum key in the left subtree and then recursively deleted from the left subtree. Following a long series of insertions and deletions, the search tree will tend to have more nodes in the right subtrees and fewer nodes in the left subtrees. Devise and conduct an experiment that demonstrates this phenomenon.
6. Consider the implementation of AVL trees. In order to check the AVL balance condition in constant time, we record in each node the height of that node. An alternative to keeping track of the height information explicitly is to record in each node the *difference* in the heights of its two subtrees. In an AVL balanced tree, this difference is either -1, 0 or +1. Replace the `_height` instance attribute of the `AVL` class defined in Program □ with one called `_diff` and rewrite the various methods

accordingly.

7. The M -way tree implementation given in Section □ is an *internal* data structure--it is assumed that all the nodes reside in the main memory. However, the motivation for using an M -way tree is that it is an efficient way to organize an *external* data structure--one that is stored on disk. Design, implement and test an external M -way tree implementation.
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Heaps and Priority Queues

In this chapter we consider priority queues. A priority queue is essentially a list of items in which each item has associated with it a *priority*. In general, different items may have different priorities and we speak of one item having a higher priority than another. Given such a list we can determine which is the highest (or the lowest) priority item in the list. Items are inserted into a priority queue in any, arbitrary order. However, items are withdrawn from a priority queue in order of their priorities starting with the highest priority item first.

For example, consider the software which manages a printer. In general, it is possible for users to submit documents for printing much more quickly than it is possible to print them. A simple solution is to place the documents in a *FIFO* queue (Chapter □). In a sense this is fair, because the documents are printed on a first-come, first-served basis.

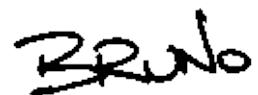
However, a user who has submitted a short document for printing will experience a long delay when much longer documents are already in the queue. An alternative solution is to use a priority queue in which the shorter a document, the higher its priority. By printing the shortest documents first, we reduce the level of frustration experienced by the users. In fact, it can be shown that printing documents in order of their length minimizes the average time a user waits for his document.

Priority queues are often used in the implementation of algorithms. Typically the problem to be solved consists of a number of subtasks and the solution strategy involves prioritizing the subtasks and then performing those subtasks in the order of their priorities. For example, in Chapter □ we show how a priority queue can improve the performance of backtracking algorithms, in Chapter □ we will see how a priority queue can be used in sorting and in Chapter □ several graph algorithms that use a priority queue are discussed.

- [Basics](#)
 - [Binary Heaps](#)
 - [Leftist Heaps](#)
 - [Binomial Queues](#)
 - [Applications](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Basics

A priority queue is a container which provides the following three operations:

enqueue

used to put objects into the container;

min

accesses the smallest object in the container; and

dequeueMin

removes the smallest object from the container.

A priority queue is used to store a finite set of keys drawn from a totally ordered set of keys K . As distinct from search trees, duplicate keys *are* allowed in priority queues.

Program □ defines the `PriorityQueue` class. The abstract `PriorityQueue` class extends the abstract `Container` class defined in Program □. In addition to the inherited methods, the `PriorityQueue` class supports the three operations listed above.

```
 1  class PriorityQueue(Container):
 2
 3      def __init__(self):
 4          super(PriorityQueue, self).__init__()
 5
 6      def enqueue(self, obj):
 7          pass
 8      enqueue = abstractmethod(enqueue)
 9
10      def getMin(self):
11          pass
12      getMin = abstractmethod(getMin)
13
14      min = property(
15          fget = lambda self: self.getMin())
16
17      def dequeueMin(self):
18          pass
19      dequeueMin = abstractmethod(dequeueMin)
```

Program: Abstract `PriorityQueue` class.

Program □ defines the `MergeablePriorityQueue` class. The abstract `MergeablePriorityQueue` class extends the abstract `PriorityQueue` class defined in Program □. A *mergeable priority queue* is one which provides the ability to merge efficiently two priority queues into one. Of course it is always possible to merge two priority queues by dequeuing the elements of one queue and enqueueing them in the other. However, the mergeable priority queue implementations we will consider allow more efficient merging than this.

```
1 class MergeablePriorityQueue(PriorityQueue):
2
3     def __init__(self):
4         super(MergeablePriorityQueue, self).__init__()
5
6     def merge(self, queue):
7         pass
8     merge = abstractmethod(merge)
```

Program: Abstract `MergeablePriorityQueue` class.

It is possible to implement the required functionality using data structures that we have already considered. For example, a priority queue can be implemented simply as a list. If an *unsorted list* is used, enqueueing can be accomplished in constant time. However, finding the minimum and removing the minimum each require $O(n)$ time where n is the number of items in the queue. On the other hand, if a *sorted list* is used, finding the minimum and removing it is easy--both operations can be done in constant time. However, enqueueing an item in a sorted list requires $O(n)$ time.

Another possibility is to use a search tree. For example, if an *AVL tree* is used to implement a priority queue, then all three operations can be done in $O(\log n)$ time. However, search trees provide more functionality than we need. Search trees support finding the largest item with `max`, deletion of arbitrary objects with `withdraw`, and the ability to visit in order all the contained objects via `depthFirstTraversal`. All these operations can be done as efficiently as the priority queue operations. Because search trees support more methods than we really need for priority queues, it is reasonable to suspect that there are more efficient ways to implement priority queues. And indeed there are!

A number of different priority queue implementations are described in this chapter. All the implementations have one thing in common--they are all based on a special kind of tree called a *min heap* or simply a *heap*.

Definition ((Min) Heap) A (*Min*) *Heap* is a tree,

$$T = \{R, T_0, T_1, T_2, \dots, T_{n-1}\},$$

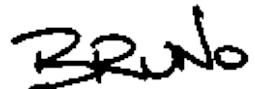
with the following properties:

1. Every subtree of T is a heap; and,
2. The root of T is less than or equal to the root of every subtree of T .
That is, $R \leq R_i$ for all i , $0 \leq i < n$, where R_i is the root of T_i .

According to Definition \square , the key in each node of a heap is less than or equal to the roots of all the subtrees of that node. Therefore, by induction, the key in each node is less than or equal to all the keys contained in the subtrees of that node. Note, however, that the definition says nothing about the relative ordering of the keys in the subtrees of a given node. For example, in a binary heap either the left or the right subtree of a given node may have the larger key.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



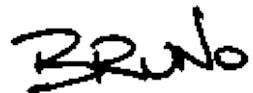
Binary Heaps

A binary heap is a heap-ordered binary tree which has a very special shape called a *complete tree*. As a result of its special shape, a binary heap can be implemented using an array as the underlying foundational data structure. Array subscript calculations are used to find the parent and the children of a given node in the tree. And since an array is used, the storage overhead associated with the subtree instance attributes contained in the nodes of the trees is eliminated.

- [Complete Trees](#)
 - [Implementation](#)
 - [Putting Items into a Binary Heap](#)
 - [Removing Items from a Binary Heap](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Complete Trees

The preceding chapter introduces the idea of a *perfect tree* (see Definition □). Complete trees and perfect trees are closely related, yet quite distinct. As pointed out in the preceding chapter, a perfect binary tree of height h has exactly $n = 2^{h+1} - 1$ internal nodes. Since, the only permissible values of n are

$$0, 1, 3, 7, 15, 31, \dots, 2^{h+1} - 1, \dots,$$

there is no *perfect* binary tree which contains, say 2, 4, 5, or 6 nodes.

However, we want a data structure that can hold an arbitrary number of objects so we cannot use a perfect binary tree. Instead, we use a *complete binary tree*, which is defined as follows:

Definition (Complete Binary Tree) A *complete binary tree* of height $h \geq 0$, is a binary tree $\{R, T_L, T_R\}$ with the following properties.

1. If $h=0$, $T_L = \emptyset$ and $T_R = \emptyset$.
2. For $h>0$ there are two possibilities:
 1. T_L is a perfect binary tree of height $h-1$ and T_R is a complete binary tree of height $h-1$; or
 2. T_L is a complete binary tree of height $h-1$ and T_R is a perfect binary tree of height $h-2$.

Figure □ shows an example of a complete binary tree of height four. Notice that the left subtree of node 1 is a complete binary tree of height three; and the right subtree is a perfect binary tree of height two. This corresponds to case 2 (b) of Definition □. Similarly, the left subtree of node 2 is a perfect binary tree of height two; and the right subtree is a complete binary tree of height two. This corresponds to case 2 (a) of Definition □.

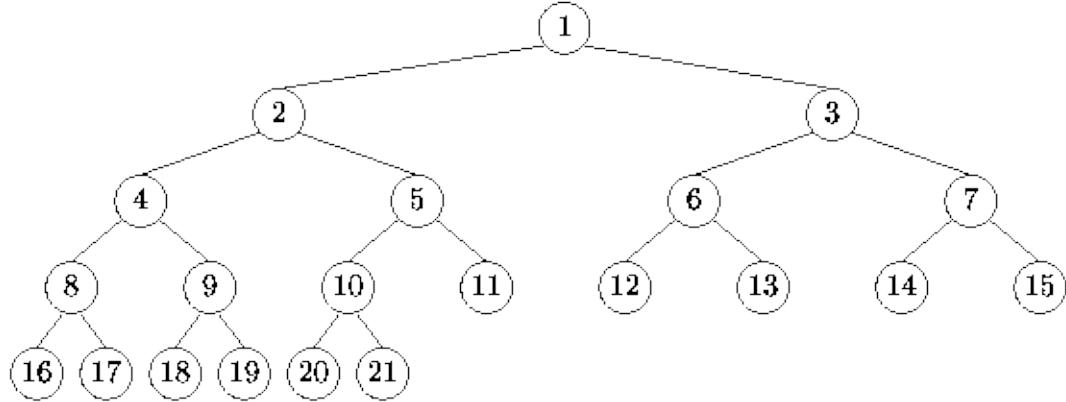


Figure: A complete binary tree.

Does there exist a complete binary with exactly n nodes for every integer $n > 0$? The following theorem addresses this question indirectly by defining the relationship between the height of a complete tree and the number of nodes it contains.

Theorem A complete binary tree of height $h \geq 0$ contains at least 2^h and at most $2^{h+1} - 1$ nodes.

extbf{Proof} First, we prove the lower bound by induction. Let m_h be the *minimum* number of nodes in a complete binary tree of height h . To prove the lower bound we must show that $m_h = 2^h$.

Base Case There is exactly one node in a tree of height zero. Therefore, $m_0 = 1 = 2^0$.

Inductive Hypothesis Assume that $m_h = 2^h$ for $h = 0, 1, 2, \dots, k$, for some $k \geq 0$. Consider the complete binary tree of height $k+1$ which has the smallest number of nodes. Its left subtree is a complete tree of height k having the smallest number of nodes and its right subtree is a perfect tree of height $k-1$.

From the inductive hypothesis, there are 2^k nodes in the left subtree and there are exactly $2^{(k-1)+1} - 1$ nodes in the perfect right subtree. Thus,

$$\begin{aligned} m_{k+1} &= 1 + 2^k + 2^{(k-1)+1} - 1 \\ &= 2^{k+1}. \end{aligned}$$

Therefore, by induction $m_h = 2^h$ for all $h \geq 0$, which proves the lower bound.

Next, we prove the upper bound by induction. Let M_h be the *maximum* number of nodes in a complete binary tree of height h . To prove the upper bound we must show that $M_h = 2^{h+1} - 1$.

Base Case There is exactly one node in a tree of height zero. Therefore, $M_0 = 1 = 2^1 - 1$.

Inductive Hypothesis Assume that $M_h = 2^{h+1} - 1$ for $h = 0, 1, 2, \dots, k$, for some $k \geq 0$. Consider the complete binary tree of height $k+1$ which has the largest number of nodes. Its left subtree is a perfect tree of height k and its right subtree is a complete tree of height k having the largest number of nodes.

There are exactly $2^{k+1} - 1$ nodes in the perfect left subtree. From the inductive hypothesis, there are $2^{k+1} - 1$ nodes in the right subtree. Thus,

$$\begin{aligned} M_{k+1} &= 1 + 2^{k+1} - 1 + 2^{k+1} - 1 \\ &= 2^{(k+1)+1} - 1. \end{aligned}$$

Therefore, by induction $M_h = 2^{h+1} - 1$ for all $h \geq 0$, which proves the upper bound.

It follows from Theorem \square that there exists exactly one complete binary tree that contains exactly n internal nodes for every integer $n \geq 0$. It also follows from Theorem \square that the height of a complete binary tree containing n internal nodes is $h = \lfloor \log_2 n \rfloor$.

Why are we interested in complete trees? As it turns out, complete trees have some useful characteristics. For example, in the preceding chapter we saw that the internal path length of a tree, i.e., the sum of the depths of all the internal nodes, determines the average time for various operations. A complete binary tree has the nice property that it has the smallest possible internal path length:

Theorem The internal path length of a binary tree with n nodes is at least as big as the internal path length of a *complete* binary tree with n nodes.

extbfProof Consider a binary tree with n nodes that has the smallest possible internal path length. Clearly, there can only be one node at depth zero--the root. Similarly, at most two nodes can be at depth one; at most four nodes can be at depth two; and so on. Therefore, the internal path length of a tree with n nodes is always at least as large as the sum of the first n terms in the series

$$0, \underbrace{1, 1}_{2}, \underbrace{2, 2, 2}_{4}, \underbrace{3, 3, 3, 3, 3, 3}_{8}, 4, \dots$$

But this summation is precisely the internal path length of a complete binary tree!

Since the depth of the average node in a tree is obtained by dividing the internal path length of the tree by n , Theorem \square tells us that complete trees are the best possible in the sense that the average depth of a node in a complete tree is the smallest possible. But how small is small? That is, does the average depth grow logarithmically with n . The following theorem addresses this question:

Theorem The *internal path length* of a complete binary tree with n nodes is

$$\sum_{i=1}^n \lfloor \log_2 i \rfloor = (n+1) \lfloor \log_2(n+1) \rfloor - 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2.$$

extbfProof The proof of Theorem \square is left as an exercise for the reader (Exercise \square).

From Theorem \square we may conclude that the internal path length of a complete tree is $O(n \log n)$. Consequently, the depth of the average node in a complete tree is $O(\log n)$.

- [Complete \$N\$ -ary Trees](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Complete N -ary Trees

The definition for complete binary trees can be easily extended to trees with arbitrary fixed degree $N \geq 2$ as follows:

Definition (Complete N -ary Tree) A *complete N -ary tree* of height $h \geq 0$, is an N -ary tree $\{R, T_0, T_1, T_2, \dots, T_{N-1}\}$ with the following properties.

1. If $h=0$, $T_i = \emptyset$ for all i , $0 \leq i < N$.
2. For $h>0$ there exists a j , $0 \leq j < N$ such that
 1. T_i is a perfect N -ary tree of height $h-1$ for all $i : 0 \leq i < j$;
 2. T_j is a complete N -ary tree of height $h-1$; and,
 3. T_i is a perfect N -ary tree of height $h-2$ for all $i : j < i < N$.

Note that while it is expressed in somewhat different terms, the definition of a complete N -ary tree is consistent with the definition of a binary tree for $N=2$. Figure □ shows an example of a complete ternary ($N=3$) tree.

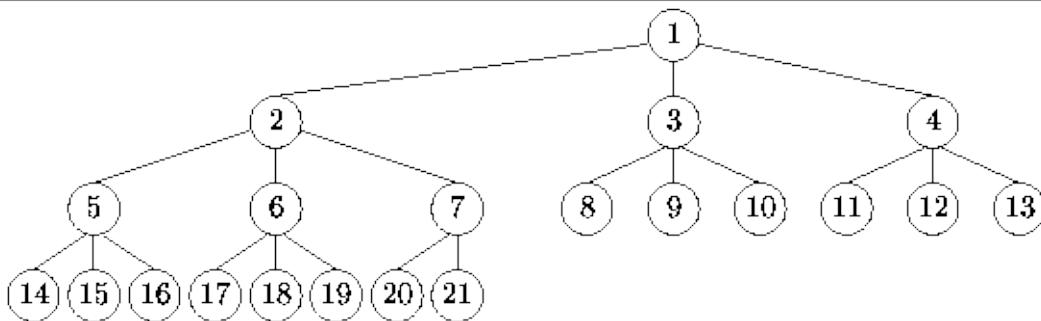


Figure: A complete ternary tree.

Informally, a complete tree is a tree in which all the levels are full except for the bottom level and the bottom level is filled from left to right. For example in Figure □, the first three levels are full. The fourth level which comprises nodes 14-21 is partially full and has been filled from left to right.

The main advantage of using complete binary trees is that they can be easily stored in an array. Specifically, consider the nodes of a complete tree numbered consecutively in *level-order* as they are in Figures □ and □. There is a simple formula that relates the number of a node with the number of its parent and the numbers of its children.

Consider the case of a complete binary tree. The root node is node 1 and its children are nodes 2 and 3. In general, the children of node i are $2i$ and $2i+1$. Conversely, the parent of node i is $\lceil i/2 \rceil$. Figure □ illustrates this idea by showing how the complete binary tree shown in Figure □ is mapped into an array.

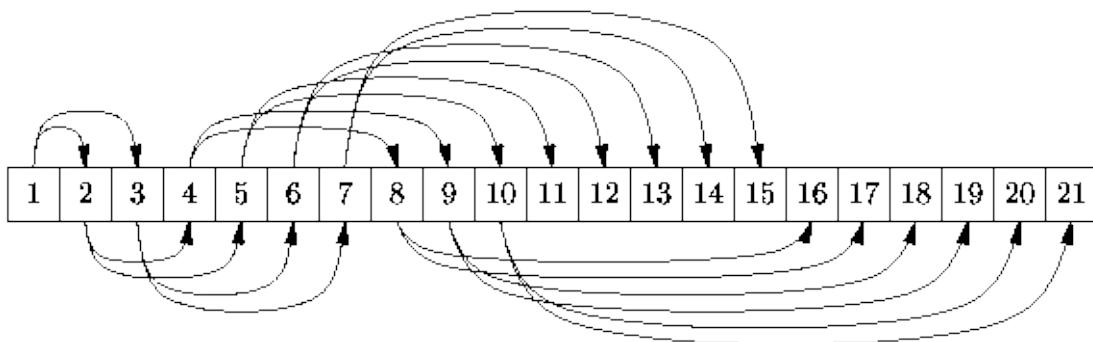


Figure: Array representation of a complete binary tree.

A remarkable characteristic of complete trees is that filling the bottom level from left to right corresponds to adding elements at the end of the array! Thus, a complete tree containing n nodes occupies the first n consecutive array positions.

The array subscript calculations given above can be easily generalized to complete N -ary trees. Assuming that the root occupies position 1 of the array, its N children occupy positions 2, 3, ..., $N+1$. In general, the children of node i occupy positions

$$N(i - 1) + 2, N(i - 1) + 3, N(i - 1) + 4, \dots, Ni + 1,$$

and the parent of node i is found at

$$\lceil (i - 1)/N \rceil.$$

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Implementation

A binary heap is a heap-ordered complete binary tree which is implemented using an array. In a heap the smallest key is found at the root and since the root is always found in the first position of the array, finding the smallest key is a trivial operation in a binary heap.

In this section we describe the implementation of a priority queue as a binary heap. As shown in Figure □, we define a concrete class called `BinaryHeap` for this purpose.

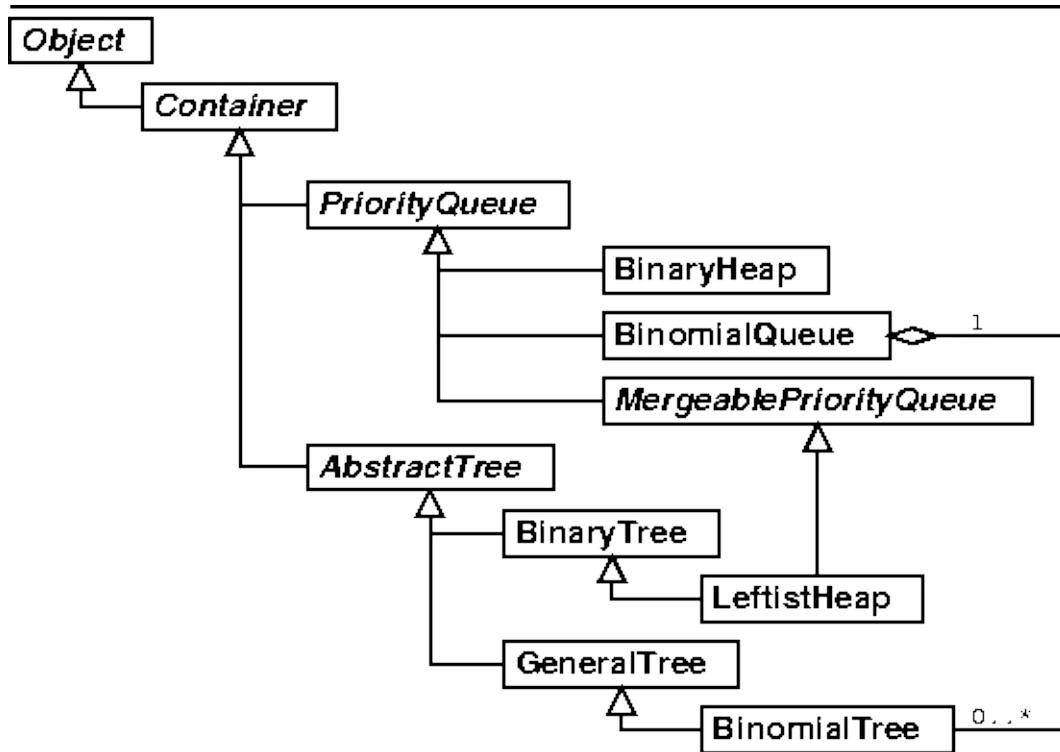


Figure: Object class hierarchy

Program □ introduces the `BinaryHeap` class. The `BinaryHeap` class extends the abstract `PriorityQueue` class defined in Program □.

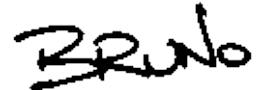
```
1 class BinaryHeap(PriorityQueue):
2
3     def __init__(self, length = 0):
4         super(BinaryHeap, self).__init__()
5         self._array = Array(length, 1) # Base index is 1.
6
7     def purge(self):
8         while self._count > 0:
9             self._array[self._count] = None
10            self._count -= 1
11
12    # ...
```

Program: BinaryHeap class `__init__` and `purge` methods.

-
- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Instance Attributes

The `BinaryHeap` class has a rather simple implementation. In particular, it requires only a single instance attribute, `_array`, which is used to hold the objects that are contained in the binary tree. When there are n items in the heap, those items occupy array positions 1, 2, ..., n .



`__init__` and `purge` Methods

Program `□` defines the `BinaryHeap __init__` method. In addition to `self`, the `__init__` method takes a single argument of type `int` which specifies the maximum capacity of the binary heap. The `__init__` method allocates an array of the specified size and sets the base index of the array to one. This is done because array position zero will not be used. The running time of the `__init__` method is $O(n)$, where n is the maximum length of the priority queue.

The purpose of the `purge` method is to make the priority queue empty. The `purge` method assigns the value `None` to the array positions one-by-one. Clearly the worst-case running time for the `purge` method is $O(n)$, where n is the maximum length of the priority queue.

Putting Items into a Binary Heap

There are two requirements which must be satisfied when an item is inserted in a binary heap. First, the resulting tree must have the correct shape. Second, the tree must remain heap-ordered. Figure □ illustrates the way in which this is done.

Since the resulting tree must be a complete tree, there is only one place in the tree where a node can be added. That is, since the bottom level must be filled from left to right, the new node must be added at the next available position in the bottom level of the tree as shown in Figure □(a).

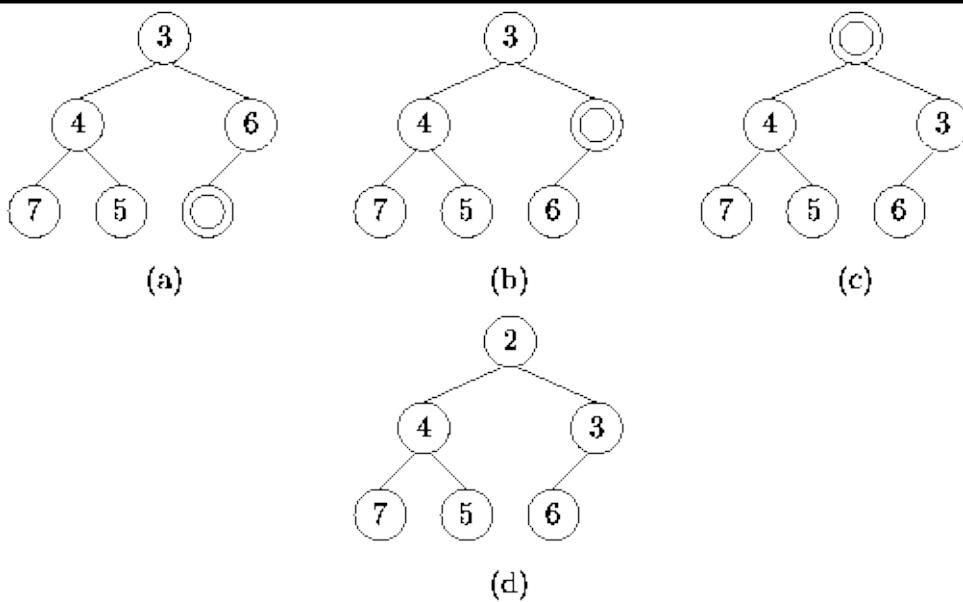


Figure: Inserting an item into a binary heap.

In this example, the new item to be inserted has the key 2. Note that we cannot simply drop the new item into the next position in the complete tree because the resulting tree is no longer heap ordered. Instead, the hole in the heap is moved toward the root by moving items down in the heap as shown in Figure □(b) and (c). The process of moving items down terminates either when we reach the root of the tree or when the hole has been moved up to a position in which when the new item is inserted the result is a heap.

Program □ gives the code for inserting an item in a binary heap. In addition to `self`, the `enqueue` method of the `BinaryHeap` class takes as its argument the item

to be inserted in the heap. If the priority queue is full an exception is thrown. Otherwise, the item is inserted as described above.

```
1 class BinaryHeap(PriorityQueue):
2
3     def enqueue(self, obj):
4         if self._count == len(self._array):
5             raise ContainerFull
6         self._count += 1
7         i = self._count
8         while i > 1 and self._array[i/2] > obj:
9             self._array[i] = self._array[i / 2]
10            i /= 2
11        self._array[i] = obj
12
13    # ...
```

Program: BinaryHeap class enqueue method.

The implementation of the algorithm is actually remarkably simple. Lines 7-10 move the hole in the heap up by moving items down. When the loop terminates, the new item can be inserted at position i . Therefore, the loop terminates either at the root, $i=1$, or when the key in the parent of i , which is found at position $\lfloor i/2 \rfloor$, is smaller than the item to be inserted.

Notice too that the subscript calculations involve only division by two. Therefore, the divisions can be replaced by bitwise right shifts which usually run much more quickly.

Since the depth of a complete binary tree with n nodes is $\lceil \log_2 n \rceil$, the worst case running time for the enqueue operation is

$$\lceil \log_2 n \rceil T(\text{gt}) + O(\log n),$$

where $T(\text{gt})$ is the time required to compare two objects. If $T(\text{gt}) = O(1)$, the enqueue operation is simply $O(\log n)$ in the worst case.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Removing Items from a Binary Heap

The `dequeueMin` method removes from a priority queue the item having the smallest key. In order to remove the smallest item, it needs first to be located. Therefore, the `dequeueMin` operation is closely related to the `getMin` operation.

The smallest item is always at the root of a min heap. Therefore, the `getMin` operation is trivial. Program □ gives the code for the `getMin` method of the `BinaryHeap` class. Assuming that no exception is thrown, the running time of `getMin` is clearly $O(1)$.

```
1  class BinaryHeap(PriorityQueue):
2
3      def getMin(self):
4          if self._count == 0:
5              raise ContainerEmpty
6          return self._array[1]
7
8      # ...
```

Program: `BinaryHeap` class `getMin` method.

Since the bottom row of a complete tree is filled from left to right as items are added, it follows that the bottom row must be emptied from right to left as items are removed. So, we have a problem: The datum to be removed from the heap by `dequeueMin` is in the root, but the node to be removed from the heap is in the bottom row.

Figure □ (a) illustrates the problem. The `dequeueMin` operation removes the key 2 from the heap, but it is the node containing key 6 that must be removed from the tree to make it into a complete tree again. When key 2 is removed from the root, a hole is created in the tree as shown in Figure □ (b).

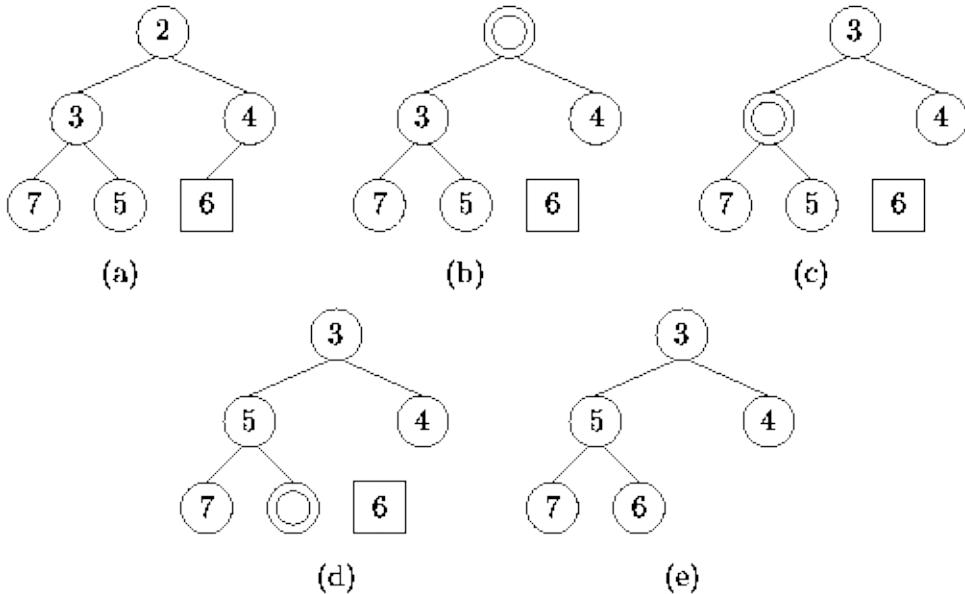


Figure: Removing an item from a binary heap.

The trick is to move the hole down in the tree to a point where the left-over key, in this case the key 6, can be reinserted into the tree. To move a hole down in the tree, we consider the children of the empty node and move up the smallest key. Moving up the smallest key ensures that the result will be a min heap.

The process of moving up continues until either the hole has been pushed down to a leaf node, or until the hole has been pushed to a point where the left over key can be inserted into the heap. In the example shown in Figure □ (b)-(c), the hole is pushed from the root node to a leaf node where the key 6 is ultimately placed is shown in Figure □ (d).

Program □ gives the code for the `dequeueMin` method of the `BinaryHeap` class. This method implements the deletion algorithm described above. The main loop (lines 10-18) moves the hole in the tree down by moving up the child with the smallest key until either a leaf node is reached or until the hole has been moved down to a point where the last element of the array can be reinserted.

```

1  class BinaryHeap(PriorityQueue):
2
3      def dequeueMin(self):
4          if self._count == 0:
5              raise ContainerEmpty
6          result = self._array[1]
7          last = self._array[self._count]
8          self._count -= 1
9          i = 1
10         while 2 * i < self._count + 1:
11             child = 2 * i
12             if child + 1 < self._count + 1 \
13                 and self._array[child + 1] < self._array[child]:
14                 child += 1
15             if last <= self._array[child]:
16                 break
17             self._array[i] = self._array[child]
18             i = child
19             self._array[i] = last
20         return result
21
22     # ...

```

Program: BinaryHeap class dequeueMin method.

In the worst case, the hole must be pushed from the root to a leaf node. Each iteration of the loop makes at most two object comparisons and moves the hole down one level. Therefore, the running time of the dequeueMin operation is

$$\lceil \log_2 n \rceil (\mathcal{T}(\text{--lt--}) + \mathcal{T}(\text{--le--})) + O(\log n),$$

where $n = \text{count}$ is the number of items in the heap. If $\mathcal{T}(\text{--lt--}) = O(1)$ and $\mathcal{T}(\text{--le--}) = O(1)$, the dequeueMin operation is simply $O(\log n)$ in the worst case.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Leftist Heaps

A leftist heap is a heap-ordered binary tree which has a very special shape called a *leftist tree*. One of the nice properties of leftist heaps is that it is possible to merge two leftist heaps efficiently. As a result, leftist heaps are suited for the implementation of mergeable priority queues.

- [Leftist Trees](#)
 - [Implementation](#)
 - [Merging Leftist Heaps](#)
 - [Putting Items into a Leftist Heap](#)
 - [Removing Items from a Leftist Heap](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Leftist Trees

A *leftist tree* is a tree which tends to ``lean'' to the left. The tendency to lean to the left is defined in terms of the shortest path from the root to an external node. In a leftist tree, the shortest path to an external node is always found on the right.

Every node in binary tree has associated with it a quantity called its *null path length* which is defined as follows:

Definition (Null Path and Null Path Length)

Consider an arbitrary node x in some binary tree T . The *null path* of node x is the shortest path in T from x to an external node of T .

The *null path length* of node x is the length of its null path.

Sometimes it is convenient to talk about the null path length of an entire tree rather than of a node:

Definition (Null Path Length of a Tree)

The *null path length* of an empty tree is zero and the null path length of a non-empty binary tree $T = \{R, T_L, T_R\}$ is the null path length its root R .

When a new node or subtree is attached to a given tree, it is usually attached in place of an external node. Since the null path length of a tree is the length of the shortest path from the root of the tree to an external node, the null path length gives a lower bound on the cost of insertion. For example, the running time for insertion in a binary search tree, Program \square , is at least

$$dT\langle \text{comp} \rangle + \Omega(d)$$

where d is the null path length of the tree.

A *leftist tree* is a tree in which the shortest path to an external node is always on the right. This informal idea is defined more precisely in terms of the null path lengths as follows:

Definition (Leftist Tree) A *leftist tree* is a binary tree T with the

following properties:

1. Either $T = \emptyset$; or
2. $T = \{R, T_L, T_R\}$, where both T_L and T_R are leftist trees which have null path lengths d_L and d_R , respectively, such that

$$d_L \geq d_R.$$

Figure □ shows an example of a leftist heap. A leftist heap is simply a heap-ordered leftist tree. The external depth of the node is shown to the right of each node in Figure □. The figure clearly shows that it is not necessarily the case in a leftist tree that the number of nodes to the left of a given node is greater than the number to the right. However, it is always the case that the null path length on the left is greater than or equal to the null path length on the right for every node in the tree.

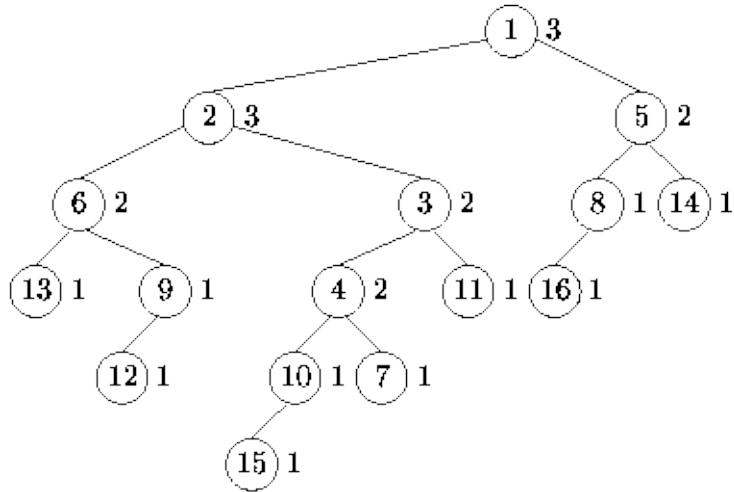


Figure: A leftist heap.

The reason for our interest in leftist trees is illustrated by the following theorems:

Theorem Consider a leftist tree T which contains n internal nodes. The path leading from the root of T downwards to the rightmost external node contains at most $\lfloor \log_2(n + 1) \rfloor$ nodes.

extbfProof Assume that T has null path length d . Then T must contain at least 2^{d-1} leaves. Otherwise, there would be a shorter path than d from the root of T to

an external node.

A binary tree with exactly l leaves has exactly $l-1$ non-leaf internal nodes. Since T has at least 2^{d-1} leaves, it must contain at least $n \geq 2^d - 1$ internal nodes altogether. Therefore, $d \leq \log_2(n + 1)$.

Since T is a leftist tree, the shortest path to an external node must be the path on the right. Thus, the length of the path to the rightmost external is at most $\lfloor \log_2(n + 1) \rfloor$.

There is an interesting dichotomy between AVL balanced trees and leftist trees. The shape of an AVL tree satisfies the AVL balance condition which stipulates that the difference in the heights of the left and right subtrees of every node may differ by at most one. The effect of AVL balancing is to ensure that the height of the tree is $O(\log n)$.

On the other hand, leftist trees have an ``imbalance condition'' which requires the null path length of the left subtree to be greater than or equal to that of the right subtree. The effect of the condition is to ensure that the length of the right path in a leftist tree is $O(\log n)$. Therefore, by devising algorithms for manipulating leftist heaps which only follow the right path of the heap, we can achieve running times which are logarithmic in the number of nodes.

The dichotomy also extends to the structure of the algorithms. For example, an imbalance sometimes results from an insertion in an AVL tree. The imbalance is rectified by doing rotations. Similarly, an insertion into a leftist tree may result in a violation of the ``imbalance condition.'' That is, the null path length of the right subtree of a node may become greater than that of the left subtree. Fortunately, it is possible to restore the proper condition simply by swapping the left and right subtrees of that node.

Implementation

This section presents an implementation of leftist heaps that is based on the binary tree implementation described in Section 1. Program 1 introduces the `LeftistHeap` class. The `LeftistHeap` class extends the `BinaryTree` class introduced in Program 1 and the abstract `MergeablePriorityQueue` class defined in Program 1.

```
 1  class LeftistHeap(BinaryTree, MergeablePriorityQueue):
 2
 3      def __init__(self, *args):
 4          if len(args) == 0:
 5              super(LeftistHeap, self).__init__()
 6              self._nullPathLength = 0
 7          else:
 8              super(LeftistHeap, self).__init__(
 9                  args[0], LeftistHeap(), LeftistHeap())
10              self._nullPathLength = 1
11
12      # ...
```

Program: `LeftistHeap` class `__init__` method.

-
- [Instance Attributes](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Instance Attributes

Since a leftist heap is a heap-ordered binary tree, it inherits from the `BinaryTree` base class the three instance attributes: `_key`, `_left`, and `_right`. The `_key` refers to the object contained in the given node and the `_left` and `_right` instance attributes refer to the left and right subtrees of the given node, respectively. In addition, the instance attribute `_nullPathLength` records the null path length of the given node. By recording the null path length in the node, it is possible to check the leftist heap balance condition in constant time.

Merging Leftist Heaps

In order to merge two leftist heaps, say h_1 and h_2 , declared as follows

```
h1 = LeftistHeap()  
h2 = LeftistHeap()
```

we invoke the `merge` method like this:

```
h1.merge(h2)
```

The effect of the `merge` method is to take all the nodes from h_2 and to attach them to h_1 , thus leaving h_2 as the empty heap.

In order to achieve a logarithmic running time, it is important for the `merge` method to do all its work on the right sides of h_1 and h_2 . It turns out that the algorithm for merging leftist heaps is actually quite simple.

To begin with, if h_1 is the empty heap, then we can simply swap the contents of h_1 and h_2 . Otherwise, let us assume that the root of h_2 is larger than the root of h_1 . Then we can merge the two heaps by recursively merging h_2 with the *right* subheap of h_1 . After doing so, it may turn out that the right subheap of h_1 now has a larger null path length than the left subheap. This we rectify by swapping the left and right subheaps so that the result is again leftist. On the other hand, if h_2 initially has the smaller root, we simply exchange the roles of h_1 and h_2 and proceed as above.

Figure □ illustrates the merge operation. In this example, we wish to merge the two trees T_1 and T_2 shown in Figure □(a). Since T_2 has the larger root, it is recursively merged with the right subtree of T_1 . The result of that merge replaces the right subtree of T_1 as shown in Figure □(b). Since the null path length of the right subtree is now greater than the left, the subtrees of T_1 are swapped giving the leftist heap shown in Figure □(c).

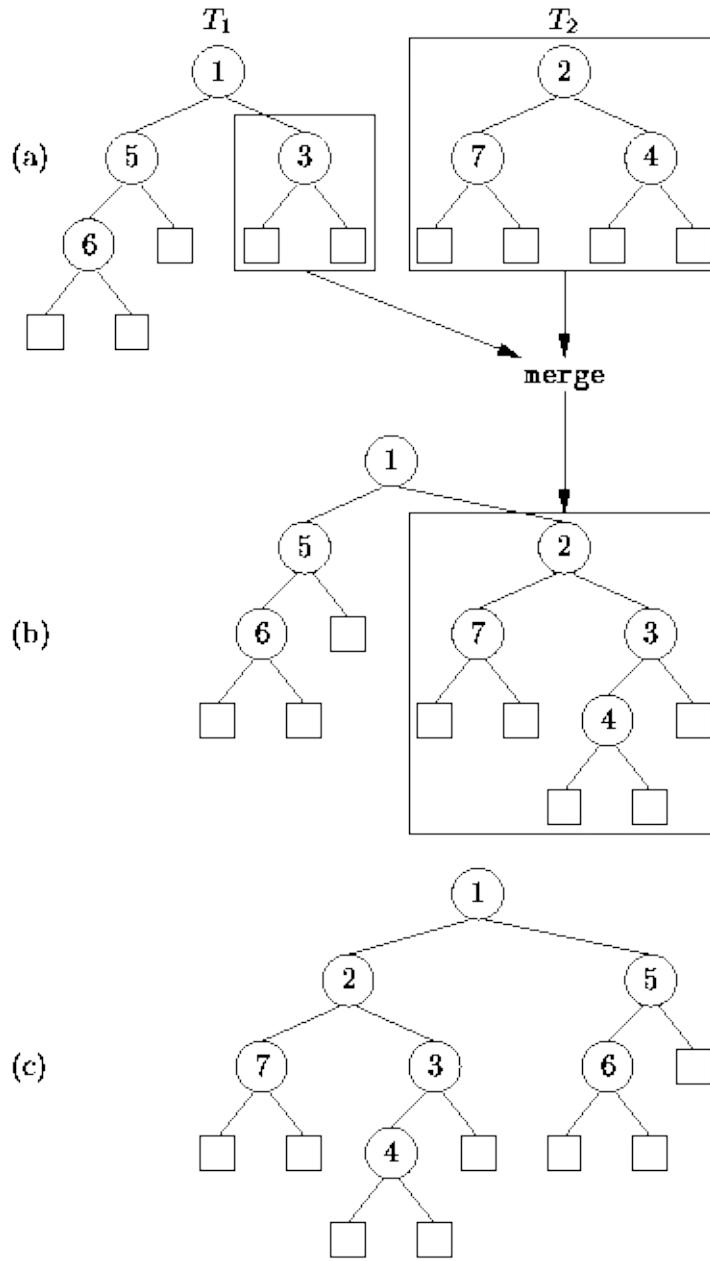


Figure: Merging leftist heaps.

Program \square gives the code for the `merge` method of the `LeftistHeap` class. The `merge` method makes use of two other methods, `swapContentsWith` and `swapSubtrees`. In addition to `self`, the `swapContentsWith` method takes as its argument a leftist heap, and exchanges all the contents (key and subtrees) of the `self` heap with the given one. The `swapSubtrees` method exchanges the left and right subtrees of a leftist heap. The implementation of these routines is trivial and is left as a project for the reader (Project \square). Clearly, the worst-case running

time for each of these routines is $O(1)$.

The `merge` method only visits nodes on the rightmost paths of the trees being merged. Suppose we are merging two trees, say T_1 and T_2 , with null path lengths d_1 and d_2 , respectively. Then the running time of the `Merge` method is

$$(d_1 - 1 + d_2 - 1)\mathcal{T}(-gt-) + O(d_1 + d_2)$$

where $\mathcal{T}(-gt-)$ is time required to compare two keys. If we assume that the time to compare two keys is a constant, then we get $O(\log n_1 + \log n_2)$, where n_1 and n_2 are the number of internal nodes in trees T_1 and T_2 , respectively.

```
1 class LeftistHeap(BinaryTree, MergeablePriorityQueue):
2
3     def merge(self, queue):
4         if self.isEmpty():
5             self.swapContentsWith(queue)
6         elif not queue.isEmpty():
7             if self._key > queue._key:
8                 self.swapContentsWith(queue)
9                 self._right.merge(queue)
10                if self._left._nullPathLength \
11                    < self._right._nullPathLength:
12                    self.swapSubtrees()
13                self._nullPathLength = 1 + min(
14                    self._left._nullPathLength,
15                    self._right._nullPathLength)
16
17     # ...
```

Program: `LeftistHeap` class `merge` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Putting Items into a Leftist Heap

The enqueue method of the `LeftistHeap` class is used to put items into the heap. `enqueue` is easily implemented using the `merge` operation. That is, to enqueue an item in a given heap, we simply create a new heap containing the one item to be enqueued and merge it with the given heap. The algorithm to do this is shown in Program □.

```
1  class LeftistHeap(BinaryTree, MergeablePriorityQueue):
2
3      def enqueue(self, obj):
4          self.merge(LeftistHeap(obj))
5
6      # ...
```

Program: `LeftistHeap` class `enqueue` method.

The expression for the running time for the `enqueue` operation follows directly from that of the `merge` operation. That is, the time required for the `enqueue` operation in the worst case is

$$(d - 1)T(\text{gt}) + O(d),$$

where d is the null path length of the heap into which the item is enqueued. If we assume that two keys can be compared in constant time, the running time for `enqueue` becomes simply $O(\log n)$, where n is the number of nodes in the tree into which the item is enqueued.

Removing Items from a Leftist Heap

The `getMin` method returns the item with the smallest key in a given priority queue and the `dequeueMin` method removes it from the queue. Since the smallest item in a heap is found at the root, the `getMin` method is easy to implement. Program □ shows how it can be done. Clearly, the running time of `getMin` is $O(1)$.

```
1 class LeftistHeap(BinaryTree, MergeablePriorityQueue):
2
3     def getMin(self):
4         if self.isEmpty:
5             raise ContainerEmpty
6         return self._key
7
8     # ...
```

Program: `LeftistHeap` class `getMin` method.

Since the smallest item in a heap is at the root, the `dequeueMin` operation must delete the root node. Since a leftist heap is a binary heap, the root has at most two children. In general when the root is deleted, we are left with two non-empty leftist heaps. Since we already have an efficient way to merge leftist heaps, the solution is to simply merge the two children of the root to obtain a single heap again! Program □ shows how the `dequeueMin` operation of the `LeftistHeap` class can be implemented.

```
1 class LeftistHeap(BinaryTree, MergeablePriorityQueue):
2
3     def dequeueMin(self):
4         if self.isEmpty:
5             raise ContainerEmpty
6         result = self._key
7         oldLeft = self._left
8         oldRight = self._right
9         self.purge()
10        self.swapContentsWith(oldLeft)
11        self.merge(oldRight)
12        return result
13
14    # ...
```

Program: `LeftistHeap` class `dequeueMin` method.

The running time of Program □ is determined by the time required to merge the two children of the root (line 11) since the rest of the work in `dequeueMin` can be done in constant time. Consider the running time to delete the root of a leftist heap T with n internal nodes. The running time to merge the left and right subtrees of T

$$(d_L - 1 + d_R - 1)\mathcal{T}(\cdot) + O(d_L + d_R),$$

where d_L and d_R are the null path lengths of the left and right subtrees T , respectively. In the worst case, $d_R = 0$ and $d_L = \lfloor \log_2 n \rfloor$. If we assume that $\mathcal{T}(\cdot) = O(1)$, the running time for `dequeueMin` is $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Binomial Queues

A binomial queue is a priority queue that is implemented not as a single tree but as a collection of heap-ordered trees. A collection of trees is called a *forest*. Each of the trees in a binomial queue has a very special shape called a binomial tree. Binomial trees are general trees. That is, the maximum degree of a node is not fixed.

The remarkable characteristic of binomial queues is that the merge operation is similar in structure to binary addition. That is, the collection of binomial trees that make up the binomial queue is like the set of bits that make up the binary representation of a non-negative integer. Furthermore, the merging of two binomial queues is done by adding the binomial trees that make up that queue in the same way that the bits are combined when adding two binary numbers.

- [Binomial Trees](#)
 - [Binomial Queues](#)
 - [Implementation](#)
 - [Merging Binomial Queues](#)
 - [Putting Items into a Binomial Queue](#)
 - [Removing an Item from a Binomial Queue](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

[*Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.*](#)



Binomial Trees

A binomial tree is a general tree with a very special shape:

Definition (Binomial Tree) The *binomial tree of order* $k \geq 0$ with root R is the tree B_k defined as follows

1. If $k=0$, $B_k = B_0 = \{R\}$. That is, the binomial tree of order zero consists of a single node, R .
2. If $k>0$, $B_k = \{R, B_0, B_1, \dots, B_{k-1}\}$. That is, the binomial tree of order $k>0$ comprises the root R , and k binomial subtrees, B_0, B_1, \dots, B_{k-1} .

Figure □ shows the first five binomial trees, $B_0 - B_4$. It follows directly from Definition □ that the root of B_k , the binomial tree of order k , has degree k . Since k may arbitrarily large, so too can the degree of the root. Furthermore, the root of a binomial tree has the largest fanout of any of the nodes in that tree.

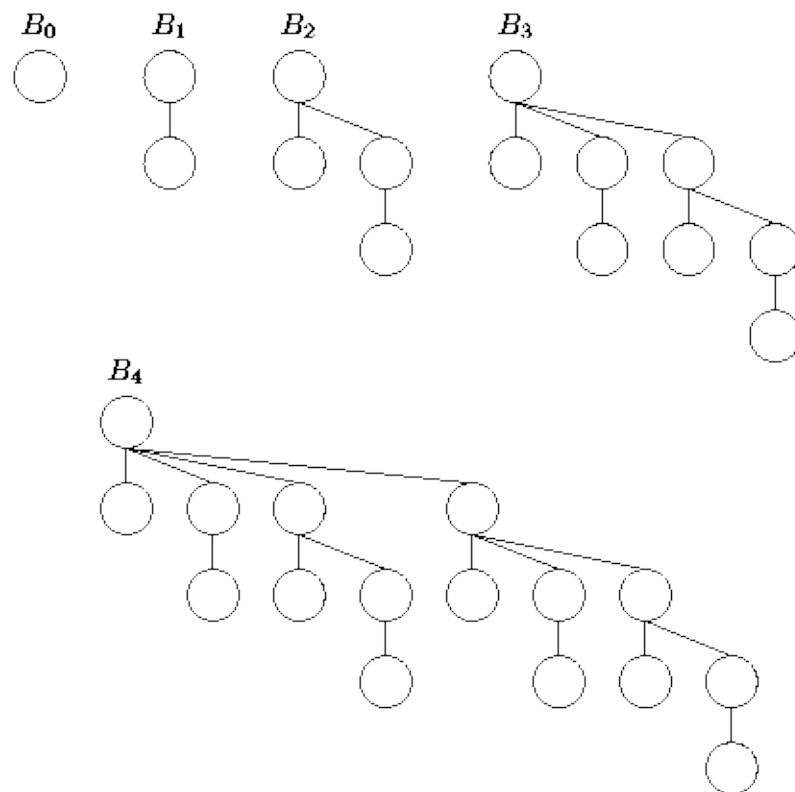


Figure: Binomial trees B_0, B_1, \dots, B_4 .

The number of nodes in a binomial tree of order k is a function of k :

Theorem The binomial tree of order k , B_k , contains 2^k nodes.

extbfProof (By induction). Let n_k be the number of nodes in B_k , a binomial tree of order k .

Base Case By definition, B_0 consists a single node. Therefore $n_0 = 1 = 2^0$.

Inductive Hypothesis Assume that $n_k = 2^k$ for $k = 0, 1, 2, \dots, l$, for some $l \geq 0$. Consider the binomial tree of order $l+1$:

$$B_{l+1} = \{R, B_0, B_1, B_2, \dots, B_l\}.$$

Therefore the number of nodes in B_{l+1} is given by

$$\begin{aligned} n_{l+1} &= 1 + \sum_{i=0}^l n_i \\ &= 1 + \sum_{i=0}^l 2^i \\ &= 1 + \frac{2^{l+1} - 1}{2 - 1} \\ &= 2^{l+1}. \end{aligned}$$

Therefore, by induction on l , $n_k = 2^k$ for all $k \geq 0$.

It follows from Theorem \square that binomial trees only come in sizes that are a power of two. That is, $n_k \in \{1, 2, 4, 8, 16, \dots\}$. Furthermore, for a given power of two, there is exactly one shape of binomial tree.

Theorem The height of B_k , the binomial tree of order k , is k .

extbfProof (By induction). Let h_k be the height of B_k , a binomial tree of order k .

Base Case By definition, B_0 consists a single node. Therefore $h_0 = 0$.

Inductive Hypothesis Assume that $h_k = k$ for $k = 0, 1, 2, \dots, l$, for some $l \geq 0$. Consider the binomial tree of order $l+1$:

$$B_{l+1} = \{R, B_0, B_1, B_2, \dots, B_l\}.$$

Therefore the height B_{l+1} is given by

$$\begin{aligned} h_{l+1} &= 1 + \max_{0 \leq i \leq l} h_i \\ &= 1 + \max_{0 \leq i \leq l} i \\ &= l + 1. \end{aligned}$$

Therefore, by induction on l , $h_k = k$ for all $k \geq 0$.

Theorem \square tells us that the height of a binomial tree of order k is k and Theorem \square tells us that the number of nodes is $n_k = 2^k$. Therefore, the height of B_k is exactly $O(\log n)$.

Figure \square shows that there are two ways to think about the construction of binomial trees. The first way follows directly from the Definition \square . That is, binomial B_k consists of a root node to which the k binomial trees B_0, B_1, \dots, B_{k-1} are attached as shown in Figure \square (a).

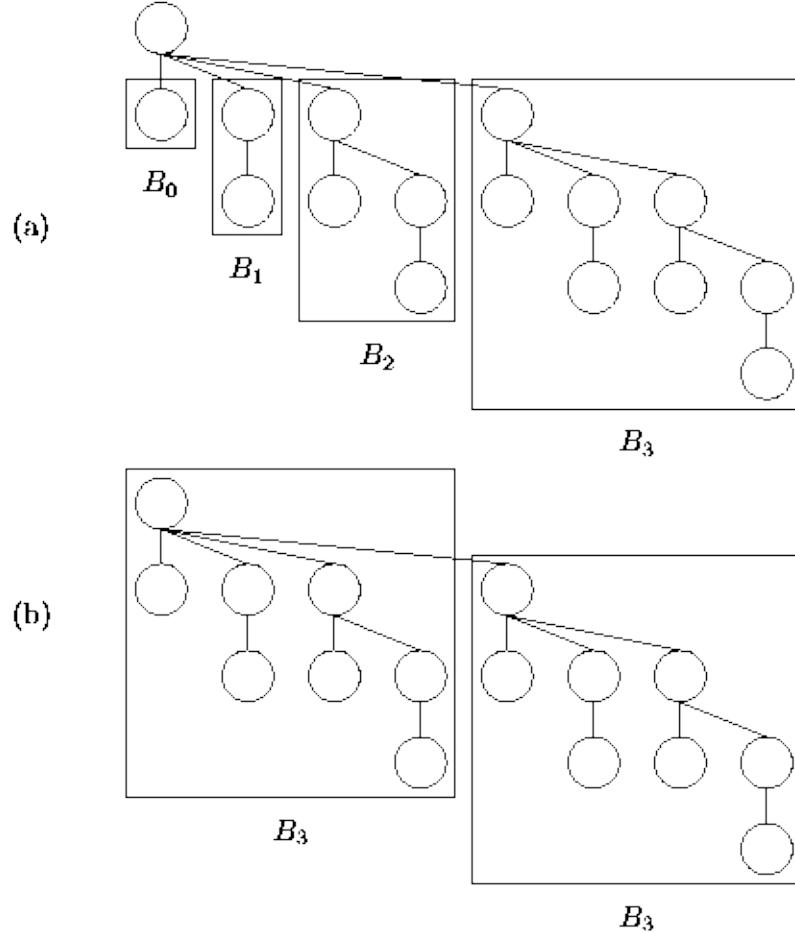


Figure: Two views of binomial tree B_4 .

Alternatively, we can think of B_k as being comprised of two binomial trees of order $k-1$. For example, Figure (b) shows that B_4 is made up of two instances of B_3 . In general, suppose we have two trees of order $k-1$, say B_{k-1}^1 and B_{k-1}^2 , where $B_{k-1}^1 = \{R^1, B_0^1, B_1^1, B_2^1, \dots, B_{k-2}^1\}$. Then we can construct a binomial tree of order k by combining the trees to get

$$B_k = \{R^1, B_0^1, B_1^1, B_2^1, \dots, B_{k-2}^1, B_{k-1}^2\}.$$

Why do we call B_k a *binomial* tree? It is because the number of nodes at a given depth in the tree is determined by the *binomial coefficient*. And the binomial coefficient derives its name from the *binomial theorem*. And the binomial theorem tells us how to compute the n^{th} power of a *binomial*. And a binomial is an expression which consists of two terms, such as $x+y$. That is why it is called a binomial tree!

Theorem (Binomial Theorem) The n^{th} power of the binomial $x+y$ for $n \geq 0$ is given by

$$(x+y)^n = \sum_{i=0}^n \binom{n}{i} x^i y^{n-i},$$

where $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ is called the *binomial coefficient*.

extbfProof The proof of the binomial theorem is left as an exercise for the reader (Exercise \square). \diamond

The following theorem gives the expression for the number of nodes at a given depth in a binomial tree:

Theorem The number of nodes at level l in B_k , the binomial tree of order k , where $0 \leq l \leq k$, is given by the *binomial coefficient* $\binom{k}{l}$.

extbfProof (By induction). Let $n_k(l)$ be the number of nodes at level l in B_k , a binomial tree of order k .

Base Case Since B_0 contains a single node, there is only one level in the tree, $n_0(0) = 1 = \binom{0}{0}$, and exactly one node at that level. Therefore,

Inductive Hypothesis Assume that $n_k(l) = \binom{k}{l}$ for $k = 0, 1, 2, \dots, h$, for some $h \geq 0$. The binomial tree of order $h+1$ is composed of two binomial trees of height h , one attached under the root of the other. Hence, the number of nodes at level l in B_{h+1} is equal to the number of nodes at level l in B_h plus the number of nodes at level $l-1$ in B_h :

$$\begin{aligned}
n_{h+1}(l) &= n_h(l) + n_h(l-1) \\
&= \binom{h}{l} + \binom{h}{l-1} \\
&= \frac{h!}{(h-l)!l!} + \frac{h!}{(h-(l-1))!(l-1)!} \\
&= \frac{h!(h+1-l)}{(h+1-l)(h-l)!l!} + \frac{h!l}{(h+1-l)!l(l-1)!} \\
&= \frac{h!(h+1-l) + h!l}{(h+1-l)!l!} \\
&= \frac{(h+1)!}{(h+1-l)!l!} \\
&= \binom{h+1}{l}
\end{aligned}$$

Therefore by induction on h , $n_k(l) = \binom{k}{l}$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Binomial Queues

If binomial trees only come in sizes that are powers of two, how do we implement a container which holds an arbitrary number number of items n using binomial trees? The answer is related to the binary representation of the number n . Every non-negative integer n can be expressed in binary form as

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i, \quad (11.1)$$

where $b_i \in \{0, 1\}$ is the i^{th} *binary digit* or *bit* in the representation of n . For example, $n=27$ is expressed as the binary number 11011_2 because $27=16+8+2+1$.

To make a container which holds exactly n items we use a collection of binomial trees. A collection of trees is called a *forest*. The forest contains binomial tree B_i if the i^{th} bit in the binary representation of n is a one. That is, the forest F_n which contains exactly n items is given by

$$F_n = \{B_i : b_i = 1\},$$

where b_i is determined from Equation \square . For example, the forest which contains 27 items is $F_{27} = \{B_4, B_3, B_1, B_0\}$.

The analogy between F_n and the binary representation of n carries over to the merge operation. Suppose we have two forests, say F_n and F_m . Since F_n contains n items and F_m contains m items, the combination of the two contains $n+m$ items. Therefore, the resulting forest is F_{n+m} .

For example, consider $n=27$ and $m=10$. In this case, we need to merge $F_{27} = \{B_4, B_3, B_1, B_0\}$ with $F_{10} = \{B_3, B_1\}$. Recall that two binomial trees of order k can be combined to obtain a binomial tree of order $k+1$. For example, $B_1 + B_1 = B_2$. But this is just like adding binary digits! In binary notation, the sum 27+10 is calculated like this:

	1	1	0	1	1
+			1	0	1

100101

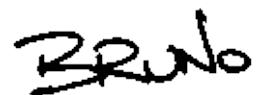
The merging of F_{27} and F_{20} is done in the same way:

$$\begin{array}{r} B_4 B_3 \emptyset B_1 B_0 F_{27} \\ + \quad B_3 \emptyset B_1 \emptyset F_{10} \\ \hline B_5 \emptyset \emptyset B_2 \emptyset B_0 F_{37} \end{array}$$

Therefore, the result is $F_{37} = \{B_5, B_2, B_0\}$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementation

- [Heap-Ordered Binomial Trees](#)
 - [Adding Binomial Trees](#)
 - [Binomial Queues](#)
 - [Instance Attributes](#)
 - [addTree and removeTree](#)
 - [getMinTree and getMin Methods](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Heap-Ordered Binomial Trees

Since binomial trees are simply general trees with a special shape, we can make use of the `GeneralTree` class presented in Section □ to implement the `BinomialTree` class. (See Figure □).

Program □ introduces the `BinomialQueue` class and the nested class `BinomialTree`. The `BinomialQueue` class extends the abstract `MergeablePriorityQueue` class defined in Program □. The `BinomialTree` class extends the `GeneralTree` class introduced in Program □.

No new instance attributes are declared in the `BinomialTree` class. Remember that the implementation of the `GeneralTree` class uses a linked list to contain the pointers to the subtrees, since the degree of a node in a general tree may be arbitrarily large. Also, the `GeneralTree` class already keeps track of the degree of a node in its `_degree` instance attribute. Since the degree of the root node of a binomial tree of order k is k , it is not necessary to keep track of the order explicitly. The `_degree` variable serves this purpose nicely.

```
1  class BinomialQueue(MergeablePriorityQueue):
2
3      class BinomialTree(GeneralTree):
4
5          def __init__(self, key):
6              super(BinomialQueue.BinomialTree, self).__init__(key)
7
8          # ...
9
10         # ...
```

Program: `BinomialQueue.BinomialTree` class `__init__` method.



Adding Binomial Trees

Recall that we can combine two binomial trees of the same order, say k , into a single binomial tree of order $k+1$. Each of the two trees to be combined is heap-ordered. Since the smallest key is at the root of a heap-ordered tree, we know that the root of the result must be the smaller root of the two trees which are to be combined. Therefore, to combine the two trees, we simply attach the tree with the larger root under the root of the tree with the smaller root. For example, Figure □ illustrates how two heap-ordered binomial trees of order two are combined into a single heap-ordered tree of order three.

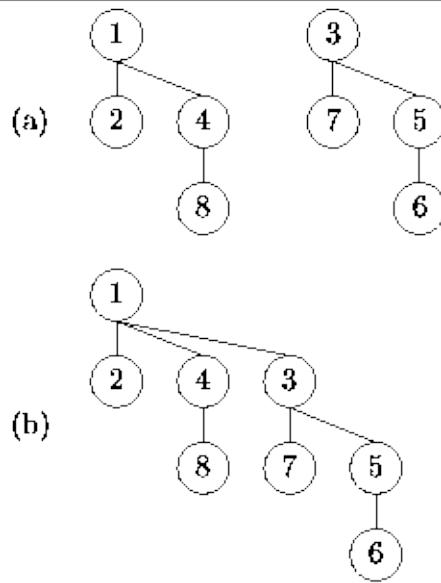


Figure: Adding binomial trees.

The add method defined in Program □ provides the means to combine two binomial trees of the same order. The add method takes two `BinomialTrees`, `self` and `tree`, and attaches the binomial tree `tree` to the binomial tree `self`. This is only permissible when both trees have the same order.

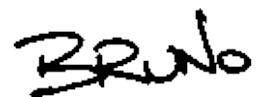
```
1 class BinomialQueue(MergeablePriorityQueue):
2
3     class BinomialTree(GeneralTree):
4
5         def add(self, tree):
6             if self._degree != tree._degree:
7                 raise ValueError
8             if self._key > tree._key:
9                 self.swapContentsWith(tree)
10            self.attachSubtree(tree)
11            return self
12
13         # ...
14
15     # ...
```

Program: BinomialTree class add method.

In order to ensure that the resulting binomial tree is heap ordered, the roots of the trees are compared. If necessary, the contents of the nodes are exchanged using swapContentsWith (lines 8-9) before the subtree is attached (line 10). Assuming swapContentsWith and attachSubtree both run in constant time, the worst-case running time of the add method is $T(-\text{gt-}) + O(1)$. That is, exactly one comparison and a constant amount of additional work is needed to combine two binomial trees.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Binomial Queues

A binomial queue is a mergeable priority queue implemented as a forest of binomial trees. In this section we present a linked-list implementation of the forest. That is, the forest is represented using a linked list of binomial trees.

Program □ shows the `__init__` method of the `BinomialQueue` class.

```
 1  class BinomialQueue(MergeablePriorityQueue):
 2
 3      def __init__(self, *args):
 4          super(BinomialQueue, self).__init__()
 5          self._treeList = LinkedList()
 6          if len(args) == 0:
 7              pass
 8          elif len(args) == 1:
 9              assert isinstance(args[0], self.BinomialTree)
10              self._treeList.append(args[0])
11          else:
12              raise ValueError
13
14      # ...
```

Program: `BinomialQueue` class `__init__` method.

Instance Attributes

The `BinomialQueue` class contains the single instance attribute `_treeList`, which is an instance of the `LinkedList` class introduced in Program □. The binomial trees contained in the linked list are stored in increasing *order*. That is, the binomial tree at the head of the list has the smallest order, and the binomial tree at the tail has the largest order.



addTree and removeTree

The `addTree` and `removeTree` methods of the `BinomialQueue` class facilitate the implementation of the various priority queue operations. These methods are defined in Program □. The `addTree` method takes a `BinomialTree` and appends that tree to `_treeList`. `addTree` also adjusts the `_count` in order to keep track of the number of items in the priority queue. It is assumed that the order of the tree which is added is larger than all the others in the list and, therefore, that it belongs at the end of the list. The running time of `addTree` is clearly $O(1)$.

```
1  class BinomialQueue(MergeablePriorityQueue):
2
3      def addTree(self, tree):
4          self._treeList.append(tree)
5          self._count += tree.count
6
7      def removeTree(self, tree):
8          self._treeList.extract(tree)
9          self._count -= tree.count
10
11     # ...
```

Program: `BinomialQueue` class `addTree` and `removeTree` methods.

The `removeTree` method takes a binomial tree and removes it from the `_treeList`. It is assumed that the specified tree is actually in the list. `removeTree` also adjust the `_count` as required. The running time of `removeTree` depends on the position of the tree in the list. A binomial queue which contains exactly n items altogether has at most $\lceil \log_2(n + 1) \rceil$ binomial trees. Therefore, the running time of `removeTree` is $O(\log n)$ in the worst case.



getMinTree and getMin Methods

A binomial queue that contains n items consists of at most $\lceil \log_2(n + 1) \rceil$ binomial trees. Each of these binomial trees is heap ordered. In particular, the smallest key in each binomial tree is at the root of that tree. So, we know that the smallest key in the queue is found at the root of one of the binomial trees, but we do not know which tree it is.

The `getMinTree` method returns the the binomial tree in the queue that has the smallest root. As shown in Program □, the `getMinTree` simply traverses the entire linked list to find the tree with the smallest key at its root. Since there are at most $\lceil \log_2(n + 1) \rceil$ binomial trees, the worst-case running time of `getMinTree` is

$$(\lceil \log_2(n + 1) \rceil - 1)T(\text{...}) + O(\log n).$$

```

1  class BinomialQueue(MergeablePriorityQueue):
2
3      def getMinTree(self):
4          minTree = None
5          ptr = self._treeList.head
6          while ptr is not None:
7              tree = ptr.datum
8              if minTree is None or tree.key < minTree.key:
9                  minTree = tree
10             ptr = ptr.next
11         return minTree
12
13     minTree = property(
14         fget = lambda self: self.getMinTree())
15
16     def getMin(self):
17         if self._count == 0:
18             raise ContainerEmpty
19         return self.minTree.key
20
21     # ...

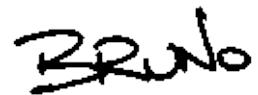
```

Program: `BinomialQueue` class `getMinTree` and `getMin` methods.

Program □ also defines the `getMin` method that returns the smallest key in the priority queue. The `getMin` method uses the `minTree` property to locate the tree with the smallest key at its root and returns that key. Clearly, the asymptotic running time of the `getMin` accessor is the same as that of the `getMinTree` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Merging Binomial Queues

Merging two binomial queues is like doing binary addition. For example, consider the addition of F_{27} and F_{10} :

$$\begin{array}{r} B_4 B_3 \emptyset B_1 B_0 F_{27} \\ + B_3 \emptyset B_1 \emptyset F_{10} \\ \hline B_5 \emptyset \emptyset B_2 \emptyset B_0 F_{37} \end{array}$$

The usual algorithm for addition begins with the least-significant ``bit.'' Since F_{27} contains a B_0 tree and F_{10} does not, the result is simply the B_0 tree from F_{27} .

In the next step, we add the B_1 from F_{27} and the B_1 from F_{10} . Combining the two B_1 's we get a B_2 which we *carry* to the next column. Since there are no B_1 's left, the result does not contain any. The addition continues in a similar manner until all the columns have been added up.

Program `merge` gives an implementation of this addition algorithm. In addition to `self`, the `merge` method of the `BinomialQueue` class takes a `BinomialQueue` and adds its subtrees to the binomial queue `self`.

```

1 class BinomialQueue(MergeablePriorityQueue):
2
3     def merge(self, queue):
4         oldList = self._treeList
5         self._treeList = LinkedList()
6         self._count = 0
7         p = oldList.head
8         q = queue._treeList.head
9         carry = None
10        i = 0
11        while p is not None or q is not None \
12            or carry is not None:
13            a = None
14            if p is not None:
15                tree = p.datum
16                if tree.degree == i:
17                    a = tree
18                    p = p.next
19            b = None
20            if q is not None:
21                tree = q.datum
22                if tree.degree == i:
23                    b = tree
24                    q = q.next
25            (sum, carry) = BinomialQueue.fullAdder(a, b, carry)
26            if sum is not None:
27                self.addTree(sum)
28            i += 1
29            queue.purge()
30
31    # ...

```

Program: BinomialQueue class merge method.

Each iteration of the main loop of the algorithm (lines 11-28) computes the i^{th} ``bit'' of the result--the i^{th} bit is a binomial tree of order i . At most three terms need to be considered: the carry from the preceding iteration and two B_i 's, one from each of the queues that are being merged.

The method fullAdder computes the result required in each iteration. Program \square defines fullAdder method. In the worst case, the fullAdder method calls the add method to combine two BinomialTrees into one. Therefore, the worst-case running time for fullAdder is

$$\mathcal{T}(\text{gt}) + O(1).$$

```

1 class BinomialQueue(MergeablePriorityQueue):
2
3     def fullAdder(a, b, c):
4         if a is None:
5             if b is None:
6                 if c is None:
7                     return (None, None)
8                 else:
9                     return (c, None)
10            else:
11                if c is None:
12                    return (b, None)
13                else:
14                    return (None, b.add(c))
15        else:
16            if b is None:
17                if c is None:
18                    return (a, None)
19                else:
20                    return (None, a.add(c))
21        else:
22            if c is None:
23                return (None, a.add(b))
24            else:
25                return (c, a.add(b))
26    fullAdder = staticmethod(fullAdder)
27
28    # ...

```

Program: BinomialQueue class fullAdder method.

Suppose the merge method of Program □ is used to combine a binomial queue with n items with another that contains m items. Since the resulting priority queue contains $n+m$ items, there are at most $\lceil \log_2(n+m+1) \rceil$ binomial trees in the result. Thus, the worst-case running time for the merge operation is

$$\lceil \log_2(n+m+1) \rceil T(\text{gt}) + O(\log(n+m)).$$

Putting Items into a Binomial Queue

With the `merge` method at our disposal, the enqueue operation is easy to implement. To enqueue an item in a given binomial queue, we create another binomial queue that contains just the one item to be enqueued and merge that queue with the original one.

Program □ shows how easily this can be done. Creating the binomial tree B_0 with the one object at its root (lines 4-5) takes a constant amount of time. Finally, the time required to merge the two queues is

$$[\log_2(n + 2)] \mathcal{T} \{ \dots \} + O(\log n),$$

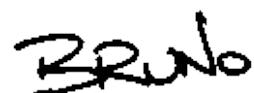
where n is the number of items originally in the queue.

```
1 class BinomialQueue(MergeablePriorityQueue):
2
3     def enqueue(self, obj):
4         self.merge(BinomialQueue(
5             BinomialQueue.BinomialTree(obj)))
6
7     # ...
```

Program: `BinomialQueue` class `enqueue` method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Removing an Item from a Binomial Queue

A binomial queue is a forest of heap-ordered binomial trees. Therefore, to dequeue the smallest item from the queue, we must withdraw the root of one of the binomial trees. But what do we do with the rest of the tree once its root has been removed?

The solution lies in realizing that the collection of subtrees of the root of a binomial tree is a forest! For example, consider the binomial tree of order k ,

$$B_k = \{R, B_0, B_1, B_2, \dots, B_{k-1}\}$$

Taken all together, its subtrees form the binomial queue F_{2^k-1} :

$$F_{2^k-1} = \{B_0, B_1, B_2, \dots, B_{k-1}\}.$$

Therefore, to delete the smallest item from a binomial queue, we first identify the binomial tree with the smallest root and remove that tree from the queue. Then, we consider all the subtrees of the root of that tree as a binomial queue and merge that queue back into the original one. Program \square shows how this can be coded.

```
1 class BinomialQueue(MergeablePriorityQueue):
2
3     def dequeueMin(self):
4         if self._count == 0:
5             raise ContainerEmpty
6         minTree = self.minTree
7         self.removeTree(minTree)
8         queue = BinomialQueue()
9         while minTree.degree > 0:
10             child = minTree.getSubtree(0)
11             minTree.detachSubtree(child)
12             queue.addTree(child)
13             self.merge(queue)
14         return minTree.key
15
16     # ...
```

Program: BinomialQueue class dequeueMin method.

The dequeueMin method begins by using the `minTree` accessor to find the tree

with the smallest root and then removing that tree using `removeTree` (lines 6-7). The time required to find the appropriate tree and to remove it is

$$(\lceil \log_2(n+1) \rceil - 1)T_{\text{removeTree}} + O(\log n),$$

where n is the number of items in the queue.

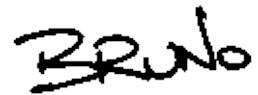
A new binomial queue is created on line 8. All the children of the root of the minimum tree are detached from the tree and added to the new binomial queue (lines 9-12). In the worst case, the minimum tree is the one with the highest order. i.e., $B_{\lfloor \log_2 n \rfloor}$, and the root of that tree has $\lfloor \log_2 n \rfloor$ children. Therefore, the running time of the loop on lines 9-12 is $O(\log n)$.

The new queue is then merged with the original one (line 13). Since the resulting queue contains $n-1$ keys, the running time for the `merge` operation in this case is

$$\lceil \log_2 n \rceil T_{\text{merge}} + O(\log n).$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

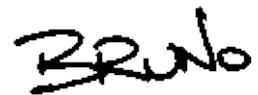


Applications

- [Discrete Event Simulation](#)
 - [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Discrete Event Simulation

One of the most important applications of priority queues is in *discrete event simulation*. Simulation is a tool which is used to study the behavior of complex systems. The first step in simulation is *modeling*. We construct a mathematical model of the system we wish to study. Then we write a computer program to evaluate the model. In a sense the behavior of the computer program mimics the system we are studying.

The systems studied using *discrete event simulation* have the following characteristics: The system has a *state* which evolves or changes with time. Changes in state occur at distinct points in simulation time. A state change moves the system from one state to another instantaneously. State changes are called *events*.

For example, suppose we wish to study the service received by customers in a bank. Suppose a single teller is serving customers. If the teller is not busy when a customer arrives at the bank, the that customer is immediately served. On the other hand, if the teller is busy when another customer arrives, that customer joins a queue and waits to be served.

We can model this system as a discrete event process as shown in Figure □. The state of the system is characterized by the state of the server (the teller), which is either busy or idle, and by the number of customers in the queue. The events which cause state changes are the arrival of a customer and the departure of a customer.

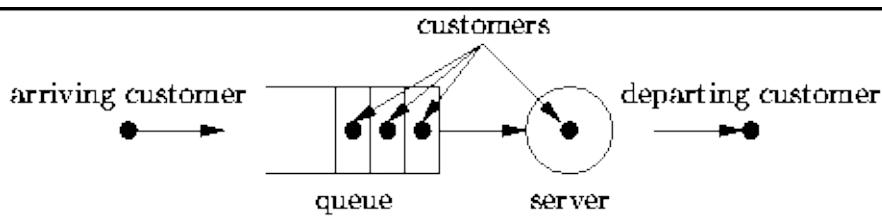


Figure: A simple queueing system.

If the server is idle when a customer arrives, the server immediately begins to serve the customer and therefore changes its state to busy. If the server is busy when a customer arrives, that customer joins the queue.

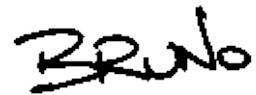
When the server finishes serving the customer, that customer departs. If the queue is not empty, the server immediately commences serving the next customer. Otherwise, the server becomes idle.

How do we keep track of which event to simulate next? Each event (arrival or departure) occurs at a discrete point in *simulation time*. In order to ensure that the simulation program is correct, it must compute the events in order. This is called the *causality constraint*--events cannot change the past.

In our model, when the server begins to serve a customer we can compute the departure time of that customer. So, when a customer arrives at the server we *schedule* an event in the future which corresponds to the departure of that customer. In order to ensure that events are processed in order, we keep them in a priority queue in which the time of the event is its priority. Since we always process the pending event with the smallest time next and since an event can schedule new events only in the future, the causality constraint will not be violated.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementation

This section presents the simulation of a system comprised of a single queue and server as shown in Figure 1. Program 1 defines the nested class `Simulation.Event` which represents events in the simulation.

```
1  class Simulation(object):
2
3      class Event(Association):
4
5          def __init__(self, type, time):
6              super(Simulation.Event, self).__init__(time, type)
7
8          time = property(
9              fget = lambda self: self.getKey())
10
11         type = property(
12             fget = lambda self: self.getValue())
13
14     # ...
```

Program: `Simulation.Event` class.

Since events will be put into a priority queue, the `Event` class is derived from the `Association` class introduced in Section 1. An association is an ordered pair comprised of a key and a value. In the case of the `Event` class, the key is the *time* of the event and the value is the *type* of the event. Therefore, the events in a priority queue are prioritized by their times.

Program 1 defines the `run` method which implements the discrete event simulation. In addition to `self`, this method takes one argument, `timeLimit`, which specifies the total amount of time to be simulated.

The `Simulation` class uses several instance attributes. The first, `_eventList`, is a priority queue. This priority queue is used to hold the events during the course of the simulation.

```
1 class Simulation(object):
2
3     ARRIVAL = 0
4     DEPARTURE = 1
5
6     def __init__(self):
7         super(Simulation, self).__init__()
8         self._eventList = LeftistHeap()
9         self._serverBusy = False
10        self._numberInQueue = 0
11        self._serviceTime = ExponentialRV(100.0)
12        self._interArrivalTime = ExponentialRV(100.0)
13
14    def run(self, timeLimit):
15        self._eventList.enqueue(self.Event(self.ARRIVAL, 0))
16        while not self._eventList.isEmpty():
17            evt = self._eventList.dequeueMin()
18            t = evt.time
19            if t > timeLimit:
20                self._eventList.purge()
21                break
22            if evt.type == self.ARRIVAL:
23                if not self._serverBusy:
24                    self._serverBusy = True
25                    self._eventList.enqueue(
26                        self.Event(self.DEPARTURE,
27                                   t + self._serviceTime.next))
28                else:
29                    self._numberInQueue += 1
30                    self._eventList.enqueue(self.Event(self.ARRIVAL,
31                                         t + self._interArrivalTime.next))
32            elif evt.type == self.DEPARTURE:
33                if self._numberInQueue == 0:
34                    self._serverBusy = False
35                else:
36                    self._numberInQueue -= 1
37                    self._eventList.enqueue(
38                        self.Event(self.DEPARTURE,
39                                   t + self._serviceTime.next))
40
41    # ...
```

Program: Application of priority queues--discrete event simulation.

The state of the system being simulated is represented by the two instance attributes `_serverBusy` and `_numberInQueue`. The first is a bool value which

indicates whether the server is busy. The second keeps track of the number of customers in the queue.

In addition to the state variables, there are two instances of the class `ExponentialRV`. The class `ExponentialRV` is a random number generator defined in Section 2. It extends the abstract `RandomVariable` class defined in Program 2. This class defines a property called `next` which is used to sample the random number generator. Every time `next` is accessed, a different (random) result is returned. The random values are exponentially distributed around a mean value which is specified in the `__init__` method. For example, in this case both `_serviceTime` and `_interArrivalTime` produce random distributions with the mean value of 100 (lines 11-12).

It is assumed that the `_eventList` priority queue is initially empty. The simulation begins by enqueueing a customer arrival at time zero (line 15). The while loop (lines 16-39) constitutes the main simulation loop. This loop continues as long as the `_eventList` is not empty, i.e., as long as there is an event to be simulated

Each iteration of the simulation loop begins by dequeuing the next event in the event list (line 17). If the time of that event exceeds `timeLimit`, the event is discarded, the `_eventList` is purged, and the simulation is terminated. Otherwise, the simulation proceeds.

The simulation of an event depends on the type of that event. The `if/elif` statements (lines 22 and 32) invoke the appropriate code for the given event. If the event is a customer arrival and the server is not busy, `_serverBusy` is set to `True` and the `_serviceTime` random number generator is sampled to determine the amount of time required to service the customer. A customer departure is scheduled at the appropriate time in the future (lines 23-27). On the other hand, if the server is already busy when the customer arrives, we add one to the `_numberInQueue` variable (line 29).

Another customer arrival is scheduled after every customer arrival. The `interArrivalTime` random number generator is sampled, and the arrival is scheduled at the appropriate time in the future (lines 30-31).

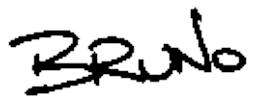
If the event is a customer departure and the queue is empty, the server becomes idle (lines 33-34). When a customer departs and there are still customers in the

queue, the next customer in the queue is served. Therefore, `_numberInQueue` is decreased by one and the `_serviceTime` random number generator is sampled to determine the amount of time required to service the next customer. A customer departure is scheduled at the appropriate time in the future (lines 36-39).

Clearly the execution of the `run` method given in Program 1 mimics the modeled system. Of course, the program given produces no output. For it to be of any practical value, the simulation program should be instrumented to allow the user to study its behavior. For example, the user may be interested in knowing statistics such as the average queue length and the average waiting time that a customer waits for service. And such instrumentation can be easily incorporated into the given framework.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exercises

1. For each of the following key sequences determine the binary heap obtained when the keys are inserted one-by-one in the order given into an initially empty heap:
 1. 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
 2. 3, 1, 4, 1, 5, 9, 2, 6, 5, 4.
 3. 2, 7, 1, 8, 2, 8, 1, 8, 2, 8.
2. For each of the binary heaps obtained in Exercise □ determine the heap obtained after three consecutive `dequeueMin` operations.
3. Repeat Exercises □ and □ for a leftist heap.
4. Show the result obtained by inserting the keys $1, 2, 3, \dots, 2^h$ one-by-one in the order given into an initially empty binomial queue.
5. A *full* binary tree is a tree in which each node is either a leaf or its is a *full node* (see Exercise □). Consider a *complete* binary tree with n nodes.
 1. For what values of n is a complete binary tree a *full* binary tree.
 2. For what values of n is a complete binary a *perfect* binary tree.
6. Prove by induction Theorem □.
7. Devise an algorithm to determine whether a given binary tree is a heap. What is the running time of your algorithm?
8. Devise an algorithm to find the *largest* item in a binary *min* heap. **Hint:** First, show that the largest item must be in one of the leaves. What is the running time of your algorithm?
9. Suppose we are given an arbitrary array of n keys to be inserted into a binary heap all at once. Devise an $O(n)$ algorithm to do this. **Hint:** See Section □.
10. Devise an algorithm to determine whether a given binary tree is a leftist tree. What is the running time of your algorithm?
11. Prove that a complete binary tree is a leftist tree.
12. Suppose we are given an arbitrary array of n keys to be inserted into a leftist heap all at once. Devise an $O(n)$ algorithm to do this. **Hint:** See Exercises □ and □.
13. Consider a complete binary tree with its nodes numbered as shown in Figure □. Let K be the number of a node in the tree. The the binary representation of K is

$$K = \sum_{i=0}^k b_i 2^i,$$

where $k = \lfloor \log_2 K \rfloor$.

1. Show that path from the root to a given node K passes through the following nodes:

$$\begin{aligned} & b_k \\ & b_k b_{k-1} \\ & b_k b_{k-1} b_{k-2} \\ & \vdots \\ & b_k b_{k-2} b_{k-2} \dots b_2 b_1 \\ & b_k b_{k-1} b_{k-2} \dots b_2 b_1 b_0. \end{aligned}$$

2. Consider a complete binary tree with n nodes. The nodes on the path from the root to the n^{th} are *special*. Show that every non-special node is the root of a perfect tree.
14. The enqueue algorithm for the `BinaryHeap` class does $O(\log n)$ object comparisons in the worst case. In effect, this algorithm does a linear search from a leaf to the root to find the point at which to insert a new key. Devise an algorithm that a binary search instead. Show that the number of comparisons required becomes $O(\log \log n)$. **Hint:** See Exercise □.
15. Prove Theorem □.
16. Do Exercise □.

Projects

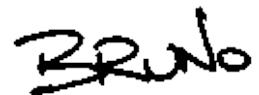
1. Design and implement a sorting algorithm using one of the priority queue implementations described in this chapter.
2. Complete the `BinaryHeap` class introduced in Program □ by providing suitable definitions for the following operations: `_compareTo`, `getIsFull`, `accept` and `_iter_`. Write a test program and test your implementation.
3. Complete the `LeftistHeap` class introduced in Program □ by providing suitable definitions for the following operations: `_init_`, `swapContentsWith` and `swapSubtrees`. You will require a complete implementation of the base class `BinaryTree`. (See Project □). Write a test program and test your implementation.
4. Complete the implementation of the `BinomialTree` class introduced in Program □ by providing suitable definitions for the following operations: `_init_`, `getCount` and `swapContentsWith`. You must also have a complete implementation of the base class `GeneralTree`. (See Project □). Write a test program and test your implementation.
5. Complete the implementation of the `BinomialQueue` class introduced in Program □ by providing suitable definitions for the following methods: `_init_`, `purge`, `_compareTo`, `accept` and `_iter_`. You must also have a complete implementation of the `BinomialTree` class. (See Project □). Write a test program and test your implementation.
6. The binary heap described in this chapter uses an array as the underlying foundational data structure. Alternatively we may base an implementation on the `BinaryTree` class described in Chapter □. Implement a priority queue class that extends the `BinaryTree` class (Program □) and the abstract `PriorityQueue` class (Program □).
7. Implement a priority queue class using the binary search tree class from Chapter □. Specifically, extend the `BinarySearchTree` class (Program □) and the abstract `PriorityQueue` class (Program □). You will require a complete implementation of the base class `BinarySearchTree`. (See Project □). Write a test program and test your implementation.
8. Devise and implement an algorithm to multiply two polynomials:

$$\left(\sum_{i=0}^n a_i x^i \right) \times \left(\sum_{j=0}^m b_j x^j \right).$$

Generate the terms of the result in order by putting intermediate product terms into a priority queue. That is, use the priority queue to group terms with the same exponent. **Hint:** See also Project □.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Sets, Multisets, and Partitions

In mathematics a *set* is a collection of elements, especially a collection having some feature or features in common. The set may have a finite number of elements, e.g., the set of prime numbers less than 100; or it may have an infinite number of elements, e.g., the set of right triangles. The *elements* of a set may be anything at all--from simple integers to arbitrarily complex objects. However, all the elements of a set are distinct--a set may contain only one instance of a given element.

For example, $\{\}$, $\{a\}$, $\{a, b, c, d\}$, and $\{d, e\}$ are all sets the elements of which are drawn from $U = \{a, b, c, d, e\}$. The set of all possible elements, U , is called the *universal set*. Note also that the elements comprising a given set are not ordered. Thus, $\{a, b, c\}$ and $\{b, c, a\}$ are the same set.

There are many possible operations on sets. In this chapter we consider the most common operations for *combining sets*--union, intersection, difference:

union

The *union* (or *conjunction*) of sets S and T , written $S \cup T$, is the set comprised of all the elements of S together with all the elements of T . Since a set cannot contain duplicates, if the same item is an element of both S and T , only one instance of that item appears in $S \cup T$. If $S = \{a, b, c, d\}$ and $T = \{d, e\}$, then $S \cup T = \{a, b, c, d, e\}$.

intersection

The *intersection* (or *disjunction*) of sets S and T is written $S \cap T$. The elements of $S \cap T$ are those items which are elements of *both* S and T . If $S = \{a, b, c, d\}$ and $T = \{d, e\}$, then $S \cap T = \{d\}$.

difference

The *difference* (or *subtraction*) of sets S and T , written $S - T$, contains those elements of S which are *not also* elements of T . That is, the result $S - T$ is obtained by taking the set S and removing from it those elements which are also found in T . If $S = \{a, b, c, d\}$ and $T = \{d, e\}$, then $S - T = \{a, b, c\}$.

Figure □ illustrates the basic set operations using a *Venn diagram*. A Venn diagram represents the membership of sets by regions of the plane. In Figure □ the two sets S and T divide the plane into the four regions labeled $I-IV$. The following table illustrates the basic set operations by enumerating the regions that comprise each set.

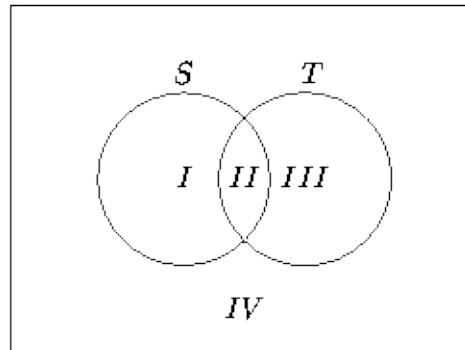


Figure: Venn diagram illustrating the basic set operations.

set region(s) of Figure □

U I, II, III, IV

S I, II

S' III, IV

T II, III

$S \cup T$ I, II, III

$S \cap T$ II

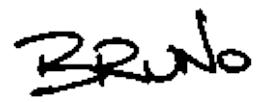
$S-T$ I

$T-S$ III

-
- [Basics](#)
 - [Array and Bit-Vector Sets](#)
 - [Multisets](#)
 - [Partitions](#)
 - [Applications](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Basics

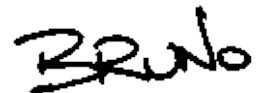
In this chapter we consider sets the elements of which are integers. By using integers as the universe rather than arbitrary objects, certain optimizations are possible. For example, we can use a bit-vector of length N to represent a set whose universe is $\{0, 1, \dots, N - 1\}$. Of course, using integers as the universe does not preclude the use of more complex objects, provided there is a one-to-one mapping between those objects and the elements of the universal set.

A crucial requirement of any set representation scheme is that it supports the common set operations including *union*, *intersection*, and set *difference*. We also need to compare sets and, specifically, to determine whether a given set is a subset of another.

-
- [Implementing Sets](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementing Sets

As discussed above, this chapter addresses the implementation of sets of integers. A set is a collection of elements. Naturally, we want to insert and withdraw objects from the collection and to test whether a given object is a member of the collection. Therefore, we consider sets as being derived from the `SearchableContainer` class defined in Chapter 1. (See Figure 1). In general, a searchable container can hold arbitrary objects. However, in this chapter we will assume that the elements of a set are integers.

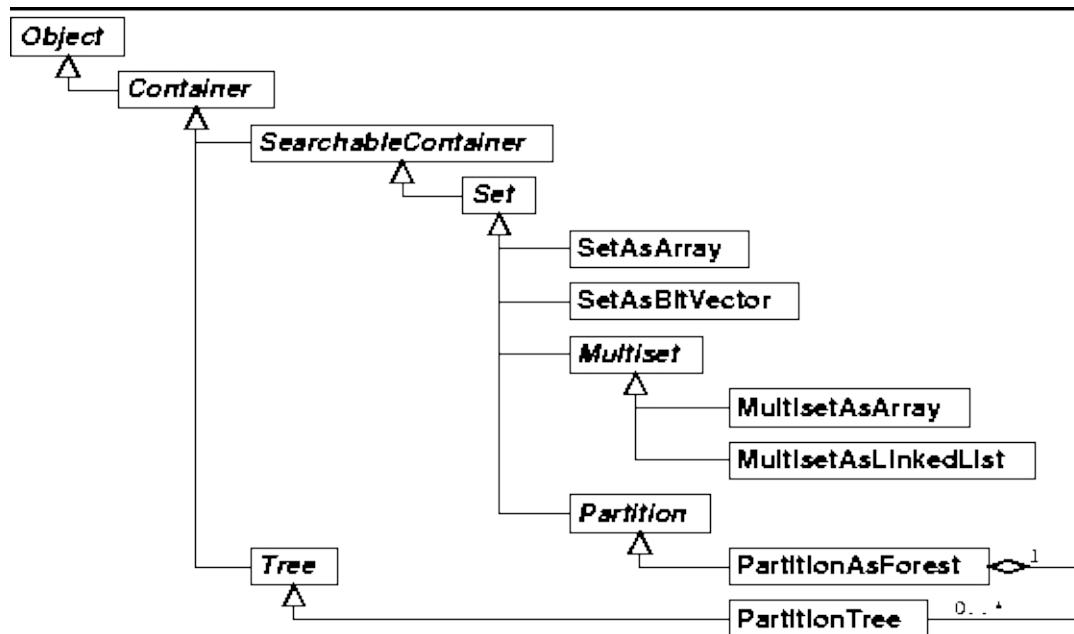


Figure: Object class hierarchy

Program 1 defines the `Set` class. The abstract `Set` class extends the abstract `SearchableContainer` class defined in Program 1. Five abstract methods are declared—`_or_`, `_and_`, `_sub_`, `_eq_`, and `_lt_`. These methods correspond to the various set operations discussed above. That is, `_or_` computes the *union*, `_and_` computes the *intersection*, `_sub_` computes the set *difference*, `_eq_` tests for set *equality*, and `_lt_` tests whether one set is a *subset* of the other.

```
 1 class Set(SearchableContainer):
 2
 3     def __init__(self, universeSize):
 4         self._universeSize = universeSize
 5
 6     def getUniverseSize(self):
 7         return self._universeSize
 8
 9     universeSize = property(
10         fget = lambda self: self.getUniverseSize())
11
12     def __or__(self, set):
13         pass
14     __or__ = abstractmethod(__or__)
15
16     def __and__(self, set):
17         pass
18     __and__ = abstractmethod(__and__)
19
20     def __sub__(self, set):
21         pass
22     __sub__ = abstractmethod(__sub__)
23
24     def __eq__(self, set):
25         pass
26     __eq__ = abstractmethod(__eq__)
27
28     def __lt__(self, set):
29         pass
30     __lt__ = abstractmethod(__lt__)
```

Program: Abstract Set class.

The Set class also defines an instance attribute called `_universeSize`. This instance attribute is used to record the size of the universal set. The `__init__` method for the Set class is given in Program □. In addition to `self`, it takes a single argument, `n`, which specifies that the universal set shall be $\{0, 1, \dots, n - 1\}$.

Array and Bit-Vector Sets

In this section we consider finite sets over a finite universe. Specifically, the universe we consider is $\{0, 1, \dots, N - 1\}$, the set of integers in the range from zero to $N-1$, for some fixed and relatively small value of N .

Let $U = \{0, 1, \dots, N - 1\}$ be the universe. Every set which we wish to represent is a subset of U . The set of all subsets of U is called the *power set* of U and is written 2^U . Thus, the sets which we wish to represent are the *elements* of 2^U . The number of elements in the set U , written $|U|$, is N . Similarly, $|2^U| = 2^{|U|} = 2^N$. This observation should be obvious: For each element of the universal set U there are only two possibilities: Either it is, or it is not, a member of the given set.

This suggests a relatively straightforward representation of the elements of 2^U --an array of `bool` values, one for each element of the universal set. By using array subscripts in U , we can represent the set implicitly. That is, i is a member of the set if the i^{th} array element is true.

Program □ introduces the class `SetAsArray`. The `SetAsArray` class extends the abstract `Set` class defined in Program □. This class uses an array of length $N = \text{numberOfItems}$ to represent the elements of 2^U where $U = \{0, 1, \dots, N - 1\}$.

```
1  class SetAsArray(Set):
2
3      def __init__(self, n):
4          super(SetAsArray, self).__init__(n)
5          self._array = Array(self._universeSize)
6          for item in xrange(0, self._universeSize):
7              self._array[item] = False
8
9      # ...
```

Program: `SetAsArray` class `__init__` method.

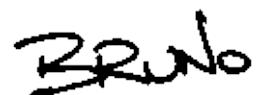
Program □ defines the `__init__` method for the `SetAsArray` class. In addition to `self`, the `__init__` method takes a single argument `n`, which defines the universe and, consequently, the size of the array of `bool` values. The `__init__` method creates the empty set by initializing all the elements of the `bool` array to `False`.

Clearly, the running time of the `__init__` method is $O(N)$, where $N = n$.

- [Basic Operations](#)
 - [Union, Intersection, and Difference](#)
 - [Comparing Sets](#)
 - [Bit-Vector Sets](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Basic Operations

Program □ shows the `insert`, `__contains__`, and `withdraw` methods of the `SetAsArray` class. The `insert` method is used to put an item into the set. The method takes an `int` argument that specifies the item to be inserted. Then the corresponding element of `_array` is set to `True` to indicate that the item has been added to the set. The running time of the `insert` operation is $O(1)$.

```
1  class SetAsArray(Set):
2
3      def insert(self, item):
4          self._array[item] = True
5
6      def __contains__(self, item):
7          return self._array[item]
8
9      def withdraw(self, item):
10         self._array[item] = False
11
12     # ...
```

Program: `SetAsArray` class `insert`, `__contains__` and `withdraw` methods.

The `__contains__` method is used to test whether a given item is an element of the set. The semantics are somewhat subtle. Since a set does not actually keep track of the specific object instances that are inserted, the membership test is based on the *value* of the argument. The method simply returns the value of the appropriate element of the `_array`. The running time of the `__contains__` operation is $O(1)$.

The `withdraw` method is used to take an item out of a set. The withdrawal operation is the opposite of insertion. Instead of setting the appropriate array element to `True`, it is set to `False`. The running time of the `withdraw` is identical to that of `insert`, viz., is $O(1)$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Union, Intersection, and Difference

Program □ defines the three methods, `__or__`, `__and__`, and `__sub__`. These methods correspond to \cup , \cap , and $-$, respectively.

```
 1  class SetAsArray(Set):
 2
 3      def __or__(self, set):
 4          assert isinstance(set, SetAsArray)
 5          assert self._universeSize == set.universeSize
 6          result = SetAsArray(self._universeSize)
 7          for i in xrange(0, self._universeSize):
 8              result._array[i] = self._array[i] or set._array[i]
 9          return result
10
11      def __and__(self, set):
12          assert isinstance(set, SetAsArray)
13          assert self._universeSize == set.universeSize
14          result = SetAsArray(self._universeSize)
15          for i in xrange(0, self._universeSize):
16              result._array[i] = self._array[i] and set._array[i]
17          return result
18
19      def __sub__(self, set):
20          assert isinstance(set, SetAsArray)
21          assert self._universeSize == set.universeSize
22          result = SetAsArray(self._universeSize)
23          for i in xrange(0, self._universeSize):
24              result._array[i] = self._array[i] \
25                  and not set._array[i]
26          return result
27
28      # ...
```

Program: SetAsArray class `__or__`, `__and__` and `__sub__` methods.

The set union method takes two arguments, `self` and `set`. The second argument is assumed to be a `SetAsArray` instance. The method computes the `SetAsArray` obtained from the union of the sets `self` and `set`. The implementation given requires that the sets be compatible. Two sets are deemed to be compatible if

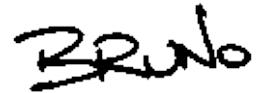
they have the same universe. The result also has the same universe. Consequently, the `bool` array in all three sets has the same length, N . The set union method creates a result array of the required size and then computes the elements of the array as required. The i^{th} element of the result is `true` if either the i^{th} element of `self` or the i^{th} element of `set` is `true`. Thus, set union is implemented using the Boolean `or` operator, or.

The set intersection method is almost identical to set union, except that the elements of the result are computed using the Boolean `and` operator, `and`. The set difference method is also very similar. In this case, an item is an element of the result only if it is a member of `self` and not a member of `set`.

Because all three methods are almost identical, their running times are essentially the same. That is, the running time of the set union, intersection, and difference operations is $O(N)$, where $N = \text{numberOfItems}$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Comparing Sets

There is a special family of operators for comparing sets. Consider two sets, say S and T . We say that S is a *subset* of T , written $S \subseteq T$, if every element of S is also an element of T . If there is at least one element of T that is not also an element of S , we say that S is a *proper subset* of T , written $S \subset T$. We can also reverse the order in which the expressions are written to get $T \supset S$ or $T \supseteq S$, which indicates that T is a (proper) *superset* of S .

The set comparison operators follow the rule that if $S \subseteq T$ and $T \subseteq S$ then $S = T$, which is analogous to a similar property of numbers: $x \leq y \wedge y \leq x \iff x = y$. However, set comparison is unlike numeric comparison in that there exist sets S and T for which neither $S \subseteq T$ nor $T \subseteq S$! For example, clearly this is the case for $S = \{1, 2\}$ and $T = \{2, 3\}$. Mathematically, the relation \subseteq is called a *partial order* because there exist some pairs of sets for which neither $S \subseteq T$ nor $T \subseteq S$ holds; whereas the relation \leq (among integers, say) is a total order.

Program `Set` defines the methods `__eq__` and `__lt__`. The former tests for equality and the latter determines whether the relation \subseteq holds between `self` and `set`. Both operators return a `bool` result. The worst-case running time of each of these operations is clearly $O(N)$.

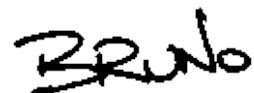
```
1 class SetAsArray(Set):
2
3     def __eq__(self, set):
4         assert isinstance(set, SetAsArray)
5         assert self._universeSize == set.universeSize
6         for i in xrange(0, self._universeSize):
7             if self._array[i] != set._array[i]:
8                 return False
9         return True
10
11     def __lt__(self, set):
12         assert isinstance(set, SetAsArray)
13         assert self._universeSize == set.universeSize
14         for i in xrange(0, self._universeSize):
15             if self._array[i] and not set._array[i]:
16                 return False
17         return True
18
19     # ...
```

Program: SetAsArray class `__eq__` and `__lt__` methods.

A complete repertoire of comparison methods would also include methods to compute \subset , \supset , \supseteq , and \neq . These operations follow directly from the implementation shown in Program □ (Exercise □).

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Bit-Vector Sets

The Python virtual machine represents bool values `False` and `True` using the `int` values `0` and `1`, respectively. In effect, it uses a 32-bit number to represent one bit of information. Therefore, we can realize a significant reduction in the memory space required to represent a set if we use an array of bits instead of an array of bools. Furthermore, by using bitwise operations to implement the basic set operations such as union and intersection, we can achieve a commensurate reduction in execution time. Unfortunately, these improvements are not free--the operations `insert`, `__contains__`, and `withdraw`, all slow down by a constant factor.

Since Python does not directly support arrays of bits, we will simulate an array of bits using an array of `ints`. Program □ illustrates how this can be done. The constant `BITS` is defined as the number of bits in a single `int`.

```
1  class SetAsBitVector(Set):
2
3      BITS = 32 # Number of bits in an integer.
4
5      def __init__(self, n):
6          super(SetAsBitVector, self).__init__(n)
7          self._vector = Array((n + self.BITS - 1) / self.BITS)
8          for i in xrange(len(self._vector)):
9              self._vector[i] = 0
10
11      # ...
```

Program: `SetAsBitVector` class `__init__` method.

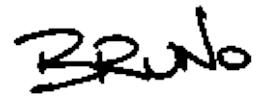
In addition to `self`, the `__init__` method takes a single argument $N = n$, which specifies the universe and, consequently, the number of bits needed in the bit array. The `__init__` method creates an array of `ints` of length $\lceil \frac{N}{w} \rceil$, where $w = \text{BITS}$ is the number of bits in an `int`, and sets the elements of the array to zero. The running time of the `__init__` method is $O(\lceil \frac{N}{w} \rceil) = O(N)$.

- [Basic Operations](#)

- [Union, Intersection, and Difference](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Basic Operations

Program □ defines the three basic operations `insert`, `__contains__`, and `withdraw` for the `SetAsBitVector` class.

```

1  class SetAsBitVector(Set):
2
3      def insert(self, item):
4          self._vector[item / self.BITS] |= 1 << item % self.BITS
5
6      def withdraw(self, item):
7          self._vector[item / self.BITS] &= \
8              ~(1 << item % self.BITS)
9
10     def __contains__(self, item):
11         return (self._vector[item / self.BITS] \
12             & (1 << item % self.BITS)) != 0
13
14     # ...

```

Program: `SetAsBitVector` `insert`, `__contains__` and `withdraw` methods.

To insert an item into the set, we need to change the appropriate bit in the array of bits to one. The i^{th} bit of the bit array is bit $i \bmod w$ of word $\lfloor i/w \rfloor$. Thus, the `insert` method is implemented using a *bitwise or* operation to change the i^{th} bit to one as shown in Program □. Even though it is slightly more complicated than the corresponding operation for the `SetAsArray` class, the running time for this operation is still $O(1)$. Since $w = \text{BITS}$ is a power of two, it is possible to replace the division and modulo operations, $/$ and $\%$, with shifts and masks like this:

```

SHIFT = 5
MASK = (1 << SHIFT) - 1
vector[item >> SHIFT] |= 1 << (item & MASK)

```

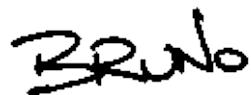
Depending on the Python interpreter and machine architecture, doing so may improve the performance of the `insert` operation by a constant factor. Of course, its asymptotic performance is still $O(1)$.

To withdraw an item from the set, we need to clear the appropriate bit in the

array of bits and to test if an item is a member of the set, we test the corresponding bit. The `__contains__` and `withdraw` methods in Program □ show how this can be done. Like `insert`, both these methods have constant worst-case running times.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Union, Intersection, and Difference

The implementations of the union, intersection, and difference methods for operands of type SetAsBitVector are shown in Program □. The code is quite similar to that for the SetAsArray class given in Program □.

```

1  class SetAsBitVector(Set):
2
3      def __or__(self, set):
4          assert isinstance(set, SetAsBitVector)
5          assert self._universeSize == set._universeSize
6          result = SetAsBitVector(self._universeSize)
7          for i in xrange(len(self._vector)):
8              result._vector[i] = self._vector[i] | set._vector[i]
9          return result
10
11     def __and__(self, set):
12         assert isinstance(set, SetAsBitVector)
13         assert self._universeSize == set._universeSize
14         result = SetAsBitVector(self._universeSize)
15         for i in xrange(len(self._vector)):
16             result._vector[i] = self._vector[i] & set._vector[i]
17         return result
18
19     def __sub__(self, set):
20         assert isinstance(set, SetAsBitVector)
21         assert self._universeSize == set._universeSize
22         result = SetAsBitVector(self._universeSize)
23         for i in xrange(len(self._vector)):
24             result._vector[i] = self._vector[i] & ~set._vector[i]
25         return result
26
27     # ...

```

Program: SetAsBitVector class __or__, __and__ and __sub__ methods.

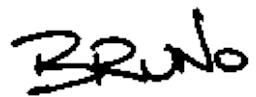
Instead of using the Boolean operators and, or, and no, we have used the bitwise operators &, |, and . By using the bitwise operators, $w = \text{BITS}$ bits of the result are computed in each iteration of the loop. Therefore, the number of iterations required is $\lceil N/w \rceil$ instead of N . The worst-case running time of each of these

operations is $O(\lceil N/w \rceil) = O(N)$.

Notice that the asymptotic performance of these SetAsBitVector class operations is the same as the asymptotic performance of the SetAsArray class operations. That is, both of them are $O(N)$. Nevertheless, the SetAsBitVector class operations are faster. In fact, the bit-vector approach is asymptotically faster than the array approach by the factor w .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Multisets

A *multiset* is a set in which an item may appear more than once. That is, whereas duplicates are not permitted in a regular set, they are permitted in a multiset. Multisets are also known simply as *bags*.

Sets and multisets are in other respects quite similar: Both support operations to insert and withdraw items; both provide a means to test the membership of a given item; and both support the basic set operations of union, intersection, and difference. As a result, the abstract `Multiset` class is essentially the same as the abstract `Set` class as shown in Program □.

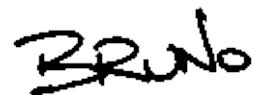
```
1 class Multiset(Set):
2
3     def __init__(self, universeSize):
4         super(Multiset, self).__init__(universeSize)
```

Program: Abstract Multiset class.

- [Array Implementation](#)
 - [Linked-List Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Array Implementation

A regular set may contain either zero or one instance of a particular item. As shown in the preceding section if the number of possible items is not excessive, we may use an array of bool variables to keep track of the number of instances of a particular item in a regular set. The natural extension of this idea for a multiset is to keep a separate count of the number of instances of each item in the multiset.

Program □ introduces the `MultisetAsArray` class. The `MultisetAsArray` class extends the abstract `Multiset` class defined in Program □. The multiset is implemented using an array of $N = n$ counters. Each counter is an `int` in this case.

```
1  class MultisetAsArray(Multiset):
2
3      def __init__(self, n):
4          super(MultisetAsArray, self).__init__(n)
5          self._array = Array(self._universeSize)
6          for item in xrange(self._universeSize):
7              self._array[item] = 0
8
9      # ...
```

Program: `MultisetAsArray` class `__init__` method.

In addition to `self`, the `__init__` method takes a single argument, $N = \text{numberOfItems}$, and initializes an array of length N counters all to zero. The running time of the `__init__` method is $O(N)$.

-
- [Basic Operations](#)
 - [Union, Intersection, and Difference](#)
-

Bruno

Basic Operations

Program □ defines the three basic operations `insert`, `__contains__` and `withdraw` for the `MultisetAsArray` class.

```
1  class MultisetAsArray(Multiset):
2
3      def insert(self, item):
4          self._array[item] += 1
5
6      def withdraw(self, item):
7          if self._array[item] == 0:
8              raise KeyError
9          self._array[item] -= 1
10
11     def __contains__(self, item):
12         return self._array[item] > 0
13
14     # ...
```

Program: `MultisetAsArray` class `insert`, `__contains__` and `withdraw` methods.

To insert an item, we simply increase the appropriate counter; to delete an item, we decrease the counter; and to test whether an item is in the set, we test whether the corresponding counter is greater than zero. In all cases the operation can be done in constant time.

Union, Intersection, and Difference

Because multisets permit duplicates but sets do not, the definitions of union, intersection, and difference are slightly modified for multisets. The *union* of multisets S and T , written $S \cup T$, is the multiset comprised of all the elements of S together with all the element of T . Since a multiset may contain duplicates, it does not matter if the same element appears in S and T .

The subtle difference between union of sets and union of multisets gives rise to an interesting and useful property. If S and T are regular sets,

$$\min(|S|, |T|) \leq |S \cup T| \leq |S| + |T|.$$

On the other hand, if S and T are *multisets*,

$$|S \cup T| = |S| + |T|.$$

The *intersection* of sets S and T is written $S \cap T$. The elements of $S \cap T$ are those items which are elements of *both* S and T . If a given element appears more than once in S or T (or both), the intersection contains m copies of that element, where m is the smaller of the number of times the element appears in S or T . For example, if $S = \{0, 1, 1, 2, 2, 2\}$ and $T = \{1, 2, 2, 3\}$, the intersection is $S \cap T = \{1, 2, 2\}$.

The *difference* of sets S and T , written $S-T$, contains those elements of S which are *not also* elements of T . That is, the result $S-T$ is obtained by taking the set S and removing from it those elements which are also found in T .

Program □ gives the implementations of the union, intersection, and difference methods of `MultisetAsArray` class. This code is quite similar to that of the `SetAsArray` class (Program □) and the `SetAsBitVector` class (Program □). The worst-case running time of each of these operations is $O(N)$.

```

1  class MultisetAsArray(Multiset):
2
3      def __or__(self, set):
4          assert isinstance(set, MultisetAsArray)
5          assert self._universeSize == set._universeSize
6          result = MultisetAsArray(self._universeSize)
7          for i in xrange(self._universeSize):
8              result._array[i] = self._array[i] + set._array[i]
9          return result
10
11
12      def __and__(self, set):
13          assert isinstance(set, MultisetAsArray)
14          assert self._universeSize == set._universeSize
15          result = MultisetAsArray(self._universeSize)
16          for i in xrange(self._universeSize):
17              result._array[i] = min(self._array[i], set._array[i])
18          return result
19
20
21      def __sub__(self, set):
22          assert isinstance(set, MultisetAsArray)
23          assert self._universeSize == set._universeSize
24          result = MultisetAsArray(self._universeSize)
25          for i in xrange(self._universeSize):
26              if set._array[i] <= self._array[i]:
27                  result._array[i] = self._array[i] - set._array[i]
28          return result
29
30      # ...

```

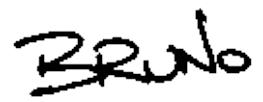
Program: MultisetAsArray class __or__, __and__ and __sub__ methods.

Instead of using the Boolean operators and, or, and not, we have used + (integer addition), min and - (integer subtraction). The following table summarizes the operators used in the various set and multiset implementations.

	class		
operation	SetAsArray	SetAsBitVector	MultisetAsArray
union	or		+
intersection	and	&	min
difference	and and not	& and	<= and -

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Linked-List Implementation

The array implementation of multisets is really only practical if the number of items in the universe, $N=|U|$, is not too large. If N is large, then it is impractical, or at least extremely inefficient, to use an array of N counters to represent the multiset. This is especially so if the number of elements in the multisets is significantly less than N .

If we use a linked list of elements to represent a multiset S , the space required is proportional to the size of the multiset, $|S|$. When the size of the multiset is significantly less than the size of the universe, $|S| \ll |U|$, it is more efficient in terms of both time and space to use a linked list.

Program □ introduces the the `MultisetAsLinkedList` class. The `MultisetAsLinkedList` extends the the abstract `Multiset` class defined in Program □. In this case a linked list of `ints` is used to record the contents of the multiset.

How should the elements of the multiset be stored in the list? Perhaps the simplest way is to store the elements in the list in no particular order. Doing so makes the `insert` operation efficient--it can be done in constant time. Furthermore, the `__contains__` and `withdraw` operations both take $O(n)$ time, where n is the number of items in the multiset, *regardless of the order of the items in the linked list*.

Consider now the union, intersection, and difference of two multisets, say S and T . If the linked list is unordered, the worst case running time for the union operation is $O(m+n)$, where $m=|S|$ and $n=|T|$. Unfortunately, intersection, and difference are both $O(mn)$.

If, on the other hand, we use an *ordered* linked list, union, intersection, and difference can all be done in $O(m+n)$ time. The trade-off is that the insertion becomes an $O(n)$ operation rather than a $O(1)$. The `MultisetAsLinkedList` implementation presented in this section records the elements of the multiset in an *ordered* linked list.

```
1 class MultisetAsLinkedList(Multiset):
2
3     def __init__(self, n):
4         super(MultisetAsLinkedList, self).__init__(n)
5         self._list = LinkedList()
6
7     # ...
```

Program: MultisetAsLinkedList class `__init__` method.

- [Union](#)
 - [Intersection](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Union

The union operation for `MultisetAsLinkedList` class requires the merging of two ordered, linked lists as shown in Program □. We have assumed that the smallest element contained in a multiset is found at the head of the linked list and the largest is at the tail.

```
 1  class MultisetAsLinkedList(Multiset):
 2
 3      def __or__(self, set):
 4          assert isinstance(set, MultisetAsLinkedList)
 5          assert self._universeSize == set._universeSize
 6          result = MultisetAsLinkedList(self._universeSize)
 7          p = self._list.head
 8          q = set._list.head
 9          while p is not None and q is not None:
10              if p.datum <= q.datum:
11                  result._list.append(p.datum)
12                  p = p.next
13              else:
14                  result._list.append(q.datum)
15                  q = q.next
16              while p is not None:
17                  result._list.append(p.datum)
18                  p = p.next
19              while q is not None:
20                  result._list.append(q.datum)
21                  q = q.next
22          return result
23
24      # ...
```

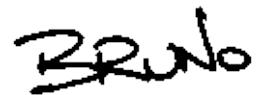
Program: `MultisetAsLinkedList` class `__or__` method.

The `__or__` method computes its result as follows: The main loop of the program (lines 9-15) traverses the linked lists of the two operands, in each iteration appending the smallest remaining element to the result. Once one of the lists has been exhausted, the remaining elements in the other list are simply appended to the result (lines 16-21). The total running time for the `__or__`

method is $O(m+n)$, where $m = |\text{this}|$ and $n = |\text{set}|$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Intersection

The implementation of the intersection operator for the `MultisetAsLinkedList` class is similar to that of union. However, instead of merging of two ordered, linked lists to construct a third, we compare the elements of two lists and append an item to the third only when it appears in both of the input lists. The `Intersection` method is shown in Program □.

```
 1  class MultisetAsLinkedList(Multiset):
 2
 3      def __and__(self, set):
 4          assert isinstance(set, MultisetAsLinkedList)
 5          assert self._universeSize == set._universeSize
 6          result = MultisetAsLinkedList(self._universeSize)
 7          p = self._list.head
 8          q = set._list.head
 9          while p is not None and q is not None:
10              diff = p.datum - q.datum
11              if diff == 0:
12                  result._list.append(p.datum)
13              if diff <= 0:
14                  p = p.next
15              if diff >= 0:
16                  q = q.next
17          return result
18
19      # ...
```

Program: `MultisetAsLinkedList` class `__and__` method.

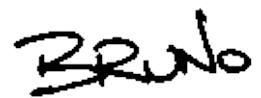
The main loop of the program traverses the linked lists of both input operands at once using two variables (lines 9-16). If the next element in each list is the same, that element is appended to the result and both variables are advanced. Otherwise, only one of the variables is advanced--the one pointing to the smaller element.

The number of iterations of the main loop actually done depends on the contents of the respective linked lists. The best case occurs when both lists are identical.

In this case, the number of iterations is m , where $m = |\text{this}| = |\text{set}|$. In the worst case case, the number of iterations done is $m+n$. Therefore, the running time of the `Intersection` method is $O(m+n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Partitions

Consider the finite universal set $U = \{0, 1, \dots, N - 1\}$. A *partition* of U is a finite set of sets $P = \{S_1, S_2, \dots, S_p\}$ with the following properties:

1. The sets S_1, S_2, \dots, S_p are pairwise *disjoint*. That is, $S_i \cap S_j = \emptyset$ for all values of i and j such that $1 \leq i < j \leq p$.
2. The sets S_1, S_2, \dots, S_p *span* the universe U . That is,

$$\begin{aligned} \bigcup_{i=1}^p S_i &= S_1 \cup S_2 \cup \dots \cup S_p \\ &= U. \end{aligned}$$

For example, consider the universe $U = \{1, 2, 3\}$. There are exactly five partitions of U :

$$\begin{aligned} P_0 &= \{\{1\}, \{2\}, \{3\}\}, \\ P_1 &= \{\{1\}, \{2, 3\}\}, \\ P_2 &= \{\{2\}, \{1, 3\}\}, \\ P_3 &= \{\{3\}, \{1, 2\}\}, \text{ and} \\ P_4 &= \{\{1, 2, 3\}\}. \end{aligned}$$

In general, given a universe U of size $n > 0$, i.e., $|U| = n$, there are $\sum_{m=0}^n \left\{ \begin{smallmatrix} n \\ m \end{smallmatrix} \right\}$ partitions of U , where $\left\{ \begin{smallmatrix} n \\ m \end{smallmatrix} \right\}$ is the so-called *Stirling number of the second kind* which denotes the number of ways to partition a set of n elements into m nonempty disjoint subsets. ◇

Applications which use partitions typically start with an initial partition and refine that partition either by joining or by splitting elements of the partition according to some application-specific criterion. The result of such a computation is the partition obtained when no more elements can be split or joined.

In this chapter we shall consider only applications that begin with the initial partition of U in which each item in U is in a separate element of the partition. Thus, the initial partition consists of $|U|$ sets, each of size one (like P_0 above). Furthermore, we restrict the applications in that we only allow elements of a partition to be joined--we do not allow elements to split.

The two operations to be performed on partitions are:

find

Given an item in the universe, say $i \in U$, find the element of the partition that contains i . That is, find $S_j \in P$ such that $i \in S_j$.

join

Given two distinct elements of a partition P , say $S_i \in P$ and $S_j \in P$ such that $i \neq j$, create a new partition P' by removing the two elements S_i and S_j from P and replacing them with a single element $S_i \cup S_j$.

For example, consider the partition $P = \{S_1, S_2, S_3\} = \{\{1\}, \{2, 3\}, \{4\}\}$. The result of the operation $\text{find}(3)$ is the set $S_2 = \{2, 3\}$ because 3 is a member of S_2 . Furthermore, when we *join* sets S_1 and S_3 , we get the partition $P' = \{\{1, 4\}, \{2, 3\}\}$.

- [Representing Partitions](#)
 - [Implementing a Partition using a Forest](#)
 - [Collapsing Find](#)
 - [Union by Size](#)
 - [Union by Height or Rank](#)
-

Representing Partitions

Program □ defines the `Partition` class. The abstract `Partition` class extends the abstract `Set` class defined in Program □. Since a partition is a set of sets, it makes sense to derive `Partition` from `Set`. The two methods, `find` and `join`, correspond to the partition operations described above.

```
1  class Partition(Set):
2
3      def __init__(self, n):
4          super(Partition, self).__init__(n)
5
6      def find(self, obj):
7          pass
8      find = abstractmethod(find)
9
10     def join(self, s, t):
11         pass
12     join = abstractmethod(join)
```

Program: Abstract Partition class.

The elements of a partition are also sets. Consequently, the objects contained in a `Partition` are also derived from the `Set` class. The `find` method of the `Partition` class expects as its argument an `int` and returns the set which contains the specified item.

The `join` method takes two arguments, both of them references to sets. The two arguments are expected to be distinct elements of the partition. The effect of the `join` operation is to remove the specified sets from the partition and replace them with a set which represents the *union* of the two.

Implementing a Partition using a Forest

A partition is a set of sets. Consequently, there are two related issues to consider when developing an approach for representing partitions:

1. How are the individual elements or parts of the partition represented?
2. How are the elements of a partition combined into the whole?

This section presents an approach in which each element of a partition is a tree. Therefore, the whole partition is a *forest*.

For example, Figure □ shows how the partition

$$\begin{aligned} P &= \{S_1, S_2, S_3, S_4\} \\ &= \{\{0, 4\}, \{2, 6, 8\}, \{10\}, \{1, 3, 5, 7, 9, 11\}\} \end{aligned}$$

can be represented using a forest. Notice that each element of the universal set $U = \{0, 1, \dots, 11\}$ appears in exactly one node of exactly one tree.

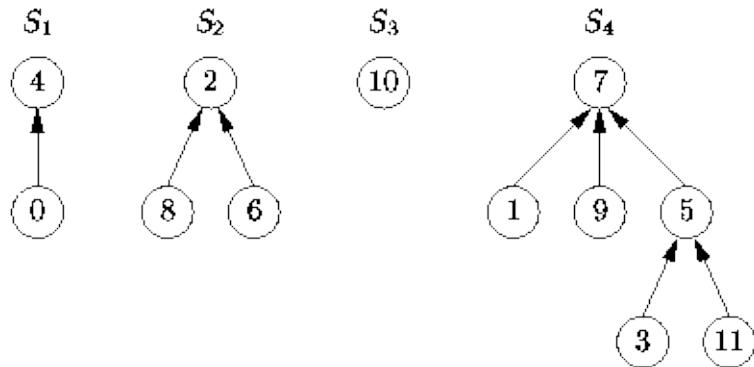


Figure: Representing a partition as a forest.

The trees in Figure □ have some very interesting characteristics. The first characteristic concerns the shapes of the trees: The nodes of the trees have arbitrary degrees. The second characteristic concerns the positions of the keys: there are no constraints on the positions of the keys in a tree. The final characteristic has to do with the way the tree is represented: Instead of pointing to its children, each node of the tree points to its parent!

Since there is no particular order to the nodes in the trees, it is necessary to keep track of the position of each node explicitly. Figure □ shows how this can be done using an array. (This figure shows the same partition as in Figure □). The array contains a node for each element of the universal set U . Specifically, the i^{th} array element holds the node that contains item i . Having found the desired node, we can follow the chain of parent pointers to find the root of the corresponding tree.

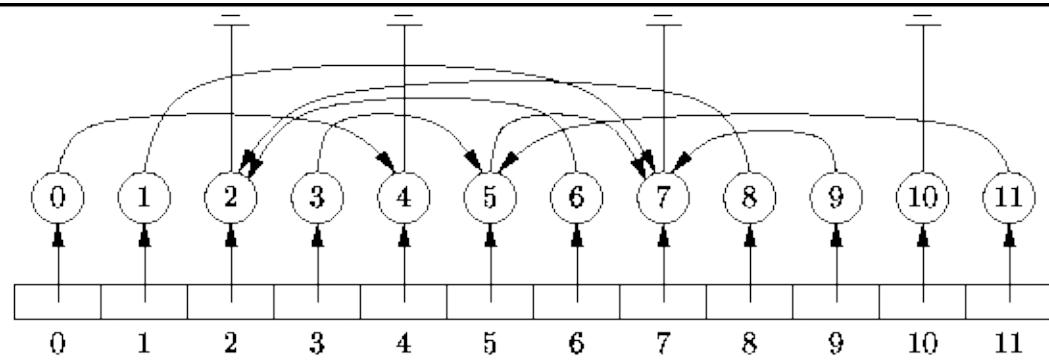


Figure: Finding the elements of a partition.

- [Implementation](#)
- [__init__ Method](#)
- [find and join Methods](#)



Implementation

Program □ introduces two classes--PartitionAsForest and the nested class PartitionTree. The latter is used to represent the individual elements or parts of a partition and the former encapsulates all of the parts that make up a given partition.

```
 1  class PartitionAsForest(Partition):
 2
 3      class PartitionTree(Set, Tree):
 4
 5          def __init__(self, partition, item):
 6              super(PartitionAsForest.PartitionTree, self) \
 7                  .__init__(partition._universeSize)
 8              self._partition = partition
 9              self._item = item
10              self._parent = None
11              self._rank = 0
12              self._count = 1
13
14          # ...
15
16      # ...
```

Program: PartitionAsForest.PartitionTree class `__init__` method.

The PartitionTree class extends the abstract Set class defined in Program □ as well as the abstract Tree class defined in Program □. Since we are representing the parts of a partition using trees, it makes sense that they extend the Tree class. On the other hand, since a partition is a set of sets, we must derive the parts of a partition from the Set class.

The PartitionTree class has five instance attributes--`_partition`, `_item`, `_parent`, `_rank` and `_count`. Each instance of this class represents one node of a tree. The `_partition` instance attribute refers to the PartitionAsForest instance that contains this tree. The `_parent` instance attribute refers to the parent of a given node and the `_item` instance attribute records the element of the universal set that the given node represents. The rank attribute is optional. While

it is not required in order to provide the basic functionality, as shown below, the `_rank` variable can be used in the implementation of the `join` operation to improve the performance of subsequent `find` operations. Finally, the `_count` attribute is used to keep track of the number of items in a given element of the partition.

Program □ gives the code for the `PartitionTree __init__` method. The `__init__` method creates a tree comprised of a single node. It takes an argument which specifies the element of the universal set that the node is to represent. The `parent` instance attribute is set to `None` to indicate that the node has no parent. Consequently, the node is a root node. Finally, the `rank` instance attribute is initialized to zero. The running time of the `__init__` method is $O(1)$.

Program □ shows the `__init__` method for the `PartitionAsForest` class. The `PartitionAsForest` class represents a complete partition. The `PartitionAsForest` class extends the abstract `Partition` class defined in Program □. The `PartitionAsForest` class contains the instance attributes `_array`, which is an array `PartitionTrees`, and `_count`, which is used to record the number of elements in the partition. The i^{th} element of the array always refers to the tree node that contains element i of the universe.



__init__ Method

The `__init__` method takes a single argument $N = n$ which specifies that the universe shall be $U = \{0, 1, \dots, N - 1\}$. It creates an initial partition of the universe consisting of N parts. Each part contains one element of the universal set and, therefore, comprises a one-node tree.

```
1 class PartitionAsForest(Partition):
2
3     def __init__(self, n):
4         super(PartitionAsForest, self).__init__(n)
5         self._array = Array(self._universeSize)
6         for item in xrange(self._universeSize):
7             self._array[item] = self.PartitionTree(self, item)
8         self._count = self._universeSize
9
10    # ...
```

Program: PartitionAsForest class `__init__` method.



find and join Methods

Two elements of the universe are in the same part of the partition if and only if they share the same root node. Since every tree has a unique root, it makes sense to use the root node as the ``handle'' for that tree. Therefore, the *find* operation takes an element of the universal set and returns the root node of the tree that contains that element. And because of way in which the trees are represented, we can follow the chain of parent pointers to find the root node.

Program □ gives the code for the *find* method of the *PartitionAsForest* class. The *find* method takes as its argument an *int* and returns a *set*. The argument specifies the item of the universe that is the object of the search.

```
1  class PartitionAsForest(Partition):
2
3      def find(self, item):
4          ptr = self._array[item]
5          while ptr._parent is not None:
6              ptr = ptr._parent
7          return ptr
8
9      # ...
```

Program: *PartitionAsForest* class *find* method.

The *find* operation begins at the node *_array[item]* and follows the chain of parent instance attributes to find the root node of the tree that contains the specified item. The result of the method is the root node.

The running time of the *find* operation is $O(d)$ where d is the depth in the tree of the node from which the search begins. If we don't do anything special to prevent it, the worst case running time is $O(N)$, where N is the size of the universe. The best performance is achieved when every non-root node points to the root node. In this case, the running time is $O(1)$.

Another advantage of having the parent instance attribute in each node is that the *join* operation can be implemented easily and efficiently. For example, suppose

we wish to *join* the two sets S_1 and S_2 shown in Figure □. While there are many possible representations for $S_1 \cup S_2$, it turns out that there are two simple alternatives which can be obtained in constant time. These are shown in Figure □. In the first alternative, the root of S_2 is made a child of the root of S_1 . This can be done in constant time simply by making the parent instance attribute of the root of S_2 refer to the root of S_1 . The second alternative is essentially the same as the first except that the roles of S_1 and S_2 are exchanged.

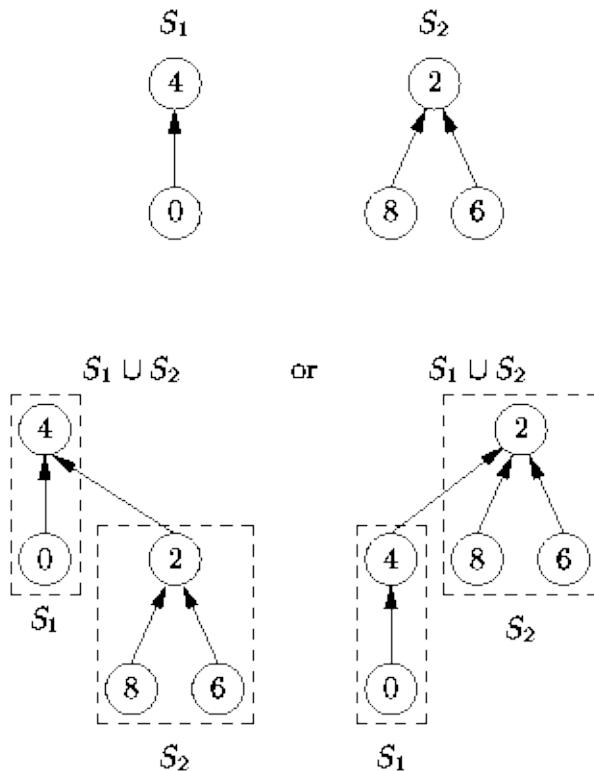


Figure: Alternatives for joining elements of a partition.

Program □ gives the simplest possible implementation for the `join` operation. In addition to `self`, the `join` method of the `PartitionAsForest` class takes two arguments. Both arguments are required to be references to distinct `PartitionTree` instances which are contained in the given partition. Furthermore, both of them are required to be root nodes. Therefore, the sets that the arguments represent are *disjoint*. The `assert` statement makes sure that the arguments satisfy these conditions.

The `join` operation is trivial and executes in constant time: It simply makes one node the parent of the other. In this case, we have arbitrarily chosen that the node

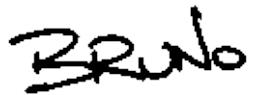
specified by the first argument shall always become the parent.

```
1 class PartitionAsForest(Partition):
2
3     def join(self, s, t):
4         assert s in self and s._parent is None and \
5             t in self and t._parent is None and s is not t
6         t._parent = s
7         self._count -= 1
8
9     # ...
```

Program: PartitionAsForest class simple join method.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Collapsing Find

Unfortunately, using the join algorithm given in Program □ can result in particularly bad trees. For example, Figure □ shows the worst possible tree that can be obtained. Such a tree is bad because its height is $O(N)$. In such a tree both the worst case and the average case running time for the `find` operation is $O(N)$.



Figure: A degenerate tree.

There is an interesting trick we can play that can improve matters significantly. Recall that the `find` operation starts from a given node and locates the root of the tree containing that node. If, having found the root, we replace the parent of the given node with the root, the next time we do a `find` it will be more efficient.

In fact, we can go one step further and replace the parent of every node along the search path to the root. This is called a *collapsing find* operation. Doing so does not change the asymptotic complexity of the `find` operation. However, a subsequent `find` operation which begins at any point along the search path to the root will run in constant time!

Program □ gives the code for a collapsing version of the `find` operation. The `find` method first determines the root node as before. Then, a second pass is made up the chain from the initial node to the root, during which the parent of each node is assigned the root. Clearly, this version of `find` is slower than the

one given in Program □ because it makes two passes up the chain rather than one. However, the running of this version of `find` is still $O(d)$, where d is the depth of the node from which the search begins.

```

1  class PartitionAsForestV2(PartitionAsForest):
2
3      def find(self, item):
4          root = self._array[item]
5          while root._parent is not None:
6              root = root._parent
7          ptr = self._array[item]
8          while ptr._parent is not None:
9              tmp = ptr._parent
10             ptr._parent = root
11             ptr = tmp
12         return root
13
14     # ...

```

Program: `PartitionAsForest` class collapsing `find` method.

Figure □ illustrates the effect of a collapsing find operation. After the find, all the nodes along the search path are attached directly to the root. That is, they have had their depths decreased to one. As a side-effect, any node which is in the subtree of a node along the search path may have its depth decreased by the collapsing find operation. The depth of a node is never increased by the find operation. Eventually, if we do enough collapsing find operations, it is possible to obtain a tree of height one in which all the non-root nodes point directly at the root.

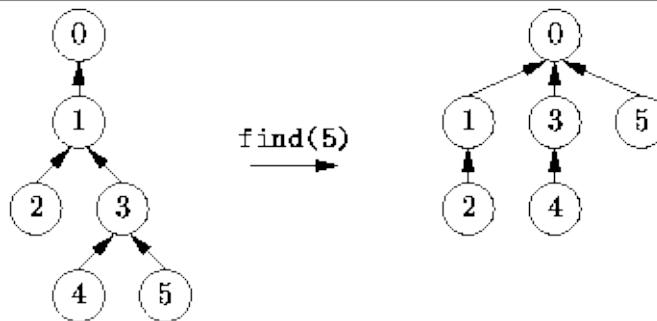


Figure: Example of collapsing find.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Union by Size

While using collapsing find does mitigate the negative effects of poor trees, a better approach is to avoid creating bad trees in the first place. As shown in Figure □, when we join two trees we have a choice--which node should we choose to be the root of the new tree. A simple, but effective choice is to attach the smaller tree under the root of the larger one. In this case, the smaller tree is the one which has fewer nodes. This is the so-called *union-by-size* join algorithm. Program □ shows how this can be done.

```
1 class PartitionAsForestV2(PartitionAsForest):
2
3     def join(self, s, t):
4         assert s in self and s._parent is None and \
5             t in self and t._parent is None and s is not t
6         if s._count > t._count:
7             t._parent = s
8             s._count += t._count
9         else:
10            s._parent = t
11            t._count += s._count
12        self._count -= 1
13
14    # ...
```

Program: PartitionAsForest class union-by-size join method.

The implementation uses the `_count` instance attribute to keep track of the number of items contained in the tree. (Since each node contains one item from the universal set, the number of items contained in a tree is equal to the number of nodes in that tree). The algorithm simply selects the tree with the largest number of nodes to become the root of the result and attaches the root of the smaller tree under that of the larger one. Clearly, the running time of the union-by-size version of `join` is $O(1)$.

The following theorem shows that when using the union-by-size join operation, the heights of the resulting trees grow logarithmically.

Theorem Consider an initial partition P of the universe $U = \{0, 1, \dots, N - 1\}$ comprised of N sets of size 1. Let S be an element of the partition obtained

from P after some sequence of *union-by-size* join operations, such that $|S|=n$ for some $n \geq 1$. Let T be the tree representing the set S . The height of tree T satisfies the inequality

$$h \leq \lfloor \log_2 n \rfloor.$$

extbf{Proof} (By induction).

Base Case Since a tree comprised of a single node has height zero, the theorem clearly holds for $n=1$.

Inductive Hypothesis Suppose the theorem holds for trees containing n nodes for $n = 1, 2, \dots, k$ for some $k \geq 1$. Consider a union-by-size join operation that produces a tree containing $k+1$ nodes. Such a tree is obtained by joining a tree T_l having $l \leq k$ nodes with another tree T_m that has $m \leq k$ nodes, such that $l+m=k+1$.

Without loss of generality, suppose $1 \leq l \leq (k+1)/2$. As a result, l is less than or equal to m . Therefore, the union-by-size algorithm will attach T_l under the root of T_m . Let h_l and h_m be the heights of T_l and T_m respectively. The height of the resulting tree is $\max(h_l + 1, h_m)$.

According to the inductive hypothesis, the height of T_m is given by

$$\begin{aligned} h_m &\leq \lfloor \log_2 m \rfloor \\ &\leq \lfloor \log_2(k+1-l) \rfloor \\ &\leq \lfloor \log_2(k+1) \rfloor. \end{aligned}$$

Similarly, the quantity $h_l + 1$ is bounded by

$$\begin{aligned} h_l + 1 &\leq \lfloor \log_2 l \rfloor + 1 \\ &\leq \lfloor \log_2((k+1)/2) \rfloor + 1 \\ &\leq \lfloor \log_2(k+1) \rfloor. \end{aligned}$$

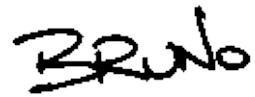
Therefore, the height of the tree containing $k+1$ nodes is no greater than $\max(h_l + 1, h_m) = \lfloor \log_2(k+1) \rfloor$. By induction on k , the theorem holds for all values of $n \geq 1$.

Note that Theorem \square and its proof does not require that we use the collapsing

find algorithm of Section □. That is, the height of a tree containing n nodes is guaranteed to be $O(\log n)$ when the simple find is used. Of course, there is nothing precluding the use of the collapsing find in conjunction with the union-by-size join method. And doing so only makes things better.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Union by Height or Rank

The union-by-size join algorithm described above controls the heights of the trees indirectly by basing the join algorithm on the sizes of the trees. If we explicitly keep track of the height of a node in the node itself, we can accomplish the same thing.

Program □ gives an implementation of the `join` method that always attaches the shorter tree under the root of the taller one. This method assumes that the `_rank` instance attribute is used to keep track of the height of a node. (The reason for calling it `_rank` rather than `height` will become evident shortly).

```
1  class PartitionAsForestV3(PartitionAsForestV2):
2
3      def join(self, s, t):
4          assert s in self and s._parent is None and \
5              t in self and t._parent is None and s is not t
6          if s._rank > t._rank:
7              t._parent = s
8          else:
9              s._parent = t
10             if s._rank == t._rank:
11                 t._rank += 1
12             self._count -= 1
13
14     # ...
```

Program: `PartitionAsForest` class union-by-rank join method.

The only time that the height of node increases is when joining two trees that have the same height. In this case, the height of the root increases by exactly one. If the two trees being joined have different heights, attaching the shorter tree under the root of the taller one has no effect on the height of the root.

Unfortunately, there is a slight complication if we combine union-by-height with the collapsing find. Since the collapsing find works by moving nodes closer to the root, it affects potentially the height of any node moved. It is not at all clear how to recompute efficiently the heights that have changed. The solution is not to do it at all!

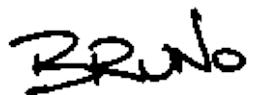
If we don't recompute the heights during the collapsing find operations, then the

height instance attributes will no longer be exact. Nevertheless, the quantities remain useful estimates of the heights of nodes. We call the estimated height of a node its *rank* and the join algorithm which uses rank instead of height is called *union by rank*.

Fortunately, Theorem \square applies equally well when union-by-rank is used. That is, the height of tree which contains n nodes is $O(\log n)$. Thus, the worst-case running time for the `find` operation grows logarithmically with n . And as before, collapsing `find` only makes things better.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Applications

One of the most important applications of partitions involves the processing of equivalence relations. Equivalence relations arise in many interesting contexts. For example, two nodes in an electric circuit are electrically equivalent if there is a conducting path (a wire) connecting the two nodes. In effect, the wires establish an electrical equivalence relation over the nodes of a circuit.

A similar relation arises among the classes in a Python program. Consider the following Python code fragment:

```
class A(object): pass
class B(A): pass
class C(A): pass
class D(B): pass
class E(C): pass
```

The classes A, B, C, D and E are equivalent in the sense that they are all subclasses of the class A. (By definition, a class is a subclass of itself.) In effect, the class declarations establish an equivalence relation over the classes in a Python program.

Definition (Equivalence Relation) An *equivalence relation* over a universal set U is a relation \equiv with the following properties:

1. The relation \equiv is *reflexive*. That is, for every $x \in U$, $x \equiv x$.
2. The relation \equiv is *symmetric*. That is, for every pair $x \in U$ and $y \in U$, if $x \equiv y$ then $y \equiv x$.
3. The relation \equiv is *transitive*. That is, for every triple $x \in U$, $y \in U$ and $z \in U$, if $x \equiv y$ and $y \equiv z$ then $x \equiv z$.

An important characteristic of an equivalence relation is that it partitions the elements of the universal set U into a set of *equivalence classes*. That is, U is partitioned into $P = \{S_1, S_2, \dots, S_p\}$, such that for every pair $x \in U$ and $y \in U$, $x \equiv y$ if and only if x and y are in the same element of the partition. That is, $x \equiv y$ if there exists a value of i such that $x \in S_i \wedge y \in S_i$.

For example, consider the universe $U = \{0, 1, \dots, 9\}$. and the equivalence relation \equiv defined over U defines as follows:

$$\begin{aligned} 0 \equiv 0, 1 \equiv 1, 1 \equiv 2, 2 \equiv 2, 3 \equiv 3, 3 \equiv 4, 3 \equiv 5, 4 \equiv 4, 4 \equiv 5, 5 \equiv 5, \\ 6 \equiv 6, 6 \equiv 7, 6 \equiv 8, 6 \equiv 9, 7 \equiv 7, 7 \equiv 8, 7 \equiv 9, 8 \equiv 8, 8 \equiv 9, 9 \equiv 9. \end{aligned} \quad (12.1)$$

This relation results in the following partition of U :

$$\{\{0\}, \{1, 2\}, \{3, 4, 5\}, \{6, 7, 8, 9\}\}.$$

The list of equivalences in Equation 12.1 contains many redundancies. Since we know that the relation \equiv is reflexive, symmetric and transitive, it is possible to infer the complete relation from the following list

$$1 \equiv 2, 3 \equiv 4, 3 \equiv 5, 6 \equiv 7, 6 \equiv 8, 6 \equiv 9.$$

The problem of finding the set of equivalence classes from a list of equivalence pairs is easily solved using a partition. Program 12.1 shows how it can be done using the `PartitionAsForest` class defined in Section 12.1.

```

1  class Algorithms(object):
2
3      def equivalenceClasses(input, output):
4          line = input.readline()
5          p = PartitionAsForest(int(line))
6          for line in input.readlines():
7              words = line.split()
8              i = int(words[0])
9              j = int(words[1])
10             s = p.find(i)
11             t = p.find(j)
12             if s is not t:
13                 p.join(s, t)
14             else:
15                 output.write("redundant pair: %d, %d\n" % (i, j))
16             output.write(str(p) + "\n")
17     equivalenceClasses = staticmethod(equivalenceClasses)

```

Program: Application of disjoint sets--finding equivalence classes.

The algorithm first gets a positive integer n from the input and creates a partition, p , of the universe $U = \{0, 1, \dots, n - 1\}$ (lines 4-5). As explained in Section 12.1, the initial partition comprises n disjoint sets of size one. That is, each

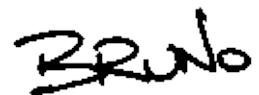
element of the universal set is in a separate element of the partition.

Each iteration of the main loop processes one equivalence pair (lines 6-15). An equivalence pair consists of two numbers, i and j , such that $i \in U$ and $j \in U$. The *find* operation is used to determine the sets s and t in partition p that contain elements i and j , respectively (lines 10-11).

If s and t are not the same set, then the disjoint sets are united using the *join* operation (lines 12-13). Otherwise, i and j are already in the same set and the equivalence pair is redundant (line 15). After all the pairs have been processed, the final partition is printed (line 16).

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exercises

1. For each of the following implementations derive an expression for the total memory space required to represent a set which contains of n elements drawn from the universe $U = \{0, 1, \dots, N - 1\}$.
 1. SetAsArray (Program □),
 2. SetAsBitVector (Program □),
 3. MultisetAsArray (Program □), and
 4. MultisetAsLinkedList (Program □).
2. In addition to $=$ and \subseteq , a complete repertoire of set operators includes \subset , \supset , \supseteq and \neq . For each of the set implementations listed in Exercise □ show how to implement the remaining operators.
3. The *symmetric difference* of two sets S and T , written $S \Delta T$ is given by

$$S \Delta T = (S \cup T) - (S \cap T).$$

For each of the set implementations listed in Exercise □ devise an algorithm to compute symmetric difference. What is the running time of your algorithm?

4. The *complement* of a set S over universe U , written S' is given by

$$S' = U - S.$$

Devise an algorithm to compute the complement of a set represented as a bit vector. What is the running time of your algorithm?

5. Devise an algorithm to sort a list of integers using a multiset. What is the running time of your algorithm? **Hint:** See Section □.
6. Consider a multiset implemented using linked lists. When the multiset contains duplicate items, each of those items occupies a separate list element. An alternative is to use a linked list of ordered pairs of the form (i, n_i) where i is the element of the universal set U and n_i is a non-negative integer that counts the number of instances of the element i in the multiset.

Derive an expression for the total memory space required to represent a multiset which contains of n instances of m distinct element drawn from the

universe $U = \{0, 1, \dots, N - 1\}$.

7. Consider a multiset implemented as described in Exercise □. Devise algorithms for set union, intersection, and difference. What are the running times of your algorithms?
8. Consider the initial partition $P = \{\{0\}, \{1\}, \{2\}, \dots, \{9\}\}$. For each of the methods of computing the union listed below show the result of the following sequence *join* operations: $\text{join}(0, 1)$, $\text{join}(2, 3)$, $\text{join}(2, 4)$, $\text{join}(2, 5)$, $\text{join}(6, 7)$, $\text{join}(8, 9)$, $\text{join}(6, 8)$, $\text{join}(0, 6)$, $\text{join}(0, 2)$.
 1. simple union,
 2. union by size,
 3. union by height, and
 4. union by rank.
9. For each final partition obtained in Exercise □, show the result of performing a *collapsing find* operation for item 9.
10. Consider the initial partition P of the universe $U = \{0, 1, \dots, N - 1\}$ comprised of N sets[24].
 1. Show that $N-1$ join operations can be performed before the number of elements in the partition is reduced to one.
 2. Show that if n join operations are done ($0 \leq n < N$), the size of the largest element of the partition is at most $n+1$.
 3. A *singleton* is an element of a partition that contains only one element of the universal set. Show that when n join operations are done ($0 \leq n < N$), at least $\max\{N - 2n, 0\}$ singletons are left.
 4. Show that if less than $\lceil N/2 \rceil$ join operations are done, at least one singleton is left.

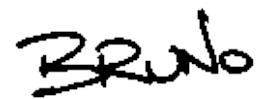
Projects

1. Complete the `SetAsArray` class introduced in Program □ by providing suitable definitions for the following operations: `purge`, `getIsEmpty`, `getIsFull`, `getCount`, `_compareTo`, `accept`, and `__iter__`. Write a test program and test your implementation.
2. Complete the `SetAsBitVector` class introduced in Program □ by providing suitable definitions for the following methods: `purge`, `getIsEmpty`, `getIsFull`, `getCount`, `_compareTo`, `accept`, and `__iter__`. Write a test program and test your implementation.
3. Rewrite the `insert`, `withdraw`, and `__contains__` methods of the `SetAsBitVector` implementation so that they use bitwise shift and mask operations rather than division and modulo operations. Compare the running times of the modified methods with the original ones and explain your observations.
4. Complete the `MultisetAsArray` class introduced in Program □ by providing suitable definitions for the following methods: `purge`, `getCount`, `_compareTo`, `accept`, and `__iter__`. Write a test program and test your implementation.
5. Complete the `MultisetAsLinkedList` class introduced in Program □ by providing suitable definitions for the following methods: `purge`, `getIsEmpty`, `getIsFull`, `getCount`, `_compareTo`, `accept`, and `iter`. Write a test program and test your implementation.
6. Design and implement a multiset class in which the contents of the set are represented by a linked list of ordered pairs of the form (i, n_i) , where i is an element of the universal set U and n_i is a non-negative integer that counts the number of instances of the element i in the multiset. (See Exercises □ and □).
7. Write a program to compute the number of ways in which a set of n elements can be partitioned. That is, compute $\sum_{m=0}^n \{ \frac{n}{m} \}$ where

$$\left\{ \frac{n}{m} \right\} = \begin{cases} 1 & n = 1, \\ 1 & n = m, \\ m \left\{ \frac{n-1}{m} \right\} + \left\{ \frac{n-1}{m-1} \right\} & \text{otherwise.} \end{cases}$$

Hint: See Section □.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

Garbage Collection and the Other Kind of Heap

A Python object is an instance of a Python type. For example, all integers have the type `int`, all floating-point numbers have the type `float`, all user-defined classes have the type `classobj`, and all instances of user-defined classes have the type `instance`. Every object in a Python program occupies some memory. The manner in which a Python object is represented in memory is left up to the implementor of the Python virtual machine and, in principle, can vary from one implementation to another. However, object data typically occupy contiguous memory locations.

The region of memory in which objects are allocated dynamically is often called *a heap*. In Chapter 1 we consider *heaps* and *heap-ordered trees* in the context of priority queue implementations. Unfortunately, the only thing that the heaps of Chapter 1 and the heap considered here have in common is the name. While it may be possible to use a heap (in the sense of Definition 1) to manage a region of memory, typical implementations do not. In this context the technical meaning of the term *heap* is closer to its dictionary definition--``a pile of many things.''

The amount of memory required to represent a Python object is determined by its type. For example, four bytes are used typically to represent the value of an `int`, eight bytes are used typically to represent the value of a `float`, 24 bytes are used typically to represent the value of a `classobj`, and 12 bytes are used typically to represent the value of a `instance`. In addition to the memory required for the value of an object, there is a fixed, constant amount of extra storage set aside in every object (eight bytes). This extra storage carries information used by the Python virtual machine to represent the type of the object and to aid the process of garbage collection.

When the Python virtual machine creates an object, it performs the following steps:

1. An unused region of memory large enough to hold an instance of the

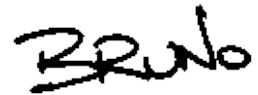
desired class is found.

2. All of the fields of the object are assigned their default initial values.
 3. The appropriate `__init__` method is run to initialize the object instance.
 4. A reference to the newly created object is returned.
-

- [What is Garbage?](#)
 - [Reference Counting Garbage Collection](#)
 - [Mark-and-Sweep Garbage Collection](#)
 - [Stop-and-Copy Garbage Collection](#)
 - [Mark-and-Compact Garbage Collection](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



What is Garbage?

While Python provides the means to create an object, the language does not provide the means to destroy an object *explicitly*. As long as a program contains a reference to some object instance, the Python virtual machine is required to ensure that the object exists. If the Python language provided the means to destroy objects, it would be possible for a program to destroy an object even when a reference to that object still existed. This situation is unsafe because the program could attempt later to invoke a method on the destroyed object, leading to unpredictable results.

The situation which arises when a program contains a reference (or pointer) to a destroyed object is called a *dangling reference* (or dangling pointer). By disallowing the explicit destruction of objects, Python eliminates the problem of dangling references.

Languages that support the explicit destruction of objects typically require the program to keep track of all the objects it creates and to destroy them explicitly when they are not longer needed. If a program somehow loses track of an object it has created then that object cannot be destroyed. And if the object is never destroyed, the memory occupied by that object cannot be used again by the program.

A program that loses track of objects before it destroys them suffers from a *memory leak*. If we run a program that has a memory leak for a very long time, it is quite possible that it will exhaust all the available memory and eventually fail because no new objects can be created.

It would seem that by disallowing the explicit destruction of objects, a Python program is doomed to eventual failure due to memory exhaustion. Indeed this would be the case, were it not for the fact that the Python language specification requires the Python virtual machine to be able to find unreferenced objects and to reclaim the memory locations allocated to those objects. An unreferenced object is called *garbage* and the process of finding all the unreferenced objects and reclaiming the storage is called *garbage collection*.

Just as the Python language does not specify precisely how objects are to be

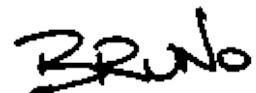
represented in the memory of a Python virtual machine, the language specification also does not stipulate how the garbage collection is to be implemented or when it should be done. Garbage collection is usually invoked when the total amount of memory allocated to a Python program exceeds some threshold. Typically, the program is suspended while the garbage collection is done.

In the analyses presented in the preceding chapters we assume that the running time to create a new object is a fixed constant, τ_{new} , and we completely ignore the garbage collection overhead. In reality, neither assumption is valid. Even if sufficient memory is available, the time required by the Python virtual machine to locate an unused region of memory depends very much on the data structures used to keep track of the memory regions allocated to a program as well as on the way in which a program uses the objects it creates. Furthermore, creating a new object may trigger the garbage collection process. The running time for garbage collection can be a significant fraction of the total running time of a program.

- [Reduce, Reuse, Recycle](#)
 - [Helping the Garbage Collector](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Reduce, Reuse, Recycle

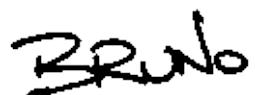
Modern societies produce an excessive amount of waste. The costs of doing so include the direct costs of waste disposal as well as the damage to the environment caused by the manufacturing, distribution, and ultimate disposal of products. The slogan ``*reduce, reuse, recycle*,'' prescribes three strategies for reducing the environmental costs associated with waste materials.

These strategies apply equally well to Python programs! A Python program that creates excessive garbage may require more frequent garbage collection than a program that creates less garbage. Since garbage collection can take a significant amount of time to do, it makes sense to use strategies that decrease the cost of garbage collection.

-
- [Reduce](#)
 - [Reuse](#)
 - [Recycle](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Reduce

A Python program that does not create any object instances or arrays does not create garbage. (Of course, a Python program that creates no objects is not very useful because it does nothing.) Similarly, a program that creates all the objects it needs at the beginning of its execution and uses the same objects until it terminates also does not create garbage. By reducing the number of objects a program creates dynamically during its execution, we can reduce or even eliminate the need for garbage collection.

Reuse

Sometimes, a Python program will create many objects which are used only once. For example, a program may create an object in the body of a loop that is used to hold "temporary" information that is only required for the particular iteration of the loop in which it is created. Consider the following:

```
for i in xrange(1000000):
    obj = SomeClass(i)
    print obj
```

This creates a million instances of the `SomeClass` class and prints them out. If the `SomeClass` class has a property, say `value`, we can reuse an a single object instance like this:

```
obj = SomeClass()
for i in xrange(1000000):
    obj.value = i
    print obj
```

Clearly, by reusing a single object instance, we have dramatically reduced the amount of garbage produced.

Recycle

Recycling of objects is a somewhat more complex strategy for reducing the overhead associated with garbage collection. Instead leaving an unused object around for the garbage collector to find, it is put into a container of unused objects. When a new object is needed, the container is searched first to see if an unused one already exists. Because a container always refers to the objects it contains, those objects are never garbage collected.

The recycling strategy can indeed reduce garbage collection overhead. However, it puts the burden back on the programmer to explicitly put unused objects into the container (avoid memory leaks) and to make sure objects put into the container are really unused (avoid dangling references). Because the recycling strategy undermines some of the benefits of garbage collection, it should be used with great care.

Helping the Garbage Collector

The preceding section presents strategies for avoiding garbage collection. However, there are times when garbage collection is actually desirable. Imagine a program that requires a significant amount of memory. Suppose the amount of memory required is very close to the amount of memory available for use by the Python virtual machine. The performance of such a program is going to depend on the ability of the garbage collector to find and reclaim as much unused storage as possible. Otherwise, the garbage collector will run too often. In this case, it pays to help out the garbage collector.

How can we help out the garbage collector? Since the garbage collector collects only unreferenced objects it is necessary to eliminate all references to objects that are no longer needed. This can be done by assigning the value `None` to every variable that refers to an object that is no longer needed. Alternatively, the Python `del` statement can be used to remove the binding for a name. Either way, helping the garbage collector requires a program to do a bit more work.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Reference Counting Garbage Collection

The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist?

A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called a *reference count*. The idea is that the reference count field is not accessible to the Python program. Instead, the reference count field is updated by the Python virtual machine itself.

Consider the statement

```
p = int(57)
```

which creates a new instance of the `int` class. Only a single variable, `p`, refers to the object. Thus, its reference count should be one.

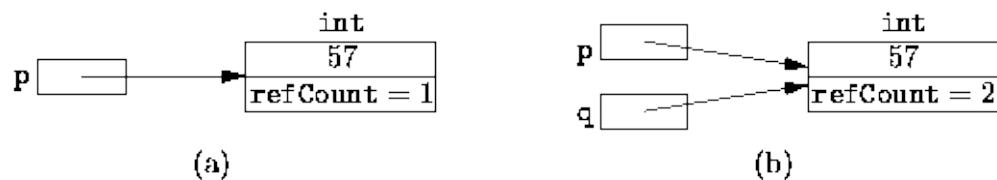


Figure: Objects with reference counters.

Now consider the following sequence of statements:

```
p = int(57)
q = p
```

This sequence creates a single `int` instance. Both `p` and `q` refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be

necessary to update several reference counts. Suppose p and q are both reference variables. The assignment

```
p = q
```

would be implemented by the Python virtual machine as follows:

```
if p is not q:  
    if p is not None:  
        p.refCount -= 1  
    p = q  
    if p is not None:  
        p.refCount += 1
```

For example suppose p and q are initialized as follows:

```
p = int(57)  
q = int(99)
```

As shown in Figure □ (a), two `int` objects are created, each with a reference count of one. Now, suppose we assign q to p using the code sequence given above. Figure □ (b) shows that after the assignment, both p and q refer to the same object--its reference count is two. And the reference count on `int` has gone to zero which indicates that it is garbage.

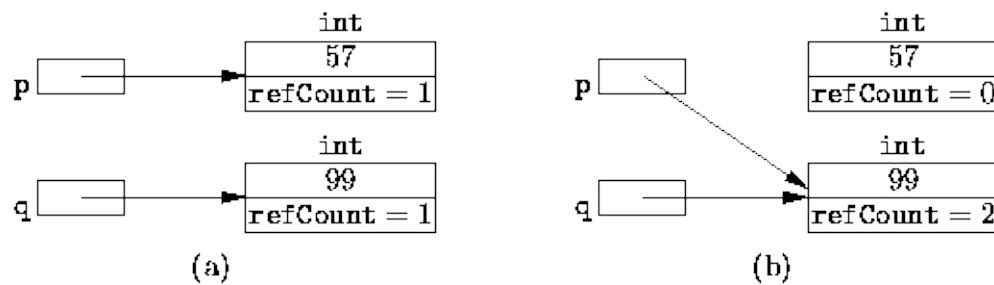


Figure: Reference counts before and after the assignment $p = q$.

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the

garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Python assignment `p = q` in the Python virtual machine as follows:

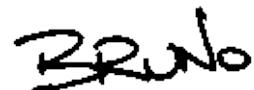
```
if p is not q:  
    if p is not None:  
        p.refCount -= 1  
        if p.refCount == 0:  
            heap.release(p)  
    p = q  
    if p is not None:  
        p.refCount += 1
```

Notice that the `release` method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.

- [When Objects Refer to Other Objects](#)
 - [Why Reference Counting Does Not Work](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



When Objects Refer to Other Objects

The `int` objects considered in the preceding examples are very simple objects--they contain no references to other objects. Reference counting is an ideal strategy for garbage collecting such objects. But what about objects that refer to other objects? For example, consider the `Association` class described in Chapter 1 which represents a $(\text{key}, \text{value})$ pair. We can still use reference counting, provided we count all references to an object *including references from other objects*.

Figure 1(a) illustrates the contents memory following the execution of this statement:

```
p = Association(int(57), int(99))
```

The reference count of the `Association` is one, because the variable `p` refers to it. Similarly, the reference counts of the two `int` instances are one because the `Association` refers to both of them.

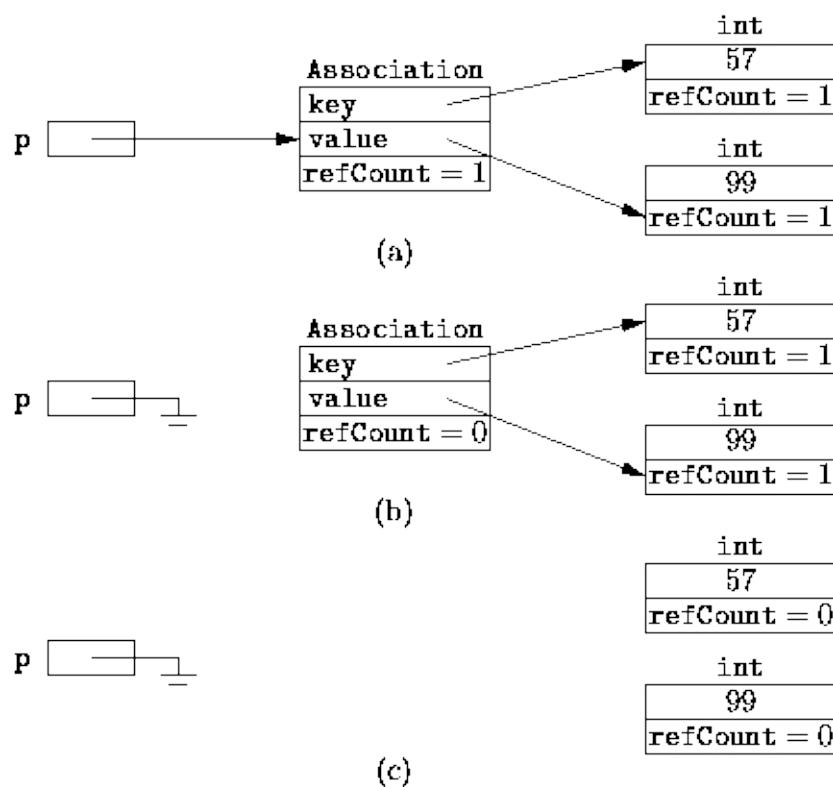


Figure: Reference counting when objects refer to other objects.

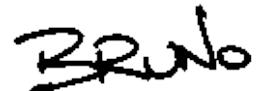
Suppose we assign the value `None` to the variable `p`. As shown in Figure □ (b), the reference count of the association becomes zero--it is now garbage.

However, until the `Association` instance continues to exist until it is garbage collected. And because it still exists, it still refers to the `int` objects.

Figure □ (d) shows that the garbage collection process adjusts the reference counts on the objects to which the association refers only when the association is garbage collected. The two `int` objects are now unreferenced and can be garbage collected as well.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Why Reference Counting Does Not Work

So far, reference counting looks like a good idea. However, the reference counting does not always work. Consider a circular, singly-linked list such as the one shown in Figure □ (a). In the figure, the variable `head` refers to the head of the linked list and the last element of the linked list also refers to the head. Therefore, the reference count on the first list element is two; whereas, the remaining list elements each has a reference count of one.

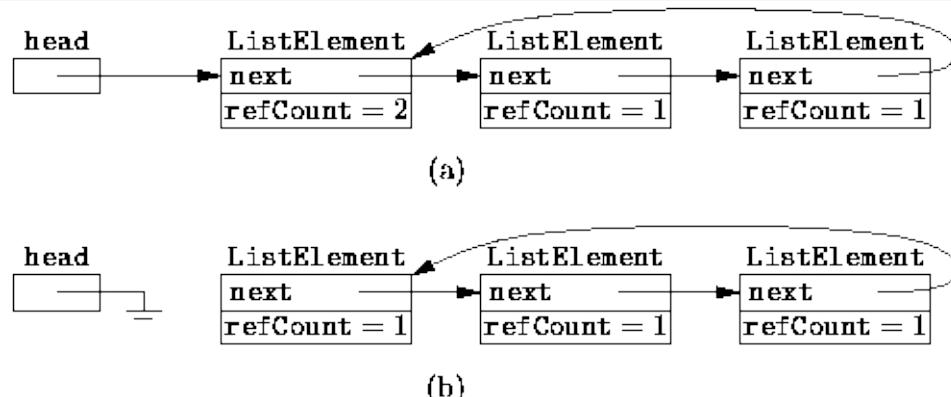


Figure: Why reference counting fails.

Consider what happens when we assign the value `None` to the `head` variable. This results in the situation shown in Figure □ (b). The reference count on the first list element has been decreased by one because the `head` variable no longer refers to it. However, its reference count is not zero, because the tail of the list still refers to the head.

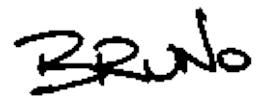
We now have a problem. The reference counts on all the lists elements are non-zero. Therefore, they are not considered to be garbage by a reference counting garbage collector. On the other hand, no external references to the linked-list elements remain. Therefore, the list elements are indeed garbage.

This example illustrates the Achilles' heel of reference counting--circular data structures. In general, reference counting will fail to work whenever the data structure contains a cycle of references. Python does not prevent the creation of cyclic structures. Therefore, reference counting by itself is not a suitable garbage collection scheme for arbitrary objects. Nevertheless, it is an extremely useful technique for dealing with simple objects that don't refer to other objects, such as

ints and floatss.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Mark-and-Sweep Garbage Collection

This section presents the *mark-and-sweep* garbage collection algorithm. The mark-and-sweep algorithm was the first garbage collection algorithm to be developed that is able to reclaim cyclic data structures. ♦ Variations of the mark-and-sweep algorithm continue to be among the most commonly used garbage collection techniques.

When using mark-and-sweep, unreferenced objects are not reclaimed immediately. Instead, garbage is allowed to accumulate until all available memory has been exhausted. When that happens, the execution of the program is suspended temporarily while the mark-and-sweep algorithm collects all the garbage. Once all unreferenced objects have been reclaimed, the normal execution of the program can resume.

The mark-and-sweep algorithm is called a *tracing* garbage collector because it *traces out* the entire collection of objects that are directly or indirectly accessible by the program. The objects that a program can access directly are those objects which are referenced by local variables on the processor stack as well as by any global variables that refer to objects. In the context of garbage collection, these variables are called the *roots*. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. An accessible object is said to be *live*. Conversely, an object which is not *live* is garbage.

The mark-and-sweep algorithm consists of two phases: In the first phase, it finds and marks all accessible objects. The first phase is called the *mark* phase. In the second phase, the garbage collection algorithm scans through the heap and reclaims all the unmarked objects. The second phase is called the *sweep* phase. The algorithm can be expressed as follows:

```
def markAndSweep():
    for r in roots:
        mark(r)
    sweep()
```

In order to distinguish the live objects from garbage, we record the state of an

object in each object. That is, we add a special `bool` field to each object called, say, `marked`. By default, all objects are unmarked when they are created. Thus, the `marked` field is initially `False`.

An object `p` and all the objects indirectly accessible from `p` can be marked by using the following recursive `mark` method:

```
def mark(p):
    if not p.marked:
        p.marked = True
        for q in successors(p):
            mark(q)
```

Notice that this recursive `mark` algorithm does nothing when it encounters an object that has already been marked. Consequently, the algorithm is guaranteed to terminate. And it terminates only when all accessible objects have been marked.

In its second phase, the mark-and-sweep algorithm scans through all the objects in the heap, in order to locate all the unmarked objects. The storage allocated to the unmarked objects is reclaimed during the scan. At the same time, the `marked` field on every live object is set back to `False` in preparation for the next invocation of the mark-and-sweep garbage collection algorithm:

```
def sweep():
    for p in heap:
        if p.marked:
            p.marked = False
        else:
            heap.release(p)
```

Figure □ illustrates the operation of the mark-and-sweep garbage collection algorithm. Figure □(a) shows the conditions before garbage collection begins. In this example, there is a single root variable. Figure □(b) shows the effect of the `mark` phase of the algorithm. At this point, all live objects have been marked. Finally, Figure □(c) shows the objects left after the `sweep` phase has been completed. Only live objects remain in memory and the `marked` fields have all been set to `False` again.

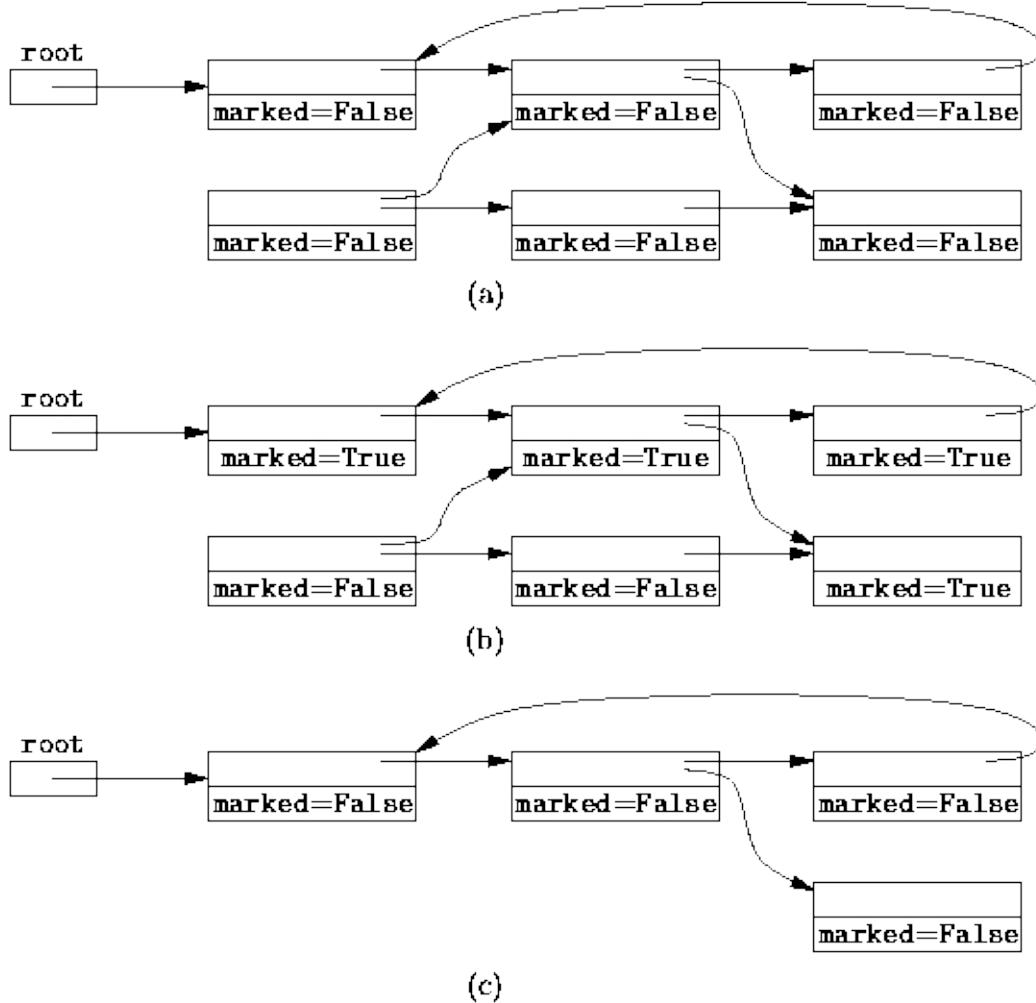


Figure: Mark-and-sweep garbage collection.

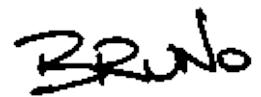
Because the mark-and-sweep garbage collection algorithm traces out the set of objects accessible from the roots, it is able to correctly identify and collect garbage even in the presence of reference cycles. This is the main advantage of mark-and-sweep over the reference counting technique presented in the preceding section. A secondary benefit of the mark-and-sweep approach is that the normal manipulations of reference variables incurs no overhead.

The main disadvantage of the mark-and-sweep approach is the fact that normal program execution is suspended while the garbage collection algorithm runs. In particular, this can be a problem in a program that interacts with a human user or that must satisfy real-time execution constraints. For example, an interactive application that uses mark-and-sweep garbage collection becomes unresponsive periodically.

- [The Fragmentation Problem](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



The Fragmentation Problem

Fragmentation is a phenomenon that occurs in a long-running program that has undergone garbage collection several times. The problem is that objects tend to become spread out in the heap. Live objects end up being separated by many, small unused memory regions. The problem in this situation is that it may become impossible to allocate memory for an object. While there may indeed be sufficient unused memory, the unused memory is not contiguous. Since objects typically occupy consecutive memory locations it is impossible to allocate storage.

The mark-and-sweep algorithm does not address fragmentation. Even after reclaiming the storage from all garbage objects, the heap may still be too fragmented to allocate the required amount of space. The next section presents an alternative to the mark-and-sweep algorithm that also *defragments* (or *compacts*) the heap.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Stop-and-Copy Garbage Collection

The section describes a garbage collection approach that collects garbage *and* defragments the heap called *stop-and-copy*. When using the stop-and-copy garbage collection algorithm, the heap is divided into two separate regions. At any point in time, all dynamically allocated object instances reside in only one of the two regions--the *active* region. The other, *inactive* region is unoccupied.

When the memory in the active region is exhausted, the program is suspended and the garbage-collection algorithm is invoked. The stop-and-copy algorithm copies all of the live objects from the active region to the inactive region. As each object is copied, all references contained in that object are updated to reflect the new locations of the referenced objects.

After the copying is completed, the active and inactive regions exchange their roles. Since the stop-and-copy algorithm copies only the live objects, the garbage objects are left behind. In effect, the storage occupied by the garbage is reclaimed all at once when the active region becomes inactive.

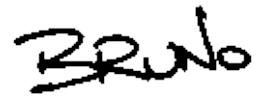
As the stop-and-copy algorithm copies the live objects from the active region to the inactive region, it stores the objects in contiguous memory locations. Thus, the stop-and-copy algorithm automatically defragments the heap. This is the main advantage of the stop-and-copy approach over the mark-and-sweep algorithm described in the preceding section.

The costs of the stop-and-copy algorithm are twofold: First, the algorithm requires that *all* live objects be copied every time garbage collection is invoked. If an application program has a large memory footprint, the time required to copy all objects can be quite significant. A second cost associated with stop-and-copy is the fact that it requires twice as much memory as the program actually uses. When garbage collection is finished, at least half of the memory space is unused.

- [The Copy Algorithm](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

The Copy Algorithm

The stop-and-copy algorithm divides the heap into two regions--an active region and an inactive region. For convenience, we can view each region as a separate heap and we shall refer to them as `activeHeap` and `inactiveHeap`. When the stop-and-copy algorithm is invoked, it copies all live objects from the `activeHeap` to the `inactiveHeap`. It does so by invoking the `copy` method given below starting at each root:

```
for r in roots:  
    r = copy(r, inactiveHeap)  
swap(activeHeap, inactiveHeap)
```

The `copy` method is complicated by the fact that it needs to update all object references contained in the objects as it copies those objects. In order to facilitate this, we record in every object a reference to its copy. That is, we add a special field to each object called `forward` which is a reference to the copy of this object.

The recursive `copy` method given below copies a given object and all the objects indirectly accessible from the given object to the destination heap. When the `forward` field of an object is `None`, it indicates that the given object has not yet been copied. In this case, the method creates a new instance of the object class in the destination heap. Then, the fields of the object are copied one-by-one. If the field is a value type, the value of that field is copied. However, if the field refers to another object, the `copy` method calls itself recursively to copy that object.

```
def copy(p, destination):  
    if p is None:  
        return None  
    if p.forward is None:  
        q = destination.newInstance(type(p))  
        p.forward = q  
        for f in successors(p):  
            q.f = copy(p.f, destination)  
        q.forward = None  
    return p.forward
```

If the `copy` method is invoked for an object whose `forward` field is non-`None`, that object has already been copied and the `forward` field refers to the copy of that object in the destination heap. In that case, the `copy` method simply returns a

reference to the previously copied object.

Figure □ traces the execution of the stop-and-copy garbage collection algorithm. When the algorithm is invoked and before any objects have been copied, the `forward` field of every object in the active region is `None` as shown in Figure □ (a). In Figure □ (b), a copy of object *A*, called *A'*, has been created in the inactive region, and the `forward` field of *A* refers to *A'*.

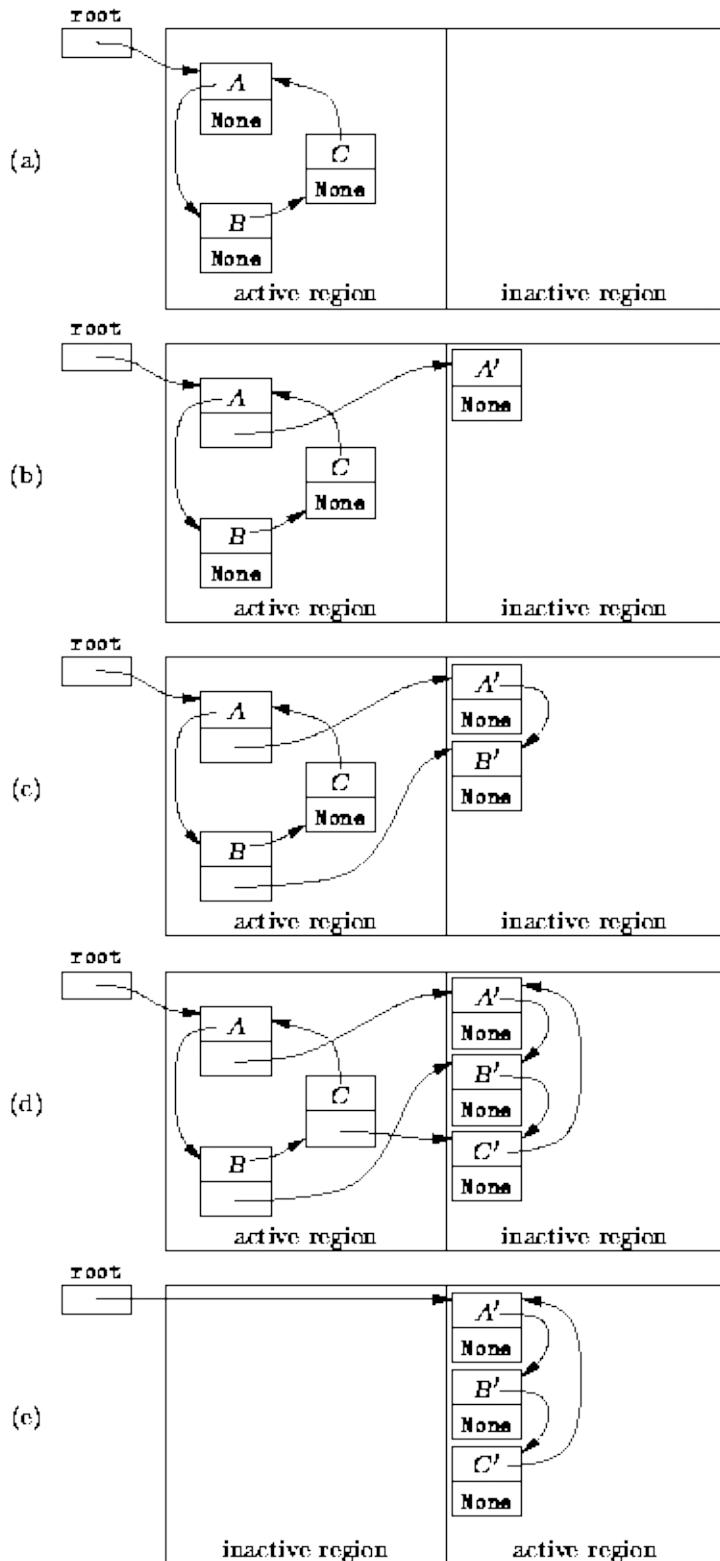


Figure: Stop-and-copy garbage collection.

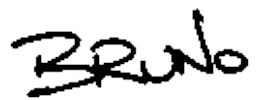
Since **A** refers to **B**, the next object copied is object **B**. As shown in Figure □ (c),

fragmentation is eliminated by allocating storage for B' immediately next to A' . Next, object C is copied. Notice that C refers to A , but A has already been copied. Object C' obtains its reference to A' from the **forward** field of A as shown in Figure □ (d).

After all the live objects have been copied from the active region to the inactive region, the regions exchange their roles. As shown in Figure □ (e), all the garbage has been collected and the heap is no longer fragmented.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Mark-and-Compact Garbage Collection

The mark-and-sweep algorithm described in Section □ has the unfortunate tendency to fragment the heap. The stop-and-copy algorithm described in Section □ avoids fragmentation at the expense of doubling the size of the heap. This section describes the *mark-and-compact* approach to garbage collection which eliminates fragmentation without the space penalty of stop-and-copy.

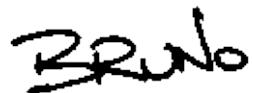
The mark-and-compact algorithm consists of two phases: In the first phase, it finds and marks all live objects. The first phase is called the *mark* phase. In the second phase, the garbage collection algorithm compacts the heap by moving all the live objects into contiguous memory locations. The second phase is called the *compaction* phase. The algorithm can be expressed as follows:

```
def markAndCompact():
    for r in roots:
        mark(r)
    compact()
```

-
- [Handles](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Handles

The Python virtual machine specification does not prescribe how reference variables are implemented. One approach is for a reference variable to be implemented as an index into an array of object *handles*. Every object has its own handle. The handle for an object typically contains a reference to a type object that describes the type of the object associated with the handle and a pointer to the region in the heap where the object data resides.

The advantage of using handles is that when the position in the heap of an object is changed, only the handle for that object needs to be modified. All other references to that object are unaffected because such references actually refer to the handle. The cost of using handles is that the handle must be dereferenced every time an object is accessed.

The mark-and-compact algorithm uses the handles in two ways: First, the marked flags which are set during the mark operation are stored in the handles rather than in the objects themselves. Second, compaction is greatly simplified because when an object is moved only its handle needs to be updated--all other objects are unaffected.

Figure □ illustrates how object references are implemented using handles. Figure □ (a) shows a circular, singly-linked list as it is usually drawn and Figure □ (b) shows how the list is represented when using handles. Each reference variable actually contains an index into the array of handles. For example, the head variable selects the handle at offset 2 and that handle points to linked list element A. Similarly, the next field of list element A selects the handle at offset 5 which refers to list element B. Notice that when an object is moved, only its handle needs to be modified.

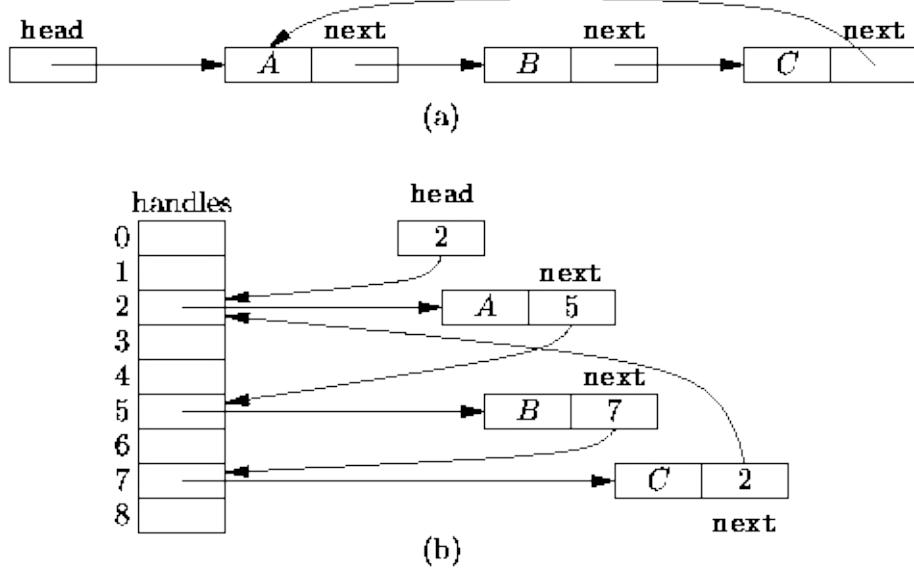


Figure: Representing object references using handles.

The handle is a convenient place in which to record information used by the garbage collection algorithm. For example, we add a `bool` field to each handle, called `marked`. The `marked` field is used to mark live objects as follows:

```
def mark(p):
    if not handle[p].marked:
        handle[p].marked = True
        for q in successors(p):
            mark(q)
```

Notice that this version of the `mark` method marks the object handles rather than the objects themselves.

Once all of the live objects in the heap have been identified, the heap needs to be defragmented. Perhaps the simplest way to defragment the heap is to *slide* the objects in the heap all to one end, removing the unused memory locations separating them. The following version of the `compact` method does just this:

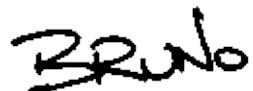
```
def compact():
    offset = 0
    for p in heap:
        if handle[p].marked:
            handle[p].obj = heap.move(p, offset)
            handle[p].marked = False
            offset += sizeof(type(p))
```

This algorithm makes a single pass through the objects in the heap, moving the

live objects towards the lower heap addresses as it goes. The compact method only modifies the object handles--object data remain unchanged. This algorithm also illustrates an important characteristic of the sliding compaction algorithm--the relative positions of the objects in the heap remains unchanged after the compaction operation. Also, when the compaction method has finished, the marked fields have all been set back to `False` in preparation for the next garbage collection operation.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

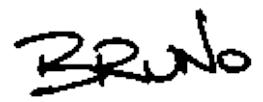


Exercises

1. Let M be the size of the heap and let f be the fraction of the heap occupied by live data. Estimate the running time of the `mark` method of the *mark-and-sweep* garbage collection scheme as a function of f and M .
 2. Repeat Exercise □ for the `copy` method. Estimate the running time of the `copy` method of the *stop-and-copy* garbage collection scheme.
 3. Repeat Exercise □ for the `copy` method. Estimate the running time of the `compact` method of the *stop-and-compact* garbage collection scheme.
 4. Using your answers to Exercises □, □ and □, show that running time of garbage collection is *inversely proportional* to the amount of storage recovered.
 5. The efficiency of a garbage collection scheme is the rate at which memory is reclaimed. Using your answers to Exercises □ and □ compare the efficiency of *mark-and-sweep* with that of *stop-and-copy*.
 6. Devise a *non-recursive* algorithm for the `mark` method of the *mark-and-sweep* garbage collection scheme.
 7. Repeat Exercise □ for the `copy` method of the *stop-and-copy* garbage collection scheme.
 8. Repeat Exercise □ for the `mark` method of the *mark-and-compact* garbage collection scheme.
 9. Consider the use of *handles* for representing object references. Is it correct to assume that the order which objects appear in the heap is the same as the order in which the corresponding handles appear in the array of handles? How does this affect *compaction* of the heap?
 10. Consider the `compact` method of the *mark-and-compact* garbage collection scheme. The algorithm visits the objects in the heap in the order in which they appear in the heap, rather than in the order in which the corresponding handles appear in the array of handles. Why is this necessary?
 11. The `compact` method of the *mark-and-compact* garbage collection scheme slides the objects in the heap all to one end, but leaves the handles where they are. As a result, the handle array becomes *fragmented*. What modifications are necessary in order to compact the handle array as well as the heap?
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Projects

1. Devise and conduct a set of experiments to measure garbage collection overhead. For example, write a program that creates a specified number of garbage objects as quickly as possible. Determine the number of objects needed to trigger garbage collection. Measure the running time of your program when no garbage collection is performed and compare it to the running time observed when garbage collection is invoked.
2. Python does not provide the means for accessing memory directly. Consequently, it is not possible to implement the Python heap in Python (without using native methods). Nevertheless, we can *simulate* a heap using a Python array of ints. Write a Python class that manages an array of ints. Your class should implement the following methods:

```
class Heap(object):

    def acquire(self, size):
        "(Heap, int) -> int"
        # ...

    def release(self, offset):
        "(Heap, int) -> None"
        # ...

    def __getitem__(self, offset):
        "(Heap, int) -> int"
        # ...

    def __setitem__(self, offset, value):
        "(Heap, int, int) -> None"
        # ...
```

The `acquire` method allocates a region of `size` consecutive ints in the array and returns the offset of the first `int` in the region. The `release` method release a region of `ints` at the specified `offset` which was obtained previously using `acquire`. The `__getitem__` and `__setitem__` methods access a value in the array at a given `offset`.

3. Using an array of `ints` simulate the *mark-and-sweep* garbage collection as follows:
 1. Write a `Handle` class that contains the methods given below:

```

class Handle(Object):

    def getSize(self):
        "(Handle) -> int"
        # ...

    getReference(self, offset):
        "(Handle, int) -> Handle"
        # ...

    setReference(self, offset, handle):
        "(Handle, int, Handle) -> None"
        # ...

    def __getitem__(self, offset):
        "(Handle, int) -> int"
        # ...

    def __setitem__(self, offset, value):
        "(Handle, int, int) -> int"
        # ...

```

A handle refers to an object that contains either ints or other handles. The size of an object is total the number of ints and handles it contains. The various store and fetch methods are used to insert and remove items from the object to which this handle refers.

2. Write a Heap class that contains the methods given below:

```

class Heap(object):

    def acquire(self, size):
        "(Heap, int) -> Handle"
        # ...

    def release (self, handle):
        "(Heap, Handle) -> None"
        # ...

    def collectGarbage(self):
        "(Heap) -> None"
        #

```

The acquire method allocates a handle and space in the heap for an object of the given size. The release method releases the given handle but does not reclaim the associated heap space. The collectGarbage method performs the actual garbage collection operation.

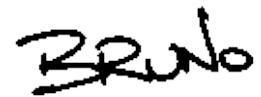
4. Using the approach described in Project □, implement a simulation of

mark-and-compact garbage collection.

5. Using the approach described in Project □ implement a simulation of *reference-counting* garbage collection.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Algorithmic Patterns and Problem Solvers

This chapter presents a number of different algorithmic patterns. Each pattern addresses a category of problems and describes a core solution strategy for that category. Given a problem to be solved, we may find that there are several possible solution strategies. We may also find that only one strategy applies or even that none of them do. A good programmer is one who is proficient at examining the problem to be solved and identifying the appropriate algorithmic technique to use. The following algorithmic patterns are discussed in this chapter:

direct solution strategies

Brute force algorithms and greedy algorithms.

backtracking strategies

Simple backtracking and branch-and-bound algorithms.

top-down solution strategies

Divide-and-conquer algorithms.

bottom-up solution strategies

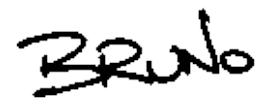
Dynamic programming.

randomized strategies

Monte Carlo algorithms and simulated annealing.

-
- [Brute-Force and Greedy Algorithms](#)
 - [Backtracking Algorithms](#)
 - [Top-Down Algorithms: Divide-and-Conquer](#)
 - [Bottom-Up Algorithms: Dynamic Programming](#)
 - [Randomized Algorithms](#)
 - [Exercises](#)
 - [Projects](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Brute-Force and Greedy Algorithms

In this section we consider two closely related algorithm types--brute-force and greedy. *Brute-force algorithms* are distinguished not by their structure or form, but by the way in which the problem to be solved is approached. A brute-force algorithm solves a problem in the most simple, direct or obvious way. As a result, such an algorithm can end up doing far more work to solve a given problem than a more clever or sophisticated algorithm might do. On the other hand, a brute-force algorithm is often easier to implement than a more sophisticated one and, because of this simplicity, sometimes it can be more efficient.

Often a problem can be viewed as a sequence of decisions to be made. For example, consider the problem of finding the best way to place electronic components on a circuit board. To solve this problem we must decide where on the board to place each component. Typically, a brute-force algorithm solves such a problem by exhaustively enumerating all the possibilities. That is, for every decision we consider each possible outcome.

A greedy algorithm is one that makes the sequence of decisions (in some order) such that once a given decision has been made, that decision is never reconsidered. For example, if we use a greedy algorithm to place the components on the circuit board, once a component has been assigned a position it is never again moved. Greedy algorithms can run significantly faster than brute force ones. Unfortunately, it is not always the case that a greedy strategy leads to the correct solution.

-
- [Example-Counting Change](#)
 - [Example-0/1 Knapsack Problem](#)
-

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Example-Counting Change

Consider the problem a cashier solves every time he counts out some amount of currency. The cashier has at his disposal a collection of notes and coins of various denominations and is required to count out a specified sum using the smallest possible number of pieces.

The problem can be expressed mathematically as follows: Let there be n pieces of money (notes or coins), $P = \{p_1, p_2, \dots, p_n\}$, and let d_i be the denomination of p_i . For example, if p_i is a dime, then $d_i = 10$. To count out a given sum of money A we find the smallest subset of P , say $S \subseteq P$, such that $\sum_{p_i \in S} d_i = A$.

One way to represent the subset S is to use n variables $X = \{x_1, x_2, \dots, x_n\}$, such that

$$x_i = \begin{cases} 1 & p_i \in S, \\ 0 & p_i \notin S. \end{cases}$$

Given $\{d_1, d_2, \dots, d_n\}$ our *objective* is to minimize

$$\sum_{i=1}^n x_i$$

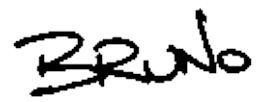
subject to the constraint

$$\sum_{i=1}^n d_i x_i = A.$$

-
- [Brute-Force Algorithm](#)
 - [Greedy Algorithm](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Brute-Force Algorithm

Since each of the elements of $X = \{x_1, x_2, \dots, x_n\}$ is either a zero or a one, there are 2^n possible values for X . A brute-force algorithm to solve this problem finds the best solution by enumerating all the possible values of X .

For each possible value of X we check first if the constraint $\sum_{i=1}^n d_i x_i = A$ is satisfied. A value which satisfies the constraint is called a *feasible solution*. The solution to the problem is the feasible solution which minimizes $\sum_{i=1}^n x_i$ which is called the *objective function*.

Since there are 2^n possible values of X the running time of a brute-force solution is $\Omega(2^n)$. The running time needed to determine whether a possible value is a feasible solution is $O(n)$ and the time required to evaluate the objective function is also $O(n)$. Therefore, the running time of the brute-force algorithm is $O(n2^n)$.

Greedy Algorithm

A cashier does not really consider all the possible ways in which to count out a given sum of money. Instead, he counts out the required amount beginning with the largest denomination and proceeding to the smallest denomination.

For example, suppose we have ten coins: five pennies, two nickels, two dimes, and one quarter. That is, $\{d_1, d_2, \dots, d_{10}\} = \{1, 1, 1, 1, 1, 5, 5, 10, 10, 25\}$. To count out 32 cents, we start with a quarter, then add a nickel followed by two pennies. This is a greedy strategy because once a coin has been counted out, it is never taken back. Furthermore, the solution obtained is the correct solution because it uses the fewest number of coins.

If we assume that the pieces of money (notes and coins) are sorted by their denomination, the running time for the greedy algorithm is $O(n)$. This is significantly better than that of the brute-force algorithm given above.

Does this greedy algorithm always produce the correct answer? Unfortunately it does not. Consider what happens if we introduce a 15-cent coin. Suppose we are asked to count out 20 cents from the following set of coins: $\{1, 1, 1, 1, 1, 10, 10, 15\}$. The greedy algorithm selects 15 followed by five ones--six coins in total. Of course, the correct solution requires only two coins. The solution found by the greedy strategy is a feasible solution, but it does not minimize the objective function.

Example-0/1 Knapsack Problem

The *0/1 knapsack problem* is closely related to the change counting problem discussed in the preceding section: We are given a set of n items from which we are to select some number of items to be carried in a knapsack. Each item has both a *weight* and a *profit*. The objective is to choose the set of items that fits in the knapsack and maximizes the profit.

Let w_i be the weight of the i^{th} item, p_i be the profit accrued when the i^{th} item is carried in the knapsack, and C be the capacity of the knapsack. Let x_i be a variable the value of which is either zero or one. The variable x_i has the value one when the i^{th} item is carried in the knapsack.

Given $\{w_1, w_2, \dots, w_n\}$ and $\{p_1, p_2, \dots, p_n\}$, our *objective* is to maximize

$$\sum_{i=1}^n p_i x_i$$

subject to the constraint

$$\sum_{i=1}^n w_i x_i \leq C.$$

Clearly, we can solve this problem by exhaustively enumerating the feasible solutions and selecting the one with the highest profit. However, since there are 2^n possible solutions, the running time required for the brute-force solution becomes prohibitive as n gets large.

An alternative is to use a greedy solution strategy which solves the problem by putting items into the knapsack one-by-one. This approach is greedy because once an item has been put into the knapsack, it is never removed.

How do we select the next item to be put into the knapsack? There are several possibilities:

Greedy by Profit

At each step select from the remaining items the one with the highest profit (provided the capacity of the knapsack is not exceeded). This approach tries

to maximize the profit by choosing the most profitable items first.

Greedy by Weight

At each step select from the remaining items the one with the least weight (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by putting as many items into the knapsack as possible.

Greedy by Profit Density

At each step select from the remaining items the one with the largest *profit density*, p_i/w_i (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing items with the largest profit per unit of weight.

While all three approaches generate feasible solutions, we cannot guarantee that any of them will always generate the optimal solution. In fact, it is even possible that none of them does! Table □ gives an example where this is the case.

Table: 0/1 knapsack problem example ($C=100$).

greedy by

i w_i p_i p_i/w_i profit weight density optimal solution

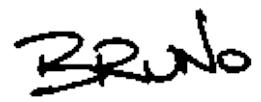
1	100	40	0.4	1	0	0	0
2	50	35	0.7	0	0	1	1
3	45	18	0.4	0	1	0	1
4	20	4	0.2	0	1	1	0
5	10	10	1.0	0	1	1	0
6	5	2	0.4	0	1	1	1
total weight		100		80	85	100	
total profit		40		34	51	55	

The bottom line about greedy algorithms is this: Before using a greedy algorithm you must make sure that it always gives the correct answer. Fortunately, in many cases this is true.



[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

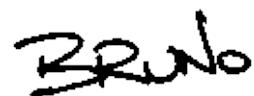
Backtracking Algorithms

In this section we consider *backtracking algorithms*. As in the preceding section, we view the problem to be solved as a sequence of decisions. A backtracking algorithm systematically considers all possible outcomes for each decision. In this sense, backtracking algorithms are like the brute-force algorithms discussed in the preceding section. However, backtracking algorithms are distinguished by the way in which the space of possible solutions is explored. Sometimes a backtracking algorithm can detect that an exhaustive search is unnecessary and, therefore, it can perform much better.

- [Example-Balancing Scales](#)
 - [Representing the Solution Space](#)
 - [Abstract Backtracking Solvers](#)
 - [Branch-and-Bound Solvers](#)
 - [Example-0/1 Knapsack Problem Again](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Balancing Scales

Consider the set of scales shown in Figure □. Suppose we are given a collection of n weights, $\{w_1, w_2, \dots, w_n\}$, and we are required to place *all* of the weights onto the scales so that they are balanced.

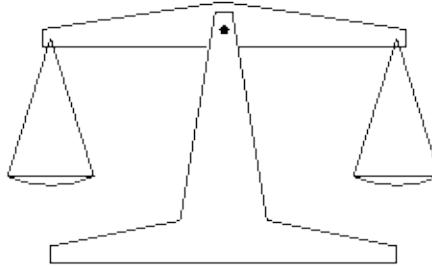


Figure: A set of scales.

The problem can be expressed mathematically as follows: Let x_i represent the pan in which weight w_i is placed such that

$$x_i = \begin{cases} 0 & w_i \text{ is placed in the left pan,} \\ 1 & w_i \text{ is placed in the right pan.} \end{cases}$$

The scales are balanced when the sum of the weights in the left pan equals the sum of the weights in the right pan,

$$\sum_{i=1}^n w_i x_i = \sum_{i=1}^n w_i (1 - x_i).$$

Given an arbitrary set of n weights, there is no guarantee that a solution to the problem exists. A solution always exists if, instead of balancing the scales, the goal is to minimize the difference between the total weights in the left and right pans. Thus, given $\{w_1, w_2, \dots, w_n\}$, our *objective* is to *minimize* δ where

$$\delta = \left| \sum_{i=1}^n w_i x_i - \sum_{i=1}^n w_i (1 - x_i) \right|$$

subject to the constraint that *all* the weights are placed on the scales.

Given a set of scales and collection of weights, we might solve the problem by trial-and-error: Place all the weights onto the pans one-by-one. If the scales balance, a solution has been found. If not, remove some number of the weights and place them back on the scales in some other combination. In effect, we search for a solution to the problem by first trying one solution and then backing-up to try another.

Figure □ shows the *solution space* for the scales balancing problem. In this case the solution space takes the form of a tree: Each node of the tree represents a *partial solution* to the problem. At the root (node A) no weights have been placed yet and the scales are balanced. Let δ be the difference between the sum of the weights currently placed in the left and right pans. Therefore, $\delta = 0$ at node A.

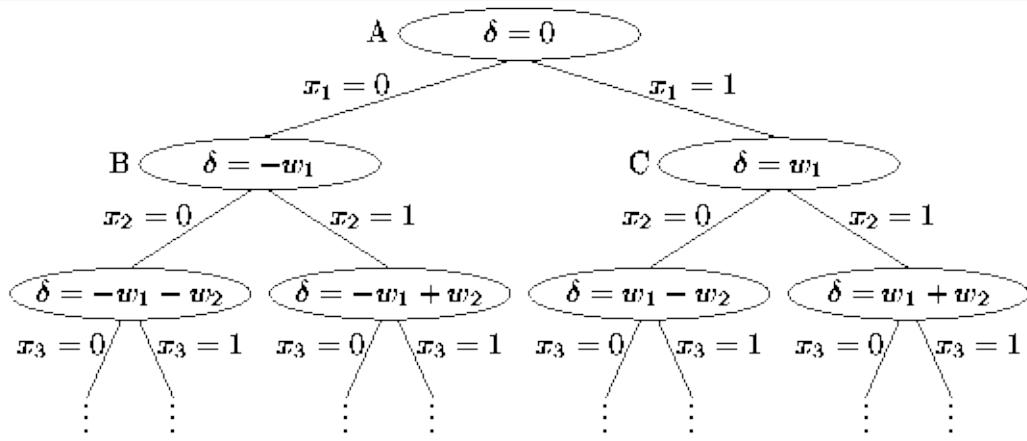


Figure: Solution space for the scales balancing problem.

Node B represents the situation in which weight w_1 has been placed in the left pan. The difference between the pans is $\delta = -w_1$. Conversely, node C represents the situation in which the weight w_1 has been placed in the right pan. In this case $\delta = +w_1$. The complete solution tree has depth n and 2^n leaves. Clearly, the solution is the leaf node having the smallest $|\delta|$ value.

In this case (as in many others) the solution space is a tree. In order to find the best solution a backtracking algorithm visits all the nodes in the solution space. That is, it does a tree *traversal*. Section □ presents the two most important tree traversals--*depth-first* and *breadth-first*. Both kinds can be used to implement a backtracking algorithm.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Representing the Solution Space

This section presents an abstract class to represent the nodes of a solution space. By using an abstract class, we hide the details of the specific problem to be solved from the backtracking algorithm. In so doing, it is possible to implement completely generic backtracking problem solvers.

Although a backtracking algorithm behaves as if it is traversing a solution tree, it is important to realize that it is not necessary to have the entire solution tree constructed at once. Instead, the backtracking algorithm creates and destroys the nodes dynamically as it explores the solution space.

Program □ defines the `Solution` class. The abstract `Solution` class extends the abstract `Object` class introduced in Program □. Each instance of a class derived from the `Solution` class represents a single node in the solution space.

```
 1 class Solution(object):
 2
 3     def __init__(self):
 4         super(Solution, self).__init__()
 5
 6     def getIsFeasible(self):
 7         pass
 8     getIsFeasible = abstractmethod(getIsFeasible)
 9
10     isFeasible = property(
11         fget = lambda self: self.getIsFeasible())
12
13     def getIsComplete(self):
14         pass
15     getIsComplete = abstractmethod(getIsComplete)
16
17     isComplete = property(
18         fget = lambda self: self.getIsComplete())
19
20     def getObjective(self):
21         pass
22     getObjective = abstractmethod(getObjective)
23
24     objective = property(
25         fget = lambda self: self.getObjective())
26
27     def getBound(self):
28         pass
29     getBound = abstractmethod(getBound)
30
31     bound = property(
32         fget = lambda self: self.getBound())
33
34     def getSuccessors(self):
35         pass
36     getSuccessors = abstractmethod(getSuccessors)
37
38     successors = property(
39         fget = lambda self: self.getSuccessors())
```

Program: Abstract Solution class.

The abstract Solution class comprises the following properties:

isFeasible

This property returns True if the solution instance is a feasible solution to

the given problem. A solution is feasible if it satisfies the problem constraints.

isComplete

This property returns `True` if the solution instance represents a complete solution. A solution is complete when all possible decisions have been made.

objective

This property returns the value of the objective function for the given solution instance.

bound

This property returns a value that is a lower bound (if it exists) on the objective function for the given solution instance as well as all the solutions that can possibly be derived from that instance. This is a hook provided to facilitate the implementation of *branch-and-bound* backtracking which is described in Section □.

successors

This property returns an iterator that enumerates all of the successors (i.e., the children) of the given solution instance. It is assumed that the children of the given node are created *dynamically*.

Abstract Backtracking Solvers

The usual way to implement a backtracking algorithm is to write a method which traverses the solution space. This section presents an alternate, object-oriented approach that is based on the notion of an *abstract solver* .

Think of a solver as an abstract machine, the sole purpose of which is to search a given solution space for the best possible solution. A machine is an object. Therefore, it makes sense that we represent it as an instance of some class.

Program □ defines the Solver class. The abstract Solver class extends the abstract Object class introduced in Program □. The Solver class contains two instance attributes, _bestSolution and _bestObjective, two concrete methods, updateBest and solve and the abstract method search. Since search is an abstract method, its implementation must be given in a derived class.

```
 1  class Solver(Object):
 2
 3      def __init__(self):
 4          super(Solver, self).__init__()
 5          self._bestSolution = None
 6          self._bestObjective = sys.maxint
 7
 8      def search(self, initial):
 9          pass
10      search = abstractmethod(search)
11
12      def solve(self, initial):
13          assert isinstance(initial, Solution)
14          self._bestSolution = None
15          self._bestObjective = sys.maxint
16          self.search(initial)
17          return self._bestSolution
18
19      def updateBest(self, solution):
20          if solution.isComplete and solution.isFeasible and \
21              solution.objective < self._bestObjective:
22              self._bestSolution = solution
23              self._bestObjective = solution.objective
24
25      # ...
```

Program: Abstract Solver class.

The purpose of the `solve` method is to solve the problem by conducting a search of the solution space. In addition to `self`, this method takes as its argument an instance of a class derived from `Solution` that is the node in the solution space from which to begin the search. The `solve` method returns the to the best solution found.

The `solve` method does not search the solution space itself--it merely sets things up for the search method. It is the `search` method, which is provided by a derived class, that does the actual searching. When `search` returns it is expected that the `_bestSolution` instance attribute will refer to the best solution and that `_bestObjective` will be the value of the objective function for the best solution.

The `updateBest` method is meant to be called by the `search` method as it explores the solution space. As each complete solution is encountered, the `updateBest` method is called to keep track of the solution which *minimizes* the objective function.

-
- [Depth-First Solver](#)
 - [Breadth-First Solver](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Depth-First Solver

This section presents a backtracking solver that finds the best solution to a given problem by performing depth-first traversal of the solution space. Program □ defines the `DepthFirstSolver` class. The `DepthFirstSolver` class extends the abstract `Solver` class defined in Program □. It provides an implementation for the search method.

```
1  class DepthFirstSolver(Solver):
2
3      def __init__(self):
4          super(DepthFirstSolver, self).__init__()
5
6      def search(self, current):
7          if current.isComplete:
8              self.updateBest(current)
9          else:
10             for successor in current.successors:
11                 self.search(successor)
```

Program: `DepthFirstSolver` class `__init__` and `search` methods.

The search method does a complete, depth-first traversal of the solution space.◊ Note that the implementation does not depend upon the characteristics of the problem being solved. In this sense the solver is a generic, *abstract solver* and can be used to solve any problem that has a tree-structured solution space!

Since the search method in Program □ visits all the nodes in the solution space, it is essentially a *brute-force* algorithm. And because the recursive method backs up and then tries different alternatives, it is called a *backtracking* algorithm.



Breadth-First Solver

If we can find the optimal solution by doing a depth-first traversal of the solution space, then we can find the solution with a breadth-first traversal too. As defined in Section □, a breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. That is, first the root is visited, then the children of the root are visited, then the grandchildren are visited, and so on.

The `BreadthFirstSolver` class is defined in Program □. The `BreadthFirstSolver` class extends the abstract `Solver` class defined in Program □. It simply provides an implementation for the `search` method.

```
 1  class BreadthFirstSolver(Solver):
 2
 3      def __init__(self):
 4          super(BreadthFirstSolver, self).__init__()
 5
 6      def search(self, initial):
 7          queue = QueueAsLinkedList()
 8          queue.enqueue(initial)
 9          while not queue.isEmpty():
10              current = queue.dequeue()
11              if current.isComplete:
12                  self.updateBest(current)
13              else:
14                  for soln in current.successors:
15                      queue.enqueue(soln)
```

Program: `BreadthFirstSolver` class `__init__` and `search` methods.

The `search` method implements a non-recursive, breadth-first traversal algorithm that uses a queue to keep track of nodes to be visited. The initial solution is enqueued first. Then the following steps are repeated until the queue is empty:

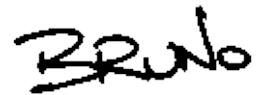
1. Dequeue the first solution in the queue.
2. If the solution is complete, call the `updateBest` method to keep track of the solution which minimizes the objective function.

3. Otherwise the solution is not complete. Enqueue all its successors.

Clearly, this algorithm does a complete traversal of the solution space.♦

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Branch-and-Bound Solvers

The depth-first and breadth-first backtracking algorithms described in the preceding sections both naively traverse the entire solution space. However, sometimes we can determine that a given node in the solution space does not lead to the optimal solution--either because the given solution and all its successors are infeasible or because we have already found a solution that is guaranteed to be better than any successor of the given solution. In such cases, the given node and its successors need not be considered. In effect, we can *prune* the solution tree, thereby reducing the number of solutions to be considered.

For example, consider the scales balancing problem described in Section \square . Consider a partial solution P_k in which we have placed k weights onto the pans ($0 \leq k < n$) and, therefore, $n-k$ weights remain to be placed. The difference between the weights of the left and right pans is given by

$$\delta = \sum_{i=1}^k w_i x_i - \sum_{i=1}^k w_i (1 - x_i),$$

and the sum of the weights still to be placed is

$$r = \sum_{i=k+1}^n w_i.$$

Suppose that $|\delta| > r$. That is, the total weight remaining is less than the difference between the weights in the two pans. Then, the best possible solution that we can obtain without changing the positions of the weights that have already been

placed is $\hat{\delta} = |\delta| - r$. The quantity $\hat{\delta}$ is a *lower bound* on the value of the objective function for all the solutions in the solution tree below the given partial solution P_k .

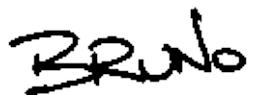
In general, during the traversal of the solution space we may have already found a complete, feasible solution for which the objective function is *less* than $\hat{\delta}$. In

that case, there is no point in considering any of the solutions below P_k . That is, we can *prune* the subtree rooted at node P_k from the solution tree. A backtracking algorithm that prunes the search space in this manner is called a *branch-and-bound* algorithm.

- [Depth-First, Branch-and-Bound Solver](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Depth-First, Branch-and-Bound Solver

Only a relatively minor modification of the simple, depth-first solver shown in Program □ is needed to transform it into a branch-and-bound solver. Program □ defines the `DepthFirstBranchAndBoundSolver` class.

```
 1  class DepthFirstBranchAndBoundSolver(Solver):
 2
 3      def __init__(self):
 4          super(DepthFirstBranchAndBoundSolver, self).__init__()
 5
 6      def search(self, current):
 7          if current.isComplete:
 8              self.updateBest(current)
 9          else:
10              for successor in current.successors:
11                  if successor.isFeasible and \
12                      successor.bound < self._bestObjective:
13                      self.search(successor)
```

Program: `DepthFirstBranchAndBoundSolver` class `__init__` and `search` methods.

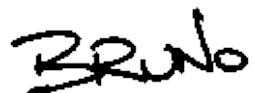
The only difference between the simple, depth-first solver and the branch-and-bound version is the `if` statement on lines 11-12. As each node in the solution space is visited two tests are done: First, the `isFeasible` accessor is called to check whether the given node represents a feasible solution. Next, the `bound` accessor is called to determine the lower bound on the best possible solution in the given subtree. The second test determines whether this bound is less than the value of the objective function of the best solution already found. The recursive call to explore the subtree is only made if both tests succeed. Otherwise, the subtree of the solution space is pruned.

The degree to which the solution space may be pruned depends strongly on the nature of the problem being solved. In the worst case, no subtrees are pruned and the branch-and-bound method visits all the nodes in the solution space. The branch-and-bound technique is really just a *heuristic* --sometimes it works and sometimes it does not.

It is important to understand the trade-off being made: The solution space is being pruned at the added expense of performing the tests as each node is visited. The technique is successful only if the savings which accrue from pruning exceed the additional execution time arising from the tests.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-0/1 Knapsack Problem Again

Consider again the 0/1 knapsack problem described in Section □. We are given a set of n items from which we are to select some number of items to be carried in a knapsack. The solution to the problem has the form $\{x_1, x_2, \dots, x_n\}$, where x_i is one if the i^{th} item is placed in the knapsack and zero otherwise. Each item has both a *weight*, w_i , and a *profit*, p_i . The goal is to maximize the total profit,

$$\sum_{i=1}^n p_i x_i,$$

subject to the knapsack capacity constraint

$$\sum_{i=1}^n w_i x_i \leq C.$$

A partial solution to the problem is one in which only the first k items have been considered. That is, the solution has the form $S_k = \{x_1, x_2, \dots, x_k\}$, where $1 \leq k < n$. The partial solution S_k is feasible if and only if

$$\sum_{i=1}^k w_i x_i \leq C. \quad (14.1)$$

Clearly if S_k is infeasible, then every possible complete solution containing S_k is also infeasible.

If S_k is feasible, the total profit of any solution containing S_k is bounded by

$$\sum_{i=1}^k p_i x_i + \sum_{i=k+1}^n p_i. \quad (14.2)$$

That is, the bound is equal the *actual* profit accrued from the k items already considered plus the sum of the profits of the remaining items.

Clearly, the 0/1 knapsack problem can be solved using a backtracking algorithm. Furthermore, by using Equations □ and □ a branch-and-bound solver can potentially prune the solution space, thereby arriving at the solution more

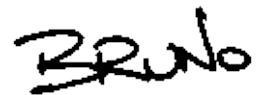
quickly.

For example, consider the 0/1 knapsack problem with $n=6$ items given in Table □. There are $2^n = 64$ possible solutions and the solution space contains $2^{n+1} - 1 = 127$

nodes. The simple DepthFirstSolver given in Program □ visits all 127 nodes and generates all 64 solutions because it does a complete traversal of the solution tree. The BreadthFirstSolver of Program □ behaves similarly. On the other hand, the DepthFirstBranchAndBoundSolver shown in Program □ visits only 67 nodes and generates only 27 complete solutions. In this case, the branch-and-bound technique prunes almost half the nodes from the solution space!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Top-Down Algorithms: Divide-and-Conquer

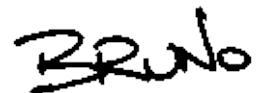
In this section we discuss a top-down algorithmic paradigm called *divide and conquer*. To solve a given problem, it is subdivided into one or more subproblems each of which is similar to the given problem. Each of the subproblems is solved independently. Finally, the solutions to the subproblems are combined in order to obtain the solution to the original problem.

Divide-and-conquer algorithms are often implemented using recursion. However, not all recursive methods are divide-and-conquer algorithms. Generally, the subproblems solved by a divide-and-conquer algorithm are *non-overlapping*.

-
- [Example-Binary Search](#)
 - [Example-Computing Fibonacci Numbers](#)
 - [Example-Merge Sorting](#)
 - [Running Time of Divide-and-Conquer Algorithms](#)
 - [Example-Matrix Multiplication](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

[*Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.*](#)



Example-Binary Search

Consider the problem of finding the position of an item in a sorted list. That is, given the sorted sequence $S = \{a_1, a_2, \dots, a_n\}$ and an item x , find i (if it exists) such that $a_i = x$. The usual solution to this problem is *binary search*.

Binary search is a divide-and-conquer strategy. The sequence S is split into two subsequences, $S_L = \{a_1, a_2, \dots, a_{\lfloor n/2 \rfloor}\}$ and $S_R = \{a_{\lfloor n/2 \rfloor + 1}, a_{\lfloor n/2 \rfloor + 2}, \dots, a_n\}$. The original problem is split into two subproblems: Find x in S_L or S_R . Of course, since the original list is sorted, we can quickly determine the list in which x must appear. Therefore, we only need to solve one subproblem.

Program □ defines the method `binarySearch` which takes four arguments, `array`, `target`, `i` and `n`. This method looks for the position in `array` at which item `target` is found. Specifically, it considers the following elements of the array:

`array[i], array[i + 1], array[i + 2], ..., array[i + n - 1].`

```
1 def binarySearch(array, target, i, n):
2     if n == 0:
3         raise KeyError
4     if n == 1:
5         if array[i] == target:
6             return i
7         raise KeyError
8     else:
9         j = i + n / 2
10        if array[j] <= target:
11            return binarySearch(array, target, j, n - n/2)
12        else:
13            return binarySearch(array, target, i, n/2)
```

Program: Divide-and-conquer example--binary search.

The running time of the algorithm is clearly a function of n , the number of elements to be searched. Although Program □ works correctly for arbitrary values of n , it is much easier to determine the running time if we assume that n is

a power of two. In this case, the running time is given by the recurrence

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ T(n/2) + O(1) & n > 1. \end{cases} \quad (14.3)$$

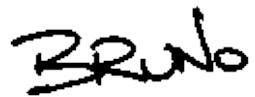
Equation \square is easily solved using repeated substitution:

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= T(n/4) + 2 \\ &= T(n/8) + 3 \\ &\vdots \\ &= T(n/2^k) + k \end{aligned}$$

Setting $n/2^k = 1$ gives $T(n) = \log n + 1 = O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Computing Fibonacci Numbers

The Fibonacci numbers are given by following recurrence

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ F_{n-1} + F_{n-2} & n \geq 2. \end{cases} \quad (14.4)$$

Section □ presents a recursive method to compute the Fibonacci numbers by implementing directly Equation □. (See Program □). The running time of that program is shown to be $T(n) = \Omega((3/2)^n)$.

In this section we present a divide-and-conquer style of algorithm for computing Fibonacci numbers. We make use of the following identities

$$F_{2k-1} = (F_k)^2 + (F_{k-1})^2$$

$$F_{2k} = (F_k)^2 + 2F_k F_{k-1}$$

for $k \geq 1$. (See Exercise □). Thus, we can rewrite Equation □ as

$$F_n = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ (F_{\lceil n/2 \rceil})^2 + (F_{\lceil n/2 \rceil - 1})^2 & n \geq 2 \text{ and } n \text{ is odd,} \\ (F_{\lceil n/2 \rceil})^2 + 2F_{\lceil n/2 \rceil} F_{\lceil n/2 \rceil - 1} & n \geq 2 \text{ and } n \text{ is even.} \end{cases} \quad (14.5)$$

Program □ defines the method `Fibonacci` which implements directly Equation □. Given $n > 1$ it computes F_n by calling itself recursively to compute $F_{\lceil n/2 \rceil}$ and $F_{\lceil n/2 \rceil - 1}$ and then combines the two results as required.

```

1 def Fibonacci(n):
2     if n == 0 or n == 1:
3         return n
4     else:
5         a = Fibonacci((n + 1) / 2)
6         b = Fibonacci((n + 1) / 2 - 1)
7         if n % 2 == 0:
8             return a * (a + 2 * b)
9         else:
10            return a * a + b * b

```

Program: Divide-and-conquer Example--computing Fibonacci numbers.

To determine a bound on the running time of the Fibonacci method in Program \square we assume that $T(n)$ is a non-decreasing function. That is, $T(n) \geq T(n-1)$ for all $n \geq 1$. Therefore $T(\lceil n/2 \rceil) \geq T(\lceil n/2 \rceil - 1)$. Although the program works correctly for all values of n , it is convenient to assume that n is a power of 2. In this case, the running time of the method is upper-bounded by $T(n)$ where

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ 2T(n/2) + O(1) & n > 1. \end{cases} \quad (14.6)$$

Equation \square is easily solved using repeated substitution:

$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 4T(n/4) + 1 + 2 \\ &= 8T(n/8) + 1 + 2 + 4 \\ &\vdots \\ &= 2^k T(n/2^k) + \sum_{i=0}^{k-1} 2^i \\ &\vdots \\ &= nT(1) + n - 1 \quad (n = 2^k). \end{aligned}$$

Thus, $T(n)=2n-1=O(n)$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Example-Merge Sorting

Sorting algorithms and sorters are covered in detail in Chapter □. In this section we consider a divide-and-conquer sorting algorithm--*merge sort*. Given an array of n items in arbitrary order, the objective is to rearrange the elements of the array so that they are ordered from the smallest element to the largest one.

The merge sort algorithm sorts a sequence of length $n > 1$ by splitting it into two subsequences--one of length $\lfloor n/2 \rfloor$, the other of length $\lceil n/2 \rceil$. Each subsequence is sorted and then the two sorted sequences are merged into one.

Program □ defines the method `mergeSort` which takes three arguments, `array`, `i`, and `n`. The method sorts the following n elements:

```
array[i], array[i + 1], array[i + 2], ..., array[i + n - 1].
```

The `mergeSort` method calls itself as well as the `merge` method. The purpose of the `merge` method is to merge two sorted sequences, one of length $\lfloor n/2 \rfloor$, the other of length $\lceil n/2 \rceil$, into a single sorted sequence of length n . This can easily be done in $O(n)$ time. (See Program □).

```
1 def mergeSort(array, i, n):
2     if n > 1:
3         mergeSort(array, i, n / 2)
4         mergeSort(array, i + n / 2, n - n / 2)
5         merge(array, i, n / 2, n - n / 2)
```

Program: Divide-and-conquer example--merge sorting.

The running time of the `mergeSort` method depends on the number of items to be sorted, n . Although Program □ works correctly for arbitrary values of n , it is much easier to determine the running time if we assume that n is a power of two. In this case, the running time is given by the recurrence

$$T(n) = \begin{cases} O(1) & n \leq 1, \\ 2T(n/2) + O(n) & n > 1. \end{cases} \quad (14.7)$$

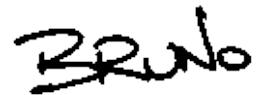
Equation \square is easily solved using repeated substitution:

$$\begin{aligned}T(n) &= 2T(n/2) + n \\&= 4T(n/4) + 2n \\&= 8T(n/8) + 3n \\&\vdots \\&= 2^k T(n/2^k) + kn\end{aligned}$$

Setting $n/2^k = 1$ gives $T(n) = n + n \log n = O(n \log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Running Time of Divide-and-Conquer Algorithms

A number of divide-and-conquer algorithms are presented in the preceding sections. Because these algorithms have a similar form, the recurrences which give the running times of the algorithms are also similar in form. Table □ summarizes the running times of Programs □, □ and □.

Table: Running times of divide-and-conquer algorithms.

program	recurrence	solution
Program □	$T(n)=T(n/2)+O(1)$	$O(\log n)$
Program □	$T(n)=2T(n/2)+O(1)$	$O(n)$
Program □	$T(n)=2T(n/2)+O(n)$	$O(n \log n)$

In this section we develop a general recurrence that characterizes the running times of many divide-and-conquer algorithms. Consider the form of a divide-and-conquer algorithm to solve a given problem. Let n be a measure of the size of the problem. Since the divide-and-conquer paradigm is essentially recursive, there must be a base case. That is, there must be some value of n , say n_0 , for which the solution to the problem is computed directly. We assume that the worst-case running time for the base case is bounded by a constant.

To solve an arbitrarily large problem using divide-and-conquer, the problem is *divided* into a number smaller problems, each of which is solved independently. Let a be the number of smaller problems to be solved ($a \in \mathbb{Z}, a \geq 1$). The size of each of these problems is some fraction of the original problem, typically either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$ ($b \in \mathbb{Z}, b \geq 1$).

The solution to the original problem is constructed from the solutions to the smaller problems. The running time required to do this depends on the problem to be solved. In this section we consider polynomial running times. That is, $O(n^k)$ for some integer $k \geq 0$.

For the assumptions stated above, the running time of a divide-and-conquer algorithm is given by

$$T(n) = \begin{cases} O(1) & n \leq n_0, \\ aT(\lceil n/b \rceil) + O(n^k) & n > n_0. \end{cases} \quad (14.8)$$

In order to make it easier to find the solution to Equation 14.8, we drop the $O(\cdot)$'s as well as the $\lceil \cdot \rceil$ from the recurrence. We can also assume (without loss of generality) that $n_0 = 1$. As a result, the recurrence becomes

$$T(n) = \begin{cases} 1 & n = 1, \\ aT(n/b) + n^k & n > 1. \end{cases}$$

Finally, we assume that n is a power of b . That is, $n = b^m$ for some integer $m \geq 0$. Consequently, the recurrence formula becomes

$$T(b^m) = \begin{cases} 1 & m = 0, \\ T(b^m) = aT(b^{m-1}) + b^{mk} & m > 0. \end{cases} \quad (14.9)$$

We solve Equation 14.9 as follows. Divide both sides of the recurrence by a^m and then *telescope*:

$$\frac{T(b^m)}{a^m} = \frac{T(b^{m-1})}{a^{m-1}} + \left(\frac{b^k}{a}\right)^m \quad (14.10)$$

$$\frac{T(b^{m-1})}{a^{m-1}} = \frac{T(b^{m-2})}{a^{m-2}} + \left(\frac{b^k}{a}\right)^{m-1}$$

$$\frac{T(b^{m-2})}{a^{m-2}} = \frac{T(b^{m-3})}{a^{m-3}} + \left(\frac{b^k}{a}\right)^{m-2}$$

⋮

$$\frac{T(b)}{a} = T(1) + \left(\frac{b^k}{a}\right) \quad (14.11)$$

Adding Equation 14.10 through Equation 14.11, substituting $T(1)=1$ and multiplying both sides by a^m gives

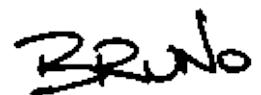
$$T(n) = a^m \sum_{i=0}^m \left(\frac{b^k}{a}\right)^i. \quad (14.12)$$

In order to evaluate the summation in Equation □ we must consider three cases:

- [Case 1 \(\$a > b^k\$ \)](#)
 - [Case 2 \(\$a = b^k\$ \)](#)
 - [Case 3 \(\$a < b^k\$ \)](#)
 - [Summary](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Case 1 ($a > b^k$)

In this case, the term b^k/a falls between zero and one. Consider the *infinite* geometric series summation:

$$\sum_{i=0}^{\infty} \left(\frac{b^k}{a}\right)^i = \frac{a}{a - b^k} = C$$

Since the infinite series summation approaches a finite constant C and since each term in the series is positive, the *finite* series summation in Equation □ is bounded from above by C :

$$\sum_{i=0}^m \left(\frac{b^k}{a}\right)^i \leq C$$

Substituting this result into Equation □ and making use of the fact that $n = b^m$, and therefore $m = \log_b n$, gives

$$\begin{aligned} T(n) &\leq Ca^m \\ &= O(a^m) \\ &= O(a^{\log_b n}) \\ &= O(a^{\log_a n \log_b a}) \\ &= O(n^{\log_b a}). \end{aligned}$$



[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Case 2 ($a = b^k$)

In this case the term $\frac{b^k}{a}$ is exactly one. Therefore, the series summation in Equation □ is simply

$$\sum_{i=0}^m \left(\frac{b^k}{a}\right)^i = m + 1.$$

Substituting this result into Equation □ and making use of the fact that $n = b^m$ and $a = b^k$ gives

$$\begin{aligned} T(n) &= (m+1)a^m \\ &= O(ma^m) \\ &= O(m(b^k)^m) \\ &= O((b^m)^k m) \\ &= O(n^k \log_b n). \end{aligned}$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Case 3 ($a < b^k$)

In this case the term b^k/a is greater than one and we make use of the general formula for a finite geometric series summation (see Section □) to evaluate the summation:

$$\sum_{i=0}^m \left(\frac{b^k}{a}\right)^i = \frac{(b^k/a)^{m+1} - 1}{b^k/a - 1}.$$

Substituting this result in Equation □ and simplifying gives:

$$\begin{aligned} T(n) &= a^m \left(\frac{(b^k/a)^{m+1} - 1}{b^k/a - 1} \right) \\ &= a^m \left(\frac{(b^k/a)^m - a/b^k}{1 - a/b^k} \right) \\ &= O(a^m(b^k/a)^m) \\ &= O(b^{km}) \\ &= O(n^k) \end{aligned}$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Summary

For many divide-and-conquer algorithms the running time is given by the general recurrence shown in Equation □. Solutions to the recurrence depend on the relative values of the constants a , b , and k . Specifically, the solutions satisfy the following bounds:

$$T(n) = \begin{cases} O(n^{\log_b a}) & a > b^k, \\ O(n^k \log_b n) & a = b^k, \\ O(n^k) & a < b^k. \end{cases} \quad (14.13)$$

Table □ shows how to apply Equation □ to find the running times of the divide-and-conquer algorithms described in the preceding sections. Comparing the solutions in Table □ with those given in Table □ shows the results obtained using the general formula agree with the analyses done in the preceding sections.

Table: Computing running times using Equation □

program	recurrence	a	b	k	case	solution
Program □	$T(n)=T(n/2)+O(1)$	1	2	0	$a = b^k$	$O(n^0 \log_2 n)$
Program □	$T(n)=2T(n/2)+O(1)$	2	2	0	$a > b^k$	$O(n^{\log_2 2})$
Program □	$T(n)=2T(n/2)+O(n)$	2	2	1	$a = b^k$	$O(n^1 \log_2 n)$

Example-Matrix Multiplication

Consider the problem of computing the product of two matrices. That is, given two $n \times n$ matrices, A and B , compute the $n \times n$ matrix $C = A \times B$, the elements of which are given by

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}. \quad (14.14)$$

Section \square shows that the direct implementation of Equation \square results in an $O(n^3)$ running time. In this section we show that the use of a divide-and-conquer strategy results in a slightly better asymptotic running time.

To implement a divide-and-conquer algorithm we must break the given problem into several subproblems that are similar to the original one. In this instance we view each of the $n \times n$ matrices as a 2×2 matrix, the elements of which are $(\frac{n}{2}) \times (\frac{n}{2})$ submatrices. Thus, the original matrix multiplication, $C = A \times B$, can be written as

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix},$$

where each $A_{i,j}$, $B_{i,j}$, and $C_{i,j}$ is an $(\frac{n}{2}) \times (\frac{n}{2})$ matrix.

From Equation \square we get that the result submatrices can be computed as follows:

$$\begin{aligned} C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} \\ C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ C_{2,1} &= A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} \\ C_{2,2} &= A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2}. \end{aligned}$$

Here the symbols $+$ and \times are taken to mean addition and multiplication (respectively) of $(\frac{n}{2}) \times (\frac{n}{2})$ matrices.

In order to compute the original $n \times n$ matrix multiplication we must compute

eight $(\frac{n}{2}) \times (\frac{n}{2})$ matrix products (*divide*) followed by four $(\frac{n}{2}) \times (\frac{n}{2})$ matrix sums (*conquer*). Since matrix addition is an $O(n^2)$ operation, the total running time for the multiplication operation is given by the recurrence:

$$T(n) = \begin{cases} O(1) & n = 1, \\ 8T(n/2) + O(n^2) & n > 1. \end{cases} \quad (14.15)$$

Note that Equation \square is an instance of the general recurrence given in Equation \square . In this case, $a=8$, $b=2$, and $k=2$. We can obtain the solution directly from Equation \square . Since $a > b^k$, the total running time is $O(n^{\log_2 a}) = O(n^{\log_2 8}) = O(n^3)$. But this is no better than the original, direct algorithm!

Fortunately, it turns out that one of the eight matrix multiplications is redundant. Consider the following series of seven $(\frac{n}{2}) \times (\frac{n}{2})$ matrices:

$$\begin{aligned} M_0 &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\ M_1 &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2}) \\ M_2 &= (A_{1,1} - A_{2,1}) \times (B_{1,1} + B_{1,2}) \\ M_3 &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\ M_4 &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\ M_5 &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\ M_6 &= (A_{2,1} + A_{2,2}) \times B_{1,1} \end{aligned}$$

Each equation above has only one multiplication. Ten additions and seven multiplications are required to compute M_0 through M_6 . Given M_0 through M_6 , we can compute the elements of the product matrix C as follows:

$$\begin{aligned} C_{1,1} &= M_0 + M_1 - M_3 + M_5 \\ C_{1,2} &= M_3 + M_4 \\ C_{2,1} &= M_5 + M_6 \\ C_{2,2} &= M_0 - M_2 + M_4 - M_6 \end{aligned}$$

Altogether this approach requires seven $(\frac{n}{2}) \times (\frac{n}{2})$ matrix multiplications and 18 $(\frac{n}{2}) \times (\frac{n}{2})$ additions. Therefore, the worst-case running time is given by the following recurrence:

$$T(n) = \begin{cases} O(1) & n = 1, \\ 7T(n/2) + O(n^2) & n > 1. \end{cases} \quad (14.16)$$

As above, Equation 14.16 is an instance of the general recurrence given in Equation 14.15, and we obtain the solution directly from Equation 14.15. In this case, $a=7$, $b=2$, and $k=2$. Therefore, $a > b^k$ and the total running time is

$$O(n^{\log_2 7}) = O(n^{2.807355}).$$

Note $\log_2 7 \approx 2.807355$. Consequently, the running time of the divide-and-conquer matrix multiplication strategy is $O(n^{2.8})$ which is better (asymptotically) than the straightforward $O(n^3)$ approach.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

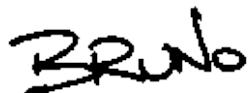
Bottom-Up Algorithms: Dynamic Programming

In this section we consider a bottom-up algorithmic paradigm called *dynamic programming*. In order to solve a given problem, a series of subproblems is solved. The series of subproblems is devised carefully in such a way that each subsequent solution is obtained by combining the solutions to one or more of the subproblems that have already been solved. All intermediate solutions are kept in a table in order to prevent unnecessary duplication of effort.

- [Example-Generalized Fibonacci Numbers](#)
 - [Example-Computing Binomial Coefficients](#)
 - [Application: Typesetting Problem](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Generalized Fibonacci Numbers

Consider the problem of computing the *generalized Fibonacci numbers*. The generalized Fibonacci numbers of order $k \geq 2$ are given by

$$F_n^{(k)} = \begin{cases} 0 & 0 \leq n < k - 1, \\ 1 & n = k - 1, \\ \sum_{i=1}^k F_{n-i}^{(k)} & n \geq k. \end{cases} \quad (14.17)$$

Notice that the ``normal'' Fibonacci numbers considered in Section □ are the same as the generalized Fibonacci numbers of order 2.

If we write a recursive method that implements directly Equation □, we get an algorithm with exponential running time. For example, in Section □ it is shown that the time to compute the second-order Fibonacci numbers is $T(n) = \Omega((3/2)^n)$.

The problem with the direct recursive implementation is that it does far more work than is needed because it solves the same subproblem many times. For example, to compute $F_{10}^{(2)}$ it is necessary to compute both $F_9^{(2)}$ and $F_8^{(2)}$. However, in computing $F_9^{(2)}$ it is also necessary to compute $F_8^{(2)}$, and so on.

An alternative to the top-down recursive implementation is to do the calculation from the bottom up. In order to do this we compute the series of sequences

$$\begin{aligned} S_0 &= \{F_0^{(k)}\} \\ S_1 &= \{F_0^{(k)}, F_1^{(k)}\} \\ &\vdots \\ S_n &= \{F_0^{(k)}, F_1^{(k)}, \dots, F_n^{(k)}\}. \end{aligned}$$

Notice that we can compute S_{i+1} from the information contained in S_i simply by using Equation □.

Program □ defines the method `Fibonacci` which takes two integer arguments n and k and computes the n^{th} Fibonacci number of order k using the approach described above. This algorithm uses an array to represent the series of

sequences S_0, S_1, \dots, S_n . As each subsequent Fibonacci number is computed it is added to the end of the array.

```
1 def Fibonacci(n, k):
2     if n < k - 1:
3         return 0
4     elif n == k - 1:
5         return 1
6     else:
7         f = [0] * (n + 1)
8         for i in xrange(k - 1):
9             f[i] = 0
10            f[k - 1] = 1
11            for i in xrange(k, n + 1):
12                sum = 0
13                for j in xrange(k + 1):
14                    sum += f[i - j]
15                f[i] = sum
16            return f[n]
```

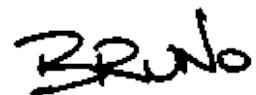
Program: Dynamic programming example--computing generalized Fibonacci numbers.

The worst-case running time of the `Fibonacci` method given in Program □ is a function of both n and k :

$$T(n, k) = \begin{cases} O(1) & 0 \leq n < k, \\ O(kn) & n \geq k. \end{cases}$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Computing Binomial Coefficients

Consider the problem of computing the *binomial coefficient*

$$\binom{n}{m} = \frac{n!}{(n-m)!m!} \quad (14.18)$$

given non-negative integers n and m (see Theorem □).

The problem with implementing directly Equation □ is that the factorials grow quickly with increasing n and m . For example, $13! = 6\,227\,020\,800 > 2^{31}$. Therefore, it is not possible to represent $n!$ for $n \geq 13$ using 32-bit integers. Nevertheless it is possible to represent the binomial coefficients $\binom{n}{m}$ up to $n=33$ without overflowing. For example, $\binom{33}{16} = 1\,166\,803\,110 < 2^{31}$.

Consider the following *recursive* definition of the binomial coefficients:

$$\binom{n}{m} = \begin{cases} 1 & m = 0, \\ 1 & n = m, \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{otherwise.} \end{cases} \quad (14.19)$$

This formulation does not require the computation of factorials. In fact, the only computation needed is addition.

If we implement Equation □ directly as a recursive method, we get a method whose running time is given by

$$T(n, m) = \begin{cases} O(1) & m = 0, \\ O(1) & n = m, \\ T(n-1, m) + T(n-1, m-1) + O(1) & \text{otherwise.} \end{cases}$$

which is very similar to Equation □. In fact, we can show that $T(n, m) = \Omega(\binom{n}{m})$ which (by Equation □) is not a very good running time at all! Again the problem with the direct recursive implementation is that it does far more work than is

needed because it solves the same subproblem many times.

An alternative to the top-down recursive implementation is to do the calculation from the bottom up. In order to do this we compute the series of sequences

$$\begin{aligned} S_0 &= \left\{ \binom{0}{0} \right\} \\ S_1 &= \left\{ \binom{1}{0}, \binom{1}{1} \right\} \\ S_2 &= \left\{ \binom{2}{0}, \binom{2}{1}, \binom{2}{2} \right\} \\ &\vdots \\ S_n &= \left\{ \binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \dots, \binom{n}{n} \right\}. \end{aligned}$$

Notice that we can compute S_{i+1} from the information contained in S_i simply by using Equation □. Table □ shows the sequence in tabular form--the i^{th} row of the table corresponds to the sequence S_i . This tabular representation of the binomial coefficients is known as *Pascal's triangle*. ◇

Table: Pascal's triangle.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Program □ defines the method `binom` which takes two integer arguments n and m and computes the binomial coefficient $\binom{n}{m}$ by computing Pascal's triangle. According to Equation □, each subsequent row depends only on the preceding row--it is only necessary to keep track of one row of data. The implementation shown uses an array of length n to represent a row of Pascal's triangle.

Consequently, instead of a table of size $O(n^2)$, the algorithm gets by with $O(n)$ space. The implementation has been coded carefully so that the computation can be done in place. That is, the elements of S_{i+1} are computed in reverse so that they can be written over the elements of S_i that are no longer needed.

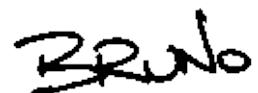
```
1 def binom(n, m):
2     b = [0] * (n + 1)
3     b[0] = 1
4     for i in xrange(1, n + 1):
5         b[i] = 1
6         j = i - 1
7         while j > 0:
8             b[j] += b[j - 1]
9             j -= 1
10    return b[m]
```

Program: Dynamic programming example--computing Binomial coefficients.

The worst-case running time of the `binom` method given in Program □ is clearly $O(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Application: Typesetting Problem

Consider the problem of typesetting a paragraph of justified text. A paragraph can be viewed as a sequence of $n > 0$ words, $\{w_1, w_2, \dots, w_n\}$. The objective is to determine how to break the sequence into individual lines of text of the appropriate size. Each word is separated from the next by some amount of space. By stretching or compressing the space between the words, the left and right ends of consecutive lines of text are made to line up. A paragraph looks best when the amount of stretching or compressing is minimized.

We can formulate the problem as follows: Assume that we are given the lengths of the words, $\{l_1, l_2, \dots, l_n\}$, and that the desired length of a line is D . Let $W_{i,j}$ represent the sequence of words from w_i to w_j (inclusive). That is,

$$W_{i,j} = \{w_i, w_{i+1}, \dots, w_j\},$$

for $1 \leq i \leq j \leq n$.

Let $L_{i,j}$ be the sum of the lengths of the words in the sequence $W_{i,j}$. That is,

$$L_{i,j} = \sum_{k=i}^j l_k.$$

The *natural length*, for the sequence $W_{i,j}$ is the sum of the lengths of the words, $L_{i,j}$ plus the normal amount of space between those words. Let s be the normal size of the space between two words. Then the natural length of $W_{i,j}$ is $L_{i,j} + (j - i)s$. Note, we can also define $L_{i,j}$ recursively as follows:

$$L_{i,j} = \begin{cases} l_i & i = j, \\ L_{i,j-1} + l_j & i < j. \end{cases} \quad (14.20)$$

In general, when we typeset the sequence $W_{i,j}$ all on a single line, we need to stretch or compress the spaces between the words so that the length of the line is the desired length D . Therefore, the amount of stretching or compressing is

given by the difference $D - (L_{i,j} + (j-i)s)$. However, if the sum of the lengths of the words, $L_{i,j}$, is longer than the desired line length D , it is not possible to typeset the sequence on a single line.

Let $P_{i,j}$ be the *penalty* associated with typesetting the sequence $L_{i,j}$ on a single line. Then,

$$P_{i,j} = \begin{cases} |D - L_{i,j} - (j-i)s| & D \geq L_{i,j}, \\ \infty & D < L_{i,j}. \end{cases} \quad (14.21)$$

This definition of penalty is consistent with the stated objectives: The penalty increases as the difference between the natural length of the sequence and the desired length increases and the infinite penalty disallows lines that are too long.

Finally, we define the quantity $C_{i,j}$ for $1 \leq i \leq j \leq n$ as the minimum total penalty required to typeset the sequence $W_{i,j}$. In this case, the text may be all on one line or it may be split over more than one line. The quantity $C_{i,j}$ is given by

$$C_{i,j} = \begin{cases} P_{i,j} & i = j, \\ \min \{ P_{i,j}, \min_{i \leq k < j} (P_{i,k} + C_{k+1,j}) \} & \text{otherwise.} \end{cases} \quad (14.22)$$

We obtain Equation \square as follows: When $i=j$ there is only one word in the paragraph. The minimum total penalty associated with typesetting the paragraph in this case is just the penalty which results from putting the one word on a single line.

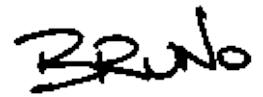
In the general case, there is more than one word in the sequence $W_{i,j}$. In order to determine the optimal way in which to typeset the paragraph we consider the cost of putting the first k words of the sequence on the first line of the paragraph, $P_{i,k}$, plus the minimum total cost associated with typesetting the rest of the paragraph $C_{k+1,j}$. The value of k which minimizes the total cost also specifies where the line break should occur.

- [Example](#)

- [Implementation](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Example

Suppose we are given a sequence of $n=5$ words, $W = \{w_1, w_2, w_3, w_4, w_5\}$ having lengths $\{10, 10, 10, 12, 50\}$, respectively, which are to be typeset in a paragraph of width $D=60$. Assume that the normal width of an inter-word space is $s=10$.

We begin by computing the lengths of all the subsequences of W using Equation □. The lengths of all $n(n-1)/2$ subsequences of W are tabulated in Table □.

Table:

Typesetting
problem.

i	l_i	$j=1$	2	3	4	5
1	10	10	20	30	42	92
2	10	10	20	32	82	
3	10	10	22	72		
4	12	12	62			
5	50		50			

Given $L_{i,j}$, D , and s , it is a simple matter to apply *Equation 14.21* to obtain the one-line penalties, $P_{i,j}$, which measure the amount of stretching or compressing needed to set all the words in a given subsequence on a single line. These are tabulated in Table □.

Table: Penalties.

$P_{i,j}$	$C_{i,j}$
$i \ j=1 \ 2 \ 3 \ 4 \ 5$	$j=1 \ 2 \ 3 \ 4 \ 5$
1 50 30 10 12 ∞	50 30 10 12 22
2 50 30 8 ∞	50 30 8 18
3 50 28 ∞	50 28 38
4 48 ∞	48 58
5 10	10

Given the one-line penalties $P_{i,j}$, we can use Equation \square to find for each subsequence of W the minimum total penalty, $C_{i,j}$, associated with forming a paragraph from the words in that subsequence. These are tabulated in Table \square .

The $C_{1,5}$ entry in Table \square gives the minimum total cost of typesetting the entire paragraph. The value 22 was obtained as follows:

$$\begin{aligned} C_{1,5} &= \min \{P_{1,1} + C_{2,5}, P_{1,2} + C_{3,5}, P_{1,3} + C_{4,5}, P_{1,4} + C_{5,5}, P_{1,5}\} \\ &= P_{1,4} + C_{5,5} \\ &= 12 + 10. \end{aligned}$$

This indicates that the optimal solution is to set words w_1 , w_2 , w_3 , and w_4 on the first line of the paragraph and leave w_5 by itself on the last line of the paragraph. Figure \square illustrates this result.

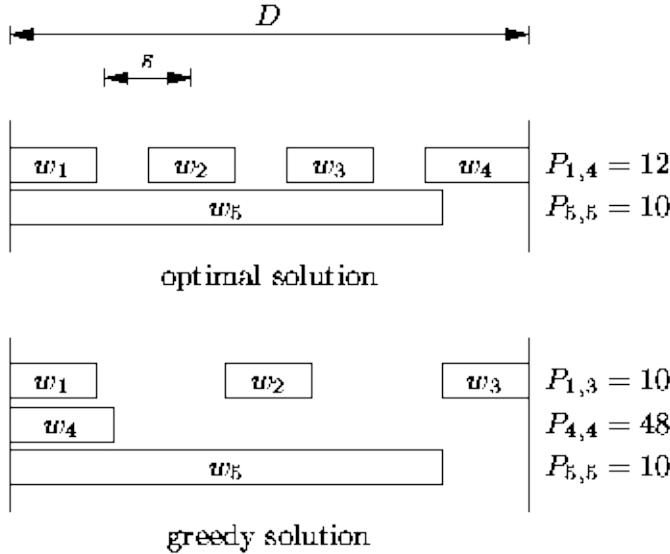


Figure: Typesetting a paragraph.

This formulation of the typesetting problem seems like overkill. Why not just typeset the lines of text one-by-one, minimizing the penalty for each line as we go? In other words why don't we just use a greedy strategy? Unfortunately, the obvious greedy solution strategy *does not work!*

For example, the greedy strategy begins by setting the first line of text. To do so it must decide how many words to put on that line. The obvious thing to do is to select the value of k for which $P_{1,k}$ is the smallest. From Table \square we see that $P_{1,3} = 10$ has the smallest penalty. Therefore, the greedy approach puts three words on the first line as shown in Figure \square .

Since the remaining two words do not both fit on a single line, they are set on separate lines. The total of the penalties for the paragraph typeset using the greedy algorithm is $P_{1,3} + P_{4,4} + P_{5,5} = 68$. Clearly, the solution is not optimal (nor is it very pleasing esthetically).



Implementation

Program □ defines the method typeset which takes three arguments. The first, l, is an array of n integers that gives the lengths of the words in the sequence to be typeset. The second, D, specifies the desired paragraph width and the third, s, specifies the normal inter-word space.

```

1 def typeset(l, D, s):
2     n = len(l)
3     L = DenseMatrix(n, n)
4     for i in xrange(n):
5         L[i, i] = l[i]
6         for j in xrange(i + 1, n):
7             L[i, j] = L[i, j - 1] + l[j]
8     P = DenseMatrix(n, n)
9     for i in xrange(n):
10        for j in xrange(i, n):
11            if L[i, j] < D:
12                P[i, j] = abs(D - L[i, j] - (j - i) * s)
13            else:
14                P[i, j] = sys.maxint
15     c = DenseMatrix(n, n)
16     for j in xrange(n):
17         c[j, j] = P[j, j]
18         i = j - 1
19         while i >= 0:
20             min = P[i, j]
21             for k in xrange(i, j):
22                 tmp = P[i, k] + c[k + 1, j]
23                 if tmp < min:
24                     min = tmp
25             c[i, j] = min
26             i -= 1

```

Program: Dynamic programming example--typesetting a paragraph.

The method first computes the lengths, $L_{i,j}$, of all possible subsequences (lines 3-7). This is done by using the dynamic programming paradigm to evaluate the recursive definition of $L_{i,j}$ given in Equation □. The running time for this

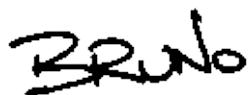
computation is clearly $O(n^2)$.

The next step computes the one-line penalties $P_{i,j}$ as given by Equation \square (lines 8-14). This calculation is a straightforward one and its running time is also $O(n^2)$.

Finally, the minimum total costs, $C_{i,j}$, of typesetting each subsequence are determined for all possible subsequences (lines 15-26). Again we make use of the dynamic programming paradigm to evaluate the recursive definition of $C_{i,j}$ given in Equation \square . The running time for this computation is $O(n^3)$. As a result, the overall running time required to determine the best way to typeset a paragraph of n words is $O(n^3)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Randomized Algorithms

In this section we discuss algorithms that behave randomly. By this we mean that there is an element of randomness in the way that the algorithm solves a given problem. Of course, if an algorithm is to be of any use, it must find a solution to the problem at hand, so it cannot really be completely random.

Randomized algorithms are said to be methods of last resort. This is because they are used often when no other feasible solution technique is known. For example, randomized methods are used to solve problems for which no closed-form, analytic solution is known. They are also used to solve problems for which the solution space is so large that an exhaustive search is infeasible.

To implement a randomized algorithm we require a source of randomness. The usual source of randomness is a random number generator. Therefore, before presenting randomized algorithms, we first consider the problem of computing random numbers.

-
- [Generating Random Numbers](#)
 - [Random Variables](#)
 - [Monte Carlo Methods](#)
 - [Simulated Annealing](#)
-

Generating Random Numbers

In this section we consider the problem of generating a sequence of *random numbers* on a computer. Specifically, we desire an infinite sequence of statistically independent random numbers uniformly distributed between zero and one. In practice, because the sequence is generated algorithmically using finite-precision arithmetic, it is neither infinite nor truly random. Instead, we say that an algorithm is ``good enough'' if the sequence it generates satisfies almost any statistical test of randomness. Such a sequence is said to be *pseudorandom*.

The most common algorithms for generating pseudorandom numbers are based on the *linear congruential* random number generator invented by Lehmer. Given a positive integer m called the *modulus* and an initial *seed* value X_0 ($0 \leq X_0 < m$), Lehmer's algorithm computes a sequence of integers between 0 and $m-1$. The elements of the sequence are given by

$$X_{i+1} = (aX_i + c) \bmod m, \quad (14.23)$$

where a and c are carefully chosen integers such that $2 \leq a < m$ and $0 \leq c < m$.

For example, the parameters $a=13$, $c=1$, $m=16$, and $X_0=0$ produce the sequence

$$0, 1, 14, 7, 12, 13, 10, 3, 8, 9, 6, 15, 4, 5, 2, 11, 0, \dots$$

The first m elements of this sequence are distinct and appear to have been drawn at random from the set $\{0, 1, 2, \dots, 15\}$. However since $X_m = X_0$ the sequence is cyclic with *period* m .

Notice that the elements of the sequence alternate between odd and even integers. This follows directly from Equation □ and the fact that $m=16$ is a multiple of 2. Similar patterns arise when we consider the elements as binary numbers:

$$0000, 0001, 1110, 0111, 1100, 1101, 1010, 0011, 1000, \dots$$

The least significant two bits are cyclic with period four and the least significant

three bits are cycle with period eight! (These patterns arise because $m=16$ is also a multiple of 4 and 8). The existence of such patterns make the sequence *less random*. This suggests that the best choice for the modulus m is a prime number.

Not all parameter values result in a period of m . For example, changing the multiplier a to 11 produces the sequence

$$0, 1, 12, 5, 8, 9, 4, 13, 0, \dots$$

the period of which is only $m/2$. In general because each subsequent element of the sequence is determined solely from its predecessor and because there are m possible values, the longest possible period is m . Such a generator is called a *full period* generator.

In practice the *increment* c is often set to zero. In this case, Equation □ becomes

$$X_{i+1} = aX_i \bmod m. \quad (14.24)$$

This is called a *multiplicative linear congruential* random number generator. (For $c \neq 0$ it is called a *mixed linear congruential* generator).

In order to prevent the sequence generated by Equation □ from collapsing to zero, the modulus m must be prime and X_0 cannot be zero. For example, the parameters $a=6$, $m=13$, and $X_0=1$ produce the sequence

$$1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, \dots$$

Notice that the first 12 elements of the sequence are distinct. Since a multiplicative congruential generator can never produce a zero, the maximum possible period is $m-1$. Therefore, this is a full period generator.

As the final step of the process, the elements of the sequence are *normalized* by division by the modulus:

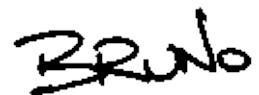
$$U_i = X_i/m.$$

In so doing, we obtain a sequence of random numbers that fall between zero and one. Specifically, a mixed congruential generator ($c \neq 0$) produces numbers in the interval $[0,1]$, whereas a multiplicative congruential generator ($c=0$) produces numbers in the interval $(0,1)$.

-
- [The Minimal Standard Random Number Generator](#)
 - [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





The Minimal Standard Random Number Generator

A great deal of research has gone into the question of finding an appropriate set of parameters to use in Lehmer's algorithm. A good generator has the following characteristics:

- It is a *full period* generator.
- The generated sequence passes statistical tests of *randomness*.
- The generator can be implemented efficiently using 32-bit integer arithmetic.

The choice of modulus depends on the arithmetic precision used to implement the algorithm. A signed 32-bit integer can represent values between -2^{31} and $2^{31} - 1 = 2\,147\,483\,647$. Fortunately, the quantity is a prime number! ◇
Therefore, it is an excellent choice for the modulus m .

Because Equation □ is slightly simpler than Equation □, we choose to implement a multiplicative congruential generator ($c=0$). The choice of a suitable multiplier is more difficult. However, a popular choice is $a = 16807$ because it satisfies all three criteria given above: It results in a full period random number generator; the generated sequence passes a wide variety of statistical tests for randomness; and it is possible to compute Equation □ using 32-bit arithmetic without overflow.

The algorithm is derived as follows: First, let $q = m \text{ div } a$ and $r = m \text{ mod } a$. ◇ In this case, $q = 127773$, $r = 2836$, and $r < q$.

Next, we rewrite Equation □ as follows:

$$\begin{aligned} X_{i+1} &= aX_i \text{ mod } m \\ &= aX_i - m(aX_i \text{ div } m) \\ &= aX_i - m(X_i \text{ div } q) + m(X_i \text{ div } q - aX_i \text{ div } m). \end{aligned}$$

This somewhat complicated formula can be simplified if we let

$$\delta(X_i) = X_i \text{ div } q - aX_i \text{ div } m.$$

$$\begin{aligned} X_{i+1} &= aX_i - m(X_i \text{ div } q) + m\delta(X_i) \\ &= a(q(X_i \text{ div } q) + X_i \bmod q) - m(X_i \text{ div } q) + m\delta(X_i) \\ &= a(X_i \bmod q) + (aq - m)(X_i \text{ div } q) + m\delta(X_i) \end{aligned}$$

Finally, we make use of the fact that $m=aq-r$ to get

$$X_{i+1} = a(X_i \bmod q) - r(X_i \text{ div } q) + m\delta(X_i). \quad (14.25)$$

Equation 14.25 has several nice properties: Both $a(X_i \bmod q)$ and $r(X_i \text{ div } q)$ are positive integers between 0 and $m-1$. Therefore the difference $(X_i \bmod q) - r(X_i \text{ div } q)$ can be represented using a signed 32-bit integer without overflow. Finally, $\delta(X_i)$ is either a zero or a one. Specifically, it is zero when the sum of the first two terms in Equation 14.25 is positive and it is one when the sum is negative. As a result, it is not necessary to compute $\delta(X_i)$ --a simple test suffices to determine whether the third term is 0 or m .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

We now describe the implementation of a random number generator based on Equation 2. Program 2 defines the RandomNumberGenerator class.

```
 1  class RandomNumberGenerator(object):
 2
 3      a = 16807
 4      m = 2147483647
 5      q = 127773
 6      r = 2836
 7
 8      def __init__(self, seed=1):
 9          super(RandomNumberGenerator, self).__init__()
10          assert seed >= 1 and seed < self.m
11          self._seed = seed
12
13      def getSeed(self):
14          return self._seed
15
16      def setSeed(self, seed):
17          assert seed >= 1 and seed < self.m
18          self._seed = seed
19
20      seed = property(
21          fget = lambda self: self.getSeed(),
22          fset = lambda self, value: self.setSeed(value))
23
24      def getNext(self):
25          self._seed = self.a * (self._seed % self.q) \
26              - self.r * (self._seed / self.q)
27          if self._seed < 0:
28              self._seed += self.m
29          return (1.0 * self._seed) / self.m
30
31      next = property(
32          fget = lambda self: self.getNext())
33
34 RandomNumberGenerator = RandomNumberGenerator() # singleton
```

Program: RandomNumberGenerator class `__init__` method, and `__seed__` and

next properties.

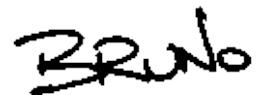
The seed property is implemented using the getSeed and setSeed methods. The setSeed method is used to specify the initial seed, X_0 . The seed must fall between 0 and $m-1$. The getSeed method returns the current seed value.

The next property is implemented using the getNext method. the getNext method generates the elements of the random sequence. Each subsequent call to getNext returns the next element of the sequence. The implementation follows directly from Equation □. Notice that the return value is normalized. Therefore, the values computed by the getNext accessor are uniformly distributed on the interval (0,1).

The assignment statement on line 34 of Program □ is interesting. It creates an *instance* of the RandomNumberGenerator class and then names that instance RandomNumberGenerator. This has the effect of hiding the class definition and thereby preventing the creation of any additional instances of the class. Because there can only be one instance of the RandomNumberGenerator class, this is an example of the *singleton* design pattern.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Random Variables

In this section we introduce the notion of an abstract *random variable*. In this context, a random variable is an object that behaves like a random number generator in that it produces a pseudorandom number sequence. The distribution of the values produced depends on the class of random variable used.

Program □ defines the `RandomVariable` class. The abstract `RandomVariable` class extends the abstract `Object` class introduced in Program □. The `RandomVariable` class defines the property `next`. Given an instance, say `rv`, of a class derived from the `RandomVariable` class, repeated accesses to the `next` property are expected to return successive elements of a pseudorandom sequence.

```
 1  class RandomVariable(Object):
 2
 3      def __init__(self):
 4          super(RandomVariable, self).__init__()
 5
 6      def getNext(self):
 7          pass
 8      getNext = abstractmethod(getNext)
 9
10      next = property(
11          fget = lambda self: self.getNext())
```

Program: Abstract `RandomVariable` class.

- [A Simple Random Variable](#)
 - [Uniformly Distributed Random Variables](#)
 - [Exponentially Distributed Random Variables](#)
-



A Simple Random Variable

Program □ defines the `SimpleRV` class. The `SimpleRV` class extends the abstract `RandomVariable` class defined in Program □. This class generates random numbers uniformly distributed in the interval (0,1).

```
1 class SimpleRV(RandomVariable):
2
3     def getNext(self):
4         return RandomNumberGenerator.next
```

Program: `SimpleRV` class.

The implementation of the `SimpleRV` class is trivial because the `RandomNumberGenerator` class generates the desired distribution of random numbers. Consequently, the `next` accessor simply calls `RandomNumberGenerator.next`.



[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Uniformly Distributed Random Variables

Program □ defines the UniformRV class. This class generates random numbers which are uniformly distributed in an arbitrary interval (u,v) , where $u < v$. The parameters u and v are specified in the `__init__` method.

```
1  class UniformRV(RandomVariable):
2
3      def __init__(self, u, v):
4          self._u = u;
5          self._v = v;
6
7      def getNext(self):
8          return self._u + (self._v - self._u) * \
9              RandomNumberGenerator.next
```

Program: UniformRV class.

The UniformRV class is also quite simple. Given that the RandomNumberGenerator class generates a sequence random numbers U_i uniformly distributed on the interval $(0,1)$, the linear transformation

$$V_i = u + (v - u)U_i$$

suffices to produce a sequence of random numbers V_i uniformly distributed on the interval (u,v) .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exponentially Distributed Random Variables

Program □ defines the `ExponentialRV` class. This class generates exponentially distributed random numbers with a mean value of μ . The mean value μ is specified in the `__init__` method.

```

1  class ExponentialRV(RandomVariable):
2
3      def __init__(self, mu):
4          self._mu = mu
5
6      def getNext(self):
7          return -self._mu * math.log(RandomNumberGenerator.next);

```

Program: `ExponentialRV` class.

The `ExponentialRV` class generates a sequence of random numbers, X_i , *exponentially distributed* on the interval $(0, \infty)$ and having a mean value μ . The numbers are said to be *exponentially distributed* because the probability that X_i falls between 0 and z is given by

$$P[0 < X_i < z] = \int_0^z p(x)dx,$$

where $p(x) = \frac{1}{\mu} e^{-x/\mu}$. The function $p(x)$ is called the *probability density function*. Thus,

$$\begin{aligned} P[0 < X_i < z] &= \int_0^z \frac{1}{\mu} e^{-x/\mu} dx \\ &= 1 - e^{-z/\mu}. \end{aligned}$$

Notice that $P[0 < X_i < z]$ is a value between zero and one. Therefore, given a random variable, U_i , uniformly distributed between zero and one, we can obtain an exponentially distributed variable X_i as follows:

$$\begin{aligned} U_i = 1 - e^{X_i/\mu} &\Rightarrow X_i = -\mu \ln(U_i - 1) \\ &= -\mu \ln(U'_i), \quad U'_i = U_i - 1 \end{aligned} \quad (14.26)$$

Note, if U_i is uniformly distributed on $(0,1)$, then so too is U'_i . The implementation of the next method follows directly from Equation \square .

Monte Carlo Methods

In this section we consider a method for solving problems using random numbers. The method exploits the statistical properties of random numbers in order to ensure that the correct result is computed in the same way that a gambling casino sets the betting odds in order to ensure that the ``house'' will always make a profit. For this reason, the problem solving technique is called a *Monte Carlo method*.

To solve a given problem using a Monte Carlo method we devise an experiment in such a way that the solution to the original problem can be obtained from the experimental results. The experiment typically consists of a series of random trials. A random number generator such as the one given in the preceding section is used to create the series of trials.

The accuracy of the final result usually depends on the number of trials conducted. That is, the accuracy usually increases with the number of trials. This trade-off between the accuracy of the result and the time taken to compute it is an extremely useful characteristic of Monte Carlo methods. If only an approximate solution is required, then a Monte Carlo method can be very fast.

- [Example-Computing \$\pi\$](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Example-Computing π

This section presents a simple, Monte Carlo algorithm to compute the value of π from a sequence of random numbers. Consider a square positioned in the x - y plane with its bottom left corner at the origin as shown in Figure □. The area of the square is r^2 , where r is the length of its sides. A quarter circle is inscribed within the square. Its radius is r and its center is at the origin of x - y plane. The area of the quarter circle is $\frac{\pi r^2}{4}$.

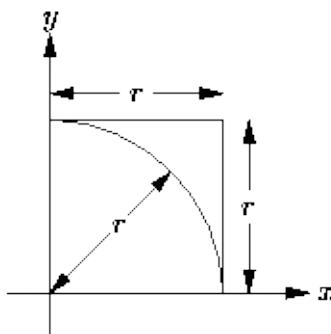


Figure: Illustration of a Monte Carlo method for computing π .

Suppose we select a large number of points at random inside the square. Some fraction of these points will also lie inside the quarter circle. If the selected points are uniformly distributed, we expect the fraction of points in the quarter circle to be

$$f = \frac{\pi r^2 / 4}{r^2} = \frac{\pi}{4}.$$

Therefore by measuring f , we can compute π . Program □ shows how this can be done.

```
1 def pi(trials):
2     hits = 0
3     for i in xrange(trials):
4         x = RandomNumberGenerator.next
5         y = RandomNumberGenerator.next
6         if x * x + y * y < 1.0:
7             hits += 1
8     return 4.0 * hits / trials
```

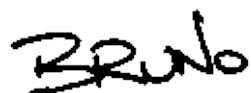
Program: Monte Carlo program to compute π .

The `pi` method uses the `RandomNumberGenerator` defined in Program □ to generate (x,y) pairs uniformly distributed on the unit square ($r=1$). Each point is tested to see if it falls inside the quarter circle. A given point is inside the circle when its distance from the origin, $\sqrt{x^2 + y^2}$, is less than r . In this case since $r=1$, we simply test whether $x^2 + y^2 < 1$.

How well does Program □ work? When 1000 trials are conducted, 792 points are found to lie inside the circle. This gives the value of 3.168 for π , which is only 0.8% too large. When 10^8 trials are conducted, 78535956 points are found to lie inside the circle. In this case, we get $\pi \approx 3.141\ 438\ 24$ which is within 0.005% of the correct value!

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Simulated Annealing

Despite its name, *simulated annealing* has nothing to do either with simulation or annealing. Simulated annealing is a problem solving technique based loosely on the way in which a metal is annealed in order to increase its strength. When a heated metal is cooled very slowly, it freezes into a regular (minimum-energy) crystalline structure.

A simulated annealing algorithm searches for the optimum solution to a given problem in an analogous way. Specifically, it moves about randomly in the solution space looking for a solution that minimizes the value of some objective function. Because it is generated randomly, a given move may cause the objective function to increase, to decrease or to remain unchanged.

A simulated annealing algorithm always accepts moves that *decrease* the value of the objective function. Moves that *increase* the value of the objective function are accepted with probability

$$p = e^{\Delta/T},$$

where Δ is the change in the value of the objective function and T is a control parameter called the *temperature*. That is, a random number generator that generates numbers distributed uniformly on the interval $(0,1)$ is sampled, and if the sample is less than p , the move is accepted.

By analogy with the physical process, the temperature T is initially high. Therefore, the probability of accepting a move that increases the objective function is initially high. The temperature is gradually decreased as the search progresses. That is, the system is *cooled* slowly. In the end, the probability of accepting a move that increases the objective function becomes vanishingly small. In general, the temperature is lowered in accordance with an *annealing schedule*.

The most commonly used annealing schedule is called *exponential cooling*. Exponential cooling begins at some initial temperature, T_0 , and decreases the temperature in steps according to $T_{k+1} = \alpha T_k$, where $0 < \alpha < 1$. Typically, a fixed

number of moves must be accepted at each temperature before proceeding to the next. The algorithm terminates either when the temperature reaches some final value, T_f , or when some other stopping criterion has been met.

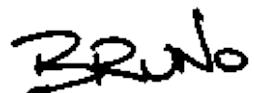
The choice of suitable values for α , T_0 , and T_f is highly problem-dependent. However, empirical evidence suggests that a good value for α is 0.95 and that T_0 should be chosen so that the initial acceptance probability is 0.8. The search is terminated typically after some fixed, total number of solutions have been considered.

Finally, there is the question of selecting the initial solution from which to begin the search. A key requirement is that it be generated quickly. Therefore, the initial solution is generated typically at random. However, sometimes the initial solution can be generated by some other means such as with a greedy algorithm.

- [Example-Balancing Scales](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Example-Balancing Scales

Consider again the *scales balancing problem* described in Section 1. That is, we are given a set of n weights, $\{w_1, w_2, \dots, w_n\}$, which are to be placed on a pair of scales in the way that minimizes the difference between the total weight in each pan. Feasible solution to the problem all have the form $X = \{x_1, x_2, \dots, x_n\}$, where

$$x_i = \begin{cases} 0 & w_i \text{ is placed in the left pan,} \\ 1 & w_i \text{ is placed in the right pan.} \end{cases}$$

To solve this problem using simulated annealing, we need a strategy for generating random moves. The move generator should make small, random changes to the current solution and it must ensure that all possible solutions can be reached. A simple approach is to use the formula

$$X_{i+1} = X_i \odot U$$

where X_i is the initial solution, X_{i+1} is a new solution, $U = \{u_1, u_2, \dots, u_n\}$ is a sequence of zeroes and ones generated randomly, and \odot denotes elementwise addition modulo two.

Exercises

1. Consider the greedy strategy for counting out change given in Section □. Let $\{d_1, d_2, \dots, d_n\}$ be the set of available denominations. For example, the set $\{1, 5, 10, 25, 100, 200\}$ represents the denominations of the commonly circulated Canadian coins. What condition(s) must the set of denominations satisfy to ensure the greedy algorithm always finds an optimal solution?
2. Devise a greedy algorithm to solve optimally the scales balancing problem described in Section □.
 1. Does your algorithm always find the optimal solution?
 2. What is the running time of your algorithm?
3. Consider the following 0/1-knapsack problem:

i	w_i	p_i
1	10	10
2	6	6
3	3	4
4	8	9
5	1	3
<hr/>		
C=18		

1. Solve the problem using the greedy by profit, greedy by weight and greedy by profit density strategies.
2. What is the optimal solution?
4. Consider the breadth-first solver shown in Program □. Suppose we replace the queue (line 3) with a *priority queue*.
 1. How should the solutions in the priority queue be prioritized?
 2. What possible benefit might there be from using a priority queue rather than a FIFO queue?
5. Repeat Exercise □, but this time consider what happens if we replace the queue with a *LIFO stack*.
6. Repeat Exercises □ and □, but this time consider a *branch-and-bound* breadth-first solver.
7. (This question should be attempted *after* reading Chapter □). For some problems the solution space is more naturally a graph rather than a tree.

1. What problem arises if we use the `DepthFirstSolver` given in Program \square to explore a search space that is not a tree.
2. Modify the `DepthFirstSolver` so that it explores a solution space that is not a tree. **Hint:** See Program \square .
3. What problem arises if we use the `BreadthFirstSolver` given in Program \square to explore a search space that is not a tree.
4. Modify the `BreadthFirstSolver` so that it explores a solution space that is not a tree. **Hint:** See Program \square .
8. Devise a backtracking algorithm to solve the *N-queens problem* : Given an $N \times N$ chess board, find a way to place N queens on the board in such a way that no queen can take another.
9. Consider a binary search tree that contains n keys, k_1, k_2, \dots, k_n , at depths d_1, d_2, \dots, d_n , respectively. Suppose the tree will be subjected to a large number of `find` operations. Let p_i be the probability that we access key k_i . Suppose we know *a priori* all the access probabilities. Then we can say that the *optimal binary search tree* is the tree which minimizes the quantity

$$\sum_{i=1}^n p_i(d_i + 1).$$

1. Devise a dynamic programming algorithm that, given the access probabilities, determines the optimal binary search tree.
2. What is the running time of your algorithm?

Hint: Let $C_{i,j}$ be the *cost* of the optimal binary search tree that contains the set of keys $\{k_i, k_{i+1}, k_{i+2}, \dots, k_j\}$ where $i \leq j$. Show that

$$C_{i,j} = \begin{cases} p_i & i = j, \\ \min_{i \leq k \leq j} \{C_{i,k-1} + C_{k+1,j} + \sum_{l=i}^j p_l\} & i < j. \end{cases}$$

10. Consider the typesetting problem discussed in Section \square . The objective is to determine how to break a given sequence of words into lines of text of the appropriate size. This was done either by stretching or compressing the space between the words. Explain why the greedy strategy always finds the optimal solution if we stretch but do not compress the space between words.
11. Consider two complex numbers, $a+bi$ and $c+di$. Show that we can compute the product $(ac-bd)+(ad+bc)i$ with only three multiplications.

12. Devise a divide-and-conquer strategy to find the root of a polynomial. For example, given a polynomial such as $p(x) = 2x^2 + 3x - 4$, and an interval $[u,v]$ such that $p(u)$ and $p(v)$ have opposite signs, find the value r , $u \leq r \leq v$, such that $p(r)=0$.
13. Devise an algorithm to compute a *normally distributed random variable*. A normal distribution is completely defined by its mean and standard deviation. The probability density function for a normal distribution is

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right),$$

where μ is the mean and σ is the standard deviation of the distribution. **Hint:** Consider the *central limit theorem*.

14. Devise an algorithm to compute a *geometrically distributed random variable*. A geometrically distributed random variable is an integer in the interval $[1, \infty)$ given by the probability density function

$$P[X = i] = \theta(1 - \theta)^{i-1},$$

where θ^{-1} is the mean of the distribution.

Hint: Use the fact $P[X = i] = P[i - 1 < Z \leq i]$, where Z is an exponentially distributed random variable with mean $\mu = -1/\ln(1 - \theta)$.

15. Do Exercise □.

Projects

1. Design a class that extends the abstract `Solution` class defined in Program □ to represent the nodes of the solution space of a *0/1-knapsack problem* described in Section □.

Devise a suitable representation for the state of a node and then implement the following properties `getIsFeasible`, `getIsComplete`, `getObjective`, `getBound`, and `getSuccessors`. Note, the `getSuccessors` method returns an iterator object that enumerates all the successors of a given node.

1. Use your class with the `DepthFirstSolver` defined in Program □ to solve the problem given in Table □.
2. Use your class with the `BreadthFirstSolver` defined in Program □ to solve the problem given in Table □.
3. Use your class with the `DepthFirstBranchAndBoundSolver` defined in Program □ to solve the problem given in Table □.
2. Do Project □ for the *change counting problem* described in Section □.
3. Do Project □ for the *scales balancing problem* described in Section □.
4. Do Project □ for the *N-queens problem* described in Exercise □.
5. Design and implement a `GreedySolver` class, along the lines of the `DepthFirstSolver` and `BreadthFirstSolver` classes, that conducts a greedy search of the solution space. To do this you will have to add a method to the abstract `Solution` class:

```
class GreedySolution(Solution):  
  
    def greedySuccessor(self):  
        pass  
    greedySuccessor = abstractmethod(greedySuccessor)
```

6. Design and implement a `SimulatedAnnealingSolver` class, along the lines of the `DepthFirstSolver` and `BreadthFirstSolver` classes, that implements the simulated annealing strategy described in Section □ To do this you will have to add a method to the abstract `Solution` class:

```
class SimulatedAnnealingSolution(Solution):
```

```

def randomSuccessor(self):
    pass
randomSuccessor = abstractmethod(randomSuccessor)

```

7. Design and implement a dynamic programming algorithm to solve the change counting problem. Your algorithm should always find the optimal solution--even when the greedy algorithm fails.
8. Consider the divide-and-conquer strategy for matrix multiplication described in Section □.
 1. Rewrite the implementation of the `__mul__` method of the `Matrix` class introduced in Program □.
 2. Compare the running time of your implementation with the $O(n^3)$ algorithm given in Program □.
9. Consider random number generator that generates random numbers uniformly distributed between zero and one. Such a generator produces a sequence of random numbers x_1, x_2, x_3, \dots . A common test of randomness evaluates the correlation between consecutive pairs of numbers in the sequence. One way to do this is to plot on a graph the points

$$(x_1, x_2), (x_2, x_3), (x_3, x_4), \dots$$

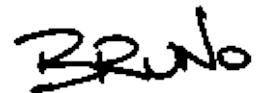
1. Write a program to compute the first 1000 pairs of numbers generated using the `UniformRV` defined in Program □.
2. What conclusions can you draw from your results?

Sorting Algorithms and Sorters

- [Basics](#)
 - [Sorting and Sorters](#)
 - [Insertion Sorting](#)
 - [Exchange Sorting](#)
 - [Selection Sorting](#)
 - [Merge Sorting](#)
 - [A Lower Bound on Sorting](#)
 - [Distribution Sorting](#)
 - [Performance Data](#)
 - [Exercises](#)
 - [Projects](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Basics

Consider an arbitrary sequence $S = \{s_1, s_2, s_3, \dots, s_n\}$ comprised of $n \geq 0$ elements drawn from a some universal set U . The goal of *sorting* is to rearrange the elements of S to produce a new sequence, say S' , in which the elements of S appear *in order*.

But what does it mean for the elements of S' to be *in order*? We shall assume that there is a relation, $<$, defined over the universe U . The relation $<$ must be a *total order*, which is defined as follows:

Definition A *total order* is a relation, say $<$, defined on the elements of some universal set U with the following properties:

1. For all pairs of elements $(i, j) \in U \times U$, exactly one of the following is true: $i < j$, $i = j$, or $j < i$.

(All elements are commensurate).

2. For all triples $(i, j, k) \in U \times U \times U$, $i < j \wedge j < k \iff i < k$.

(The relation $<$ is transitive).

In order to *sort* the elements of the sequence S , we determine the *permutation* $P = \{p_1, p_2, p_3, \dots, p_n\}$ of the elements of S such that

$$s_{p_1} \leq s_{p_2} \leq s_{p_3} \leq \dots \leq s_{p_n}.$$

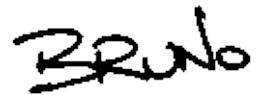
In practice, we are not interested in the permutation P , *per se*. Instead, our objective is to compute the sorted sequence $S' = \{s'_1, s'_2, s'_3, \dots, s'_n\}$ in which $s'_i = s_{p_i}$ for $1 \leq i \leq n$.

Sometimes the sequence to be sorted, S , contains duplicates. That is, there exist values i and j , $1 \leq i < j \leq n$, such that $s_i = s_j$. In general when a sequence that contains duplicates is sorted, there is no guarantee that the duplicated elements retain their relative positions. That is, s_i could appear either before or after s_j in

the sorted sequence S' . If duplicates retain their relative positions in the sorted sequence the sort is said to be *stable* . In order for s_i and s_j to retain their relative order in the sorted sequence, we require that p'_i precedes p'_j in S' . Therefore, the sort is stable if $p_i < p_j$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Sorting and Sorters

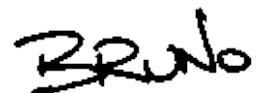
The traditional way to implement a sorting algorithm is to write a method that sorts an array of data. This chapter presents an alternate, object-oriented approach that is based on the notion of an *abstract sorter* .

Think of a sorter as an abstract machine, the sole purpose of which is to sort arrays of data. A machine is an object. Therefore, it makes sense that we represent it as an instance of some class. The machine sorts data. Therefore, the class will have a method, say `sort`, which sorts an array of data.

-
- [Abstract Sorters](#)
 - [Sorter Class Hierarchy](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Abstract Sorters

Program □ defines the Sorter class. The abstract Sorter class extends the abstract object class introduced in Program □.

```
1  class Sorter(Object):
2
3      def __init__(self):
4          super(Sorter, self).__init__()
5          self._array = None
6          self._n = 0
7
8      def _sort(self):
9          pass
10     _sort = abstractmethod(_sort)
11
12     def sort(self, array):
13         assert isinstance(array, Array)
14         self._array = array
15         self._n = len(array)
16         if self._n > 0:
17             self._sort()
18         self._array = None
19
20     def swap(self, i, j):
21         tmp = self._array[i]
22         self._array[i] = self._array[j]
23         self._array[j] = tmp
```

Program: Abstract Sorter class.

The Sorter comprises the two instance attributes, `_array` and `_n`, the concrete methods `sort` and `swap`, and the abstract method `_sort`. Since the `_sort` method is an abstract method, an implementation must be given in a derived class.

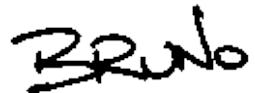
The `sort` method does not sort the data itself. It is the `_sort` method, which is provided by a derived class, that does the actual sorting. The `sort` method merely sets-up things by initializing the instance attributes of `Sorter` as follows: The `_array` instance attribute refers to the array of objects to be sorted and the

length of that array is assigned to `_n`.

The `swap` method is used to implement most of the sorting algorithms presented in this chapter. In addition to `self`, the `swap` method takes two integers arguments. It exchanges the contents of the array at the positions specified by those arguments. The exchange is done as a sequence of three assignments. Therefore, the `swap` method runs in constant time.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Sorter Class Hierarchy

This chapter describes nine different sorting algorithms. These are organized into the following five categories:

- insertion sorts
- exchange sorts
- selection sorts
- merge sorts
- distribution sorts .

As shown in Figure □, the sorter classes have been arranged in a class hierarchy that reflects this classification scheme.

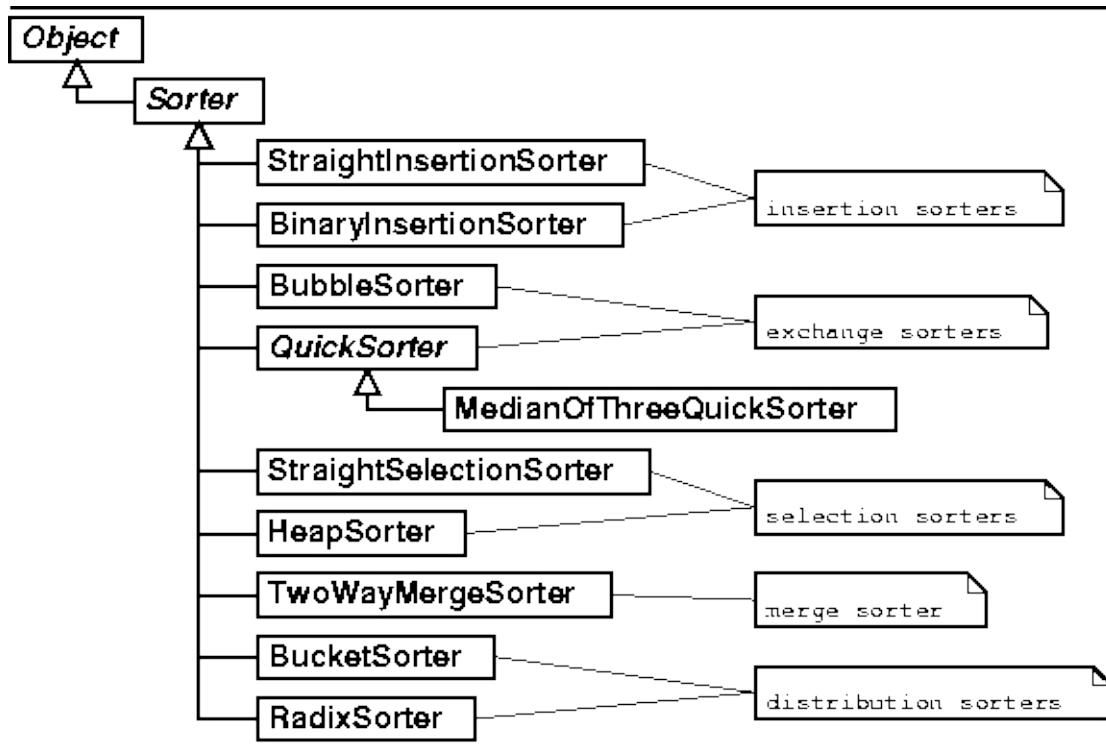


Figure: Sorter class hierarchy

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Insertion Sorting

The first class of sorting algorithm that we consider comprises algorithms that *sort by insertion*. An algorithm that sorts by insertion takes the initial, unsorted sequence, $S = \{s_1, s_2, s_3, \dots, s_n\}$, and computes a series of *sorted* sequences $S'_0, S'_1, S'_2, \dots, S'_n$, as follows:

1. The first sequence in the series, S'_0 is the empty sequence. That is, $S'_0 = \{\}$.
2. Given a sequence S'_i in the series, for $0 \leq i < n$, the next sequence in the series, S'_{i+1} , is obtained by inserting the $(i + 1)^{th}$ element of the unsorted sequence s_{i+1} into the correct position in S'_i .

Each sequence S'_i , $0 \leq i < n$, contains the first i elements of the unsorted sequence S . Therefore, the final sequence in the series, S'_n , is the sorted sequence we seek. That is, $S' = S'_n$.

Figure  illustrates the insertion sorting algorithm. The figure shows the progression of the insertion sorting algorithm as it sorts an array of ten integers. The array is sorted *in place*. That is, the initial unsorted sequence, S , and the series of sorted sequences, S'_0, S'_1, \dots , occupy the same array.

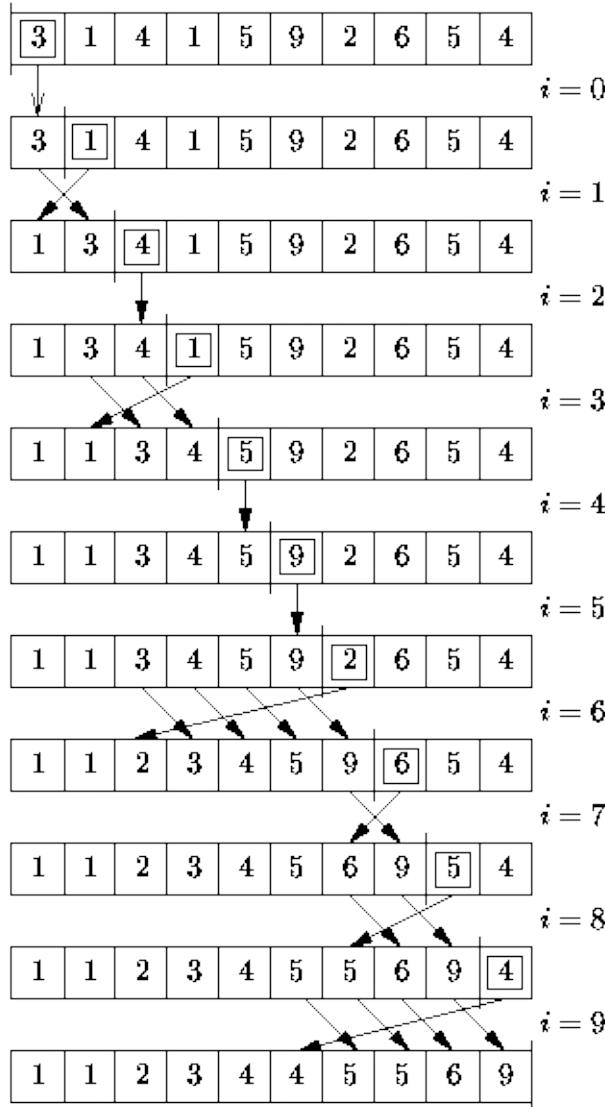


Figure: Insertion sorting.

In the i^{th} step, the element at position i in the array is inserted into the sorted sequence S'_i which occupies array positions 0 to $(i-1)$. After this is done, array positions 0 to i contain the $i+1$ elements of S'_{i+1} . Array positions $(i+1)$ to $(n-1)$ contain the remaining $n-i-1$ elements of the unsorted sequence S .

As shown in Figure □, the first step ($i=0$) is trivial--inserting an element into the empty list involves no work. Altogether, $n-1$ non-trivial insertions are required to sort a list of n elements.

-
- [Straight Insertion Sort](#)
 - [Average Running Time](#)
 - [Binary Insertion Sort](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Straight Insertion Sort

The key step of any insertion sorting algorithm involves the insertion of an item into a sorted sequence. There are two aspects to an insertion--finding the correct position in the sequence at which to insert the new element and moving all the elements over to make room for the new one.

This section presents the *straight insertion sorting* algorithm. Straight insertion sorting uses a *linear search* to locate the position at which the next element is to be inserted.

-
- [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementation

Program □ defines the `StraightInsertionSorter` class. The `StraightInsertionSorter` extends the abstract `Sorter` class defined in Program □. It simply provides an implementation for the `_sort` method.

```
1  class StraightInsertionSorter(Sorter):
2
3      def __init__(self):
4          super(StraightInsertionSorter, self).__init__()
5
6      def _sort(self):
7          for i in xrange(1, self._n):
8              j = i
9              while j > 0 and self._array[j - 1] > self._array[j]:
10                  self.swap(j, j - 1)
11                  j -= 1
```

Program: `StraightInsertionSorter` class `__init__` and `_sort` methods.

In order to determine the running time of the `_sort` method, we need to determine the number of iterations of the inner loop (lines 9-11). The number of iterations of the inner loop in the i^{th} iteration of the outer loop depends on the positions of the values in the array. In the best case, the value in position i of the array is larger than that in position $i-1$ and zero iterations of the inner loop are done. In this case, the running time for insertion sort is $O(n)$. Notice that the best case performance occurs when we sort an array that is already sorted!

In the worst case, i iterations of the inner loop are required in the i^{th} iteration of the outer loop. This occurs when the value in position i of the array is smaller than the values at positions 0 through $i-1$. Therefore, the worst case arises when we sort an array in which the elements are initially sorted in reverse. In this case the running time for insertion sort is $O(n^2)$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Average Running Time

The best case running time of insertion sorting is $O(n)$ but the worst-case running time is $O(n^2)$. Therefore, we might suspect that the average running time falls somewhere in between. In order to determine it, we must define more precisely what we mean by the *average* running time. A simple definition of average running time is to say that it is the running time needed to sort the average sequence. But what is the average sequence?

The usual way to determine the average running time of a sorting algorithm is to consider only sequences that contain no duplicates. Since every sorted sequence of length n is simply a permutation of an unsorted one, we can represent every such sequence by a permutation of the sequence $S = \{1, 2, 3, \dots, n\}$. When computing the average running time, we assume that every permutation is equally likely. Therefore, the average running time of a sorting algorithm is the running time averaged over all permutations of the sequence S .

Consider a permutation $P = \{p_1, p_2, p_3, \dots, p_n\}$ of the sequence S . An *inversion* in P consists of two elements, say p_i and p_j , such that $p_i > p_j$ but $i < j$. That is, an inversion in P is a pair of elements that are in the wrong order. For example, the permutation $\{1, 4, 3, 2\}$ contains three inversions--(4,3), (4,2), and (3,2). The following theorem tells us how many inversions we can expect in the average sequence:

Theorem The average number of inversions in a permutation of n distinct elements is $n(n-1)/4$.

extbf{Proof} Let S be an arbitrary sequence of n distinct elements and let S^R be the same sequence, but in reverse.

For example, if $S = \{s_1, s_2, s_3, \dots, s_n\}$, then $S^R = \{s_n, s_{n-1}, s_{n-2}, \dots, s_1\}$.

Consider any pair of distinct elements in S , say s_i and s_j where $1 \leq i < j \leq n$. There are two distinct possibilities: Either $s_i < s_j$, in which case (s_j, s_i) is an

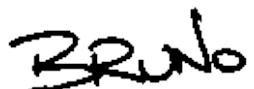
inversion in S^R ; or $s_j < s_i$, in which case (s_i, s_j) is an inversion in S . Therefore, every pair contributes exactly one inversion either to S or to S^R .

The total number of pairs in S is $\binom{n}{2} = n(n - 1)/2$. Since every such pair contributes an inversion either to S or to S^R , we expect *on average* that half of the inversions will appear in S . Therefore, the average number of inversions in a sequence of n distinct elements is $n(n-1)/4$.

What do inversions have to do with sorting? As a list is sorted, inversions are removed. In fact, since the inner loop of the insertion sort method swaps *adjacent* array elements, inversions are removed *one at a time!* Since a swap takes constant time, and since the average number of inversions is $n(n-1)/4$, the $O(n^2)$ *average* running time for the insertion sort method is .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Binary Insertion Sort

The straight insertion algorithm presented in the preceding section does a linear search to find the position in which to do the insertion. However, since the element is inserted into a sequence that is already sorted, we can use a binary search instead of a linear search. Whereas a linear search requires $O(n)$ comparisons in the worst case, a binary search only requires $O(\log n)$ comparisons. Therefore, if the cost of a comparison is significant, the binary search may be preferred.

Program □ defines the `BinaryInsertionSorter` class. The `BinaryInsertionSorter` class extends the abstract `Sorter` class defined in Program □. The framework of the `_sort` method is essentially the same as that of the `StraightInsertionSorter` class.

```
1  class BinaryInsertionSorter(Sorter):
2
3      def __init__(self):
4          super(BinaryInsertionSorter, self).__init__()
5
6      def _sort(self):
7          for i in xrange(1, self._n):
8              tmp = self._array[i]
9              left = 0
10             right = i
11             while left < right:
12                 middle = (left + right) / 2
13                 if tmp >= self._array[middle]:
14                     left = middle + 1
15                 else:
16                     right = middle
17             j = i
18             while j > left:
19                 self.swap(j - 1, j)
20                 j -= 1
```

Program: `BinaryInsertionSorter` class `__init__` and `_sort` methods.

Exactly, $n-1$ iterations of the outer loop are done (lines 7-20). In each iteration, a binary search search is done to determine the position at which to do the insertion (lines 9-16). In the i^{th} iteration of the outer loop, the binary search

considers array positions 0 to i (for $1 \leq i < n$). The running time for the binary search in the i^{th} iteration is $O(\lfloor \log_2(i+1) \rfloor) = O(\log i)$. Once the correct position is found, at most i swaps are needed to insert the element in its place (lines 17-20).

The worst-case running time of the binary insertion sort is dominated by the i swaps needed to do the insertion. Therefore, the worst-case running time is $O(n^2)$. Furthermore, since the algorithm only swaps adjacent array elements, the average running time is also $O(n^2)$ (see Section □). Asymptotically, the binary insertion sort is no better than straight insertion.

However, the binary insertion sort does fewer array element comparisons than insertion sort. In the i^{th} iteration of the outer loop, the binary search requires $\lfloor \log_2(i+1) \rfloor$ comparisons, for $1 \leq i < n$. Therefore, the total number of comparisons is

$$\begin{aligned} \sum_{i=1}^{n-1} \lfloor \log_2(i+1) \rfloor &= \sum_{i=1}^n \lfloor \log_2 i \rfloor \\ &= (n+1)\lfloor \log_2(n+1) \rfloor + 2^{\lfloor \log_2(n+1) \rfloor + 1} + 2 \\ &= O(n \log n) \end{aligned}$$

(This result follows directly from Theorem □).

The number of comparisons required by the straight insertion sort is $O(n^2)$ in the worst case as well as on average. Therefore on average, the binary insertion sort uses fewer comparisons than straight insertion sort. On the other hand, the previous section shows that in the best case the running time for straight insertion is $O(n)$. Since the binary insertion sort method *always* does the binary search, its best case running time is $O(n \log n)$. Table □ summarizes the asymptotic running times for the two insertion sorts.

Table: Running times for insertion sorting.

running time

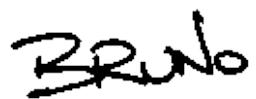
algorithm	best case	average case	worst case
-----------	-----------	--------------	------------

straight insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
-------------------------	--------	----------	----------

binary insertion sort	$O(n \log n)$	$O(n^2)$	$O(n^2)$
-----------------------	---------------	----------	----------

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exchange Sorting

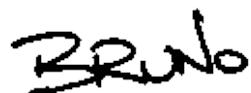
The second class of sorting algorithm that we consider comprises algorithms that *sort by exchanging* pairs of items until the sequence is sorted. In general, an algorithm may exchange adjacent elements as well as widely separated ones.

In fact, since the insertion sorts considered in the preceding section accomplish the insertion by swapping adjacent elements, insertion sorting can be considered as a kind of exchange sort. The reason for creating a separate category for insertion sorts is that the essence of those algorithms is insertion into a sorted list. On the other hand, an exchange sort does not necessarily make use of such a sorted list.

-
- [Bubble Sort](#)
 - [Quicksort](#)
 - [Running Time Analysis](#)
 - [Average Running Time](#)
 - [Selecting the Pivot](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Bubble Sort

The simplest and, perhaps, the best known of the exchange sorts is the *bubble sort*. ◇ Figure □ shows the operation of bubble sort.

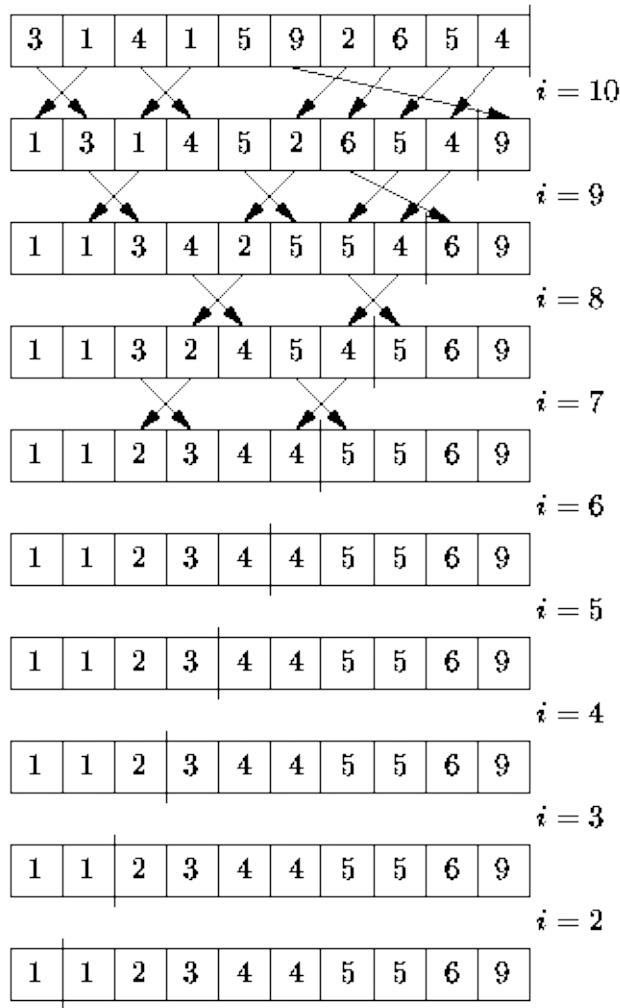


Figure: Bubble sorting.

To sort the sequence $S = \{s_0, s_1, s_2, \dots, s_{n-1}\}$, bubble sort makes $n-1$ passes through the data. In each pass, adjacent elements are compared and swapped if necessary. First, s_0 and s_1 are compared; next, s_1 and s_2 ; and so on.

Notice that after the first pass through the data, the largest element in the sequence has *bubbled up* into the last array position. In general, after k passes

through the data, the last k elements of the array are correct and need not be considered any longer. In this regard the bubble sort differs from the insertion sort algorithms--the sorted subsequence of k elements is never modified (by an insertion).

Figure □ also shows that while $n-1$ passes through the data are required to guarantee that the list is sorted in the end, it is possible for the list to become sorted much earlier! When no exchanges at all are made in a given pass, then the array is sorted and no additional passes are required. A minor algorithmic modification would be to count the exchanges made in a pass, and to terminate the sort when none are made.

Program □ defines the `BubbleSorter` class. The `BubbleSorter` class extends the abstract `Sorter` class defined in Program □. It simply provides an implementation for the `_sort` method.

```

1  class BubbleSorter(Sorter):
2
3      def __init__(self):
4          super(BubbleSorter, self).__init__()
5
6      def _sort(self):
7          i = self._n
8          while i > 1:
9              for j in xrange(i - 1):
10                  if self._array[j] > self._array[j + 1]:
11                      self.swap(j, j + 1)
12          i -= 1

```

Program: `BubbleSorter` class `__init__` and `_sort` methods.

The outer loop (lines 8-12) is done for $i = n - 1, n - 2, n - 3, \dots, 2$. That makes $n-1$ iterations in total. During the i^{th} iteration of the outer loop, exactly $i-1$ iterations of the inner loop are done (lines 9-11). Therefore, the number of iterations of the inner loop, summed over all the passes of the outer loop is

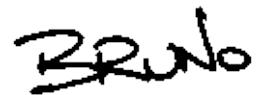
$$\sum_{i=2}^n (i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Consequently, the running time of bubble sort is $\Theta(n^2)$.

The body of the inner loop compares adjacent array elements and swaps them if necessary (lines 10-11). This takes at most a constant amount of time. Of course, the algorithm will run slightly faster when no swapping is needed. For example, this occurs if the array is already sorted to begin with. In the worst case, it is necessary to swap in every iteration of the inner loop. This occurs when the array is sorted initially in reverse order. Since only adjacent elements are swapped, bubble sort removes inversions one at time. Therefore, the average number of swaps required is $\Theta(n^2)$. Nevertheless, the running time of bubble sort is always $\Theta(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Quicksort

The second exchange sort we consider is the *quicksort* algorithm. Quicksort is a *divide-and-conquer* style algorithm. A divide-and-conquer algorithm solves a given problem by splitting it into two or more smaller subproblems, recursively solving each of the subproblems, and then combining the solutions to the smaller problems to obtain a solution to the original one.

To sort the sequence $S = \{s_1, s_2, s_3, \dots, s_n\}$, quicksort performs the following steps:

1. Select one of the elements of S . The selected element, p , is called the *pivot*.
2. Remove p from S and then partition the remaining elements of S into two distinct sequences, L and G , such that every element in L is less than or equal to the pivot and every element in G is greater than or equal to the pivot. In general, both L and G are *unsorted*.
3. Rearrange the elements of the sequence as follows:

$$S' = \underbrace{\{l_1, l_2, \dots, l_{|L|}\}}_L, \underbrace{p}_{\text{pivot}}, \underbrace{\{g_1, g_2, \dots, g_{|G|}\}}_G$$

Notice that the pivot is now in the position in which it belongs in the sorted sequence, since all the elements to the left of the pivot are less than or equal to the pivot and all the elements to the right are greater than or equal to it.

4. Recursively quicksort the unsorted sequences L and G .

The first step of the algorithm is a crucial one. We have not specified how to select the pivot. Fortunately, the sorting algorithm works no matter which element is chosen to be the pivot. However, the pivot selection affects directly the running time of the algorithm. If we choose poorly the running time will be poor.

Figure  illustrates the detailed operation of quicksort as it sorts the sequence $\{3, 1, 4, 1, 5, 9, 2, 6, 5, 4\}$. To begin the sort, we select a pivot. In this example, the value 4 in the last array position is chosen. Next, the remaining elements are partitioned into two sequences, one which contains values less than or equal to 4 ($L = \{3, 1, 2, 1\}$) and one which contains values greater than or equal to 4 (

$G = \{5, 9, 4, 6, 5\}$). Notice that the partitioning is accomplished by exchanging elements. This is why quicksort is considered to be an exchange sort.

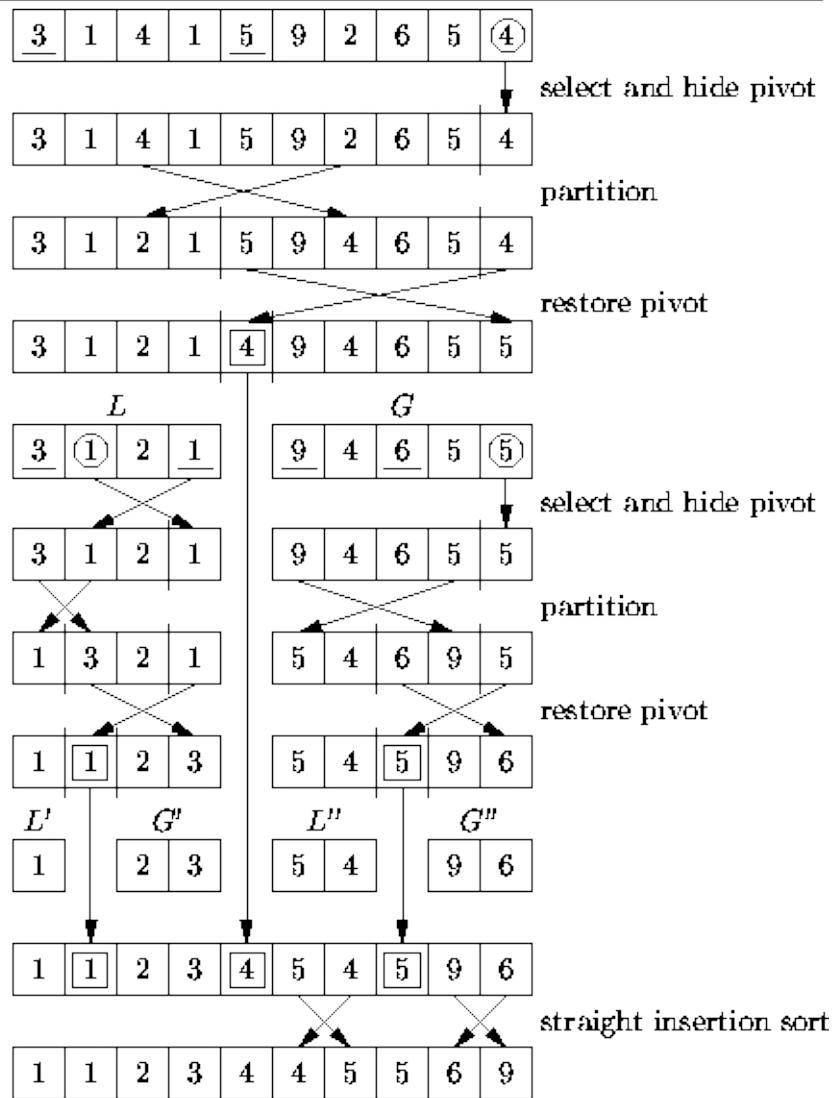


Figure: ``Quick" sorting.

After the partitioning, the pivot is inserted between the two sequences. This is called *restoring* the pivot. To restore the pivot, we simply exchange it with the first element of G . Notice that the 4 is in its correct position in the sorted sequence and it is not considered any further.

Now the quicksort algorithm calls itself recursively, first to sort the sequence $L = \{3, 1, 2, 1\}$; second to sort the sequence $G = \{9, 4, 6, 5, 5\}$. The quicksort of L

selects 1 as the pivot, and creates the two subsequences $L' = \{1\}$ and $G' = \{2, 3\}$. Similarly, the quicksort of G uses 5 as the pivot and creates the two subsequences $L'' = \{5, 4\}$ and $G'' = \{9, 6\}$.

At this point in the example the recursion has been stopped. It turns out that to keep the code simple, quicksort algorithms usually stop the recursion when the length of a subsequence falls below a critical value called the *cut-off*. In this example, the cut-off is two (i.e., a subsequence of two or fewer elements is not sorted). This means that when the algorithm terminates, the sequence is not yet sorted. However as Figure □ shows, the sequence is *almost* sorted. In fact, every element is guaranteed to be less than two positions away from its final resting place.

We can complete the sorting of the sequence by using a straight insertion sort. In Section □ it is shown that straight insertion is quite good at sorting sequences that are almost sorted. In fact, if we know that every element of the sequence is at most d positions from its final resting place, the running time of straight insertion is $O(dn)$ and since $d=2$ is a constant, the running time is $O(n)$.

- [Implementation](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Program □ introduces the QuickSorter class. The abstract QuickSorter class extends the abstract Sorter class defined in Program □. It declares the abstract method selectPivot the implementation of which is provided by a derived class.

```
 1  class QuickSorter(Sorter):
 2
 3      def __init__(self):
 4          super(QuickSorter, self).__init__()
 5
 6      def selectPivot(self):
 7          pass
 8      selectPivot = abstractmethod(selectPivot)
 9
10      # ...
```

Program: Abstract QuickSorter class `__init__` and `selectPivot` methods.

Program □ defines the quicksort method of the QuickSorter class. In addition to `self`, the quicksort method takes two integer arguments, `left` and `right`, which denote left and right ends, respectively, of the section of the array to be sorted. That is, this quicksort method sorts

`array[left], array[left + 1], ..., array[right]`

```
1 class QuickSorter(Sorter):
2
3     CUTOFF = 2 # minimum cut-off
4
5     def quicksort(self, left, right):
6         if right - left + 1 > self.CUTOFF:
7             p = self.selectPivot(left, right)
8             self.swap(p, right)
9             pivot = self._array[right]
10            i = left
11            j = right - 1
12            while True:
13                while i < j and self._array[i] < pivot:
14                    i += 1
15                while i < j and self._array[j] > pivot:
16                    j -= 1
17                if i >= j:
18                    break
19                self.swap(i, j)
20                i += 1
21                j -= 1
22            if self._array[i] > pivot:
23                self.swap(i, right)
24            if left < i:
25                self.quicksort(left, i - 1)
26            if right > i:
27                self.quicksort(i + 1, right)
28
29     # ...
```

Program: Abstract QuickSorter class quicksort method.

As discussed above, the QuickSorter only sorts sequences whose length exceeds the *cut-off* value. Since the implementation shown only works correctly when the number of elements in the sequence to be sorted is three or more, the CUTOFF value of two is used (line 3).

The algorithm begins by calling the method selectPivot which chooses one of the elements to be the pivot (line 7). The implementation of selectPivot is discussed below. All that we require here is that the value p returned by selectPivot satisfies $\text{left} \leq p \leq \text{right}$. Having selected an element to be the pivot, we *hide* the pivot by swapping it with the right-most element of the sequence (line 8). The pivot is *hidden* in order to get it out of the way of the next step.

The next step partitions the remaining elements into two sequences--one comprised of values less than or equal to the pivot, the other comprised of values greater than or equal to the pivot. The partitioning is done using two array indices, i and j . The first, i , starts at the left end and moves to the right; the second, j , starts at the right end and moves to the left.

The variable i is increased as long as $\text{array}[i]$ is less than the pivot (lines 13-14). Then the variable j is decreased as long as $\text{array}[j]$ is greater than the pivot (lines 15-16). When i and j meet (or pass each other), the partitioning is done (lines 17-18). Otherwise, $i < j$ but $\text{array}[i] \geq \text{pivot} \geq \text{array}[j]$. This situation is remedied by swapping $\text{array}[i]$ and $\text{array}[j]$ (line 19).

When the partitioning loop terminates, the pivot is still in $\text{array}[\text{right}]$; the value in $\text{array}[i]$ is greater than or equal to the pivot; everything to the left is less than or equal to the pivot; and everything to the right is greater than or equal to the pivot. We can now put the pivot in its proper place by swapping it with $\text{array}[i]$ (lines 22-23). This is called *restoring* the pivot. With the pivot in its final resting place, all we need to do is sort the subsequences on either side of the pivot (lines 24-27).

Program □ defines the `_sort` method of the abstract `QuickSorter` class. The `_sort` acts as the front end to the recursive `quicksort` method given in Program □. It calls the `quicksort` method with `left` set to zero and `right` set to $n-1$, where n is the length of the array to be sorted. Finally, it uses a `StraightInsertionSorter` to finish sorting the list.

```
1  class QuickSorter(Sorter):
2
3      def _sort(self):
4          self.quicksort(0, self._n - 1)
5          sorter = StraightInsertionSorter()
6          sorter.sort(self._array)
7
8      # ...
```

Program: Abstract `QuickSorter` class `_sort` method.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Running Time Analysis

The running time of the recursive quicksort method (Program □) is given by

$$T(n) = \begin{cases} O(1) & n \leq 2, \\ T(\text{selectPivot}) + T(i) + T(n - i - 1) + O(n) & n > 2, \end{cases} \quad (15.1)$$

where n is the number of elements in sequence to be sorted, $T(\text{selectPivot})$ is the running time of the `selectPivot` method, and i is the number of elements which end up to the left of the pivot, $0 \leq i \leq n - 1$.

The running time of quicksort is affected by the `selectPivot` method in two ways: First, the value of the pivot chosen affects the sizes of the subsequences. That is, the pivot determines the value i in Equation □. Second, the running time of the `selectPivot` method itself, $T(\text{selectPivot})$, must be taken into account. Fortunately, if $T(\text{selectPivot}) = O(n)$, we can ignore its running time because there is already an $O(n)$ term in the expression.

In order to solve Equation □, we assume that $T(\text{selectPivot}) = O(n)$ and then drop the $O(\cdot)$'s from the recurrence to get

$$T(n) = \begin{cases} 1 & n \leq 2, \\ T(i) + T(n - i - 1) + n & n > 2, \quad 0 \leq i \leq n - 1. \end{cases} \quad (15.2)$$

Clearly the solution depends on the value of i .

-
- [Worst-Case Running Time](#)
 - [Best-Case Running Time](#)
-

Bruno



Worst-Case Running Time

In the worst case the i in Equation \square is always zero. \diamond In this case, we solve the recurrence using repeated substitution like this:

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(n-k) + \sum_{j=n-k}^n j \\ &\vdots \\ &= T(2) + \sum_{j=2}^n j \\ &= n(n+1)/2 \\ &= O(n^2). \end{aligned}$$

The worst case occurs when the two subsequences are as unbalanced as they can be--one sequence has all the remaining elements and the other has none.

Best-Case Running Time

In the best case, the partitioning step divides the remaining elements into two sequences with exactly the same number of elements. For example, suppose that $n = 2^m - 1$ for some integer $m > 0$. After removing the pivot $2^m - 2$ elements remain. If these are divided evenly, each sequence will have $2^{m-1} - 1$ elements. In this case Equation □ gives

$$\begin{aligned}
 T(2^m - 1) &= 2T(2^{m-1} - 1) + 2^m - 1 \\
 &= 2^2T(2^{m-2} - 1) + 2 \cdot 2^m - 2 - 1 \\
 &= 2^3T(2^{m-3} - 1) + 3 \cdot 2^m - 3 - 2 - 1 \\
 &\vdots \\
 &= 2^kT(2^{m-k} - 1) + k2^m - \sum_{j=1}^k j \\
 &= 2^{m-1}T(1) + (m-1)2^m - \sum_{j=1}^{m-1} j, \quad m - k = 1 \\
 &= (2^m(2m - 1) - m(m - 1))/2 \\
 &= [(n + 1)(2\log_2(n + 1) - 1) - (\log_2(n + 1) - 1)\log_2(n + 1)]/2 \\
 &= O(n \log n).
 \end{aligned}$$

Average Running Time

To determine the average running time for the quicksort algorithm, we shall assume that each element of the sequence has an equal chance of being selected for the pivot. Therefore, if i is the number of elements in a sequence of length n less than the pivot, then i is uniformly distributed in the interval $[0, n-1]$.

Consequently, the average value of $T(i) = \frac{1}{n} \sum_{j=0}^{n-1} T(j)$. Similarly, the average value of $T(n-i-1) = \frac{1}{n} \sum_{j=0}^{n-1} T(n-j-1)$. To determine the average running time, we rewrite Equation □ thus:

$$\begin{aligned} T(n) &= \begin{cases} 1 & n \leq 2, \\ \frac{1}{n} \sum_{j=0}^{n-1} T(j) + \frac{1}{n} \sum_{j=0}^{n-1} T(n-j-1) + n & n > 2 \end{cases} \\ &= \begin{cases} 1 & n \leq 2, \\ \frac{2}{n} \sum_{j=0}^{n-1} T(j) + n & n > 2. \end{cases} \end{aligned} \quad (15.3)$$

To solve this recurrence we consider the case $n > 2$ and then multiply Equation □ by n to get

$$nT(n) = 2 \sum_{j=0}^{n-1} T(j) + n^2. \quad (15.4)$$

Since this equation is valid for any $n > 2$, by substituting $n-1$ for n we can also write

$$(n-1)T(n-1) = 2 \sum_{j=0}^{n-2} T(j) + n^2 - 2n + 1. \quad (15.5)$$

which is valid for $n > 3$. Subtracting Equation □ from Equation □ gives

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1$$

which can be rewritten as

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} - \frac{1}{n(n+1)}. \quad (15.6)$$

Equation □ can be solved by telescoping like this:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} - \frac{1}{(n)(n+1)} \quad (15.7)$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2}{n} - \frac{1}{(n-1)(n)}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2}{n-1} - \frac{1}{(n-2)(n-1)}$$

⋮

$$\frac{T(n-k)}{n-k+1} = \frac{T(n-k-1)}{n-k} + \frac{2}{n-k+1} - \frac{1}{(n-k)(n-k+1)}$$

⋮

$$\frac{T(3)}{4} = \frac{T(2)}{2} + \frac{2}{4} - \frac{1}{(3)(4)}. \quad (15.8)$$

Adding together Equation 15.7 through Equation 15.8 gives

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(2)}{3} + 2 \sum_{i=4}^{n+1} \frac{1}{i} - \sum_{i=3}^n \frac{1}{i(i+1)} \\ &= 2 \sum_{i=1}^{n+1} \frac{1}{i} - \sum_{i=1}^n \frac{1}{i(i+1)} - 2 \\ &= 2H_{n+1} + \frac{1}{n+1} - 3, \end{aligned}$$

where H_{n+1} is the $(n+1)^{\text{th}}$ harmonic number. Finally, multiplying through by $n+1$ gives

$$T(n) = 2(n+1)H_{n+1} - 3n - 2.$$

In Section 15.2 it is shown that $H_n \approx \ln n + \gamma$, where $\gamma \approx 0.577\,215$ is called Euler's constant. Thus, we get that the average running time of quicksort is

$$\begin{aligned} T(n) &\approx 2(n+1)(\ln(n+1) + \gamma) - 3n - 3 \\ &= O(n \log n). \end{aligned}$$

Table 15.2 summarizes the asymptotic running times for the quicksort method and compares it to those of bubble sort. Notice that the best-case and average case running times for the quicksort algorithm have the same asymptotic bound!

Table: Running times for exchange sorting.

algorithm	best case	average case	worst case
bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
quicksort (random pivot selection)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Selecting the Pivot

The analysis in the preceding section shows that selecting a good pivot is important. If we do a bad job of choosing the pivot, the running time of quicksort is $O(n^2)$. On the other hand, the average-case analysis shows that if every element of a sequence is equally likely to be chosen for the pivot, the running time is $O(n \log n)$. This suggests that we can expect to get good performance simply by selecting a *random pivot*!

If we expect to be sorting random input sequences, then we can achieve random pivot selection simply by always choosing, say, the first element of the sequence to be the pivot. Clearly this can be done in constant time. (Remember, the analysis requires that $T(\text{selectPivot}) = O(n)$). As long as each element in the sequence is equally likely to appear in the first position, the average running time will be $O(n \log n)$.

In practice it is often the case that the sequence to be sorted is almost sorted. In particular, consider what happens if the sequence to be sorted using quicksort is already sorted. If we always choose the first element as the pivot, then we are guaranteed to have the worst-case running time! This is also true if we always pick the last element of the sequence. And it is also true if the sequence is initially sorted in reverse.

Therefore, we need to be more careful when choosing the pivot. Ideally, the pivot divides the input sequence exactly in two. That is, the ideal pivot is the *median* element of the sequence. This suggests that the `selectPivot` method should find the median. To ensure that the running time analysis is valid, we need to find the median in $O(n)$ time.

How do you find the median? One way is to sort the sequence and then select the $\lceil n/2 \rceil^{\text{th}}$ element. But this is not possible, because we need to find the median to sort the sequence in the first place!

While it is possible to find the median of a sequence of n elements in $O(n)$ time, it is usually not necessary to do so. All that we really need to do is select a random element of the sequence while avoiding the problems described above.

A common way to do this is the *median-of-three pivot selection* technique. In this approach, we choose as the pivot the median of the element at the left end of the sequence, the element at the right end of the sequence, and the element in the middle of the sequence. Clearly, this does the *right thing* if the input sequence is initially sorted (either in forward or reverse order).

Program □ defines the `MedianOfThreeQuickSorter` class. The `MedianOfThreeQuickSorter` class extends the abstract `QuickSorter` class introduced in Program □. It provides an implementation for the `selectPivot` method based on median-of-three pivot selection. Notice that this algorithm does exactly three comparisons to select the pivot. As a result, its running time is $O(1)$. In practice this scheme performs sufficiently well that more complicated pivot selection approaches are unnecessary.

```
1  class MedianOfThreeQuickSorter(QuickSorter):
2
3      def __init__(self):
4          super(MedianOfThreeQuickSorter, self).__init__()
5
6      def selectPivot(self, left, right):
7          middle = (left + right) / 2
8          if self._array[left] > self._array[middle]:
9              self.swap(left, middle)
10         if self._array[left] > self._array[right]:
11             self.swap(left, right)
12         if self._array[middle] > self._array[right]:
13             self.swap(middle, right)
14
15     return middle
```

Program: `MedianOfThreeQuickSorter` class `__init__` and `selectPivot` methods.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Selection Sorting

The third class of sorting algorithm that we consider comprises algorithms that sort by *selection*. Such algorithms construct the sorted sequence one element at a time by adding elements to the sorted sequence *in order*. At each step, the next element to be added to the sorted sequence is selected from the remaining elements.

Because the elements are added to the sorted sequence in order, they are always added at one end. This is what makes selection sorting different from insertion sorting. In insertion sorting elements are added to the sorted sequence in an arbitrary order. Therefore, the position in the sorted sequence at which each subsequent element is inserted is arbitrary.

Both selection sorts described in this section sort the arrays *in place*. Consequently, the sorts are implemented by exchanging array elements. Nevertheless, selection differs from exchange sorting because at each step we *select* the next element of the sorted sequence from the remaining elements and then we move it into its final position in the array by exchanging it with whatever happens to be occupying that position.

-
- [Straight Selection Sorting](#)
 - [Sorting with a Heap](#)
 - [Building the Heap](#)
-

Straight Selection Sorting

The simplest of the selection sorts is called *straight selection*. Figure □ illustrates how straight selection works. In the version shown, the sorted list is constructed from the right (i.e., from the largest to the smallest element values).

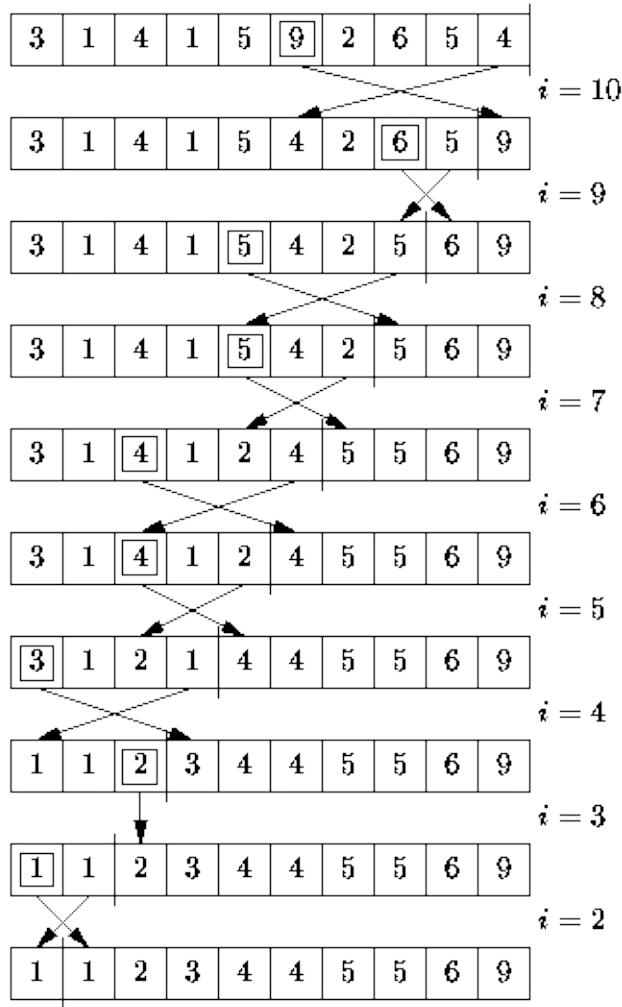


Figure: Straight selection sorting.

At each step of the algorithm, a linear search of the unsorted elements is made in order to determine the position of the largest remaining element. That element is then moved into the correct position of the array by swapping it with the element which currently occupies that position.

For example, in the first step shown in Figure □, a linear search of the entire

array reveals that 9 is the largest element. Since 9 is the largest element, it belongs in the last array position. To move it there, we swap it with the 4 that initially occupies that position. The second step of the algorithm identifies 6 as the largest remaining element and moves it next to the 9. Each subsequent step of the algorithm moves one element into its final position. Therefore, the algorithm is done after $n-1$ such steps.

- [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Program 1 defines the `StraightSelectionSorter` class. This class is derived from the abstract `Sorter` class defined in Program 1 and it provides an implementation for the `_sort` method. The `_sort` method follows directly from the algorithm discussed above. In each iteration of the main loop (lines 8-14), exactly one element is selected from the unsorted elements and moved into the correct position. A linear search of the unsorted elements is done in order to determine the position of the largest remaining element (lines 9-12). That element is then moved into the correct position (line 13).

```

1  class StraightSelectionSorter(Sorter):
2
3      def __init__(self):
4          super(StraightSelectionSorter, self).__init__()
5
6      def _sort(self):
7          i = self._n
8          while i > 1:
9              max = 0
10             for j in xrange(i):
11                 if self._array[j] > self._array[max]:
12                     max = j
13             self.swap(i - 1, max)
14             i -= 1

```

Program: `StraightSelectionSorter` class `__init__` and `_sort` methods.

In all $n-1$ iterations of the outer loop are needed to sort the array. Notice that exactly one swap is done in each iteration of the outer loop. Therefore, $n-1$ data exchanges are needed to sort the list.

Furthermore, in the i^{th} iteration of the outer loop, $i-1$ iterations of the inner loop are required and each iteration of the inner loop does one data comparison.

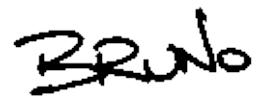
Therefore, $O(n^2)$ data comparisons are needed to sort the list.

The total running time of the straight selection `_sort` method is $O(n^2)$. Because

the same number of comparisons and swaps are always done, this running time bound applies in all cases. That is, the best-case, average-case and worst-case running times are all $O(n^2)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Sorting with a Heap

Selection sorting involves the repeated selection of the next element in the sorted sequence from the set of remaining elements. For example, the straight insertion sorting algorithm given in the preceding section builds the sorted sequence by repeatedly selecting the largest remaining element and prepending it to the sorted sequence developing at the right end of the array.

At each step the largest remaining element is withdrawn from the set of remaining elements. A linear search is done because the order of the remaining elements is arbitrary. However, if we consider the value of each element as its priority, we can view the set of remaining elements as a priority queue. In effect, a selection sort repeatedly dequeues the highest priority element from a priority queue.

Chapter 1 presents a number of priority queue implementations, including binary heaps, leftist heaps and binomial queues. In this section we present a version of selection sorting that uses a *binary heap* to hold the elements that remain to be sorted. Therefore, it is called a *heapsort*. The principal advantage of using a binary heap is that it is easily implemented using an array and the entire sort can be done in place.

As explained in Section 1, a binary heap is a *complete binary tree* which is easily represented in an array. The n nodes of the heap occupy positions 1 through n of the array. The root is at position 1. In general, the children of the node at position i of the array are found at positions $2i$ and $2i+1$, and the parent is found at position $\lfloor i/2 \rfloor$.

The heapsort algorithm consists of two phases. In the first phase, the unsorted array is transformed into a heap. (This is called *heapifying* the array). In this case, a *max-heap* rather than a min-heap is used. The data in a max heap satisfies the following condition: For every node in the heap that has a parent, the item contained in the parent is greater than or equal to the item contained in the given node.

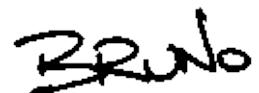
The second phase of heapsort builds the sorted list. The sorted list is built by

repeatedly selecting the largest element, withdrawing it from the heap, and adding it to the sorted sequence. As each element is withdrawn from the heap, the remaining elements are heapified.

- [Implementation](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

In the first phase of heapsort, the unsorted array is transformed into a max heap. Throughout the process we view the array as a complete binary tree. Since the data in the array is initially unsorted, the tree is not initially heap-ordered. We make the tree into a max heap from the bottom up. That is, we start with the leaves and work towards the root. Figure □ illustrates this process.

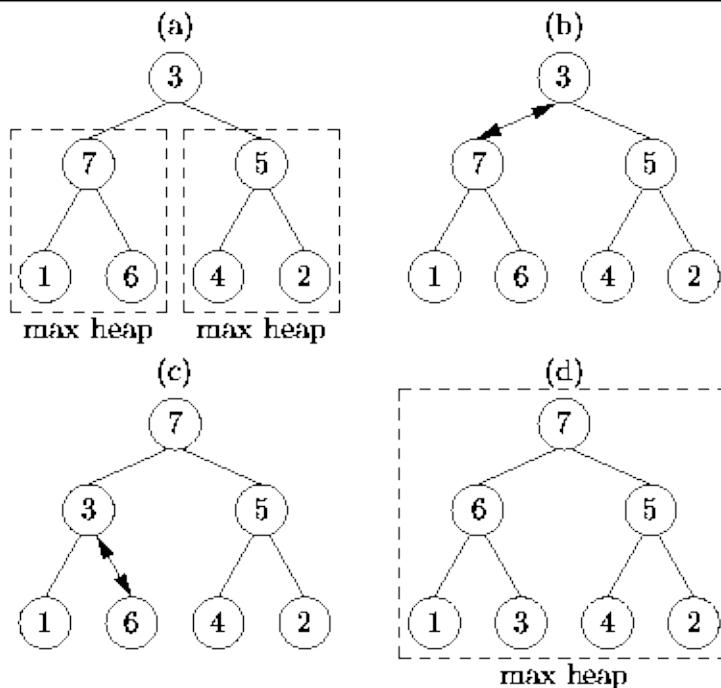


Figure: Combining heaps by percolating values.

Figure □ (a) shows a complete tree that is not yet heap ordered--the root is smaller than both its children. However, the two subtrees of the root *are* heap ordered. Given that both of the subtrees of the root are already heap ordered, we can heapify the tree by *percolating* the value in the root down the tree.

To percolate a value down the tree, we swap it with its largest child. For example, in Figure □ (b) we swap 3 and 7. Swapping with the largest child ensures that after the swap, the new root is greater than or equal to *both* its children.

Notice that after the swap the heap-order is satisfied at the root, but not in the left subtree of the root. We continue percolating the 3 down by swapping it with 6 as shown in Figure □(c). In general, we percolate a value down either until it arrives in a position in which the heap order is satisfied or until it arrives in a leaf. As shown in Figure □(d), the tree obtained when the percolation is finished is a max heap

Program □ introduces the `HeapSorter` class. The `HeapSorter` class extends the abstract `Sorter` class defined in Program □. The `percolateDown` method shown in Program □ implements the algorithm described above. In addition to `self`, the `percolateDown` method takes two arguments: the number of elements in the array to be considered, $n = \text{length}$, and the position, i , of the node to be percolated.

```

1  class HeapSorter(Sorter):
2
3      def __init__(self):
4          super(HeapSorter, self).__init__()
5
6
7      def percolateDown(self, i, length):
8          while 2 * i <= length:
9              j = 2 * i
10             if j < length and self._array[j + 1] \
11                 > self._array[j]:
12                 j = j + 1
13             if self._array[i] \
14                 >= self._array[j]:
15                 break
16             self.swap(i, j)
17             i = j
18
19     # ...

```

Program: `HeapSorter` class `__init__` and `percolateDown` methods.

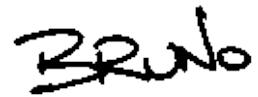
The purpose of the `percolateDown` method is to transform the subtree rooted at position i into a max heap. It is assumed that the left and right subtrees of the node at position i are already max heaps. Recall that the children of node i are found at positions $2i$ and $2i+1$. `percolateDown` percolates the value in position i down the tree by swapping elements until the value arrives in a leaf node or until both children of i contain smaller value.

A constant amount of work is done in each iteration. Therefore, the running time of the `percolateDown` method is determined by the number of iterations of its main loop (lines 8-17). In fact, the number of iterations required in the worst case is equal to the height in the tree of node i .

Since the root of the tree has the greatest height, the worst-case occurs for $i=1$. In Chapter \square it is shown that the height of a complete binary tree is $\lceil \log_2 n \rceil$. Therefore the worst-case running time of the `percolateDown` method is $O(\log n)$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Building the Heap

The `buildHeap` method shown in Program □ transforms an unsorted array into a max heap. It does so by calling the `percolateDown` method for $i = \lfloor n/2 \rfloor, \lfloor n/2 \rfloor - 1, \lfloor n/2 \rfloor - 2, \dots, 1$.

```
1 class HeapSorter(Sorter):
2
3     def buildHeap(self):
4         i = self._n / 2
5         while i > 0:
6             self.percolateDown(i, self._n)
7             i -= 1
8
9     # ...
```

Program: `HeapSorter` class `buildHeap` method.

Why does `buildHeap` start percolating at $\lfloor n/2 \rfloor$? A complete binary tree with n nodes has exactly $\lfloor n/2 \rfloor$ leaves. Therefore, the last node in the array which has a child is in position $\lfloor n/2 \rfloor$. Consequently, the `buildHeap` method starts doing percolate down operations from that point.

The `buildHeap` visits the array elements in reverse order. In effect the algorithm starts at the deepest node that has a child and works toward the root of the tree. Each array position visited is the root of a subtree. As each such subtree is visited, it is transformed into a max heap. Figure □ illustrates how the `buildHeap` method heapifies an array that is initially unsorted.

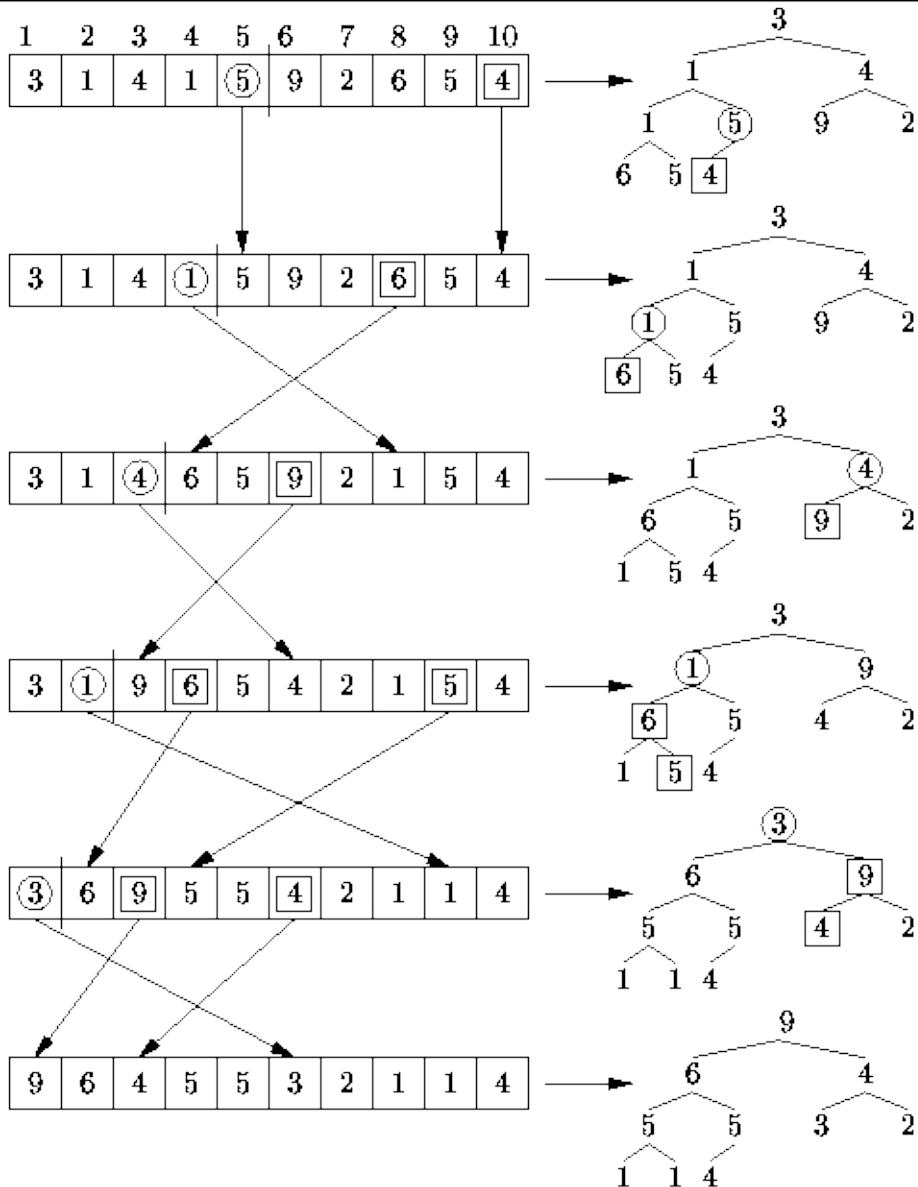


Figure: Building a heap.

- [Running Time Analysis](#)
 - [The Sorting Phase](#)
-

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Running Time Analysis

The `buildHeap` method does exactly $\lfloor \frac{n}{2} \rfloor$ `percolateDown` operations. As discussed above, the running time for `percolateDown` is $O(h_i)$, where h_i is the height in the tree of the node at array position i . The highest node in the tree is the root and its height is $O(\log n)$. If we make the simplifying assumption that the running time for `percolateDown` is $O(\log n)$ for every value of i , we get that the total running time for `buildHeap` is $O(n \log n)$.

However, $n \log n$ is not a tight bound. The maximum number of iterations of the `percolateDown` loop done during the entire process of building the heap is equal to the sum of the heights of all of the nodes in the tree! The following theorem shows that this is $O(n)$.

Theorem Consider a *perfect* binary tree T of height h having $n = 2^{h+1} - 1$ nodes. The sum of the heights of the nodes in T is $2^{h+1} - 1 - (h + 1) = n - \log_2(n + 1)$.

extbfProof A perfect binary tree has 1 node at height h , 2 nodes at height $h-1$, 4 nodes at height $h-2$ and so on. In general, there are 2^i nodes at height $h-i$.

Therefore, the sum of the heights of the nodes is $\sum_{i=0}^h (h - i)2^i$.

The summation can be solved as follows: First, we make the simple variable substitution $i=j-1$:

$$\begin{aligned}
\sum_{i=0}^h (h-i)2^i &= \sum_{j=1=0}^h (h-(j-1))2^{j-1} \\
&= \frac{1}{2} \sum_{j=1}^{h+1} (h-j+1)2^j \\
&= \frac{1}{2} \sum_{j=0}^h (h-j+1)2^j - (h+1)/2 \\
&= \frac{1}{2} \sum_{j=0}^h (h-j)2^j + \sum_{j=0}^h 2^j - (h+1)/2 \\
&= \frac{1}{2} \sum_{j=0}^h (h-j)2^j + (2^{h+1} - 1 - h + 1)/2
\end{aligned} \tag{15.9}$$

Note that the summation which appears on the right hand side is identical to that on the left. Rearranging Equation □ and simplifying gives:

$$\begin{aligned}
\sum_{i=0}^h (h-i)2^i &= 2^{h+1} - 1 - h + 1 \\
&= n - \log_2(n + 1).
\end{aligned}$$

It follows directly from Theorem □ that the sum of the heights of a perfect binary tree is $O(n)$. But a heap is not a *perfect* tree--it is a *complete* tree. Nevertheless, it is easy to show that the same bound applies to a complete tree. The proof is left as an exercise for the reader (Exercise □). Therefore, the running time for the buildHeap method is $O(n)$, where n is the length of the array to be heapified.



The Sorting Phase

Once the max heap has been built, heapsort proceeds to the selection sorting phase. In this phase the sorted sequence is obtained by repeatedly withdrawing the largest element from the max heap. Figure □ illustrates how this is done.

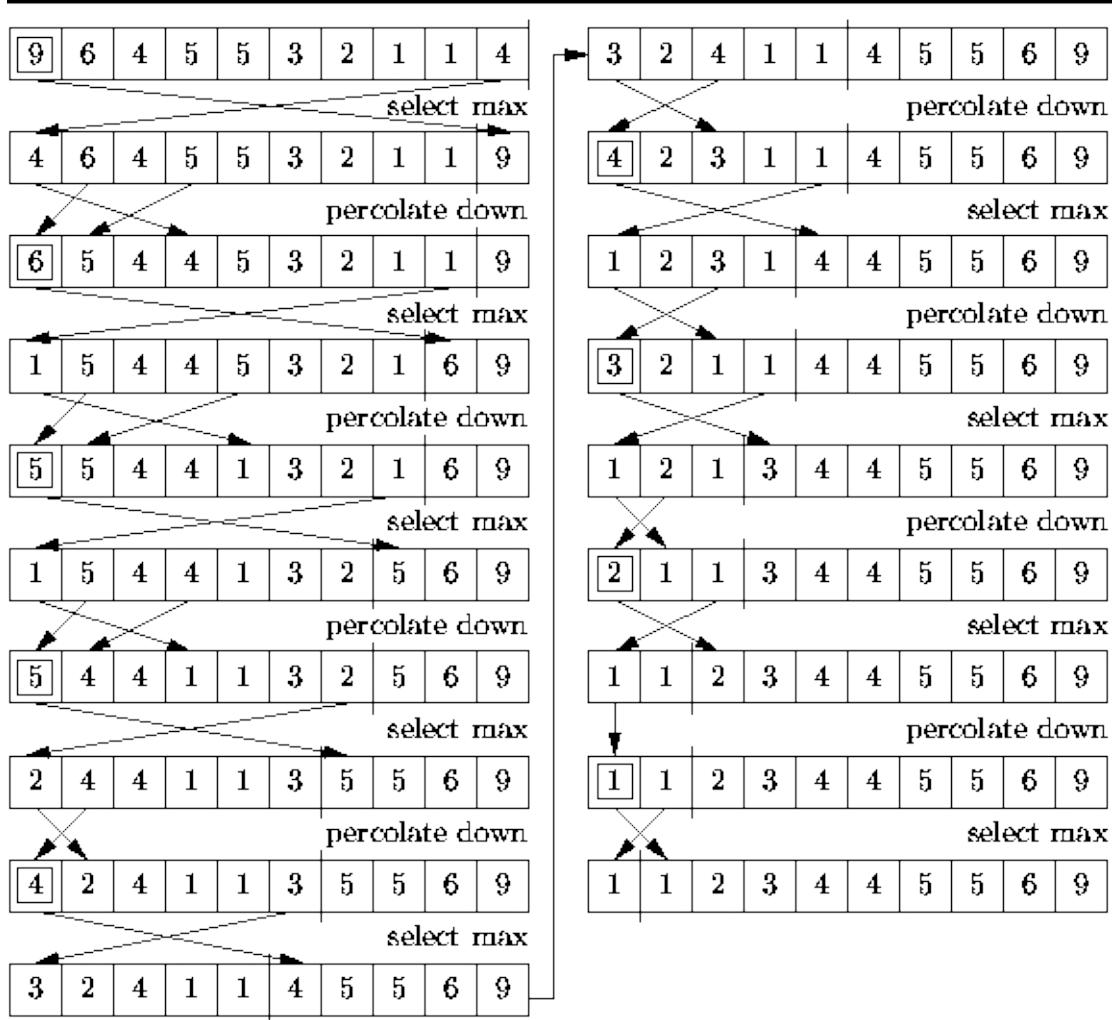


Figure: Heap sorting.

The largest element of the heap is always found at the root and the root of a complete tree is always in array position one. Suppose the heap occupies array

positions 1 through k . When an element is withdrawn from the heap, its length decreases by one. That is, after the withdrawal the heap occupies array positions 1 through $k-1$. Thus, array position k is no longer required by the max heap. However, the next element of the sorted sequence belongs in position k !

So, the sorting phase of heapsort works like this: We repeatedly swap the largest element in the heap (always in position 1) into the next position of the sorted sequence. After each such swap, there is a new value at the root of the heap and this new value is pushed down into the correct position in the heap using the `percolateDown` method.

Program □ gives the `_sort` method of the `HeapSorter` class. The `_sort` method embodies both phases of the heapsort algorithm. In the first phase of heapsort the `buildHeap` method is called to transform the array into a max heap. As discussed above, this is done in $O(n)$ time.

```
 1  class HeapSorter(Sorter):
 2
 3      def _sort(self):
 4          base = self._array.baseIndex
 5          self._array.baseIndex = 1
 6          self.buildHeap()
 7          i = self._n
 8          while i >= 2:
 9              self.swap(i, 1)
10              self.percolateDown(1, i - 1)
11              i -= 1
12          self._array.baseIndex = base
13
14      # ...
```

Program: `HeapSorter` class `_sort` method.

The second phase of the heapsort algorithm builds the sorted list. In all $n-1$ iterations of the loop on lines 8-11 are required. Each iteration involves one swap followed by a `percolateDown` operation. Since the worst-case running time for `percolateDown` is $O(\log n)$, the total running time of the loop is $O(n \log n)$. The running time of the second phase asymptotically dominates that of the first phase. As a result, the worst-case running time of heapsort is $O(n \log n)$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Merge Sorting

The fourth class of sorting algorithm we consider comprises algorithms that sort by *merging*. Merging is the combination of two or more sorted sequences into a single sorted sequence.

Figure \square illustrates the basic, two-way merge operation. In a two-way merge, two sorted sequences are merged into one. Clearly, two sorted sequences each of length n can be merged into a sorted sequence of length $2n$ in $O(2n)=O(n)$ steps. However in order to do this, we need space in which to store the result. That is, it is not possible to merge the two sequences *in place* in $O(n)$ steps.

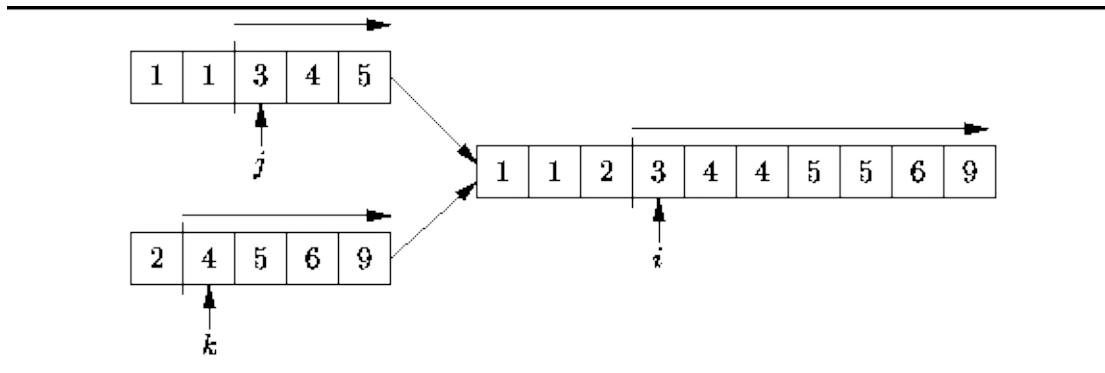


Figure: Two-way merging.

Sorting by merging is a recursive, divide-and-conquer strategy. In the base case, we have a sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done. To sort a sequence of $n > 1$ elements:

1. Divide the sequence into two sequences of length $\lceil n/2 \rceil$ and $\lceil n/2 \rceil$;
2. recursively sort each of the two subsequences; and then,
3. merge the sorted subsequences to obtain the final result.

Figure \square illustrates the operation of the two-way merge sort algorithm.

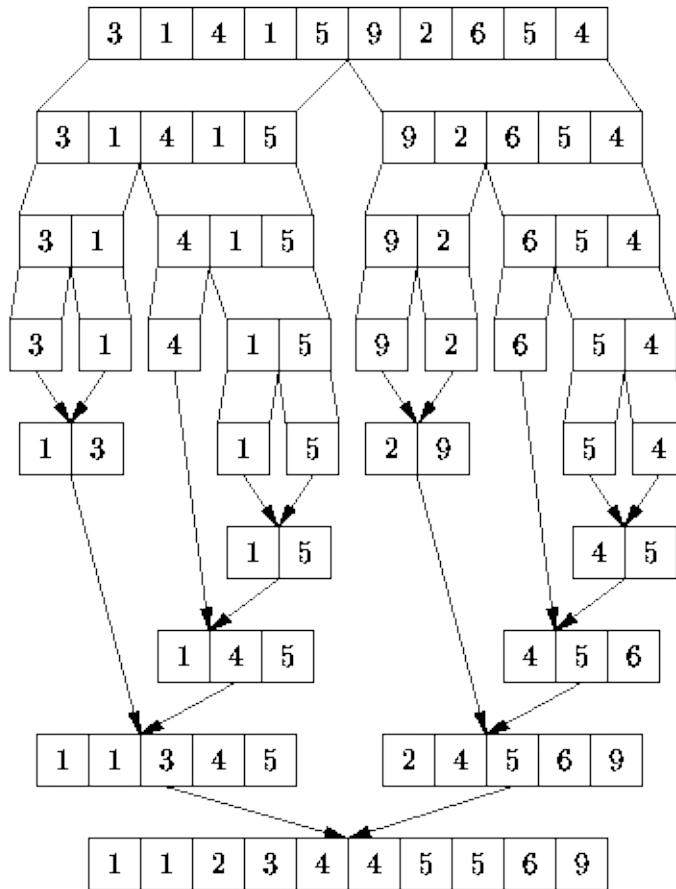


Figure: Two-way merge sorting.

- [Implementation](#)
- [Merging](#)
- [Two-Way Merge Sorting](#)
- [Running Time Analysis](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

Program □ declares the TwoWayMergeSorter class. The TwoWayMergeSorter class extends the abstract Sorter class defined in Program □. A single instance attribute, _tempArray, is declared. Since merge operations cannot be done in place, a second, temporary array is needed. The _tempArray instance attribute is used to keep track of that array.

```
1  class TwoWayMergeSorter(Sorter):
2
3      def __init__(self):
4          super(TwoWayMergeSorter, self).__init__()
5          self._tempArray = None
6
7      # ...
```

Program: TwoWayMergeSorter __init__ method.



Merging

The `merge` method of the `TwoWayMergeSorter` class is defined in Program □. In addition to `self`, this method takes three integer parameters, `left`, `middle`, and `right`. It is assumed that

$$\text{left} \leq \text{middle} < \text{right}.$$

Furthermore, it is assumed that the two subsequences of the array,

`array[left], array[left + 1], …, array[middle],`

and

`array[middle + 1], array[middle + 2], …, array[right],`

are both sorted. The `merge` method merges the two sorted subsequences using the temporary array, `_tempArray`. It then copies the merged (and sorted) sequence into the array at

`array[left], array[left + 1], …, array[right].`

```
1 class TwoWayMergeSorter(Sorter):
2
3     def merge(self, left, middle, right):
4         i = left
5         j = left
6         k = middle + 1
7         while j <= middle and k <= right:
8             if self._array[j] < self._array[k]:
9                 self._tempArray[i] = self._array[j]
10                i += 1
11                j += 1
12            else:
13                self._tempArray[i] = self._array[k]
14                i += 1
15                k += 1
16            while j <= middle:
17                self._tempArray[i] = self._array[j]
18                i += 1
19                j += 1
20            for i in xrange(left, k):
21                self._array[i] = self._tempArray[i]
22
23     # ...
```

Program: TwoWayMergeSorter class merge method.

In order to determine the running time of the merge method it is necessary to recognize that the total number of iterations of the two loops (lines 7-15, lines 16-19) is $\frac{right - left + 1}{2}$, in the worst case. The total number of iterations of the third loop (lines 20-21) is the same. Since all the loop bodies do a constant amount of work, the total running time for the merge method is $O(n)$, where $n = right - left + 1$ is the total number of elements in the two subsequences that are merged.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Two-Way Merge Sorting

Program □ gives the code for the `_sort` and `mergesort` methods of the `TwoWayMergeSorter` class. The `_sort` method sets things up for the recursive `mergesort` method. First, it allocates a temporary array, the length of which is equal to the length of the array to be sorted (line 4). Then it calls the recursive `mergesort` method which sorts the array (line 8). After the array has been sorted, the `_sort` method discards the temporary array (line 6).

```
 1  class TwoWayMergeSorter(Sorter):
 2
 3      def _sort(self):
 4          self._tempArray = Array(self._n)
 5          self.mergesort(0, self._n - 1)
 6          self._tempArray = None
 7
 8      def mergesort(self, left, right):
 9          if left < right:
10              middle = (left + right) / 2
11              self.mergesort(left, middle)
12              self.mergesort(middle + 1, right)
13              self.merge(left, middle, right)
14
15      # ...
```

Program: `TwoWayMergeSorter` class `_sort` and `mergesort` methods.

The `mergesort` method implements the recursive, divide-and-conquer merge sort algorithm described above. In addition to `self`, the `mergesort` method takes two parameters, `left` and `right`, that specify the subsequence of the array to be sorted. If the sequence to be sorted contains more than one element, the sequence is split in two (line 10), each half is recursively sorted (lines 11-12), and then two sorted halves are merged (line 13).

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Running Time Analysis

The running time of merge sort is determined by the running time of the recursive `mergesort` method. (The `_sort` method adds only a constant amount of overhead). The running time of the recursive `mergesort` method is given by the following recurrence:

$$T(n) = \begin{cases} O(1) & n = 1, \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & n > 1, \end{cases} \quad (15.10)$$

where $n = \text{right} - \text{left} + 1$.

In order to simplify the solution of Equation □ we shall assume that $n = 2^k$ for some integer $k \geq 0$. Dropping the $O(\cdot)$'s from the equation we get

$$T(n) = \begin{cases} 1 & n = 1, \\ 2T(n/2) + n & n > 1, \end{cases}$$

which is easily solved by repeated substitution:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \\ &= 2^k T(n/2^k) + kn \\ &\vdots \\ &= nT(1) + n \log_2 n \\ &= n + n \log_2 n \end{aligned}$$

Therefore, the running time of merge sort is $O(n \log n)$.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

A Lower Bound on Sorting

The preceding sections present three $O(n \log n)$ sorting algorithms--quicksort, heapsort, and the two-way merge sort. But is $O(n \log n)$ the best we can do? In this section we answer the question by showing that any sorting algorithm that sorts using only binary comparisons must make $\Omega(n \log n)$ such comparisons. If each binary comparison takes a constant amount of time, then running time for any such sorting algorithm is also $\Omega(n \log n)$.

Consider the problem of sorting the sequence $S = \{a, b, c\}$ comprised of three distinct items. That is, $a \neq b \wedge a \neq c \wedge b \neq c$. Figure \square illustrates a possible sorting algorithm in the form of a *decision tree*. Each node of the decision tree represents one binary comparison. That is, in each node of the tree, exactly two elements of the sequence are compared. Since there are exactly two possible outcomes for each comparison, each non-leaf node of the binary tree has degree two.

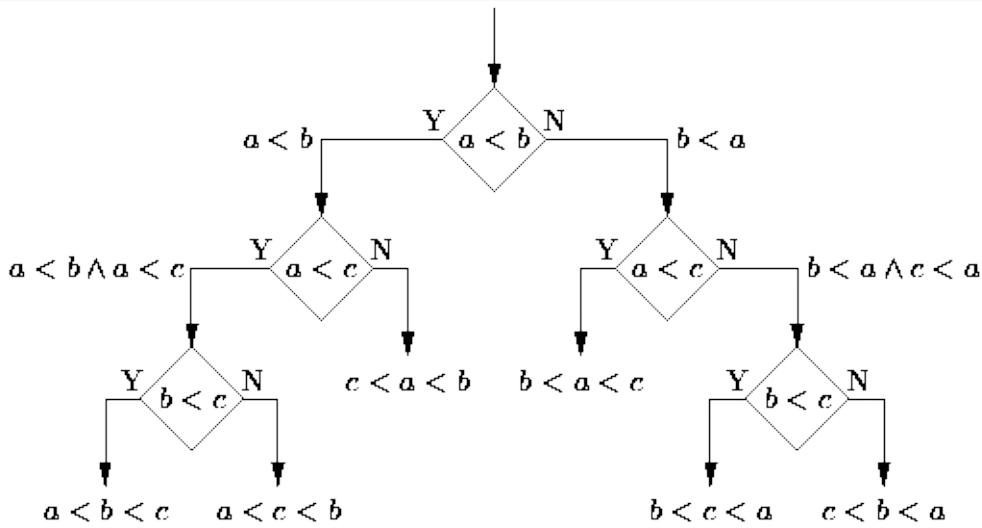


Figure: A decision tree for comparison sorting.

For example, suppose that $a < b < c$. Consider how the algorithm shown in Figure \square discovers this. The first comparison compares a and b which reveals that $a < b$. The second comparison compares a and c to find that $a < c$. At this point it has

been determined that $a < b$ and $a < c$ -- the relative order of b and c is not yet known. Therefore, one more comparison is required to determine that $b < c$. Notice that the algorithm shown in Figure □ works correctly in all cases because every possible permutation of the sequence S appears as a leaf node in the decision tree. Furthermore, the number of comparisons required in the worst case is equal to the height of the decision tree!

Any sorting algorithm that uses only binary comparisons can be represented by a binary decision tree. Furthermore, it is the height of the binary decision tree that determines the worst-case running time of the algorithm. In general, the size and shape of the decision tree depends on the sorting algorithm and the number of items to be sorted.

Given an input sequence of n items to be sorted, every binary decision tree that correctly sorts the input sequence must have *at least* $n!$ leaves--one for each permutation of the input. Therefore, it follows directly from Theorem □ that the height of the binary decision tree is *at least* $\lceil \log_2 n! \rceil$:

$$\begin{aligned} \lceil \log_2 n! \rceil &\geq \log_2 n! \\ &\geq \sum_{i=1}^n \log_2 i \\ &\geq \sum_{i=1}^{n/2} \log_2 n/2 \\ &\geq n/2 \log_2 n/2 \\ &= \Omega(n \log n). \end{aligned}$$

Since the height of the decision tree is $\Omega(n \log n)$, the number of comparisons done by any sorting algorithm that sorts using only binary comparisons is $\Omega(n \log n)$. Assuming each comparison can be done in constant time, the running time of any such sorting algorithm is $\Omega(n \log n)$.

Distribution Sorting

The final class of sorting algorithm considered in this chapter consists of algorithms that sort *by distribution*. The unique characteristic of a distribution sorting algorithm is that it does *not* make use of comparisons to do the sorting.

Instead, distribution sorting algorithms rely on *a priori* knowledge about the universal set from which the elements to be sorted are drawn. For example, if we know *a priori* that the size of the universe is a small, fixed constant, say m , then we can use the bucket sorting algorithm described in Section □.

Similarly, if we have a universe the elements of which can be represented with a small, finite number of bits (or even digits, letters, or symbols), then we can use the radix sorting algorithm given in Section □.

-
- [Bucket Sort](#)
 - [Radix Sort](#)
-

Bucket Sort

Bucket sort is possibly the simplest distribution sorting algorithm. The essential requirement is that the size of the universe from which the elements to be sorted are drawn is a small, fixed constant, say m .

For example, suppose that we are sorting elements drawn from $\{0, 1, \dots, m - 1\}$, i.e., the set of integers in the interval $[0, m-1]$. Bucket sort uses m counters. The i^{th} counter keeps track of the number of occurrences of the i^{th} element of the universe. Figure □ illustrates how this is done.

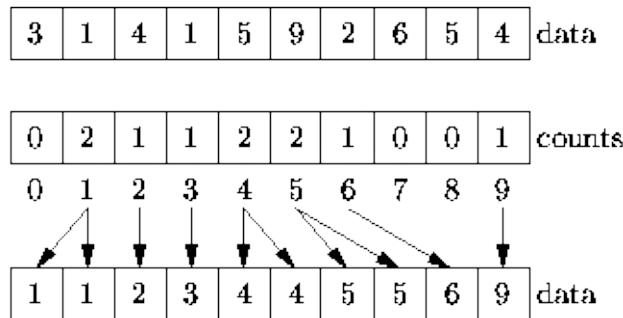


Figure: Bucket sorting.

In Figure □, the universal set is assumed to be $\{0, 1, \dots, 9\}$. Therefore, ten counters are required--one to keep track of the number of zeroes, one to keep track of the number of ones, and so on. A single pass through the data suffices to count all of the elements. Once the counts have been determined, the sorted sequence is easily obtained. For example, the sorted sequence contains no zeroes, two ones, one two, and so on.

- [Implementation](#)
-

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

Program □ introduces the `BucketSorter` class. The `BucketSorter` class extends the abstract `Sorter` class defined in Program □. This bucket sorter is designed to sort specifically an array `ints`. The `BucketSorter` class contains two instance attributes, `_m` and `_count`. The integer `_m` simply keeps track of the size of the universe. The `_count` variable is an array of integers used to count the number of occurrences of each element of the universal set.

```
1  class BucketSorter(Sorter):
2
3      def __init__(self, m):
4          super(BucketSorter, self).__init__()
5          self._m = m
6          self._count = Array(self._m)
7
8          # ...
9 }
```

Program: `BucketSorter` class `__init__` method.

In addition to `self`, the `__init__` method for the `BucketSorter` class takes a single argument which specifies the size of the universal set. The variable `_m` is set to the specified value, and the `_count` array is initialized to have the required size.

Program □ defines the `_sort` method. It begins by setting all of the counters to zero (lines 4-5). This can clearly be done in $O(m)$ time.

```

1 class BucketSorter(Sorter):
2
3     def _sort(self):
4         for i in xrange(self._m):
5             self._count[i] = 0
6         for j in xrange(self._n):
7             self._count[self._array[j]] += 1
8         j = 0
9         for i in xrange(self._m):
10            while self._count[i] > 0:
11                self._array[j] = i
12                j += 1
13                self._count[i] -= 1
14
15    # ...
16 }
```

Program: BucketSorter class `_sort` method.

Next, a single pass is made through the data to count the number of occurrences of each element of the universe (lines 6-7). Since each element of the array is examined exactly once, the running time is $O(n)$.

In the final step, the sorted output sequence is created (lines 8-13). Since the output sequence contains exactly n items, the body of the inner loop (lines 10-13) is executed exactly n times. During the i^{th} iteration of the outer loop (lines 9-13), the loop termination test of the inner loop (line 10) is evaluated $\text{count}[i] + 1$ times. As a result, the total running time of the final step is $O(m+n)$.

Thus, the running time of the bucket sort method is $O(m+n)$. Note that if $m=O(n)$, the running time for bucket sort is $O(n)$. That is, the bucket sort algorithm is a *linear-time* sorting algorithm! Bucket sort breaks the $\Omega(n \log n)$ bound associated with sorting algorithms that use binary comparisons because bucket sort does not do any binary comparisons. The cost associated with breaking the $\Omega(n \log n)$ running time bound is the $O(m)$ space required for the array of counters. Consequently, bucket sort is practical only for small m . For example, to sort 16-bit integers using bucket sort requires the use of an array of $2^{16} = 65\,536$ counters.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Radix Sort

This section presents a sorting algorithm known as *least-significant-digit-first radix sorting*. Radix sorting is based on the bucket sorting algorithm discussed in the preceding section. However, radix sorting is practical for much larger universal sets than it is practical to handle with a bucket sort.

Radix sorting can be used when each element of the universal set can be viewed as a sequences of digits (or letters or any other symbols). For example, we can represent each integer between 0 and 99 as a sequence of two, decimal digits. (For example, the number five is represented as ``05").

To sort an array of two-digit numbers, the algorithm makes two sorting passes through the array. In the first pass, the elements of the array are sorted by the *least significant* decimal digit. In the second pass, the elements of the array are sorted by the *most significant* decimal digit. The key characteristic of the radix sort is that the second pass is done in such a way that it does not destroy the effect of the first pass. Consequently, after two passes through the array, the data is contained therein is sorted.

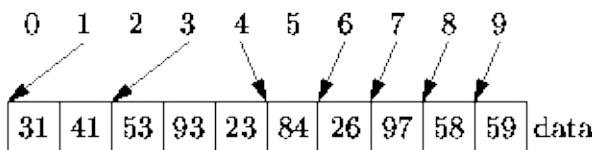
Each pass of the radix sort is implemented as a bucket sort. In the example we base the sort on *decimal* digits. Therefore, this is called a *radix-10* sort and ten buckets are required to do each sorting pass.

Figure □ illustrates the operation of the radix-10 sort. The first radix sorting pass considers the least significant digits. As in the bucket sort, a single pass is made through the unsorted data, counting the number of times each decimal digit appears as the least-significant digit. For example, there are no elements that have a 0 as the least-significant digit; there are two elements that have a 1 as the least-significant digit; and so on.

31	41	59	26	53	58	97	93	23	84
data									

0	1	2	3	4	5	6	7	8	9
0	2	0	3	1	0	1	1	1	1

0	0	2	2	5	6	6	7	8	9
0	0	2	2	5	6	6	7	8	9



0	1	2	3	4	5	6	7	8	9
0	0	2	1	1	3	0	0	1	2

0	0	0	2	3	4	7	7	7	8
0	0	0	2	3	4	7	7	7	8

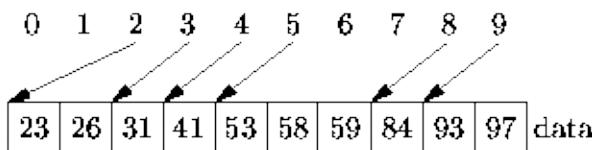


Figure: Radix sorting.

After the counts have been determined, it is necessary to permute the input sequence so that it is sorted by the least-significant digits. To do this permutation efficiently, we compute the sequence of *offsets* given by

$$\text{offset}[i] = \begin{cases} 0 & i = 0, \\ \sum_{j=0}^{i-1} \text{count}[j] & 0 < i < R, \end{cases} \quad (15.11)$$

where R is the sorting radix. Note that $\text{offset}[i]$ is the position in the permuted sequence of the first occurrence of an element whose least significant digit is i . By making use of the offsets, it is possible to permute the input sequence by making a single pass through the sequence.

The second radix sorting pass considers the most significant digits. As above a single pass is made through the permuted data sequence counting the number of times each decimal digit appears as the most-significant digit. Then the sequence of *offsets* is computed as above. The sequence is permuted again using the offsets producing the final, sorted sequence.

In general, radix sorting can be used when the elements of the universe can be viewed as p -digit numbers with respect to some radix, R . That is, each element of the universe has the form

$$\sum_{i=0}^{p-1} d_i R^i,$$

where $d_i \in \{0, 1, \dots, R-1\}$ for $0 \leq i < p$. In this case, the radix sort algorithm must make p sorting passes from the least significant digit, d_0 , to the most significant digit, d_{p-1} , and each sorting pass uses exactly R counters.

Radix sorting can also be used when the universe can be viewed as the cross-product of a finite number of finite sets. That is, when the universe has the form

$$U = U_1 \times U_2 \times U_3 \times \dots \times U_p,$$

where $p > 0$ is a fixed integer constant and U_i is a finite set for $1 \leq i \leq p$. For example, each card in a 52-card deck of playing cards can be represented as an element of $U = U_1 \times U_2$, where $U_1 = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ and $U_2 = \{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$.

Before we can sort over the universe U , we need to define what it means for one element to precede another in U . The usual way to do this is called *lexicographic ordering*. For example in the case of the playing cards we may say that one card precedes another if its suit precedes the other suit or if the suits are equal but the face value precedes that of the other.

In general, given the universe $U = U_1 \times U_2 \times U_3 \times \dots \times U_p$, and two elements of U , say x and y , represented by the p -tuples $x = (x_1, x_2, \dots, x_p)$ and $y = (y_1, y_2, \dots, y_p)$, respectively, we say that x *lexicographically precedes* y if there exists $1 \leq k \leq p$ such that $x_k < y_k$ and $x_i = y_i$ for all $1 \leq i < k$.

With this definition of precedence, we can radix sort a sequence of elements drawn from U by sorting with respect to the components of the p -tuples.

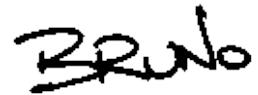
Specifically, we sort first with respect to U_p , then U_{p-1} , and so on down to U_1 . Notice that the algorithm does p sorting passes and in the i^{th} pass it requires $|U_i|$ counters. For example to sort a deck of cards, two passes are required. In first

pass the cards are sorted into 13 piles according to their face values. In the second pass the cards are sorted into four piles according to their suits.

- [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Program □ introduces the RadixSorter class. The RadixSorter class extends the abstract Sorter class defined in Program □. This radix sorter is designed to sort specifically an array of ints.

```

1  class RadixSorter(Sorter):
2
3      r = 8
4      R = 1 << r
5      p = (32 + r - 1) / r
6
7      def __init__(self):
8          self._count = Array(self.R)
9          self._tempArray = None
10
11      # ...

```

Program: RadixSorter class `__init__` method.

Three constants are declared in the RadixSorter class--R, r, and p. The constant R represents the radix and $r = \log_2 R$. The constant p is the number sorting passes needed to sort the data. In this case $r=8$ and $R = 2^r = 256$. Therefore, a radix-256 sort is being done. We have chosen R as a power of two because that way the computations required to implement the radix sort can be implemented efficiently using simple bit shift and mask operations. In order to sort b -bit integers, it is necessary to make $p = \lceil \log_R 2^b \rceil = \lceil b/r \rceil$ sorting passes.

Two instance attributes are defined in the RadixSorter class--`_count` and `_tempArray`. The `_count` instance attribute is an array of integers used to implement the sorting passes. An array of integers of length R is created and assigned to the `_count` array. The `_tempArray` instance attribute is used for temporary storage in the `_sort` method.

The `_sort` method shown in Program □ begins by creating a temporary array of length n . Each iteration of the main loop corresponds to one pass of the radix sort (lines 5-21). In all p iterations are required.

```

1 class RadixSorter(Sorter):
2
3     def _sort(self):
4         self._tempArray = Array(self._n)
5         for i in xrange(self.p):
6             for j in xrange(self.R):
7                 self._count[j] = 0
8             for k in xrange(self._n):
9                 self._count[(self._array[k] \
10                   >> (self.r*i)) & (self.R-1)] += 1
11             self._tempArray[k] = self._array[k]
12         pos = 0
13         for j in xrange(self.R):
14             tmp = pos
15             pos += self._count[j]
16             self._count[j] = tmp
17         for k in xrange(self._n):
18             j = (self._tempArray[k] \
19                   >> (self.r*i)) & (self.R-1)
20             self._array[self._count[j]] = self._tempArray[k]
21             self._count[j] += 1
22
23     # ...

```

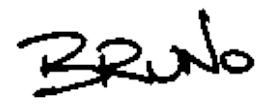
Program: RadixSorter class `_sort` method.

During the i^{th} pass of the main loop the following steps are done: First, the R counters are all set to zero (lines 6-7). This takes $O(R)$ time. Then a pass is made through the input array during which the number of occurrences of each radix- R digit in the i^{th} digit position are counted (lines 8-11). This pass takes $O(n)$ time. Notice that during this pass all the input data is copied into the temporary array.

Next, the array of counts is transformed into an array of offsets according to Equation \square . This requires a single pass through the counter array (lines 12-16). Therefore, it takes $O(R)$ time. Finally, the data sequence is permuted by copying the values from the temporary array back into the input array (lines 17-21). Since this requires a single pass through the data arrays, the running time is $O(n)$.

After the p sorting passes have been done, the array of data is sorted. The running time for the `_sort` method of the `RadixSorter` class is $O(p(R + n))$. If we assume that the size of an integer is 32 bits and given that $R=256$, the number of sorting passes required is $p=4$. Therefore, the running time for the radix sort is simply $O(n)$. That is, radix sort is a linear-time sorting algorithm.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Performance Data

In order to better understand the actual performance of the various sorting algorithms presented in this chapter, it is necessary to conduct some experiments. Only by conducting experiments is it possible to determine the relative performance of algorithms with the same asymptotic running time.

To measure the performance of a sorting algorithm, we need to provide it with some data to sort. To obtain the results presented here, random sequences of integers were sorted. That is, for each value of n , the `RandomNumberGenerator` class defined in Section 1 was used to create a sequence of n integers. In all cases (except for bucket sort) the random numbers are uniformly distributed in the interval $[1, 2^{31} - 1]$. For the bucket sort the numbers are uniformly distributed in $[0, 2^{10} - 1]$.

Figures 1, 2 and 3 show the actual running times of the sorting algorithms presented in this chapter. These running times were measured on an Intel Pentium III, which has a 1 GHz clock and 256MB RAM under the Red Hat Linux 7.1 operating system. The programs were executed using the Python version 2.2.3 interpreter. The times shown are elapsed CPU times, measured in seconds.

Figure 1 shows the running times of the $O(n^2)$ sorts for sequences of length n , $10 \leq n \leq 2000$. Notice that the bubble sort has the worst performance and that the straight selection sort has the best performance. Figure 2 clearly shows that, as predicted, binary insertion is better than straight insertion. Notice too that all of the $O(n^2)$ sorts require more than 25 seconds of execution time to sort an array of 2000 integers.

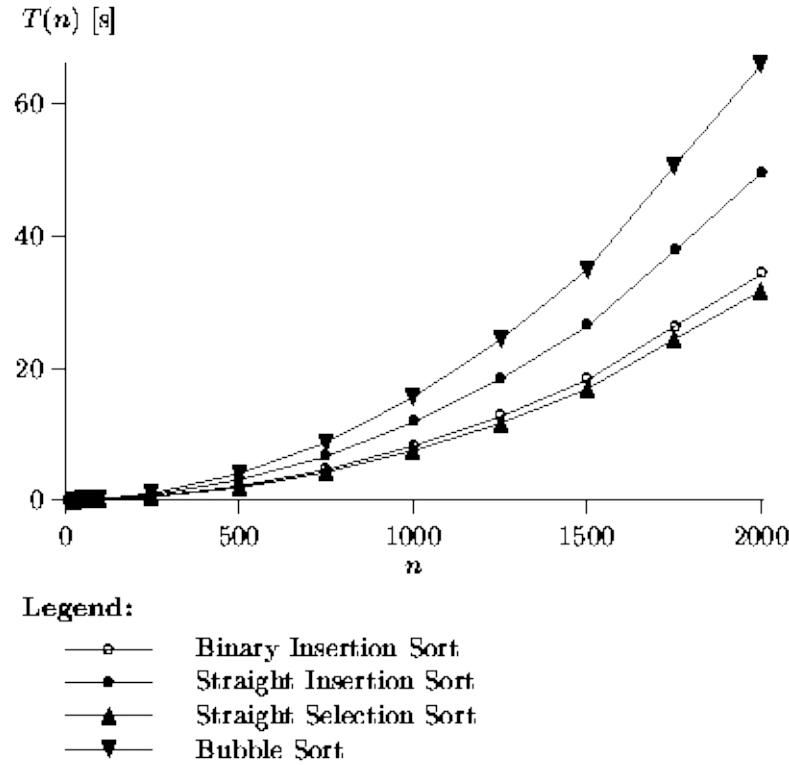


Figure: Actual running times of the $\mathcal{O}(n^2)$ sorts.

The performance of the $\mathcal{O}(n \log n)$ sorts is shown in Figure □. In this case, the length of the sequence varies between $n=10$ and $n = 10\,000$. The graph clearly shows that the $\mathcal{O}(n \log n)$ algorithms are significantly faster than the $\mathcal{O}(n^2)$ ones. All three algorithms sort 10000 integers in under 8 seconds. Quicksort is clear the best of the three.

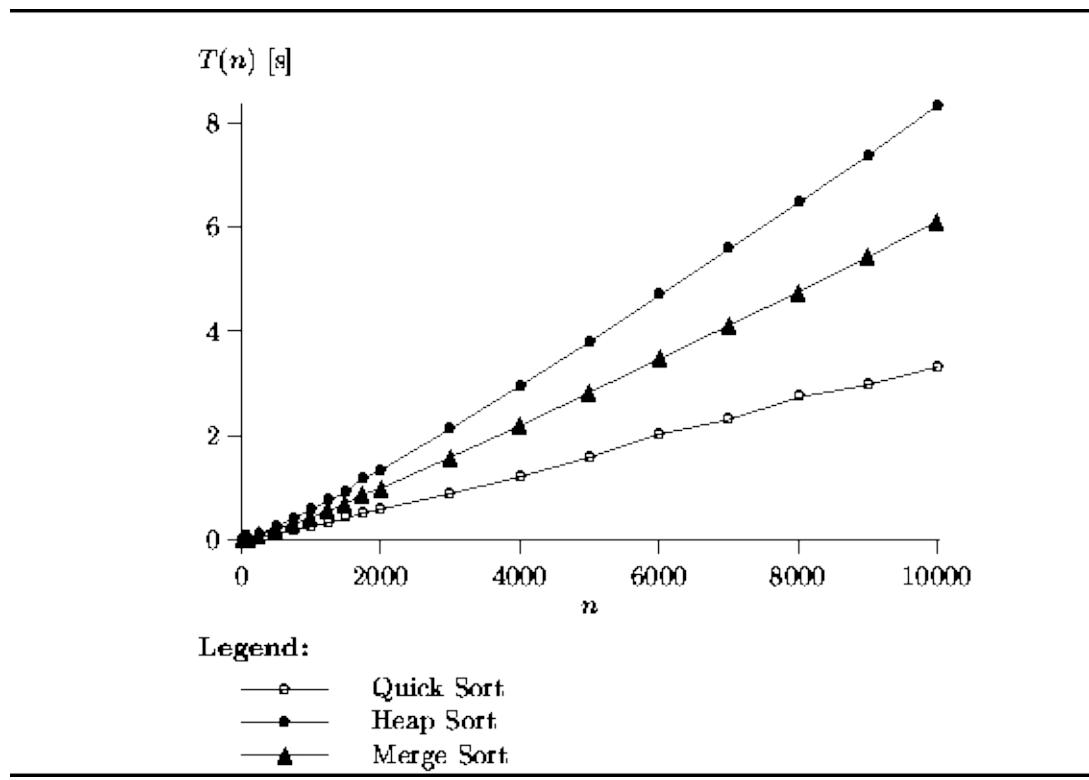


Figure: Actual running times of the $O(n \log n)$ sorts.

Figure □ shows the actual running times for the bucket sort and radix sort algorithms. Both these algorithms were shown to be $O(n)$ sorts. The graph shows results for n between 10 and 100 000. The universe used to test bucket sort was $\{0, 1, \dots, 1023\}$. That is, a total of $m=1024$ counters (buckets) were used. For the radix sort, 32-bit integers were sorted by using the radix $R=256$ and doing $p=4$ sorting passes.

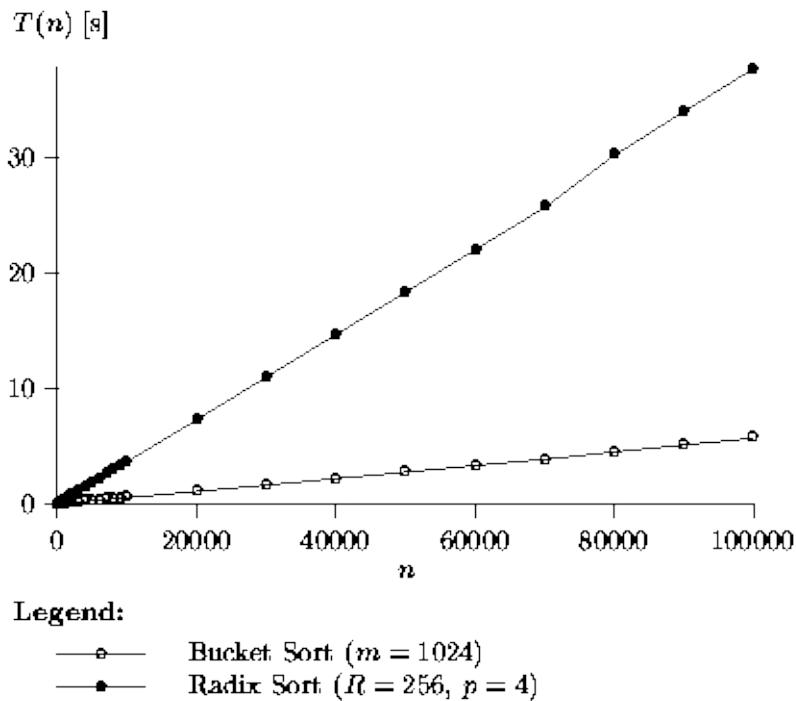


Figure: Actual running times of the $O(n)$ sorts.

Clearly, the bucket sort has the better running time. For example, it sorts 100000 10-bit integers in under 5 seconds. Radix sort performs extremely well too. It sorts 100000 32-bit integers in about 35 seconds. Given that the radix sort makes four passes through the data set, we can expect that it will be at least four times slower than the bucket sort.

Exercises

1. Consider the sequence of integers

$$S = \{8, 9, 7, 9, 3, 2, 3, 8, 4, 6\}.$$

For each of the following sorting algorithms, draw a sequence of diagrams that traces the execution of the algorithm as it sorts the sequence S : straight insertion sort, binary insertion sort, bubble sort, quick sort, straight selection sort, heapsort, merge sort, and bucket sort.

2. Draw a sequence of diagrams that traces the execution of a radix-10 sort of the sequence

$$S = \{89, 79, 32, 38, 46, 26, 43, 38, 32, 79\}.$$

3. For each of the sorting algorithms listed in Exercises 1 and 2 indicate whether the sorting algorithm is *stable*.
4. Consider a sequence of three distinct keys $\{a, b, c\}$. Draw the binary decision tree that represents each of the following sorting algorithms: straight insertion sort, straight selection sort, and bubble sort.
5. Devise an algorithm to sort a sequence of exactly five elements. Make your algorithm as efficient as possible.
6. Prove that the swapping of a pair of adjacent elements removes at most one inversion from a sequence.
7. Consider the sequence of elements $\{s_1, s_2, \dots, s_n\}$. What is the maximum number of inversions that can be removed by the swapping of a pair of distinct elements s_i and s_j ? Express the result in terms of the *distance* between s_i and s_j : $d(s_i, s_j) = j - i + 1$.
8. Devise a sequence of keys such that *exactly* eleven inversions are removed by the swapping of one pair of elements.
9. Prove that *binary insertion sort* requires $O(n \log n)$ comparisons.
10. Consider an arbitrary sequence $\{s_1, s_2, \dots, s_n\}$. To sort the sequence, we determine the permutation $\{p_1, p_2, \dots, p_n\}$ such that

$$s_{p_1} \leq s_{p_2} \leq \dots \leq s_{p_n}.$$

Prove that *bubble sort* requires at least p passes where

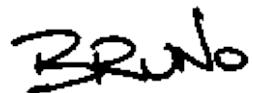
$$p = \max_{1 \leq i \leq n} (i - p_i).$$

11. Modify the bubble sort algorithm (Program \square) so that it terminates the outer loop when it detects that the array is sorted. What is the running time of the modified algorithm? **Hint:** See Exercise \square .
12. A variant of the bubble sorting algorithm is the so-called *odd-even transposition sort*. Like bubble sort, this algorithm makes a total of $n-1$ passes through the array. Each pass consists of two phases: The first phase compares $\text{array}[i]$ with $\text{array}[i + 1]$ and swaps them if necessary for all the odd values of i . The second phase does the same for the even values of i .
 1. Show that the array is guaranteed to be sorted after $n-1$ passes.
 2. What is the running time of this algorithm?
13. Another variant of the bubble sorting algorithm is the so-called *cocktail shaker sort*. Like bubble sort, this algorithm makes a total of $n-1$ passes through the array. However, alternating passes go in opposite directions. For example, during the first pass the largest item bubbles to the end of the array and during the second pass the smallest item bubbles to the beginning of the array.
 - Show that the array is guaranteed to be sorted after $n-1$ passes.
 - What is the running time of this algorithm?
14. Consider the following algorithm for selecting the k^{th} largest element from an unsorted sequence of n elements, $S = \{s_1, s_2, \dots, s_n\}$.
 1. If $n \leq 5$, sort S and select directly the k^{th} largest element.
 2. Otherwise $n > 5$: Partition the sequence S into subsequences of length five. In general, there will be $\lfloor n/5 \rfloor$ subsequences of length five and one of length $n \bmod 5$.
 3. Sort by any means each of the subsequences of length five. (See Exercise \square).
 4. Form the sequence $M = \{m_1, m_2, \dots, m_{\lfloor n/5 \rfloor}\}$ containing the $\lfloor n/5 \rfloor$ median values of each of the subsequences of length five.
 5. Apply the selection method recursively to find the median element of M . Let m be the median of the medians.
 6. Partition S into three subsequences, $S = \{L, E, G\}$, such that all the elements in L are less than m , all the elements in E are equal to m , and all the elements of G are greater than m .

7. If $k \leq |L|$ then apply the method recursively to select the k^{th} largest element of L ; if $|L| < k \leq |L| + |E|$, the result is m ; otherwise apply the method recursively to select the $(k - (|L| + |E|))^{\text{th}}$ largest element of G .
1. What is the running time of this algorithm?
 2. Show that if we use this algorithm to select the pivot the worst-case running time of *quick sort* is $O(n \log n)$.
15. Show that the sum of the heights of the nodes in a complete binary tree with n nodes altogether is $n-b(n)$, where $b(n)$ is the number of ones in the binary representation of n .

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Projects

1. Design and implement an algorithm that finds all the duplicates in a random sequence of keys.
2. Suppose we wish to sort a sequence of data represented using the linked-list class `LinkedList` introduced in Program □. Which of the sorting algorithms described in this chapter is the most appropriate for sorting a linked list? Design and implement a linked list sorter class that implements this algorithm.
3. Replace the `mergesort` method of the `MergeSorter` class with a non-recursive version. What is the running time of the non-recursive merge sort?
4. Replace the `quicksort` method of the `QuickSorter` class with a non-recursive version. What is the running time of the non-recursive quick sort?
Hint: Use a stack.
5. Design and implement a radix-sorter class that sorts an array of *strings*.
6. Design and implement a `RandomPivotQuickSorter` class that uses a random number generator (see Section □) to select a pseudorandom pivot. Run a sequence of experiments to compare the running times of random pivot selection with median-of-three pivot selection.
7. Design and implement a `MeanPivotQuickSorter` class that partitions the sequence to be sorted into elements that are less than the mean and elements that are greater than the mean. Run a sequence of experiments to compare the running times of the mean pivot quick sorter with median-of-three pivot selection.
8. Design and implement a `MedianPivotQuickSorter` class that uses the algorithm given in Exercise □ to select the median element for the pivot. Run a sequence of experiments to compare the running times of median pivot selection with median-of-three pivot selection.
9. Design and implement a sorter class that sorts using a `PriorityQueue` instance. (See Chapter □).

Bruno

Graphs and Graph Algorithms

A graph is simply a set of points together with a set of lines connecting various points. Myriad real-world application problems can be reduced to problems on graphs.

Suppose you are planning a trip by airplane. From a map you have determined the distances between the airports in the various cities that you wish to visit. The information you have gathered can be represented using a graph as shown in Figure □ (a). The points in the graph represent the cities and the lines represent the distances between them. Given such a graph, you can answer questions such as ``What is the shortest distance between LAX and JFK?'' or ``What is the shortest route that visits all of the cities?''

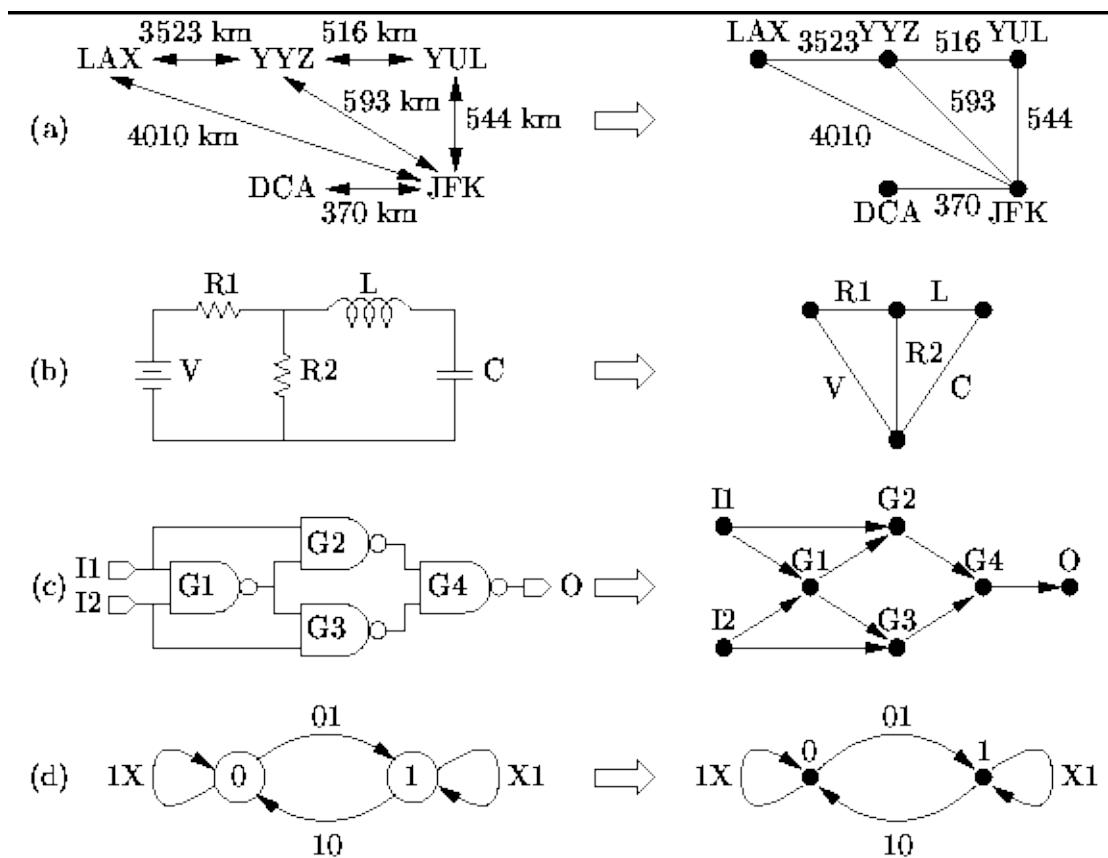


Figure: Real-world examples of graphs.

An electric circuit can also be viewed as a graph as shown in Figure □(b). In this case the points in the graph indicate where the components are connected (i.e., the wires) and the lines represent the components themselves (e.g, resistors and capacitors). Given such a graph, we can answer questions such as ``What are the mesh equations that describe the circuit's behavior?''

Similarly, a logic circuit can be reduced to a graph as shown in Figure □(c). In this case the logic gates are represented by the points and arrows represent the signal flows from gate outputs to gate inputs. Given such a graph, we can answer questions such as ``How long does it take for the signals to propagate from the inputs to the outputs?'' or ``Which gates are on the critical path?''

Finally, Figure □(d) illustrates that a graph can be used to represent a *finite state machine*. The points of the graph represent the states and labeled arrows indicate the allowable state transitions. Given such a graph, we can answer questions such as ``Are all the states reachable?'' or ``Can the finite state machine deadlock?''

This chapter is a brief introduction to the body of knowledge known as *graph theory*. It covers the most common data structures for the representation of graphs and introduces some fundamental graph algorithms.

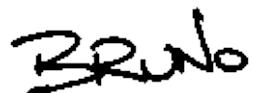
-
- [Basics](#)
 - [Implementing Graphs](#)
 - [Graph Traversals](#)
 - [Shortest-Path Algorithms](#)
 - [Minimum-Cost Spanning Trees](#)
 - [Application: Critical Path Analysis](#)
 - [Exercises](#)
 - [Projects](#)
-

Basics

- [Directed Graphs](#)
 - [Terminology](#)
 - [More Terminology](#)
 - [Directed Acyclic Graphs](#)
 - [Undirected Graphs](#)
 - [Terminology](#)
 - [Labeled Graphs](#)
 - [Representing Graphs](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Directed Graphs

We begin with the definition of a directed graph:

Definition (Directed Graph) A *directed graph*, or *digraph*, is an ordered pair $G = (\mathcal{V}, \mathcal{E})$ with the following properties:

1. The first component, \mathcal{V} , is a finite, non-empty set. The elements of \mathcal{V} are called the *vertices* of G .
2. The second component, \mathcal{E} , is a finite set of ordered pairs of vertices. That is, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The elements of \mathcal{E} are called the *edges* of G .

For example, consider the directed graph $G_1 = (\mathcal{V}_1, \mathcal{E}_1)$ comprised of four vertices and six edges:

$$\begin{aligned}\mathcal{V}_1 &= \{a, b, c, d\} \\ \mathcal{E}_1 &= \{(a, b), (a, c), (b, c), (c, a), (c, d), (d, d)\}.\end{aligned}$$

The graph G can be represented *graphically* as shown in Figure □. The vertices are represented by appropriately labeled circles, and the edges are represented by arrows that connect associated vertices.

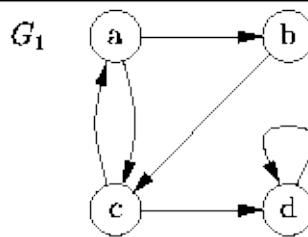
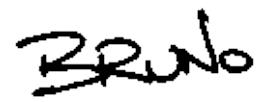


Figure: A directed graph.

Notice that because the pairs that represent edges are *ordered*, the two edges (a, c) and (c, a) are distinct. Furthermore, since \mathcal{E}_1 is a mathematical set, it cannot contain more than one instance of a given edge. And finally, an edge such as (d, d) may connect a node to itself.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Terminology

Consider a directed graph $G = (\mathcal{V}, \mathcal{E})$ as given by Definition □.

- Each element of \mathcal{V} is called a *vertex* or a *node* of G . Hence, \mathcal{V} is the set of *vertices* (or *nodes*) of G .
- Each element of \mathcal{E} is called an *edge* or an *arc* of G . Hence, \mathcal{E} is the set of *edges* (or *arcs*) of G .
- An edge $(v, w) \in \mathcal{E}$ can be represented as $v \rightarrow w$. An arrow that points from v to w is known as a *directed arc*. Vertex w is called the *head* of the arc because it is found at the arrow head. Conversely, v is called the *tail* of the arc. Finally, vertex w is said to be *adjacent* to vertex v .
- An edge $e=(v,w)$ is said to *emanate* from vertex v . We use notation $\mathcal{A}(v)$ to denote the set of edges emanating from vertex v . That is,

$$\mathcal{A}(v) = \{(v_0, v_1) \in \mathcal{E} : v_0 = v\}.$$
- The *out-degree* of a node is the number of edges emanating from that node. Therefore, the out-degree of v is $|\mathcal{A}(v)|$.
- An edge $e=(v,w)$ is said to be *incident* on vertex w . We use notation $\mathcal{I}(w)$ to denote the set of edges incident on vertex w . That is,

$$\mathcal{I}(w) = \{(v_0, v_1) \in \mathcal{E} : v_1 = w\}.$$
- The *in-degree* of a node is the number of edges incident on that node. Therefore, the in-degree of w is $|\mathcal{I}(w)|$.

For example, Table □ enumerates the sets of emanating and incident edges and the in- and out-degrees for each of the vertices in graph G_1 shown in Figure □.

Table: Emanating and incident edge sets for graph G_1 in Figure □.

vertex v	$\mathcal{A}(v)$	out-degree	$\mathcal{I}(v)$	in-degree
------------	------------------	------------	------------------	-----------

a	$\{(a, b), (a, c)\}$	2	$\{(c, a)\}$	1
b	$\{(b, c)\}$	1	$\{(a, b)\}$	1
c	$\{(c, a), (c, d)\}$	2	$\{(a, c), (b, c)\}$	2
d	$\{(d, d)\}$	1	$\{(c, d), (d, d)\}$	2

There is still more terminology to be introduced, but in order to do that, we need the following definition:

Definition (Path and Path Length)

A *path* in a directed graph $G = (\mathcal{V}, \mathcal{E})$ is a non-empty sequence of vertices

$$P = \{v_1, v_2, \dots, v_k\},$$

where $v_i \in \mathcal{V}$ for $1 \leq i \leq k$ such that $(v_i, v_{i+1}) \in \mathcal{E}$ for $1 \leq i < k$. The *length* of path P is $k-1$.

For example, consider again the graph G_1 shown in Figure \square . Among the paths contained in G_1 there is the path of length zero, $\{a\}$; the path of length one, $\{b, c\}$; the path of length two, $\{a, b, c\}$; and so on. In fact, this graph generates an infinite number of paths! (To see how this is possible, consider that $\{a, c, a, c, a, c, a, c, a, c, a\}$ is a path in G_1). Notice too the subtle distinction between a path of length zero, say $\{d\}$, and the path of length one $\{d, d\}$.



More Terminology

Consider the path $P = \{v_1, v_2, \dots, v_k\}$ in a directed graph $G = (\mathcal{V}, \mathcal{E})$.

- Vertex v_{i+1} is the *successor* of vertex v_i for $1 \leq i < k$. Each element v_i of path P (except the last) has a *successor*.
- Vertex v_{i-1} is the *predecessor* of vertex v_i for $1 < i \leq k$. Each element v_i of path P (except the first) has a *predecessor*.
- A path P is called a *simple* path if and only if $v_i \neq v_j$ for all i and j such that $1 \leq i < j \leq k$. However, it is permissible for v_1 to be the same as v_k in a simple path.
- A *cycle* is a path P of non-zero length in which $v_1 = v_k$. The *length of a cycle* is just the length of the path P .
- A *loop* is a cycle of length one. That is, it is a path of the form $\{v, v\}$.
- A *simple cycle* is a path that is both a *cycle* and *simple*.

Referring again to graph G_1 in Figure □ we find that the path $\{a, b, c, d\}$ is a simple path of length three. Conversely, the path $\{c, a, c, d\}$ also has length three but is not simple because vertex c occurs twice in the sequence (but not at the ends). The graph contains the path $\{a, b, c, a\}$ which is a cycle of length three, as well as $\{a, c, a, c, a\}$, a cycle of length four. The former is a simple cycle but the latter is not.



Directed Acyclic Graphs

For certain applications it is convenient to deal with graphs that contain no cycles. For example, a tree (see Chapter □) is a special kind of graph that contains no cycles.

Definition (Directed Acyclic Graph (DAG))

A *directed, acyclic graph* is a directed graph that contains no cycles.

Obviously, all trees are DAGs. However, not all DAGs are trees. For example consider the two directed, acyclic graphs, G_2 and G_3 , shown in Figure □. Clearly G_2 is a tree but G_3 is not.

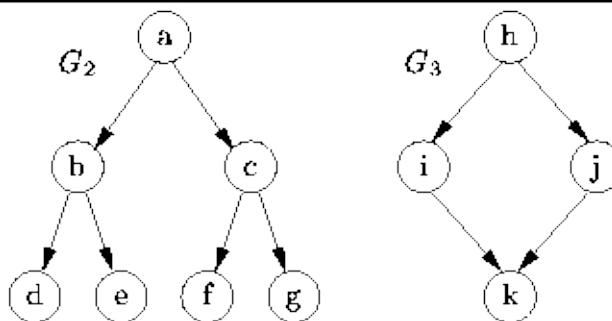


Figure: Two directed, acyclic graphs.

Undirected Graphs

An undirected graph is a graph in which the nodes are connected by *undirected arcs*. An undirected arc is an edge that has no arrow. Both ends of an undirected arc are equivalent--there is no head or tail. Therefore, we represent an edge in an undirected graph as a set rather than an ordered pair:

Definition (Undirected Graph) An *undirected graph* is an ordered pair $G = (\mathcal{V}, \mathcal{E})$ with the following properties:

1. The first component, \mathcal{V} , is a finite, non-empty set. The elements of \mathcal{V} are called the *vertices* of G .
2. The second component, \mathcal{E} , is a finite set of sets. Each element of \mathcal{E} is a set that is comprised of exactly two (distinct) vertices. The elements of \mathcal{E} are called the *edges* of G .

For example, consider the undirected graph $G_4 = (\mathcal{V}_4, \mathcal{E}_4)$ comprised of four vertices and four edges:

$$\begin{aligned}\mathcal{V}_4 &= \{a, b, c, d\} \\ \mathcal{E}_4 &= \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}\end{aligned}$$

The graph G_4 can be represented *graphically* as shown in Figure □. The vertices are represented by appropriately labeled circles, and the edges are represented by lines that connect associated vertices.

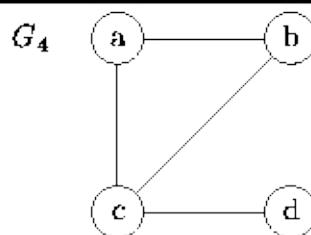


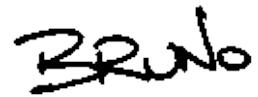
Figure: An undirected graph.

Notice that because an edge in an undirected graph is a set, $\{a, b\} \equiv \{b, a\}$, and

since \mathcal{E}_4 is also a set, it cannot contain more than one instance of a given edge. Another consequence of Definition \square is that there cannot be an edge from a node to itself in an undirected graph because an edge is a set of size two and a set cannot contain duplicates.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Terminology

Consider an undirected graph $G = (V, \mathcal{E})$ as given by Definition □.

- An edge $\{v, w\} \in \mathcal{E}$ emanates from and is *incident on* both vertices v and w .
 - The set of edges emanating from a vertex v is the set $\mathcal{A}(v) = \{(v_0, v_1) \in \mathcal{E} : v_0 = v \vee v_1 = v\}$. The set of edges incident on a vertex w is $\mathcal{I}(w) \equiv \mathcal{A}(w)$.
-



Labeled Graphs

Practical applications of graphs usually require that they be annotated with additional information. Such information may be attached to the edges of the graph and to the nodes of the graph. A graph which has been annotated in some way is called a *labeled graph*. Figure □ shows two examples of this.

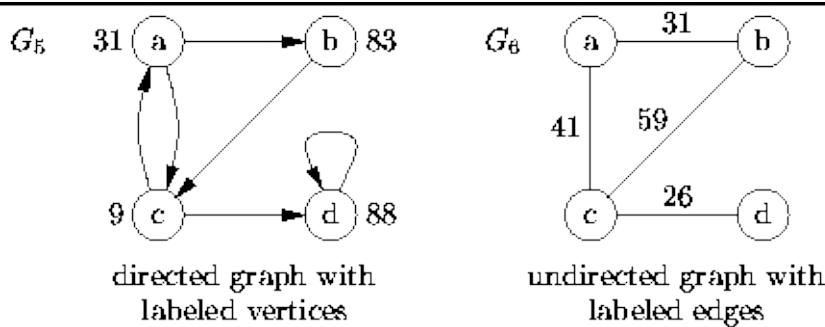


Figure: Labeled graphs.

For example, we can use a directed graph with labeled vertices such as G_5 in Figure □ to represent a finite state machine. Each vertex corresponds to a state of the machine and each edge corresponds to an allowable state transition. In such a graph we can attach a label to each vertex that records some property of the corresponding state such as the latency time for that state.

We can use an undirected graph with labeled edges such as G_6 in Figure □ to represent geographic information. In such a graph, the vertices represent geographic locations and the edges represent possible routes between locations. In such a graph we might use a label on each edge to represent the distance between the end points.

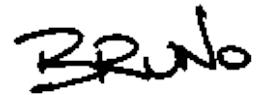
Representing Graphs

Consider a directed graph $G = (\mathcal{V}, \mathcal{E})$. Since $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, graph G contains at most $|\mathcal{V}|^2$ edges. There are $2^{|\mathcal{V}|^2}$ possible sets of edges for a given set of vertices \mathcal{V} . Therefore, the main concern when designing a graph representation scheme is to find a suitable way to represent the set of edges.

- [Adjacency Matrices](#)
 - [Sparse vs. Dense Graphs](#)
 - [Adjacency Lists](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Adjacency Matrices

Consider a directed graph $G = (V, \mathcal{E})$ with n vertices, $V = \{v_1, v_2, \dots, v_n\}$. The simplest graph representation scheme uses an $n \times n$ matrix A of zeroes and ones given by

$$A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

That is, the $(i, j)^{\text{th}}$ element of the matrix, is a one only if $v_i \rightarrow v_j$ is an edge in G . The matrix A is called an *adjacency matrix*.

For example, the adjacency matrix for graph G_1 in Figure [□](#) is

$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Clearly, the number of ones in the adjacency matrix is equal to the number of edges in the graph.

One advantage of using an adjacency matrix is that it is easy to determine the sets of edges emanating from a given vertex. For example, consider vertex v_i . Each one in the i^{th} row corresponds to an edge that emanates from vertex v_i . Conversely, each one in the i^{th} column corresponds to an edge incident on vertex v_i .

We can also use adjacency matrices to represent undirected graphs. That is, we represent an undirected graph $G = (V, \mathcal{E})$ with n vertices, using an $n \times n$ matrix A of zeroes and ones given by

$$A_{i,j} = \begin{cases} 1 & \{v_i, v_j\} \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

Since the two sets $\{v_i, v_j\}$ and $\{v_j, v_i\}$ are equivalent, matrix A is symmetric about the diagonal. That is, $A_{i,j} = A_{j,i}$. Furthermore, all of the entries on the diagonal are zero. That is, $A_{i,i} = 0$ for $1 \leq i \leq n$.

For example, the adjacency matrix for graph G_4 in Figure \square is

$$A_4 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In this case, there are twice as many ones in the adjacency matrix as there are edges in the undirected graph.

A simple variation allows us to use an adjacency matrix to represent an edge-labeled graph. For example, given numeric edge labels, we can represent a graph (directed or undirected) using an $n \times n$ matrix A in which the $A_{i,j}$ is the numeric label associated with edge (v_i, v_j) in the case of a directed graph, and edge $\{v_i, v_j\}$, in an undirected graph.

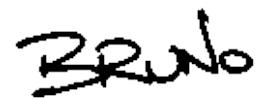
For example, the adjacency matrix for the graph G_6 in Figure \square is

$$A_6 = \begin{bmatrix} \infty & 31 & 41 & \infty \\ 31 & \infty & 59 & \infty \\ 41 & 59 & \infty & 26 \\ \infty & \infty & 26 & \infty \end{bmatrix}$$

In this case, the array entries corresponding to non-existent edges have all been set to ∞ . Here ∞ serves as a kind of *sentinel*. The value to use for the sentinel depends on the application. For example, if the edges represent routes between geographic locations, then a route of length ∞ is much like one that does not exist.

Since the adjacency matrix has $|\mathcal{V}|^2$ entries, the amount of space needed to represent the edges of a graph is $O(|\mathcal{V}|^2)$, regardless of the actual number of edges in the graph. If the graph contains relatively few edges, e.g., if $|\mathcal{E}| \ll |\mathcal{V}|^2$, then most of the elements of the adjacency matrix will be zero (or ∞). A matrix in which most of the elements are zero (or ∞) is a *sparse matrix*.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with "Bruno" written in a larger, more prominent style than the "R." preceding it.

Sparse vs. Dense Graphs

Informally, a graph with relatively few edges is *sparse*, and a graph with many edges is *dense*. The following definition defines precisely what we mean when we say that a graph ``has relatively few edges'':

Definition (Sparse Graph) A *sparse graph* $G = (\mathcal{V}, \mathcal{E})$ in which $|\mathcal{E}| = O(|\mathcal{V}|)$.

For example, consider a graph $G = (\mathcal{V}, \mathcal{E})$ with n nodes. Suppose that the out-degree of each vertex in G is some fixed constant k . Graph G is a *sparse graph* because $|\mathcal{E}| = k|\mathcal{V}| = O(|\mathcal{V}|)$.

A graph that is not sparse is said to be *dense*:

Definition (Dense Graph) A *dense graph* $G = (\mathcal{V}, \mathcal{E})$ in which $|\mathcal{E}| = \Theta(|\mathcal{V}|^2)$.

For example, consider a graph $G = (\mathcal{V}, \mathcal{E})$ with n nodes. Suppose that the out-degree of each vertex in G is some fraction f of n , $0 < f \leq 1$. For example, if $n=16$ and $f=0.25$, the out-degree of each node is 4. Graph G is a *dense graph* because $|\mathcal{E}| = f|\mathcal{V}|^2 = \Theta(|\mathcal{V}|^2)$.

Adjacency Lists

One technique that is often used for a sparse graph, say $G = (V, \mathcal{E})$, uses $|V|$ linked lists--one for each vertex. The linked list for vertex $v_i \in V$ contains the elements of $\{w : (v_i, w) \in \mathcal{A}(v_i)\}$, the set of nodes adjacent to v_i . As a result, the lists are called *adjacency lists*.

Figure □ shows the adjacency lists for the directed graph G_1 of Figure □ and the directed graph G_4 of Figure □. Notice that the total number of list elements used to represent a directed graph is $|\mathcal{E}|$ but the number of lists elements used to represent an undirected graph is $2 \times |\mathcal{E}|$. Therefore, the space required for the adjacency lists is $O(|\mathcal{E}|)$.

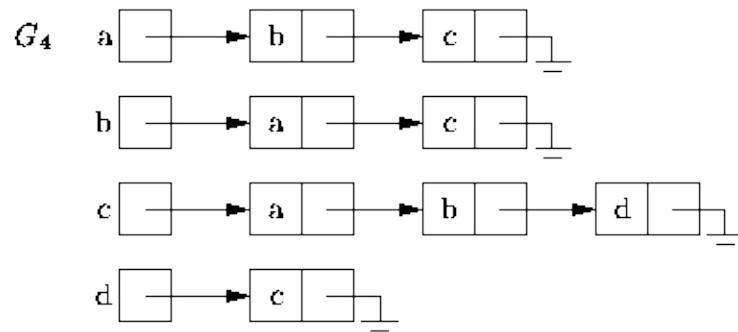
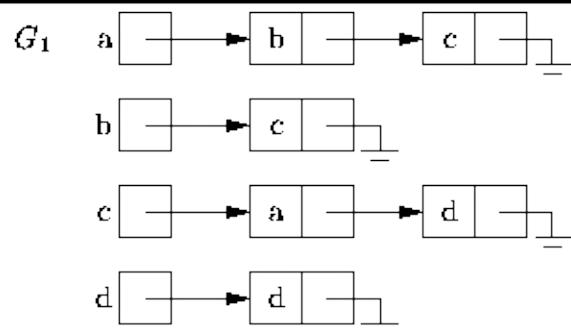
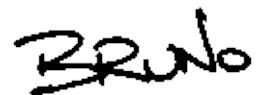


Figure: Adjacency lists.

By definition, a sparse graph has $|\mathcal{E}| = O(|\mathcal{V}|)$. Hence the space required to represent a sparse graph using adjacency lists is $O(|\mathcal{V}|)$. Clearly this is asymptotically better than using adjacency matrices which require $O(|\mathcal{V}|^2)$ space.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Implementing Graphs

In keeping with the design framework used throughout this text, we view graphs as specialized containers. Formally, the graph $G = (\mathcal{V}, \mathcal{E})$ is an ordered pair comprised of two sets--a set of vertices and a set of edges. Informally, we can view a graph as a container with two compartments, one which holds vertices and one which holds edges. There are four kinds of objects--vertices, edges, undirected graphs, and directed graphs. Accordingly, we define four abstract classes: `Vertex`, `Edge`, `Graph`, and `Digraph`. (See Figure □).

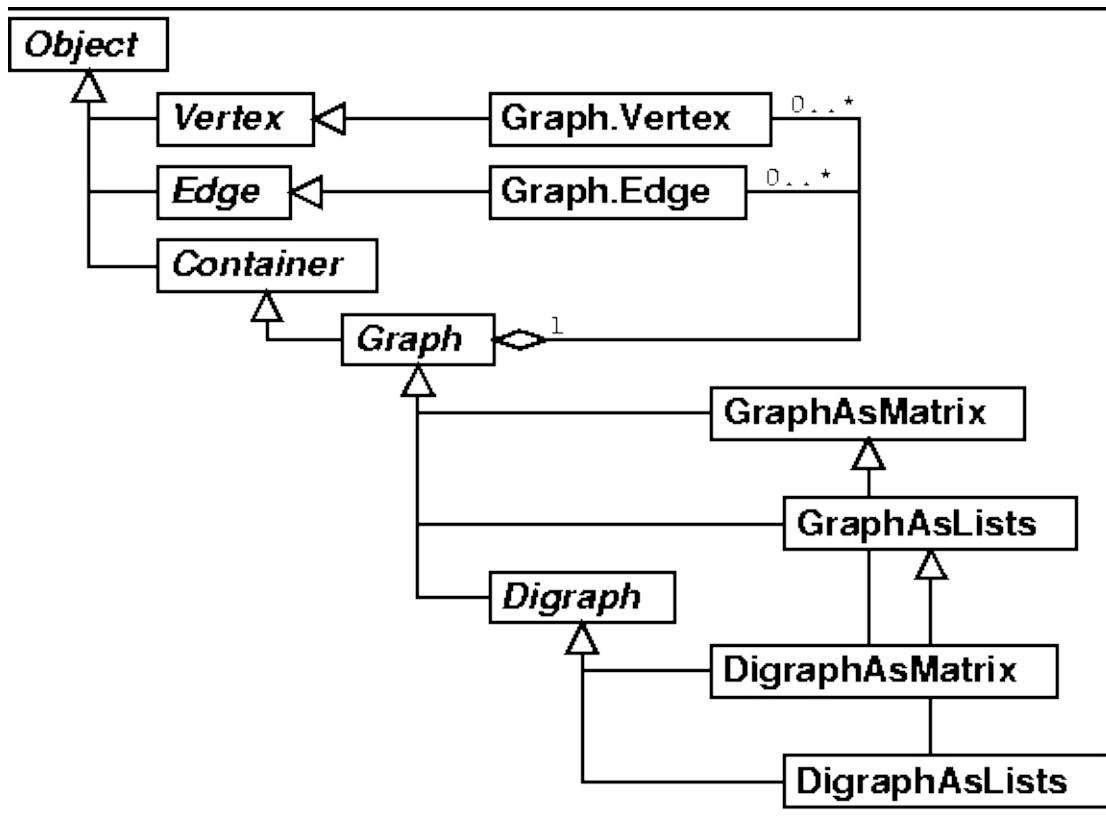


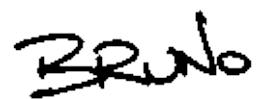
Figure: Object class hierarchy

- [Vertices](#)
- [Edges](#)

- [Graphs and Digraphs](#)
 - [Abstract Graphs](#)
 - [Accessors and Mutators](#)
 - [Directed Graphs](#)
 - [Implementing Undirected Graphs](#)
 - [Comparison of Graph Representations](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Vertices

What exactly is a vertex? The answer to this question depends on the application. At the very minimum, every vertex in a graph must be distinguishable from every other vertex in that graph. We can do this by numbering consecutively the vertices of a graph. In addition, some applications require vertex-weighted graphs. A weighted vertex can be viewed as one which carries a ``payload''. The payload is an object that represents the weight on the vertex.

Program □ defines the `vertex` class. The abstract `vertex` class extends the abstract `object` class introduced in Program □.

```
 1 class Vertex(Object):
 2
 3     def __init__(self):
 4         super(Vertex, self).__init__()
 5
 6     def getNumber(self): pass
 7     getNumber = abstractmethod(getNumber)
 8     number = property(
 9         fget = lambda self: self.getNumber())
10
11     def getWeight(self): pass
12     getWeight = abstractmethod(getWeight)
13     weight = property(
14         fget = lambda self: self.getWeight())
15
16     def getIncidentEdges(self): pass
17     getIncidentEdges = abstractmethod(getIncidentEdges)
18     incidentEdges = property(
19         fget = lambda self: self.getIncidentEdges())
20
21     def getEmanatingEdges(self): pass
22     getEmanatingEdges = abstractmethod(getEmanatingEdges)
23     emanatingEdges = property(
24         fget = lambda self: self.getEmanatingEdges())
25
26     def getPredecessors(self): pass
27     getPredecessors = abstractmethod(getPredecessors)
28     predecessors = property(
29         fget = lambda self: self.getPredecessors())
30
31     def getSuccessors(self): pass
32     getSuccessors = abstractmethod(getSuccessors)
33     successors = property(
34         fget = lambda self: self.getSuccessors())
```

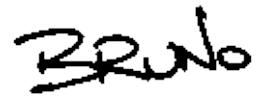
Program: Abstract Vertex class.

Every vertex in a graph is assigned a unique number. The `number` property returns the number of a vertex. The `weight` property returns an object that represents the weight associated with a weighted vertex. If the vertex is unweighted, the `weight` property returns `None`.

- [Iterators](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Iterators

Program `□` also declares four properties that return iterators. The `incidentEdges` property returns an iterator that enumerates the elements of the $\mathcal{I}(v)$; the `emanatingEdges` property returns an iterator that enumerates the elements of $\mathcal{A}(v)$. Similarly, the `predecessors` property returns an iterator that enumerates the elements of $\mathcal{P}(v) = \{u : (u, v) \in \mathcal{I}(v)\}$ and the `successors` property returns an iterator that enumerates the elements of $\mathcal{S}(v) = \{w : (v, w) \in \mathcal{A}(v)\}$. The elements of $\mathcal{P}(v)$ and $\mathcal{S}(v)$ are vertices whereas the elements of $\mathcal{I}(v)$ and $\mathcal{A}(v)$ are edges.

Edges

An edge in a directed graph is an ordered pair of vertices; an edge in an undirected graph is a set of two vertices. Because of the similarity of these concepts, we use the same class for both--the context in which an edge is used determines whether it is directed or undirected.

Program □ defines the Edge class. The abstract Edge class extends the abstract Object class introduced in Program □.

```
 1  class Edge(Object):
 2
 3      def __init__(self):
 4          super(Edge, self).__init__()
 5
 6      def getV0(self): pass
 7      getV0 = abstractmethod(getV0)
 8      v0 = property(
 9          fget = lambda self: self.getV0())
10
11      def getV1(self): pass
12      getV1 = abstractmethod(getV1)
13      v1 = property(
14          fget = lambda self: self.getV1())
15
16      def getWeight(self): pass
17      getWeight = abstractmethod(getWeight)
18      weight = property(
19          fget = lambda self: self.getWeight())
20
21      def getIsDirected(self): pass
22      getIsDirected = abstractmethod(getIsDirected)
23      isDirected = property(
24          fget = lambda self: self.getIsDirected())
25
26      def mateOf(self, vertex): pass
27      mateOf = abstractmethod(mateOf)
```

Program: Abstract Edge class.

An edge connects two vertices, v_0 and v_1 . The properties v_0 and v_1 return these vertices. The $isDirected$ property returns True if the edge is directed. When an

Edge is directed, it represents $v_0 \rightarrow v_1$. That is, v_1 is the head and v_0 is the tail.
Alternatively, when an Edge is undirected, it represents $\{v_0, v_1\}$.

For every instance e of a class derived from the Edge class the mateOf method satisfies the following identities:

`e.mateOf(e.v0) ≡ e.v1`
`e.mateOf(e.v1) ≡ e.v0.`

Therefore, if we know that a vertex v is one of the vertices of e, then we can find the other vertex by calling `e.Mate(v)`.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

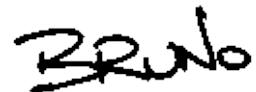
Bruno

Graphs and Digraphs

Directed graphs and undirected graphs have many common characteristics. In addition, we can view a directed graph as an undirected graph with arrowheads added. As shown in Figure □, we have chosen to define the abstract Graph class to represent undirected graphs and to derive abstract Digraph class from it. We have chosen this approach because many algorithms for undirected graphs can also be used with directed graphs. On the other hand, it is often the case that algorithms for directed graphs cannot be used with undirected graphs.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Abstract Graphs

Program □ introduces the `Graph` class. The abstract `Graph` class extends the abstract `Container` class introduced in Program □.

```
 1  class Graph(Container):
 2
 3      def __init__(self, size):
 4          super(Graph, self).__init__()
 5          self._numberOfVertices = 0
 6          self._numberOfEdges = 0
 7          self._vertex = Array(size)
 8          self._isDirected = False
 9
10      class Vertex(Vertex):
11
12          def __init__(self, graph, number, weight):
13              super(Graph.Vertex, self).__init__()
14              self._graph = graph
15              self._number = number
16              self._weight = weight
17
18          # ...
19
20      class Edge(Edge):
21
22          def __init__(self, graph, v0, v1, weight):
23              super(Graph.Edge, self).__init__()
24              self._graph = graph
25              self._v0 = v0
26              self._v1 = v1
27              self._weight = weight
28
29          # ...
30
31      # ...
```

Program: Abstract `Graph`, `Graph.Vertex` and `Graph.Edge` `__init__` methods.

The abstract `Graph` class serves as the base class from which the various concrete graph implementations discussed in Section □ are derived. The abstract `Graph` class also provides implementations for the graph traversals described in Section □ and for the algorithms that test for cycles and connectedness described

in Section □.

As shown in Program □, the abstract `Graph` class defines two nested classes, `Vertex` and `Edge`.

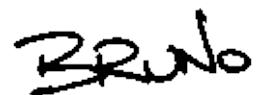
The concrete `Graph.Vertex` class extends the abstract `Vertex` class defined in Program □. It comprises three instance attributes--`_graph`, `_number` and `_weight`. The `_graph` instance attribute refers to the graph instance that contains this vertex. Each vertex in a graph with n vertices is assigned a unique number in the interval $[0,n-1]$. The `_number` instance attribute records this number. The `_weight` instance attribute is used to record the weight on a weighted vertex.

The concrete `Graph.Edge` class extends the abstract `Edge` class defined in Program □. It comprises four instance attributes--`_graph`, `_v0`, `_v1`, and `_weight`. The `_graph` instance attribute refers to the graph instance that contains this edge. The `_v0` and `_v1` attributes record the vertices that are the end-points of the edge. The `_weight` instance attribute is used to record the weight on a weighted edge.

- [Properties](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Properties

Program □ continues the definition of the abstract `Graph` class. The following properties of graphs are defined in Program □

`numberOfEdges`

This property returns the number of edges contained by the graph.

`numberOfVertices`

This property returns the number of vertices contained by the graph.

`isDirected`

This bool-valued property is `True` if the graph is a directed graph.

`isCyclic`

This bool-valued property is `True` if the graph is *cyclic*.

`isConnected`

This bool-valued property is `True` if the graph is *connected*. Connectedness of graphs is discussed in Section □.

`vertices`

This property returns an iterator that enumerates the elements of \mathcal{V} .

`edges`

This property returns an iterator that enumerates the elements of \mathcal{E} .

```
 1 class Graph(Container):
 2
 3     def getNumberOfEdges(self): pass
 4     getNumberOfEdges = abstractmethod(getNumberOfEdges)
 5     numberOfEdges = property(
 6         fget = lambda self: self.getNumberOfEdges())
 7
 8     def getNumberOfVertices(self): pass
 9     getNumberOfVertices = abstractmethod(getNumberOfVertices)
10     numberOfVertices = property(
11         fget = lambda self: self.getNumberOfVertices())
12
13     def getIsDirected(self): pass
14     getIsDirected = abstractmethod(getIsDirected)
15     isDirected = property(
16         fget = lambda self: self.getIsDirected())
17
18     def getIsConnected(self): pass
19     getIsConnected = abstractmethod(getIsConnected)
20     isConnected = property(
21         fget = lambda self: self.getIsConnected())
22
23     def getIsCyclic(self): pass
24     getIsCyclic = abstractmethod(getIsCyclic)
25     isCyclic = property(
26         fget = lambda self: self.getIsCyclic())
27
28     def getVertices(self): pass
29     getVertices = abstractmethod(getVertices)
30     vertices = property(
31         fget = lambda self: self.getVertices())
32
33     def getEdges(self): pass
34     getEdges = abstractmethod(getEdges)
35     edges = property(
36         fget = lambda self: self.getEdges())
37
38     # ...
```

Program: Abstract Graph class properties.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Accessors and Mutators

Program □ continues the definition of the abstract `Graph` class. The following additional methods are defined in Program □:

`addVertex`

This method inserts a vertex into a graph. All the vertices contained in a given graph must have a unique vertex number. Furthermore, if a graph contains n vertices, those vertices shall be numbered $0, 1, \dots, n-1$.

Therefore, the next vertex inserted into the graph shall have the number n .

`getVertex`

In addition to `self`, this method takes an integer, say i where $0 \leq i < n$, and returns the i^{th} vertex contained in the graph.

`addEdge`

This method inserts an edge into a graph. If the graph contains n vertices, both arguments must fall in the interval $[0, n-1]$.

`isEdge`

In addition to `self`, this `bool`-valued method takes two integer arguments. It returns `True` if the graph contains an edge that connects the corresponding vertices.

`getEdge`

In addition to `self`, this method takes two integer arguments. It returns the edge instance (if it exists) that connects the corresponding vertices. The behavior of this method is undefined when the edge does not exist. (An implementation will typically throw an exception).

`getEmanatingEdges`

In addition to `self`, this method takes an integer argument that specifies a vertex and returns and iterator that enumerates the edges incident upon the specified vertex.

`getIncidentEdges`

In addition to `self`, this method takes an integer argument that specifies a vertex and returns and iterator that enumerates the edges emanating from the specified vertex.

`__len__`

This method returns the number of vertices in the graph.

`getitem__`

This method is equivalent to the `getVertex` described above.

```
 1 class Graph(Container):
 2
 3     def addVertex(self, *args): pass
 4     addVertex = abstractmethod(addVertex)
 5
 6     def getVertex(self, v): pass
 7     getVertex = abstractmethod(getVertex)
 8
 9     def addEdge(self, *args): pass
10     addEdge = abstractmethod(addEdge)
11
12     def getEdge(self, v, w): pass
13     getEdge = abstractmethod(getEdge)
14
15     def isEdge(self, v, w): pass
16     isEdge = abstractmethod(isEdge)
17
18     def depthFirstTraversal(self, visitor, start): pass
19     depthFirstTraversal = abstractmethod(depthFirstTraversal)
20
21     def breadthFirstTraversal(self, visitor, start): pass
22     breadthFirstTraversal = abstractmethod(breadthFirstTraversal)
23
24     def getIncidentEdges(self, v): pass
25     getIncidentEdges = abstractmethod(getIncidentEdges)
26
27     def getEmanatingEdges(self, v): pass
28     getEmanatingEdges = abstractmethod(getEmanatingEdges)
29
30     def __len__(self):
31         return self.numberOfVertices
32
33     def __getitem__(self, v):
34         return self.getVertex(v)
35
36     # ...
```

Program: Abstract Graph class accessors and mutators.

-
- [Graph Traversals](#)
-

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Graph Traversals

Program `graph` also defines the following traversal methods. These methods are analogous to the `accept` method of the container class (see Section `Container`). Each of these methods takes a *visitor* and performs a traversal. That is, all the *vertices* of the graph are visited systematically. When a vertex is visited, the `visit` method of the visitor is applied to that vertex.

DepthFirstTraversal

In addition to `self`, this methods accepts two arguments--a `PrePostVisitor` and an integer. The integer specifies the starting vertex for a depth-first traversal of the graph.

BreadthFirstTraversal

In addition to `self`, this methods accepts two arguments--a `visitor` and an integer. The integer specifies the starting vertex for a breadth-first traversal of the graph.

Graph traversal algorithms are discussed in Section `Traversal`.

Directed Graphs

Program 1 introduces the `Digraph` class. The abstract `Digraph` class extends the abstract `Graph` class defined in Programs 1, 2 and 3.

```
1  class Digraph(Graph):
2
3      def __init__(self, size):
4          super(Digraph, self).__init__(size)
5          self._isDirected = True
6
7      def getIsStronglyConnected(self): pass
8      getIsStronglyConnected = abstractmethod(
9          getIsStronglyConnected)
10     isStronglyConnected = property(
11         fget = lambda self: self.getIsStronglyConnected())
12
13     def topologicalOrderTraversal(self): pass
14     topologicalOrderTraversal = abstractmethod(
15         topologicalOrderTraversal)
16
17     # ...
```

Program: Abstract `Digraph` class.

The abstract `Digraph` class adds the following operations which apply only to directed graphs:

isStronglyConnected

This bool-valued property is `True` if the directed graph is *strongly connected*. Strong connectedness is discussed in Section 1.

topologicalOrderTraversal

A topological sort is an ordering of the nodes of a directed graph. This traversal visits the nodes of a directed graph in the order specified by a topological sort.

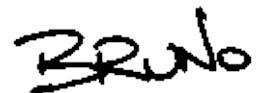
Implementing Undirected Graphs

This section describes two concrete classes--`GraphAsMatrix` and `GraphAsLists`. These classes both represent *undirected graphs*. The `GraphAsMatrix` class represents the edges of a graph using an adjacency matrix. The `GraphAsLists` class represents the edges of a graph using adjacency lists.

- [Using Adjacency Matrices](#)
 - [Using Adjacency Lists](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Using Adjacency Matrices

The `GraphAsMatrix` class is introduced in Program □. The `GraphAsMatrix` class extends the abstract `Graph` class introduced in Programs □, □ and □.

```
1  class GraphAsMatrix(Graph):
2
3      def __init__(self, size):
4          super(GraphAsMatrix, self).__init__(size)
5          self._matrix = DenseMatrix(size, size)
6
7      # ...
```

Program: `GraphAsMatrix` class `__init__` method.

Each instance of the `GraphAsMatrix` class represents an undirected graph, say $G = (\mathcal{V}, \mathcal{E})$. The set of vertices, \mathcal{V} , is represented using the `_vertex` array inherited from the abstract `Graph` base class. Each vertex is represented by a separate `Graph.Vertex` instance.

Similarly, The set of edges, \mathcal{E} , is represented using the `_matrix` instance attribute which is a two-dimensional array of `Edges`. Each edge is represented by a separate `Graph.Edge` instance.

In addition to `self`, the `GraphAsMatrix __init__` method takes a single argument of type `int` that specifies the maximum number of vertices that the graph may contain. This quantity specifies the length of the array of vertices and the dimensions of the adjacency matrix. The implementation of the `GraphAsMatrix` class is left as programming project for the reader (Project □).

Using Adjacency Lists

Program 1 introduces the `GraphAsLists` class. The `GraphAsLists` extends the abstract `Graph` class introduced in Programs 1, 2 and 3. The `GraphAsLists` class represents the edges of a graph using adjacency lists.

```
1  class GraphAsLists(Graph):
2      "Graph implemented using adjacency lists."
3
4      def __init__(self, size):
5          super(GraphAsLists, self).__init__(size)
6          self._adjacencyList = Array(size)
7          for i in xrange(size):
8              self._adjacencyList[i] = LinkedList()
9
10     # ...
```

Program: `GraphAsLists` class `__init__` method.

Each instance of the `GraphAsLists` class represents an undirected graph, say $G = (\mathcal{V}, \mathcal{E})$. The set of vertices, \mathcal{V} , is represented using the `_vertex` array inherited from the abstract `Graph` base class. The set of edges, \mathcal{E} , is represented using the `_adjacencyList` instance attribute, which is an array of linked lists. The i^{th} linked list, `_adjacencyList[i]`, represents the set $\mathcal{A}(v_i)$ which is the set of edges emanating from vertex v_i . The implementation uses the `LinkedList` class given in Section 1.

In addition to `self`, the `GraphAsLists __init__` method takes a single argument of type `int` that specifies the maximum number of vertices that the graph may contain. This quantity specifies the lengths of the array of vertices and the array of adjacency lists. The implementation of the `GraphAsLists` class is left as programming project for the reader (Project 1).

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

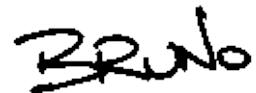
Comparison of Graph Representations

In order to make the appropriate choice when selecting a graph representation scheme, it is necessary to understand the time/space trade-offs. Although the details of the implementations have been omitted, we can still make meaningful conclusions about the performance that we can expect from those implementations. In this section we consider the space required as well as the running times for basic graph operations.

- [Space Comparison](#)
 - [Time Comparison](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Space Comparison

Consider the representation of a directed graph $G = (\mathcal{V}, \mathcal{E})$. In addition to the $|\mathcal{V}|$ `GraphVertex` class instances and the $|\mathcal{E}|$ `GraphEdge` class instances contained by the graph, there is the storage required by the adjacency matrix. In this case, the matrix is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix of `Edges`. Therefore, the amount of storage required by an adjacency matrix implementation is

$$|\mathcal{V}| \times \text{sizeof}(\text{GraphVertex}) + |\mathcal{E}| \times \text{sizeof}(\text{GraphEdge}) \\ + |\mathcal{V}| \times \text{sizeof}(\text{id}(\text{Vertex})) + |\mathcal{V}|^2 \times \text{sizeof}(\text{id}(\text{Edge})) + O(1). \quad (16.1)$$

On the other hand, consider the amount of storage required when we represent the same graph using adjacency lists. In addition to the vertices and the edges themselves, there are $|\mathcal{V}|$ linked lists. If we use the `LinkedList` class defined in Section □, each such list has a `head` and `tail` instance attribute. Altogether there are $|\mathcal{E}|$ linked lists elements each of which refers to the linked list itself, to the next element of the list and contains an `Edge`. Therefore, the total space required is

$$|\mathcal{V}| \times \text{sizeof}(\text{GraphVertex}) + |\mathcal{E}| \times \text{sizeof}(\text{GraphEdge}) \\ + |\mathcal{V}| \times \text{sizeof}(\text{id}(\text{Vertex})) + |\mathcal{V}| \times \text{sizeof}(\text{id}(\text{LinkedList})) \\ + 2|\mathcal{V}| \times \text{sizeof}(\text{id}(\text{LinkedList.Element})) + \\ + |\mathcal{E}| \times (\text{sizeof}(\text{id}(\text{LinkedList})) + \text{sizeof}(\text{id}(\text{LinkedList.Element}))) + \\ \text{sizeof}(\text{id}(\text{Edge}))) + O(1). \quad (16.2)$$

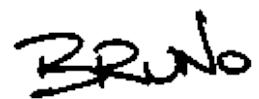
Notice that the space for the vertices and edges themselves cancels out when we compare Equation □ with Equation □. If we assume that all object ids require the same amount of space, we can conclude that adjacency lists use less space than adjacency matrices when

$$|\mathcal{E}| < \frac{|\mathcal{V}|^2 - |\mathcal{V}|}{3}.$$

For example, given a 10 node graph, the adjacency lists version uses less space when there are fewer than 30 edges. As a rough rule of thumb, we can say that adjacency lists use less space when the average degree of a node, $\bar{d} = |\mathcal{E}|/|\mathcal{V}|$, satisfies $\bar{d} \lesssim |\mathcal{V}|/3$.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Time Comparison

The following four operations are used extensively in the implementations of many different graph algorithms:

find edge (v, w)

Given vertices v and w , this operation locates the corresponding Edge instance. When using an adjacency matrix, we can find an edge in constant time.

When adjacency lists are used, the worst-case running time is $O(|\mathcal{A}(v)|)$, since $|\mathcal{A}(v)|$ is the length of the adjacency list associated with vertex v .

This is the operation performed by the `getEdge` method of the Graph interface.

enumerate all edges

In order to locate all the edges in when using adjacency matrices, it is necessary to examine all $|V|^2$ matrix entries. Therefore, the worst-case running time needed to enumerate all the edges is $O(|V|^2)$.

On the other hand, to enumerate all the edges when using adjacency lists requires the traversal of $|V|$ lists. In all there are $|E|$ edges. Therefore the worst case running time is $O(|V| + |E|)$.

This operation is performed using the enumerator obtained using the `edges` property of the Graph interface.

enumerate edges emanating from v

To enumerate all the edges that emanate from vertex v requires a complete scan of the v^{th} row of an adjacency matrix. Therefore, the worst-case running time when using adjacency matrices is $O(|V|)$.

Enumerating the edges emanating from vertex v is a trivial operation when

using adjacency lists. All we need do is traverse the v^{th} list. This takes $O(|\mathcal{A}(v)|)$ time in the worst case.

This operation is performed using the enumerator obtained using the `getEmanatingEdges` property of the `Vertex` interface.

enumerate edges incident on w

To enumerate all the edges are incident on vertex w requires a complete scan of the w^{th} column of an adjacency matrix. Therefore, the worst-case running time when using adjacency matrices is $O(|\mathcal{V}|)$.

Enumerating the edges incident on vertex w is a non-trivial operation when using adjacency lists. It is necessary to search every adjacency list in order to find all the edges incident on a given vertex. Therefore, the worst-case running time is $O(|\mathcal{V}| + |\mathcal{E}|)$.

This operation is performed using the enumerator obtained using the `getIncidentEdges` property of the `Vertex` interface.

Table □ summarizes these running times.

Table: Comparison of graph representations.

operation	adjacency matrix	adjacency list
find edge (v,w)	$O(1)$	$O(\mathcal{A}(v))$
enumerate all edges	$O(\mathcal{V} ^2)$	$O(\mathcal{V} + \mathcal{E})$
enumerate edges emanating from v	$O(\mathcal{V})$	$O(\mathcal{A}(v))$
enumerate edges incident on w	$O(\mathcal{V})$	$O(\mathcal{V} + \mathcal{E})$

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Graph Traversals

There are many different applications of graphs. As a result, there are many different algorithms for manipulating them. However, many of the different graph algorithms have in common the characteristic that they systematically visit all the vertices in the graph. That is, the algorithm walks through the graph data structure and performs some computation at each vertex in the graph. This process of walking through the graph is called a *graph traversal* .

While there are many different possible ways in which to systematically visit all the vertices of a graph, certain traversal methods occur frequently enough that they are given names of their own. This section presents three of them--depth-first traversal, breadth-first traversal and topological sort.

-
- [Depth-First Traversal](#)
 - [Breadth-First Traversal](#)
 - [Topological Sort](#)
 - [Graph Traversal Applications:
Testing for Cycles and Connectedness](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

[Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.](#)



Depth-First Traversal

The *depth-first traversal* of a graph is like the depth-first traversal of a tree discussed in Section □. A depth-first traversal of a tree always starts at the root of the tree. Since a graph has no root, when we do a depth-first traversal, we must specify the vertex at which to begin.

A depth-first traversal of a tree visits a node and then recursively visits the subtrees of that node. Similarly, depth-first traversal of a graph visits a vertex and then recursively visits all the vertices adjacent to that node. The catch is that the graph may contain cycles, but the traversal must visit every vertex at most once. The solution to the problem is to keep track of the nodes that have been visited, so that the traversal does not suffer the fate of infinite recursion.

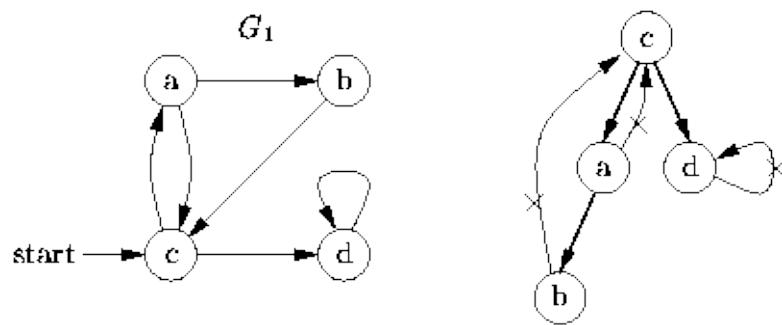


Figure: Depth-first traversal.

For example, Figure □ illustrates the depth-first traversal of the directed graph G_1 starting from vertex c . The depth-first traversal visits the nodes in the order

$$c, a, b, d.$$

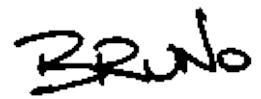
A depth-first traversal only follows edges that lead to unvisited vertices. As shown in Figure □, if we omit the edges that are not followed, the remaining edges form a tree. Clearly, the depth-first traversal of this tree is equivalent to the depth-first traversal of the graph

-
- [Implementation](#)

- [Running Time Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Program □ gives the code for the `depthFirstTraversal` method of the abstract `Graph` class. In addition to `self`, the `DepthFirstTraversal` method takes any `PrePostVisitor` and an integer. The idea is that the `visit` method of the visitor is called once for each vertex in the graph and the vertices are visited in depth-first traversal order starting from the vertex specified by the integer.

```

1  class Graph(Container):
2
3      def depthFirstTraversal(self, visitor, start):
4          assert isinstance(visitor, PrePostVisitor)
5          visited = Array(self._numberOfVertices)
6          for v in xrange(self._numberOfVertices):
7              visited[v] = False
8          self._depthFirstTraversal(visitor, self[start], visited)
9
10     def _depthFirstTraversal(self, visitor, v, visited):
11         if visitor.isDone:
12             return
13         visitor.preVisit(v)
14         visited[v.number] = True
15         for to in v.successors:
16             if not visited[to.number]:
17                 self._depthFirstTraversal(visitor, to, visited)
18         visitor.postVisit(v)
19
20     # ...

```

Program: Abstract Graph class `depthFirstTraversal` method.

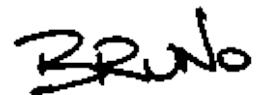
In order to ensure that each vertex is visited at most once, an array of length $|V|$ of bool values called `visited` is used (line 10). That is, $\text{visited}[i] = \text{True}$ only if vertex i has been visited. All the array elements are initially `False` (lines 5-7). After initializing the array, the `depthFirstTraversal` method calls the recursive `_depthFirstTraversal` method, passing it the array as the third argument.

The recursive `_depthFirstTraversal` emthod returns immediately if the visitor

is done. Otherwise, it visits the specified node, and then it follows all the edges emanating from that node and recursively visits the adjacent vertices *if those vertices have not already been visited.*

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Running Time Analysis

The running time of the depth-first traversal method depends on the graph representation scheme used. The traversal visits each node in the graph at most once. When a node is visited, all the edges emanating from that node are considered. During a complete traversal enumerates every edge in the graph.

Therefore, the worst-case running time for the depth-first traversal of a graph is represented using an adjacency matrix is

$$|\mathcal{V}| \times (\mathcal{T}(\text{preVisit}) + \mathcal{T}(\text{postVisit})) + O(|\mathcal{V}|^2).$$

When adjacency lists are used, the worst case running time for the depth-first traversal method is

$$|\mathcal{V}| \times (\mathcal{T}(\text{preVisit}) + \mathcal{T}(\text{postVisit})) + O(|\mathcal{V}| + |\mathcal{E}|).$$

Recall that for a sparse graph graph $|\mathcal{E}| = O(|\mathcal{V}|)$. If the sparse graph is represented using adjacency lists and if $\mathcal{T}(\text{preVisit}) = O(1)$ and $\mathcal{T}(\text{postVisit}) = O(1)$ the worst-case running time of the depth-first traversal is simply $O(|\mathcal{V}|)$.

Breadth-First Traversal

The *breadth-first traversal* of a graph is like the breadth-first traversal of a tree discussed in Section □. The breadth-first traversal of a tree visits the nodes in the order of their depth in the tree. Breadth-first tree traversal first visits all the nodes at depth zero (i.e., the root), then all the nodes at depth one, and so on.

Since a graph has no root, when we do a breadth-first traversal, we must specify the vertex at which to start the traversal. Furthermore, we can define the depth of a given vertex to be the length of the shortest path from the starting vertex to the given vertex. Thus, breadth-first traversal first visits the starting vertex, then all the vertices adjacent to the starting vertex, and then all the vertices adjacent to those, and so on.

Section □ presents a non-recursive breadth-first traversal algorithm for N -ary trees that uses a queue to keep track vertices that need to be visited. The breadth-first graph traversal algorithm is very similar.

First, the starting vertex is enqueued. Then, the following steps are repeated until the queue is empty:

1. Remove the vertex at the head of the queue and call it v .
2. Visit v .
3. Follow each edge emanating from v to find the adjacent vertex and call it to . If to has not already been put into the queue, enqueue it.

Notice that a vertex can be put into the queue at most once. Therefore, the algorithm must somehow keep track of the vertices that have been enqueued.

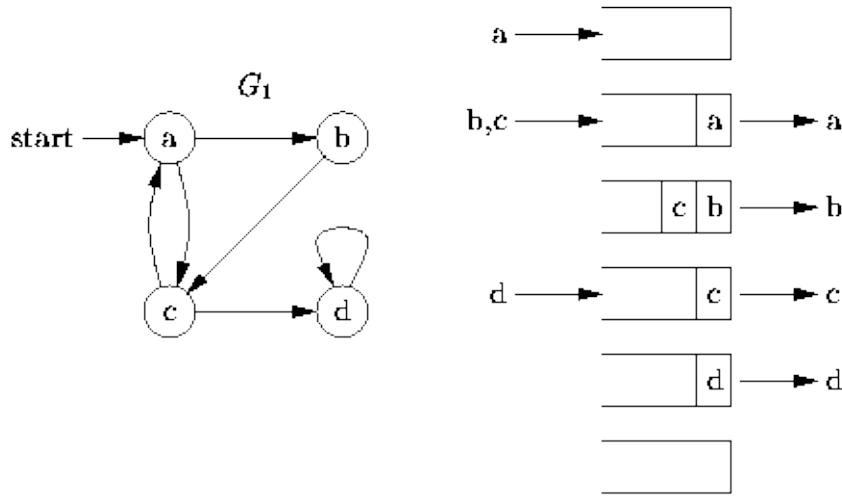


Figure: Breadth-first traversal.

Figure \square illustrates the breadth-first traversal of the directed graph G_1 starting from vertex a . The algorithm begins by inserting the starting vertex, a , into the empty queue. Next, the head of the queue (vertex a) is dequeued and visited, and the vertices adjacent to it (vertices b and c) are enqueue. When, b is dequeued and visited we find that there is only adjacent vertex, c , and that vertex is already in the queue. Next vertex c is dequeued and visited. Vertex c is adjacent to a and d . Since a has already been enqueue (and subsequently dequeued) only vertex d is put into the queue. Finally, vertex d is dequeued and visited. Therefore, the breadth-first traversal of G_1 starting from a visits the vertices in the sequence

a, b, c, d .

-
- [Implementation](#)
 - [Running Time Analysis](#)
-



Implementation

Program □ gives the code for the `breadthFirstTraversal` method of the abstract `Graph` class. In addition to `self`, this method takes any `Visitor` and an integer. The `visit` method of the visitor is called once for each vertex in the graph and the vertices are visited in breadth-first traversal order starting from the vertex specified by the integer.

```

1  class Graph(Container):
2
3      def breadthFirstTraversal(self, visitor, start):
4          assert isinstance(visitor, Visitor)
5          enqueueued = Array(self._numberOfVertices)
6          for v in xrange(self._numberOfVertices):
7              enqueueued[v] = False
8          queue = QueueAsLinkedList()
9          queue.enqueue(self[start])
10         enqueueued[start] = True
11         while not queue.isEmpty and not visitor.isDone:
12             v = queue.dequeue()
13             visitor.visit(v)
14             for to in v.successors:
15                 if not enqueueued[to.number]:
16                     queue.enqueue(to)
17                     enqueueued[to.number] = True
18
19     # ...

```

Program: Abstract `Graph` class `breadthFirstTraversal` method.

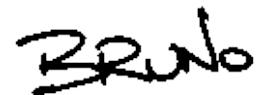
A bool-valued array, `enqueueued`, is used to keep track of the vertices that have been put into the queue. The elements of the array are all initialized to `False` (lines 5-7). Next, a new queue is created and the starting vertex is enqueued (lines 8-9).

The main loop of the `breadthFirstTraversal` method comprises lines 11-17. This loop continues as long as there is a vertex in the queue and the visitor is willing to do more work (line 11). In each iteration exactly one vertex is dequeued and visited (lines 12-13). After a vertex is visited, all the successors of

that node are examined (lines 14-17). Every successor of the node that has not yet been enqueued is put into the queue and the fact that it has been enqueued is recorded in the array enqueueued (lines 15-17).

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Running Time Analysis

The breadth-first traversal enqueues each node in the graph at most once. When a node is dequeued, all the edges emanating from that node are considered. Therefore, a complete traversal enumerates every edge in the graph.

The actual running time of the breadth-first traversal method depends on the graph representation scheme used. The worst-case running time for the traversal of a graph represented using an adjacency matrix is

$$|\mathcal{V}| \times T(\text{visit}) + O(|\mathcal{V}|^2).$$

When adjacency lists are used, the worst case running time for the breadth-first traversal method is

$$|\mathcal{V}| \times T(\text{visit}) + O(|\mathcal{V}| + |\mathcal{E}|).$$

If the graph is sparse, then $|\mathcal{E}| = O(|\mathcal{V}|)$. Therefore, if a sparse graph is represented using adjacency lists and if $T(\text{visit}) = O(1)$, the worst-case running time of the breadth-first traversal is just $O(|\mathcal{V}|)$.

Topological Sort

A topological sort is an ordering of the vertices of a *directed acyclic graph* given by the following definition:

Definition (Topological Sort) Consider a directed acyclic graph $G = (\mathcal{V}, \mathcal{E})$.

A *topological sort* of the vertices of G is a sequence $S = \{v_1, v_2, \dots, v_{|\mathcal{V}|}\}$ in which each element of \mathcal{V} appears exactly once. For every pair of distinct vertices v_i and v_j in the sequence S , if $v_i \rightarrow v_j$ is an edge in G , i.e., $(v_i, v_j) \in \mathcal{E}$, then $i < j$.

Informally, a topological sort is a list of the vertices of a DAG in which all the successors of any given vertex appear in the sequence after that vertex. Consider the directed acyclic graph G_7 shown in Figure □. The sequence $S = \{a, b, c, d, e, f, g, h, i\}$ is a topological sort of the vertices of G_7 . To see that this is so, consider the set of vertices:

$$\mathcal{E} = \{(a, b), (a, c), (a, e), (b, d), (b, e), (c, f), (c, h), (d, g), (e, g), (e, h), (e, i), (f, h), (g, i), (h, i)\}.$$

The vertices in each edge are in alphabetical order, and so is the sequence S .

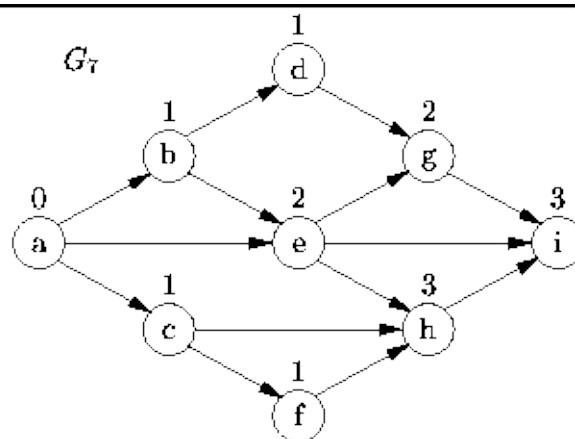


Figure: A directed acyclic graph.

It should also be evident from Figure □ that a topological sort is not unique. For example, the following are also valid topological sorts of the graph G_7 :

$$\begin{aligned}S' &= \{a, c, b, f, e, d, h, g, i\} \\S'' &= \{a, b, d, e, g, c, f, h, i\} \\S''' &= \{a, c, f, h, b, e, d, g, i\} \\&\vdots\end{aligned}$$

One way to find a topological sort is to consider the *in-degrees* of the vertices. (The number above a vertex in Figure □ is the in-degree of that vertex). Clearly the first vertex in a topological sort must have in-degree zero and every DAG must contain at least one vertex with in-degree zero. A simple algorithm to create the sort goes like this:

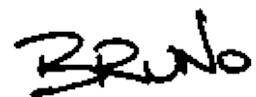
Repeat the following steps until the graph is empty:

1. Select a vertex that has in-degree zero.
2. Add the vertex to the sort.
3. Delete the vertex and all the edges emanating from it from the graph.

-
- [Implementation](#)
 - [Running Time Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Instead of implementing an algorithm that computes a topological sort, we have chosen to implement a traversal that visits the vertices of a DAG in the order given by the topological sort. The topological order traversal can be used to implement many other graph algorithms. Furthermore, given such a traversal, it is easy to define a visitor that computes a topological sort.

In order to implement the algorithm described in the preceding section, an array of integers of length $|V|$ is used to record the in-degrees of the vertices. As a result, it is not really necessary to remove vertices or edges from the graph during the traversal. Instead, the effect of removing a vertex and all the edges emanating from that vertex is simulated by decreasing the apparent in-degrees of all the successors of the removed vertex.

In addition, we use a queue to keep track of the vertices that have not yet been visited, but whose in-degree is zero. Doing so eliminates the need to search the array for zero entries.

Program □ defines the `topologicalOrderTraversal` method of the abstract `Digraph` class. This method takes as its argument a `visitor`. The `visit` method of the visitor is called once for each vertex in the graph. The order in which the vertices are visited is given by a topological sort of those vertices.

```

1 class Digraph(Graph):
2
3     def topologicalOrderTraversal(self, visitor):
4         inDegree = Array(self.numberOfVertices)
5         for v in xrange(self.numberOfVertices):
6             inDegree[v] = 0
7             for e in self.edges:
8                 inDegree[e.v1.number] += 1
9             queue = QueueAsLinkedList()
10            for v in xrange(self.numberOfVertices):
11                if inDegree[v] == 0:
12                    queue.enqueue(self[v])
13                while not queue.isEmpty and not visitor.isDone:
14                    v = queue.dequeue()
15                    visitor.visit(v)
16                    for to in v.successors:
17                        inDegree[to.number] -= 1
18                        if inDegree[to.number] == 0:
19                            queue.enqueue(to)
20
21    # ...

```

Program: Abstract Digraph class `topologicalOrderTraversal` method.

The algorithm begins by computing the in-degrees of all the vertices. An array of integers of length V called `inDegree` is used for this purpose. First, all the array elements are set to zero (lines 2-4). Then, for each edge $(v_0, v_1) \in V$, array element `inDegree(v_1)` is increased by one (lines 5-6). Next, a queue to hold vertices is created. All vertices with in-degree zero are put into this queue (lines 7-10).

The main loop of the `topologicalOrderTraversal` method comprises lines 11-17. This loop continues as long as the queue is not empty and the visitor is not finished. In each iteration of the main loop exactly one vertex is dequeued and visited (lines 12-13).

Once a vertex has been visited, the effect of removing that vertex from the graph is simulated by decreasing by one the in-degrees of all the successors of that vertex. When the in-degree of a vertex becomes zero, that vertex is enqueued (lines 14-17).

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Running Time Analysis

The topological-order traversal enqueues each node in the graph at most once. When a node is dequeued, all the edges emanating from that node are considered. Therefore, a complete traversal enumerates every edge in the graph.

The worst-case running time for the traversal of a graph represented using an adjacency matrix is

$$|\mathcal{V}| \times T(\text{visit}) + O(|\mathcal{V}|^2).$$

When adjacency lists are used, the worst case running time for the topological-order traversal method is

$$|\mathcal{V}| \times T(\text{visit}) + O(|\mathcal{V}| + |\mathcal{E}|).$$

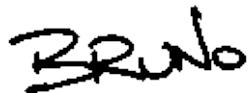
Graph Traversal Applications: Testing for Cycles and Connectedness

This section presents several graph algorithms that are based on graph traversals. The first two algorithms test undirected and directed graphs for connectedness. Both algorithms are implemented using the depth-first traversal. The third algorithm tests a directed graph for cycles. It is implemented using a topological-order traversal.

-
- [Connectedness of an Undirected Graph](#)
 - [Connectedness of a Directed Graph](#)
 - [Testing Strong Connectedness](#)
 - [Testing for Cycles in a Directed Graph](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Connectedness of an Undirected Graph

Definition (Connectedness of an Undirected Graph) An undirected graph $G = (\mathcal{V}, \mathcal{E})$ is *connected* if there is a path in G between every pair of vertices in \mathcal{V} .

Consider the undirected graph shown in Figure □. It is tempting to interpret this figure as a picture of two graphs. However, the figure actually represents the undirected graph $G_8 = (\mathcal{V}, \mathcal{E})$, given by

$$\begin{aligned}\mathcal{V} &= \{a, b, c, d, e, f\} \\ \mathcal{E} &= \{\{a, b\}, \{a, c\}, \{b, c\}, \{d, e\}, \{e, f\}\}.\end{aligned}$$

Clearly, the graph G_8 is not connected. For example, there is no path between vertices a and d . In fact, the graph G_8 consists of two, unconnected parts, each of which is a connected sub-graph. The connected sub-graphs of a graph are called *connected components*.

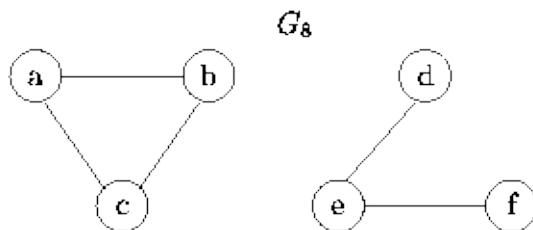


Figure: An unconnected, undirected graph with two (connected) components.

A traversal of an undirected graph (either depth-first or breadth-first) starting from any vertex will only visit all the other vertices of the graph if that graph is connected. Therefore, there is a very simple way to test whether an undirected graph is connected: Count the number of vertices visited during a traversal of the graph. Only if all the vertices are visited is the graph connected.

Program □ shows how this can be implemented. The `getIsConnected` method of the abstract `Graph` class returns `True` if the graph is connected.

```

1  class Graph(Container):
2
3      class CountingVisitor(Visitor):
4
5          def __init__(self):
6              super(Graph.CountingVisitor, self).__init__()
7              self._count = 0
8
9          def visit(self, obj):
10             self._count += 1
11
12         def getCount(self):
13             return self._count
14
15         count = property(
16             fget = lambda self: self.getCount())
17
18     def getIsConnected(self):
19         visitor = self.CountingVisitor()
20         self.depthFirstTraversal(PreOrder(visitor), 0)
21         return visitor.count == self.numberOfVertices
22
23     # ...

```

Program: Abstract Graph class getIsConnected method.

The property is implemented using a the depthFirstTraversal method and a visitor that simply counts the number of vertices it visits. The visit method adds one the _count instance attribute of the counter each time it is called.

The worst-case running time of the getIsConnected method is determined by the time taken by the depthFirstTraversal. Clearly in this case $T(\text{visit}) = O(1)$. Therefore, the running time of getIsConnected is $O(|V|^2)$ when adjacency matrices are used to represent the graph and $O(|V| + |E|)$ when adjacency lists are used.

[Next] [Up] [Previous] [Contents] [Index]

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Connectedness of a Directed Graph

When dealing with directed graphs, we define two kinds of connectedness, *strong* and *weak*. Strong connectedness of a directed graph is defined as follows:

Definition (Strong Connectedness of a Directed Graph) A directed graph $G = (\mathcal{V}, \mathcal{E})$ is *strongly connected* if there is a path in G between every pair of vertices in \mathcal{V} .

For example, Figure shows the directed graph $G_9 = (\mathcal{V}, \mathcal{E})$ given by

$$\begin{aligned}\mathcal{V} &= \{a, b, c, d, e, f\} \\ \mathcal{E} &= \{(a, b), (b, c), (b, e), (c, a), (d, e), (e, f), (f, d)\}.\end{aligned}$$

Notice that the graph G_9 is *not* connected! For example, there is no path from any of the vertices in $\{d, e, f\}$ to any of the vertices in $\{a, b, c\}$. Nevertheless, the graph ``looks'' connected in the sense that it is not made up of separate parts in the way that the graph G_8 in Figure is.

This idea of ``looking'' connected is what *weak connectedness* represents. To define weak connectedness we need to introduce first the notion of the undirected graph that underlies a directed graph: Consider a directed graph $G = (\mathcal{V}, \mathcal{E})$. The underlying undirected graph is the graph $\hat{G} = (\mathcal{V}, \hat{\mathcal{E}})$ where $\hat{\mathcal{E}}$ represents the set of undirected edges that is obtained by removing the arrowheads from the directed edges in G :

$$\hat{\mathcal{E}} = \{\{v, w\} : (v, w) \in \mathcal{E} \vee (w, v) \in \mathcal{E}\}.$$

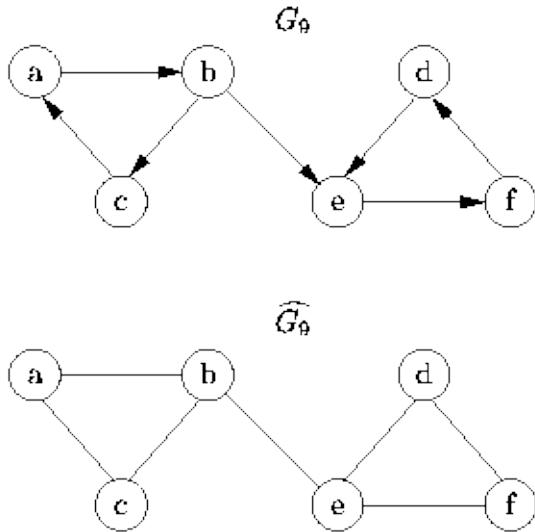


Figure: An weakly connected directed graph and the underlying undirected graph.

Weak connectedness of a directed graph is defined with respect to its underlying, undirected graph:

Definition (Weak Connectedness of a Directed Graph) A directed graph $G = (\mathcal{V}, \mathcal{E})$ is *weakly connected* if the underlying undirected graph \widehat{G} is connected.

For example, since the undirected graph \widehat{G}_9 in Figure □ is connected, the directed graph G_9 is *weakly connected*. Consider what happens when we remove the edge (b,e) from the directed graph G_9 . The underlying undirected graph that we get is \widehat{G}_8 in Figure □. Therefore, when we remove edge (b,e) from G_9 , the graph that remains is neither strongly connected nor weakly connected.



Testing Strong Connectedness

A traversal of a directed graph (either depth-first or breadth-first) starting from a given vertex will only visit all the vertices of an undirected graph if there is a path from the start vertex to every other vertex. Therefore, a simple way to test whether a directed graph is strongly connected uses $|V|$ traversals--one starting from each vertex in V . Each time the number of vertices visited is counted. The graph is strongly connected if all the vertices are visited in each traversal.

Program □ shows how this can be implemented. It shows the `getIsStronglyConnected` method of the abstract `Digraph` class which returns the `bool` value `True` if the graph is *strongly* connected.

```

1  class Digraph(Graph):
2
3      def getIsStronglyConnected(self):
4          for v in xrange(self.numberOfVertices):
5              visitor = self.CountingVisitor()
6              self.depthFirstTraversal(PreOrder(visitor), v)
7              if visitor.count != self.numberOfVertices:
8                  return False
9          return True
10
11      # ...

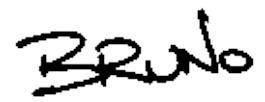
```

Program: Abstract `Digraph` class `getIsConnected` method.

The accessor consists of a loop over all the vertices of the graph. Each iteration does a `depthFirstTraversal` using a visitor that counts the number of vertices it visits. The running time for one iteration is essentially that of the `depthFirstTraversal` since $T(\text{visit}) = O(1)$ for the counting visitor. Therefore, the worst-case running time for the `getIsConnected` method is $O(|V|^3)$ when adjacency matrices are used and $O(|V|^2 + |V| \cdot |E|)$ when adjacency lists are used to represent the graph.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Testing for Cycles in a Directed Graph

The final application of graph traversal that we consider in this section is to test a directed graph for cycles. An easy way to do this is to attempt a topological-order traversal using the algorithm given in Section □. This algorithm only visits all the vertices of a directed graph if that graph contains no cycles.

To see why this is so, consider the directed cyclic graph G_{10} shown in Figure □. The topological traversal algorithm begins by computing the *in-degrees* of the vertices. (The number shown below each vertex in Figure □ is the in-degree of that vertex).

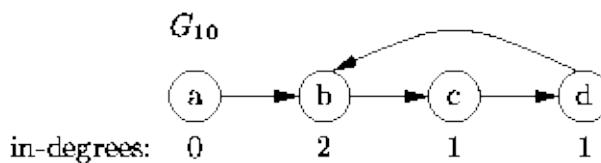


Figure: A directed cyclic graph.

At each step of the traversal, a vertex with in-degree of zero is visited. After a vertex is visited, the vertex and all the edges emanating from that vertex are removed from the graph. Notice that if we remove vertex a and edge (a,b) from G_{10} , all the remaining vertices have in-degrees of one. The presence of the cycle prevents the topological-order traversal from completing.

Therefore, the a simple way to test whether a directed graph is cyclic is to attempt a topological traversal of its vertices. If all the vertices are not visited, the graph must be cyclic.

Program □ gives the implementation of the `getIsCyclic` method of the `Digraph` class. This method returns `True` if the graph is cyclic. The implementation uses a visitor that counts the number of vertices visited during a `topologicalOrderTraversal` of the graph.

```
1 class Digraph(Graph):
2
3     def getIsCyclic(self):
4         visitor = self.CountingVisitor()
5         self.topologicalOrderTraversal(visitor)
6         return visitor.count != self.numberOfVertices
7
8     # ...
```

Program: Abstract Digraph class getIsCyclic method.

The worst-case running time of the getIsCyclic method is determined by the time taken by the topologicalOrderTraversal. Since $T_{\text{visit}} = O(1)$, the running time of getIsCyclic is $O(|V|^2)$ when adjacency matrices are used to represent the graph and $O(|V| + |E|)$ when adjacency lists are used.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Shortest-Path Algorithms

In this section we consider edge-weighted graphs, both directed and undirected, in which the weight measures the *cost* of traversing that edge. The units of cost depend on the application.

For example, we can use a directed graph to represent a network of airports. In such a graph the vertices represent the airports and the edges correspond to the available flights between airports. In this scenario there are several possible cost metrics: If we are interested in computing travel time, then we use an edge-weighted graph in which the weights represent the flying time between airports. If we are concerned with the financial cost of a trip, then the weights on the edges represent the monetary cost of a ticket. Finally, if we are interested in the actual distance traveled, then the weights represent the physical distances between airports.

If we are interested in traveling from point A to B , we can use a suitably labeled graph to answer the following questions: What is the fastest way to get from A to B ? Which route from A to B has the least expensive airfare? What is the shortest possible distance traveled to get from A to B ?

Each of these questions is an instance of the same problem: Given an edge-weighted graph, $G = (\mathcal{V}, \mathcal{E})$, and two vertices, $v_s \in \mathcal{V}$ and $v_d \in \mathcal{V}$, find the path that starts at v_s and ends at v_d that has the smallest weighted path length. The weighted length of a path is defined as follows:

Definition (Weighted Path Length) Consider an edge-weighted graph $G = (\mathcal{V}, \mathcal{E})$. Let $C(v_i, v_j)$ be the weight on the edge connecting v_i to v_j . A path in G is a non-empty sequence of vertices $P = \{v_1, v_2, \dots, v_k\}$. The *weighted path length* of path P is given by

$$\sum_{i=1}^{k-1} C(v_i, v_{i+1}).$$

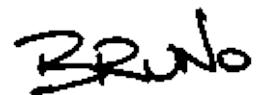
The *weighted* length of a path is the sum of the weights on the edges in that path. Conversely, the *unweighted* length of a path is simply the number of edges in

that path. Therefore, the *unweighted* length of a path is equivalent to the weighted path length obtained when all edge weights are one.

- [Single-Source Shortest Path](#)
 - [All-Pairs Source Shortest Path](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Single-Source Shortest Path

In this section we consider the *single-source shortest path* problem: Given an edge-weighted graph $G = (\mathcal{V}, \mathcal{E})$ and a vertex $v_s \in \mathcal{V}$, find the shortest weighted path from v_s to every other vertex in \mathcal{V} .

Why do we find the shortest path to every other vertex if we are interested only in the shortest path from, say, v_s to v_d ? It turns out that in order to find the shortest path from v_s to v_d , it is necessary to find the shortest path from v_s to every other vertex in G ! If a vertex is ignored, say v_i , then we will not consider any of the paths from v_s to v_d that pass through v_i . But if we fail to consider all the paths from v_s to v_d , we cannot be assured of finding the shortest one.

Furthermore, suppose the shortest path from v_s to v_d passes through some intermediate node v_i . That is, the shortest path is of the form

$P = \{v_s, \dots, v_i, \dots, v_d\}$. It must be the case that the portion of P between v_s to v_i is also the shortest path from v_s to v_i . Suppose it is not. Then there exists another shorter path from v_s to v_i . But then, P would not be the shortest path from v_s to v_d , because we could obtain a shorter one by replacing the portion of P between v_s and v_i by the shorter path.

Consider the directed graph G_{11} shown in Figure □. The shortest *weighted* path between vertices b and f is the path $\{b, a, c, e, f\}$, which has the weighted path length nine. On the other hand, the shortest *unweighted* path is from b to f is the path of length three, $\{b, c, e, f\}$.

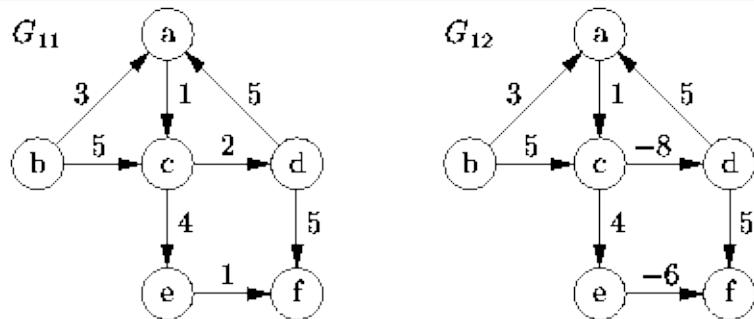


Figure: Two edge-weighted directed graphs.

As long as all the edge weights are non-negative (as is the case for G_{11}), the shortest-path problem is well defined. Unfortunately, things get a little tricky in the presence of negative edge weights.

For example, consider the graph G_{12} shown in Figure □. Suppose we are looking for the shortest path from d to f . Exactly two edges emanate from vertex d , both with the same edge weight of five. If the graph contained only positive edge weights, there could be no shorter path than the direct path $\{d, f\}$.

However, in a graph that contains negative weights, a long path gets ``shorter'' when we add edges with negative weights to it. For example, the path $\{d, a, c, e, f\}$ has a total weighted path length of four, even though the first edge, (d, a) , has the weight five.

But negative weights are even more insidious than this: For example, the path $\{d, a, c, d, a, e, f\}$, which also joins vertex d to f , has a weighted path length of two but the path $\{d, a, c, d, a, c, d, a, e, f\}$ has length zero. That is, as the number of edges in the path increases, the weighted path length decreases! The problem in this case is the existence of the cycle $\{d, a, c, d\}$ the weighted path length of which is less than zero. Such a cycle is called a *negative cost cycle*.

Clearly, the shortest-path problem is not defined for graphs that contain negative cost cycles. However, negative edges are not intrinsically bad. Solutions to the problem do exist for graphs that contain both positive and negative edge weights, as long as there are no negative cost cycles. Nevertheless, the problem is greatly simplified when all edges carry non-negative weights.

-
- [Dijkstra's Algorithm](#)
 - [Data Structures for Dijkstra's Algorithm](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
-

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm for solving the single-source, shortest-path problem on an edge-weighted graph in which all the weights are non-negative. It finds the shortest paths from some initial vertex, say v_s , to all the other vertices one-by-one. The essential feature of Dijkstra's algorithm is the order in which the paths are determined: The paths are discovered in the order of their weighted lengths, starting with the shortest, proceeding to the longest.

For each vertex v , Dijkstra's algorithm keeps track of three pieces of information, k_v , d_v , and p_v :

k_v

The bool-valued flag k_v indicates that the shortest path to vertex v is *known*. Initially, $k_v = \text{False}$ for all $v \in V$.

d_v

The quantity d_v is the length of the shortest known path from v_s to v . When the algorithm begins, no shortest paths are known. The distance d_v is a *tentative* distance. During the course of the algorithm candidate paths are examined and the *tentative* distances are modified.

Initially, $d_v = \infty$ for all $v \in V$ such that $v \neq v_s$, while $d_{v_s} = 0$.

p_v

The predecessor of vertex v on the shortest path from v_s to v . That is, the shortest path from v_s to v has the form $\{v_s, \dots, p_v, v\}$.

Initially, p_v is unknown for all $v \in V$.

Dijkstra's algorithm proceeds in phases. The following steps are performed in each pass:

- From the set of vertices for which $k_v = \text{False}$, select the vertex v having the smallest tentative distance d_v .
- Set $k_v \leftarrow \text{True}$.
- For each vertex w adjacent to v for which $k_w \neq \text{True}$, test whether the tentative distance d_w is greater than $d_v + C(v, w)$. If it is, set $d_w \leftarrow d_v + C(v, w)$ and set $p_w \leftarrow v$.

In each pass exactly one vertex has its k_v set to True. The algorithm terminates after $|V|$ passes are completed at which time all the shortest paths are known.

Table \square illustrates the operation of Dijkstra's algorithm as it finds the shortest paths starting from vertex b in graph G_{11} shown in Figure \square .

Table: Operation of Dijkstra's algorithm.

	vertex initially	1	2	3	4	5	6
a	∞	$3 b \checkmark$	$3 b \checkmark$	$3 b \checkmark$	$3 b \checkmark$	$3 b \checkmark$	$3 b$
b	0	-- \checkmark	$0 -- \checkmark$	$0 -- \checkmark$	$0 -- \checkmark$	$0 -- \checkmark$	$0 --$
c	∞	$5 b$	$4 a \checkmark$	$4 a \checkmark$	$4 a \checkmark$	$4 a \checkmark$	$4 a$
d	∞	∞	∞	$6 c \checkmark$	$6 c \checkmark$	$6 c \checkmark$	$6 c$
e	∞	∞	∞	$8 c \checkmark$	$8 c \checkmark$	$8 c \checkmark$	$8 c$
f	∞	∞	∞	∞	$11 d$	$9 e \checkmark$	$9 e$

Initially all the tentative distances are ∞ , except for vertex b which has tentative distance zero. Therefore, vertex b is selected in the first pass. The mark \checkmark beside an entry in Table \square indicates that the shortest path is known ($k_v = \text{True}$).

Next we follow the edges emanating from vertex b , $b \rightarrow a$ and $b \rightarrow c$, and update the distances accordingly. The new tentative distances for a becomes 3 and the

new tentative distance for c is 5. In both cases, the next-to-last vertex on the shortest path is vertex b .

In the second pass, vertex a is selected and its entry is marked with \checkmark indicating the shortest path is known. There is one edge emanating from a , $a \rightarrow c$. The distance to c via a is 4. Since this is less than the tentative distance to c , vertex c is given the new tentative distance 4 and its predecessor on the shortest-path is set to a . The algorithm continues in this fashion for a total of V passes until all the shortest paths have been found.

The shortest-path information contained in the right-most column of Table \square can be represented in the form of a vertex-weighted graph as shown in Figure \square .

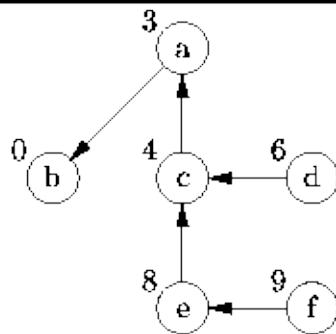


Figure: The shortest-path graph for G_{11} .

This graph contains the same set of vertices as the problem graph G_{11} . Each vertex v is labeled with the length d_v of the shortest path from b to v . Each vertex (except b) has a single emanating edge that connects the vertex to the next-to-last vertex on the shortest-path. By following the edges in this graph from any vertex v to vertex b , we can construct the shortest path from b to v in reverse.

Data Structures for Dijkstra's Algorithm

The implementation of Dijkstra's algorithm described below uses the `Entry` class declared in Program □. Each `Entry` value has three instance attributes, known, distance, and predecessor, which correspond to the variables k_v , d_v , and p_v , respectively.

```
1  class Algorithms(object):
2
3      class Entry(object):
4
5          def __init__(self):
6              self.known = False
7              self.distance = sys.maxint
8              self.predecessor = sys.maxint
```

Program: `Algorithms.Entry` class `__init__` method.

In each pass of its operation, Dijkstra's algorithm selects from the set of vertices for which the shortest-path is not yet known the one with the smallest tentative distance. Therefore, we use a *priority queue* to represent this set of vertices.

The priority assigned to a vertex is its tentative distance. The class `Association` class introduced in Program □ is used to associate a priority with a given vertex instance.

Implementation

An implementation of Dijkstra's algorithm is shown in Program □. The `DijkstrasAlgorithm` method takes two arguments. The first is a directed graph. It is assumed that the directed graph is an edge-weighted graph in which the weights are `ints`. The second argument is the number of the start vertex, v_s .

The `DijkstrasAlgorithm` method returns its result in the form of a shortest-path graph. Therefore, the return value is a `Digraph` instance.

```

1  class Algorithms(object):
2
3      def DijkstrasAlgorithm(g, s):
4          n = g.numberOfVertices
5          table = Array(n)
6          for v in xrange(n):
7              table[v] = Algorithms.Entry()
8          table[s].distance = 0
9          queue = BinaryHeap(g.numberOfEdges)
10         queue.enqueue(Association(0, g[s]))
11         while not queue.isEmpty():
12             assoc = queue.dequeueMin()
13             v0 = assoc.value
14             if not table[v0.number].known:
15                 table[v0.number].known = True
16                 for e in v0.emanatingEdges:
17                     v1 = e.mateOf(v0)
18                     d = table[v0.number].distance + e.weight
19                     if table[v1.number].distance > d:
20
21                         table[v1.number].distance = d
22                         table[v1.number].predecessor = v0.number
23                         queue.enqueue(Association(d, v1))
24         result = DigraphAsLists(n)
25         for v in xrange(n):
26             result.addVertex(v, table[v].distance)
27         for v in xrange(n):
28             if v != s:
29                 result.addEdge(v, table[v].predecessor)
30         return result
31     DijkstrasAlgorithm = staticmethod(DijkstrasAlgorithm)

```

Program: Dijkstra's algorithm.

The main data structures used are called `table` and `queue` (lines 5 and 9). The former is an array of $n = |\mathcal{V}|$ `Entry` elements. The latter is a priority queue. In this case, a `BinaryHeap` of length $|\mathcal{E}|$ is used. (See Section □).

The algorithm begins by setting the tentative distance for the start vertex to zero and inserting the start vertex into the priority queue with priority zero (lines 8-10).

The main loop of the method comprises lines 11-23. In each iteration of this loop the vertex with the smallest distance is dequeued (line 12). The vertex is

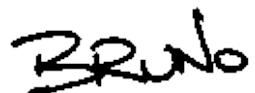
processed only if its table entry indicates that the shortest path is not already known (line 14).

When a vertex v_0 is processed, its shortest path is deemed to be *known* (line 15). Then each vertex v_1 adjacent to v_0 is considered (lines 16-23). The distance to v_1 along the path that passes through v_0 is computed (line 18). If this distance is less than the tentative distance associated with v_1 , entries in the table for v_1 are updated, and the v_1 is given a new priority and inserted into the priority queue (lines 19-23).

The main loop terminates when all the shortest paths have been found. The shortest-path graph is then constructed using the information in the table (lines 24-29).

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Running Time Analysis

The running time of the `DijkstrasAlgorithm` method is dominated by the running time of the main loop (lines 11-23). (It is easy to see that lines 4-10 and 24-39 run in $O(|V|)$ time).

To determine the running time of the main loop, we proceed as follows: First, we ignore temporarily the time required for the enqueue and dequeue operations in the priority queue. Clearly, each vertex in the graph is processed exactly once. When a vertex is processed all the edges emanating from it are considered. Therefore, the time (ignoring the priority queue operations) taken is $O(|V|+|E|)$ when adjacency lists are used and $O(|V|^2)$ when adjacency matrices are used.

Now, we add back the worst-case time required for the priority queue operations. In the worst case, a vertex is enqueued and subsequently dequeued once for every edge in the graph. Therefore, the length of the priority queue is at most $|E|$. As a result, the worst-case time for each operation is $O(\log |E|)$.

Thus, the worst-case running time for Dijkstra's algorithm is

$$O(|V| + |E| \log |E|),$$

when adjacency lists are used, and

$$O(|V|^2 + |E| \log |E|),$$

when adjacency matrices are used to represent the input graph.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

All-Pairs Source Shortest Path

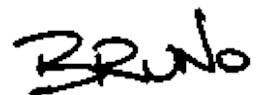
In this section we consider the *all-pairs, shortest path* problem: Given an edge-weighted graph $G = (\mathcal{V}, \mathcal{E})$, for each pair of vertices in \mathcal{V} find the *length* of the shortest weighted path between the two vertices.

One way to solve this problem is to run Dijkstra's algorithm $|\mathcal{V}|$ times in turn using each vertex in \mathcal{V} as the initial vertex. Therefore, we can solve the all-pairs problem in $O(|\mathcal{V}|^2 + |\mathcal{V}||\mathcal{E}| \log |\mathcal{E}|)$ time when adjacency lists are used, and $O(|\mathcal{V}|^3 + |\mathcal{V}||\mathcal{E}| \log |\mathcal{E}|)$, when adjacency matrices are used. However, for a dense graph ($|\mathcal{E}| = \Theta(|\mathcal{V}|^2)$) the running time of Dijkstra's algorithm is $O(|\mathcal{V}|^3 \log |\mathcal{V}|)$, regardless of the representation scheme used.

- [Floyd's Algorithm](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Floyd's Algorithm

Floyd's algorithm uses the dynamic programming method to solve the all-pairs shortest-path problem on a dense graph. The method makes efficient use of an adjacency matrix to solve the problem. Consider an edge-weighted graph $G = (\mathcal{V}, \mathcal{E})$, where $C(v, w)$ represents the weight on edge (v, w) . Suppose the vertices are numbered from 1 to $|\mathcal{V}|$. That is, let $\mathcal{V} = \{v_1, v_2, \dots, v_{|\mathcal{V}|}\}$. Furthermore, let \mathcal{V}_k be the set comprised of the first k vertices in \mathcal{V} . That is, $\mathcal{V}_k = \{v_1, v_2, \dots, v_k\}$, for $0 \leq k \leq |\mathcal{V}|$.

Let $P_k(v, w)$ be the shortest path from vertex v to w that passes only through vertices in \mathcal{V}_k , if such a path exists. That is, the path $P_k(v, w)$ has the form

$$P_k(v, w) = \{v, \underbrace{\dots}_{\in \mathcal{V}_k}, w\}.$$

Let $D_k(v, w)$ be the *length* of path $P_k(v, w)$:

$$D_k(v, w) = \begin{cases} |P_k(v, w)| & P_k(v, w) \text{ exists,} \\ \infty & \text{otherwise.} \end{cases}$$

Since $\mathcal{V}_0 = \emptyset$, the P_0 paths are correspond to the edges of G :

$$P_0(v, w) = \begin{cases} \{v, w\} & (v, w) \in \mathcal{E}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Therefore, the D_0 path lengths correspond to the weights on the edges of G :

$$D_0(v, w) = \begin{cases} C(v, w) & (v, w) \in \mathcal{E}, \\ \infty & \text{otherwise.} \end{cases}$$

Floyd's algorithm computes the sequence of matrices $D_0, D_1, \dots, D_{|\mathcal{V}|}$. The distances in D_i represent paths with intermediate vertices in \mathcal{V}_i . Since

$\mathcal{V}_{i+1} = \mathcal{V}_i \cup \{v_{i+1}\}$, we can obtain the distances in D_{i+1} from those in D_i by considering only the paths that pass through vertex v_{i+1} . Figure \square illustrates how this is done.

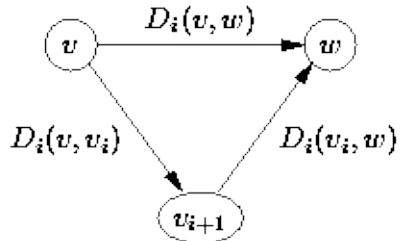


Figure: Calculating D_{i+1} in Floyd's algorithm.

For every pair of vertices (v, w) , we compare the distance $D_i(v, w)$, (which represents the shortest path from v to w that does not pass through v_{i+1}) with the sum $D_i(v, v_{i+1}) + D_i(v_{i+1}, w)$ (which represents the shortest path from v to w that does pass through v_{i+1}). Thus, D_{i+1} is computed as follows:

$$D_{i+1}(v, w) = \min\{D_i(v, v_{i+1}) + D_i(v_{i+1}, w), D_i(v, w)\}.$$

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

An implementation of Floyd's algorithm is shown in Program □. The `FloydsAlgorithm` method takes as its argument a directed graph. The directed graph is assumed to be an edge-weighted graph in which the weights are ints.

```

1  class Algorithms(object):
2
3      def FloydsAlgorithm(g):
4          n = g.numberOfVertices
5          distance = DenseMatrix(n, n)
6          for v in xrange(n):
7              for w in xrange(n):
8                  distance[v, w] = sys.maxint
9          for e in g.edges:
10              distance[e.v0.number, e.v1.number] = e.weight
11          for i in xrange(n):
12              for v in xrange(n):
13                  for w in xrange(n):
14                      if distance[v, i] != sys.maxint and \
15                          distance[i, w] != sys.maxint:
16                          d = distance[v, i] + distance[i, w]
17                          if distance[v, w] > d:
18                              distance[v, w] = d
19          result = DigraphAsMatrix(n)
20          for v in xrange(n):
21              result.addVertex(v)
22          for v in xrange(n):
23              for w in xrange(n):
24                  if distance[v, w] != sys.maxint:
25                      result.addEdge(v, w, distance[v, w])
26          return result
27      FloydsAlgorithm = staticmethod(FloydsAlgorithm)

```

Program: Floyd's algorithm.

The `FloydsAlgorithm` method returns its result in the form of an edge-weighted directed graph. Therefore, the return value is a `Digraph`.

The principal data structure used by the algorithm is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix of integers

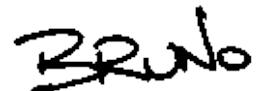
called distance. All the elements of the matrix are initially set to ∞ (lines 6-8). Next, an edge iterator is used to visit all the edges in the input graph in order to transfer the weights from the graph to the distance matrix (lines 9-10).

The main work of the algorithm is done in three, nested loops (lines 11-18). The outer loop computes the sequence of distance matrices $D_1, D_2, \dots, D_{|v|}$. The inner two loops consider all possible pairs of vertices. Notice that as D_{i+1} is computed, its entries overwrite those of D_i .

Finally, the values in the distance matrix are transferred to the result graph (lines 19-25). The result graph contains the same set of vertices as the input graph. For each finite entry in the distance matrix, a weighted edge is added to the result graph.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Running Time Analysis

The worst-case running time for Floyd's algorithm is easily determined. Creating and initializing the distance matrix is $O(|V|^2)$ (lines 5-8). Transferring the weights from the input graph to the distance matrix requires $O(|V| + |E|)$ time if adjacency lists are used, and $O(|V|^2)$ time when an adjacency matrix is used to represent the input graph (lines 9-10).

The running time for the three nested loops is $O(|V|^3)$ in the worst case. Finally, constructing the result graph and transferring the entries from the distance matrix to the result requires $O(|V|^2)$ time. As a result, the worst-case running time of Floyd's algorithm is $O(|V|^3)$.

Minimum-Cost Spanning Trees

In this section we consider undirected graphs and their subgraphs. A *subgraph* of a graph $G = (\mathcal{V}, \mathcal{E})$ is any graph $G' = (\mathcal{V}', \mathcal{E}')$ such that $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' \subseteq \mathcal{E}$. In particular, we consider *connected* undirected graphs and their *minimal subgraphs*. The minimal subgraph of a connected graph is called a *spanning tree*:

Definition (Spanning Tree) Consider a *connected, undirected* graph $G = (\mathcal{V}, \mathcal{E})$. A *spanning tree* of G is a subgraph of G , say $T = (\mathcal{V}', \mathcal{E}')$, with the following properties:

1. $\mathcal{V}' = \mathcal{V}$.
2. T is connected.
3. T is acyclic.

Figure \square shows an undirected graph, G_{13} , together with three of its spanning trees. A spanning tree is called a *tree* because every *acyclic* undirected graph can be viewed as a general, unordered tree. Because the edges are undirected, any vertex may be chosen to serve as the root of the tree. For example, the spanning tree of G_{13} given in Figure \square (c) can be viewed as the general, unordered tree

$$\{b, \{a\}, \{c\}, \{e, \{d\}, \{f\}\}\}.$$

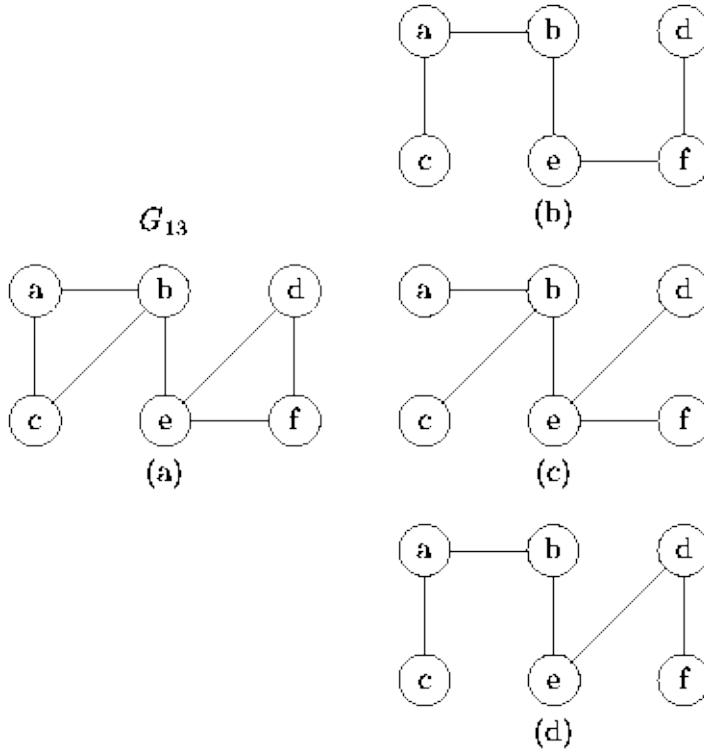


Figure: An undirected graph and three spanning trees.

According to Definition \square , a spanning tree is connected. Therefore, as long as the tree contains more than one vertex, there can be no vertex with degree zero. Furthermore, the following theorem guarantees that there is always at least one vertex with degree one:

Theorem Consider a connected, undirected graph $G = (\mathcal{V}, \mathcal{E})$, where $|\mathcal{V}| > 1$.

Let $T = (\mathcal{V}, \mathcal{E}')$ be a spanning tree of G . The spanning tree T contains at least one vertex of degree one.

extbfProof (By contradiction). Assume that there is no vertex in T of degree one. That is, all the vertices in T have degree two or greater. Then by following edges into and out of vertices we can construct a path that is cyclic. But a spanning tree is acyclic--a contradiction. Therefore, a spanning tree always contains at least one vertex of degree one.

According to Definition \square , the edge set of a spanning tree is a subset of the edges in the spanned graph. How many edges must a spanning tree have? The following theorem answers the question:

Theorem Consider a connected, undirected graph $G = (\mathcal{V}, \mathcal{E})$. Let $T = (\mathcal{V}, \mathcal{E}')$ be a spanning tree of G . The number of edges in the spanning tree is given by

$$|\mathcal{E}'| = |\mathcal{V}| - 1.$$

extbfProof (By induction). We can prove Theorem \square by induction on $|\mathcal{V}|$, the number of vertices in the graph.

Base Case Consider a graph that contains only one node, i.e., $|\mathcal{V}| = 1$. Clearly, the spanning tree for such a graph contains no edges. Since $|\mathcal{V}| - 1 = 0$, the theorem is valid.

Inductive Hypothesis Assume that the number of edges in a spanning tree for a graph with $|\mathcal{V}|$ has been shown to be $|\mathcal{V}| - 1$ for $|\mathcal{V}| = 1, 2, \dots, k$.

Consider a graph $G_{k+1} = (\mathcal{V}, \mathcal{E})$ with $k+1$ vertices and its spanning tree $T_{k+1} = (\mathcal{V}, \mathcal{E}')$. According to Theorem \square , G_{k+1} contains at least one vertex of degree one. Let $v \in \mathcal{V}$ be one such vertex and $\{v, w\} \in \mathcal{E}'$ be the one edge emanating from v in T_{k+1} .

Let T_k be the graph of k nodes obtained by removing v and its emanating edge from the graph T_{k+1} . That is, $T_k = (\mathcal{V} - \{v\}, \mathcal{E}' - \{v, w\})$.

Since T_{k+1} is connected, so too is T_k . Similarly, since T_{k+1} is acyclic, so too is T_k . Therefore T_k is a spanning tree with k vertices. By the inductive hypothesis T_k has $k-1$ edges. Thus, T_{k+1} has k edges.

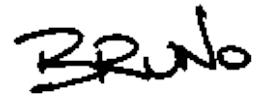
Therefore, by induction on k , the spanning tree for a graph with $|\mathcal{V}|$ vertices contains $|\mathcal{V}| - 1$ edges.

- [Constructing Spanning Trees](#)
- [Minimum-Cost Spanning Trees](#)

- [Prim's Algorithm](#)
 - [Kruskal's Algorithm](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".



Constructing Spanning Trees

Any traversal of a connected, undirected tree visits all the vertices in that tree, regardless of the node from which the traversal is started. During the traversal certain edges are traversed while the remaining edges are not. Specifically, an edge is traversed if it leads from a vertex that has been visited to a vertex that has not been visited. The set of edges which are traversed during a traversal forms a spanning tree.

The spanning tree obtained from a breadth-first traversal starting at vertex v of graph G is called the *breadth-first spanning tree* of G rooted at v . For example, the spanning tree shown in Figure □ (c) is the breadth-first spanning tree of G_{13} rooted at vertex b .

Similarly, the spanning tree obtained from a depth-first traversal is the *depth-first spanning tree* of G rooted at v . The spanning tree shown in Figure □ (d) is the depth-first spanning tree of G_{13} rooted at vertex c .



Minimum-Cost Spanning Trees

The total *cost* of an edge-weighted undirected graph is simply the sum of the weights on all the edges in that graph. A minimum-cost spanning tree of a graph is a spanning tree of that graph that has the least total cost:

Definition (Minimal Spanning Tree) Consider an *edge-weighted, undirected, connected graph* $G = (\mathcal{V}, \mathcal{E})$, where $C(v, w)$ represents the weight on edge $\{v, w\} \in \mathcal{E}$. The *minimum spanning tree* of G is the spanning tree $T = (\mathcal{V}, \mathcal{E}')$ that has the smallest total cost,

$$\sum_{\{v, w\} \in \mathcal{E}'} C(v, w).$$

Figure □ shows edge-weighted graph G_{14} together with its minimum-cost spanning tree T_{14} . In general, it is possible for a graph to have several different minimum-cost spanning trees. However, in this case there is only one.

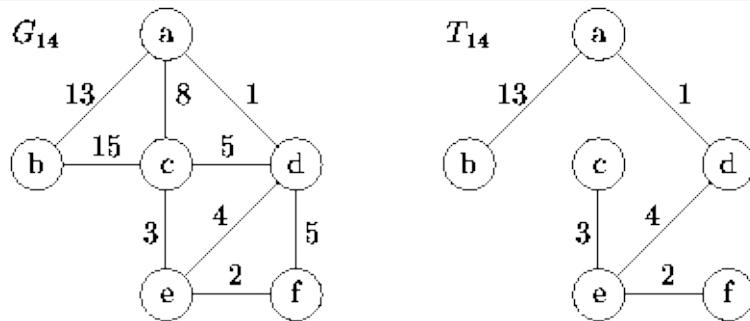


Figure: An edge-weighted, undirected graph and a minimum-cost spanning tree.

The two sections that follow present two different algorithms for finding the minimum-cost spanning tree. Both algorithms are similar in that they build the tree one edge at a time.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Prim's Algorithm

Prim's algorithm finds a minimum-cost spanning tree of an edge-weighted, connected, undirected graph $G = (\mathcal{V}, \mathcal{E})$. The algorithm constructs the minimum-cost spanning tree of a graph by selecting edges from the graph one-by-one and adding those edges to the spanning tree.

Prim's algorithm is essentially a minor variation of *Dijkstra's algorithm* (see Section □). To construct the spanning tree, the algorithm constructs a sequence of spanning trees $T_0, T_1, \dots, T_{|\mathcal{V}|-1}$, each of which is a subgraph of G . The algorithm begins with a tree that contains one selected vertex, say $v_n \in \mathcal{V}$. That is, $T_0 = \{\{v_n\}, \emptyset\}$.

Given $T_i = \{\mathcal{V}_i, \mathcal{E}_i\}$, we obtain the next tree in the sequence as follows. Consider the set of edges given by

$$\mathcal{H}_i = \bigcup_{u \in \mathcal{V}_i} \mathcal{A}(u) - \bigcup_{u \in \mathcal{V}_i} \mathcal{I}(u).$$

The set \mathcal{H}_i contains all the edges $\{v, w\}$ such that exactly one of v or w is in \mathcal{V}_i (but not both). Select the edge $\{v, w\} \in \mathcal{H}_i$ with the smallest edge weight,

$$C(v, w) = \min_{\{v', w'\} \in \mathcal{H}_i} C(v', w').$$

Then $T_{i+1} = \{\mathcal{V}_{i+1}, \mathcal{E}_{i+1}\}$, where $\mathcal{V}_{i+1} = \mathcal{V}_i \cup \{v\}$ and $\mathcal{E}_{i+1} = \mathcal{E}_i \cup \{\{v, w\}\}$. After $|\mathcal{V}| - 1$ such steps we get $T_{|\mathcal{V}|-1}$ which is the minimum-cost spanning tree of G .

Figure □ illustrates how Prim's algorithm determines the minimum-cost spanning tree of the graph G_{14} shown in Figure □. The circled vertices are the elements of \mathcal{V}_i , the solid edges represent the elements of \mathcal{E}_i and the dashed edges represent the elements of \mathcal{H}_i .

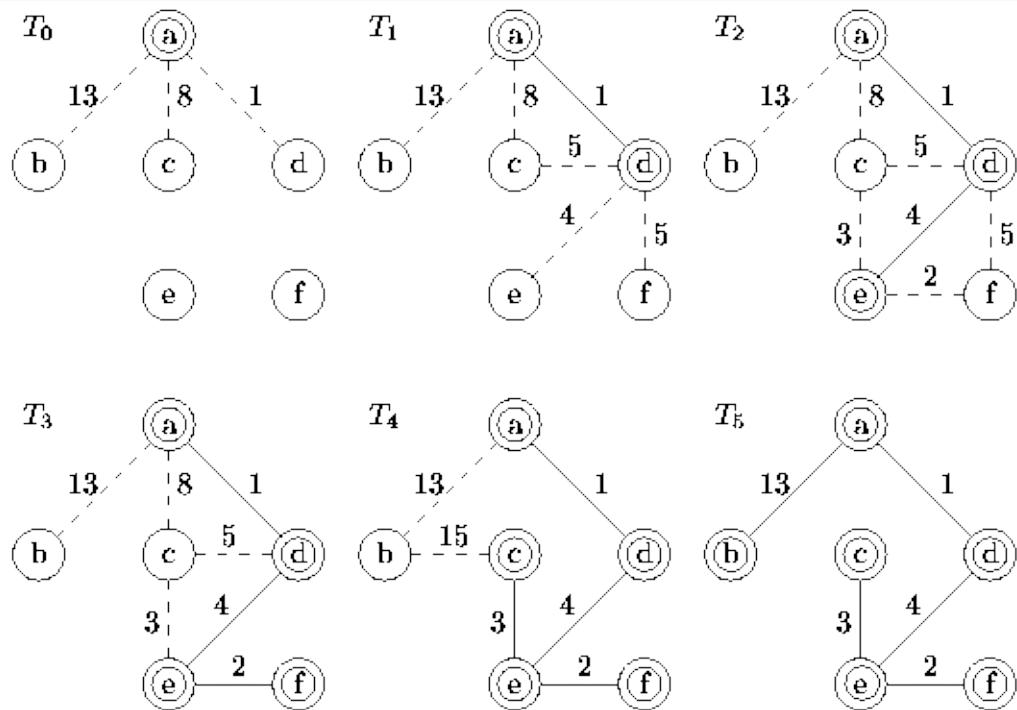


Figure: Operation of Prim's algorithm.

- [Implementation](#)

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

An implementation of Prim's algorithm is shown in Program □. This implementation is almost identical to the version of *Dijkstra's* algorithm given in Program □. In fact, there are only four differences between the two algorithms. These are found on lines 3, 18-20, 24, and 26.

```
 1 class Algorithms(object):
 2
 3     def PrimsAlgorithm(g, s):
 4         n = g.numberOfVertices
 5         table = Array(n)
 6         for v in xrange(n):
 7             table[v] = Algorithms.Entry()
 8         table[s].distance = 0
 9         queue = BinaryHeap(g.numberOfEdges)
10         queue.enqueue(Association(0, g[s]))
11         while not queue.isEmpty():
12             assoc = queue.dequeueMin()
13             v0 = assoc.value
14             if not table[v0.number].known:
15                 table[v0.number].known = True
16                 for e in v0.emanatingEdges:
17                     v1 = e.mateOf(v0)
18                     d = e.weight
19                     if not table[v1.number].known and \
20                         table[v1.number].distance > d:
21                         table[v1.number].distance = d
22                         table[v1.number].predecessor = v0.number
23                         queue.enqueue(Association(d, v1))
24         result = GraphAsLists(n)
25         for v in xrange(n):
26             result.addVertex(v)
27         for v in xrange(n):
28             if v != s:
29                 result.addEdge(v, table[v].predecessor)
30         return result
31     PrimsAlgorithm = staticmethod(PrimsAlgorithm)
```

Program: Prim's algorithm.

The `PrimsAlgorithm` method takes two arguments. The first is an undirected graph instance. We assume that the graph is edge-weighted and that the weights are `ints`. The second argument is the number of the start vertex, v_s .

The `PrimsAlgorithm` method returns a minimum-cost spanning tree represented as an undirected graph. Therefore, the return value is a `Graph`.

The running time of Prim's algorithm is asymptotically the same as Dijkstra's algorithm. That is, the worst-case running time is

$$O(|V| + |E| \log |E|),$$

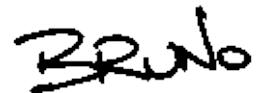
when adjacency lists are used, and

$$O(|V|^2 + |E| \log |E|),$$

when adjacency matrices are used to represent the input graph.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Kruskal's Algorithm

Like Prim's algorithm, *Kruskal's algorithm* also constructs the minimum spanning tree of a graph by adding edges to the spanning tree one-by-one. At all points during its execution the set of edges selected by Prim's algorithm forms exactly one tree. On the other hand, the set of edges selected by Kruskal's algorithm forms a forest of trees.

Kruskal's algorithm is conceptually quite simple. The edges are selected and added to the spanning tree in increasing order of their weights. An edge is added to the tree only if it does not create a cycle.

The beauty of Kruskal's algorithm is the way that potential cycles are detected. Consider an undirected graph $G = (\mathcal{V}, \mathcal{E})$. We can view the set of vertices, \mathcal{V} , as a *universal set* and the set of edges, \mathcal{E} , as the definition of an *equivalence relation* over the universe \mathcal{V} . (See Definition \square). In general, an equivalence relation partitions a universal set into a set of equivalence classes. If the graph is connected, there is only one equivalence class--all the elements of the universal set are *equivalent*. Therefore, a *spanning tree* is a minimal set of equivalences that result in a single equivalence class.

Kruskal's algorithm computes, $P_0, P_1, \dots, P_{|\mathcal{V}-1|}$, a sequence of *partitions* of the set of vertices \mathcal{V} . (Partitions are discussed in Section \square). The initial partition consists of $|\mathcal{V}|$ sets of size one:

$$P_0 = \{\{v_1\}, \{v_2\}, \dots, \{v_{|\mathcal{V}|\}}\}.$$

Each subsequent element of the sequence is obtained from its predecessor by *joining* two of the elements of the partition. Therefore, P_i has the form

$$P_i = \{S_0^i, S_1^i, \dots, S_{|\mathcal{V}|-1-i}^i\},$$

for $0 \leq i \leq |\mathcal{V}| - 1$.

To construct the sequence the edges in \mathcal{E} are considered one-by-one in increasing order of their weights. Suppose we have computed the sequence up to P_i and the next edge to be considered is $\{v, w\}$. If v and w are both members of the same

element of partition P_i , then the edge forms a cycle, and is not part of the minimum-cost spanning tree.

On the other hand, suppose v and w are members of two different elements of partition P_i , say S_k^i and S_l^i (respectively). Then $\{v, w\}$ must be an edge in the minimum-cost spanning tree. In this case, we compute P_{i+1} by *joining* S_k^i and S_l^i . That is, we replace S_k^i and S_l^i in P_i by the *union* $S_k^i \cup S_l^i$.

Figure □ illustrates how Kruskal's algorithm determines the minimum-cost spanning tree of the graph G_{14} shown in Figure □. The algorithm computes the following sequence of partitions:

$$\begin{aligned} P_0 &= \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}\} \\ P_1 &= \{\{a, d\}, \{b\}, \{c\}, \{e\}, \{f\}\} \\ P_2 &= \{\{a, d\}, \{b\}, \{c\}, \{e, f\}\} \\ P_3 &= \{\{a, d\}, \{b\}, \{c, e, f\}\} \\ P_4 &= \{\{a, c, d, e, f\}, \{b\}\} \\ P_5 &= \{\{a, b, c, d, e, f\}\} \end{aligned}$$

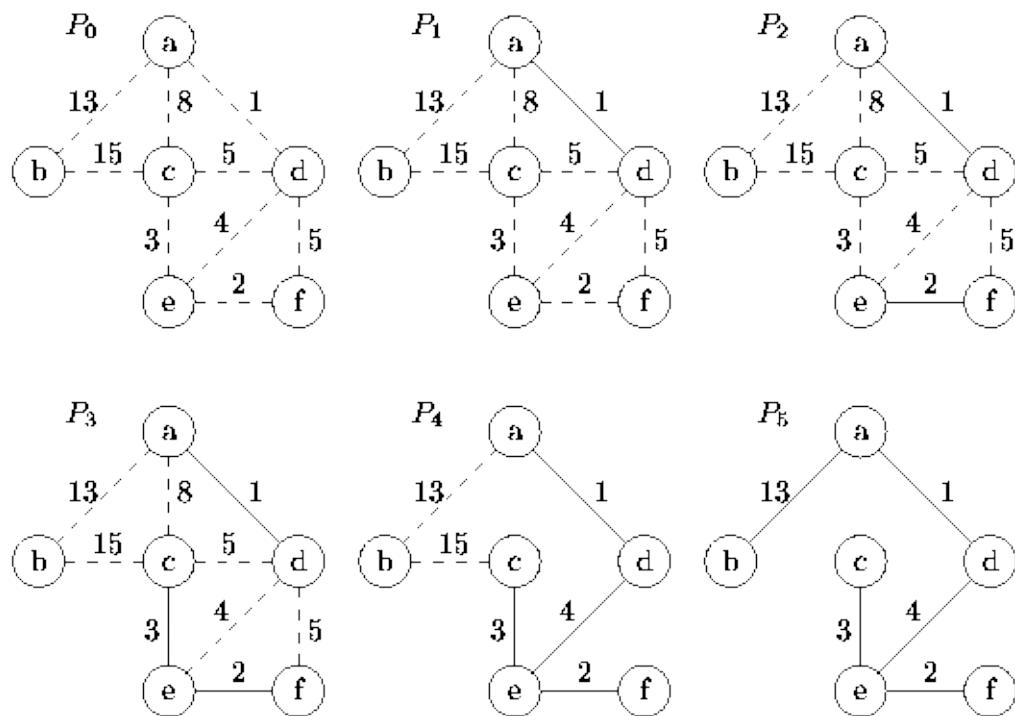


Figure: Operation of Kruskal's algorithm.

-
- [Implementation](#)
 - [Running Time Analysis](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno



Implementation

An implementation of Kruskal's algorithm is shown in Program □. The `KruskalsAlgorithm` method takes as its argument an edge-weighted, undirected graph. This implementation assumes that the edge weights are `ints`. The method computes the minimum-cost spanning tree and returns it in the form of an edge-weighted undirected graph.

```
 1 class Algorithms(object):
 2
 3     def KruskalsAlgorithm(g):
 4         n = g.numberOfVertices
 5         result = GraphAsLists(n)
 6         for v in xrange(n):
 7             result.addVertex(v)
 8         queue = BinaryHeap(g.numberOfEdges)
 9         for e in g.edges:
10             weight = e.weight
11             queue.enqueue(Association(weight, e))
12         partition = PartitionAsForest(n)
13         while not queue.isEmpty and partition.count > 1:
14             assoc = queue.dequeueMin()
15             e = assoc.value
16             n0 = e.v0.number
17             n1 = e.v1.number
18             s = partition.find(n0)
19             t = partition.find(n1)
20             if s != t:
21                 partition.join(s, t)
22                 result.addEdge(n0, n1)
23         return result
24     KruskalsAlgorithm = staticmethod(KruskalsAlgorithm)
```

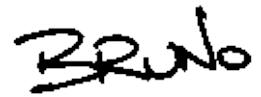
Program: Kruskal's algorithm.

The main data structures used by the method are a priority queue to hold the edges, a partition to detect cycles and a graph for the result. This implementation uses a `BinaryHeap` (Section □) for the priority queue (line 9), a `PartitionAsForest` (Section □) for the partition (line 12) and a `GraphAsLists`

for the spanning tree (line 5).

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Running Time Analysis

The `KruskalsAlgorithm` method begins by creating an graph to hold the result spanning tree (lines 5-7). Since a spanning tree is a sparse graph the `GraphAsLists` class is used to represent it. Initially the graph contains $|V|$ vertices but no edges. The running time for lines 5-7 is $O(|V|)$.

Next all of the edges in the input graph are inserted one-by-one into the priority queue (lines 8-11). Since there are $|\mathcal{E}|$ edges, the worst-case running time for a single insertion is $O(\log |\mathcal{E}|)$. Therefore, the worst-case running time to initialize the priority queue is

$$O(|V| + |\mathcal{E}| \log |\mathcal{E}|),$$

when adjacency lists are used, and

$$O(|V|^2 + |\mathcal{E}| \log |\mathcal{E}|),$$

when adjacency matrices are used to represent the input graph.

The main loop of the method comprises lines 13-22. This loop is done at most $|\mathcal{E}|$ times. In each iteration of the loop, one edge is removed from the priority queue (lines 14-15). In the worst-case this takes $O(\log |\mathcal{E}|)$ time.

Then, two partition *find* operations are done to determine the elements of the partition that contain the two end-points of the given edge (lines 16-19). Since the partition contains at most $|V|$ elements, the running time for the find operations is $O(\log |V|)$. If the two elements of the partition are distinct, then an edge is added to the spanning tree and a *join* operation is done to unite the two elements of the partition (lines 20-22). The join operation also requires $O(\log |V|)$ time in the worst-case. Therefore, the total running time for the main loop is $O(|\mathcal{E}| \log |\mathcal{E}| + |\mathcal{E}| \log |V|)$.

Thus, the worst-case running time for Kruskal's algorithm is

$$O(|\mathcal{V}| + |\mathcal{E}| \log |\mathcal{E}| + |\mathcal{E}| \log |\mathcal{V}|),$$

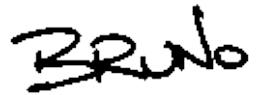
when adjacency lists are used, and

$$O(|\mathcal{V}|^2 + |\mathcal{E}| \log |\mathcal{E}| + |\mathcal{E}| \log |\mathcal{V}|),$$

when adjacency matrices are used to represent the input graph.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Application: Critical Path Analysis

In the introduction to this chapter it is stated that there are myriad applications of graphs. In this section we consider one such application--*critical path analysis*. Critical path analysis crops up in a number of different contexts, from the planning of construction projects to the analysis of combinational logic circuits.

For example, consider the scheduling of activities required to construct a building. Before the foundation can be poured, it is necessary to dig a hole in the ground. After the building has been framed, the electricians and the plumbers can rough-in the electrical and water services and this rough-in must be completed before the insulation is put up and the walls are closed in.

We can represent the set of activities and the scheduling constraints using a vertex-weighted, directed acyclic graph (DAG). Each vertex represents an activity and the weight on the vertex represents the time required to complete the activity. The directed edges represent the sequencing constraints. That is, an edge from vertex v to vertex w indicates that activity v must complete before w may begin. Clearly, such a graph must be *acyclic*.

A graph in which the vertices represent activities is called an *activity-node graph*. Figure □ shows an example of an activity-node graph. In such a graph it is understood that independent activities may proceed in parallel. For example, after activity A is completed, activities B and C may proceed in parallel. However, activity D cannot begin until *both* B and C are done.

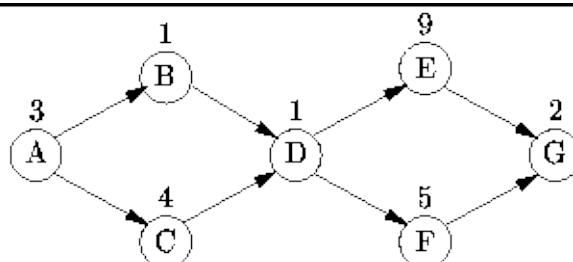


Figure: An activity-node graph.

Critical path analysis answers the following questions:

1. What is the minimum amount of time needed to complete all activities?
2. For a given activity v , is it possible to delay the completion of that activity without affecting the overall completion time? If yes, by how much can the completion of activity v be delayed?

The activity-node graph is a vertex-weighted graph. However, the algorithms presented in the preceding sections all require edge-weighted graphs. Therefore, we must convert the vertex-weighted graph into its edge-weighted *dual*. In the dual graph the edges represent the activities, and the vertices represent the commencement and termination of activities. For this reason, the dual graph is called an *event-node graph*.

Figure □ shows the event-node graph corresponding to the activity node graph given in Figure □. Where an activity depends on more than one predecessor it is necessary to insert *dummy* edges.

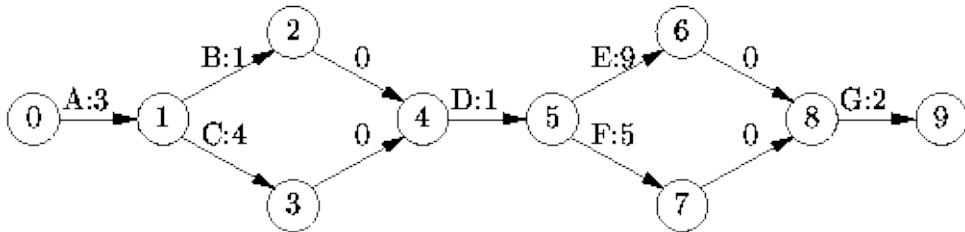


Figure: The event-node graph corresponding to Figure □.

For example, activity D cannot commence until both B and C are finished. In the event-node graph vertex 2 represents the termination of activity B and vertex 3 represents the termination of activity C . It is necessary to introduce vertex 4 to represent the event that *both* B and C have completed. Edges $2 \rightarrow 4$ and $3 \rightarrow 4$ represent this synchronization constraint. Since these edges do not represent activities, the edge weights are zero.

For each vertex v in the event node graph we define two times. The first E_v^* is the *earliest event time* for event v . It is the earliest time at which event v can occur assuming the first event begins at time zero. The earliest event time is given by

$$E_w = \begin{cases} 0 & w = v_i, \\ \min_{(v,w) \in \mathcal{I}(w)} E_v + C(v,w) & \text{otherwise,} \end{cases} \quad (16.3)$$

where v_i is the *initial* event, $\mathcal{I}(w)$ is the set of incident edges on vertex w and

$C(v,w)$ is the weight on vertex (v,w) .

Similarly, L_v is the *latest event time* for event v . It is the latest time at which event v can occur. The latest event time is given by

$$L_v = \begin{cases} E_{v_f}, & w = v_f, \\ \max_{(v,w) \in A(w)} E_w - C(v,w) & \text{otherwise,} \end{cases} \quad (16.4)$$

where v_f is the *final* event.

Given the earliest and latest event times for all events, we can compute time available for each activity. For example, consider an activity represented by edge (v,w) . The amount of time available for the activity is $L_w - E_v$ and the time required for that activity is $C(v,w)$. We define the *slack time* for an activity as the amount of time by which an activity can be delayed without affecting the overall completion time of the project. The slack time for the activity represented by edge (v,w) is given by

$$S(v,w) = L_w - E_v - C(v,w). \quad (16.5)$$

Activities with zero slack are *critical*. That is, critical activities must be completed on time--any delay affects the overall completion time. A *critical path* is a path in the event-node graph from the initial vertex to the final vertex comprised solely of critical activities.

Table □ gives the results from obtained from the critical path analysis of the activity-node graph shown in Figure □. The tabulated results indicate the critical path is

$$\{A, C, D, E, G\}.$$

Table: Critical path analysis results for the activity-node graph in Figure □.

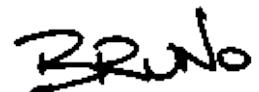
activity	$C(v,w)$	E_v	L_w	$S(v,w)$

<i>A</i>	3	0	3	0
<i>B</i>	1	3	7	3
<i>C</i>	4	3	7	0
<i>D</i>	1	7	8	0
<i>E</i>	9	8	17	0
<i>F</i>	5	8	17	4
<i>G</i>	2	17	18	0

- [Implementation](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Implementation

Given an activity-node graph, the objective of critical path analysis is to determine the slack time for each activity and thereby to identify the critical activities and the critical path. We shall assume that the activity node graph has already been transformed to an edge-node graph. The implementation of this transformation is left as a project for the reader (Project □). Therefore, the first step is to compute the earliest and latest event times.

According to Equation □, the earliest event time of vertex w is obtained from the earliest event times of all its predecessors. Therefore, must compute the earliest event times *in topological order*. To do this, we define the `EarliestTimeVisitor` shown in Program □.

```

1  class Algorithms(object):
2
3      class EarliestTimeVisitor(Visitor):
4
5          def __init__(self, earliestTime):
6              super(Algorithms.EarliestTimeVisitor, self).__init__()
7              self._earliestTime = earliestTime
8
9          def visit(self, w):
10             t = self._earliestTime[0]
11             for e in w.incidentEdges:
12                 t = max(t,
13                         self._earliestTime[e.v0.number] + e.weight)
14             self._earliestTime[w.number] = t

```

Program: Critical path analysis--computing earliest event times.

The `EarliestTimeVisitor` has one instance attribute, `_earliestTime`, which is an array used to record the E_v values. The `visit` method of the `EarliestTimeVisitor` class implements directly Equation □. It uses the `incidentEdges` property to obtain an iterator that enumerates all the predecessors of a given node and computes $\min_{v,w \in \mathcal{I}(w)} E_v + C(v,w)$.

In order to compute the latest event times, it is necessary to define also a `LatestTimeVisitor`. This visitor must visit the vertices of the event-node graph in *reverse topological order*. Its implementation follows directly from Equation □ and Program □.

Program □ defines the method called `criticalPathAnalysis` that does what its name implies. This method takes as its argument a `Digraph` that represents an event-node graph. This implementation assumes that the edge weights are `ints`.

```

1  class Algorithms(object):
2
3      def criticalPathAnalysis(g):
4          n = g.numberOfVertices
5
6          earliestTime = Array(n)
7          earliestTime[0] = 0
8          g.topologicalOrderTraversal(
9              Algorithms.EarliestTimeVisitor(earliestTime))
10
11         latestTime = Array(n)
12         latestTime[n - 1] = earliestTime[n - 1]
13         g.depthFirstTraversal(PostOrder(
14             Algorithms.LatestTimeVisitor(latestTime)), 0)
15
16         slackGraph = DigraphAsLists(n)
17         for v in xrange(n):
18             slackGraph.addVertex(v)
19         for e in g.edges:
20             slack = latestTime[e.v1.number] - \
21                 earliestTime[e.v0.number] - e.weight
22             slackGraph.addEdge(
23                 e.v0.number, e.v1.number, e.weight)
24     return Algorithms.DijkstrasAlgorithm(slackGraph, 0)
25
26     criticalPathAnalysis = staticmethod(criticalPathAnalysis)

```

Program: Critical path analysis--finding the critical paths.

The method first uses the `EarliestTimeVisitor` in a topological order traversal to compute the earliest event times which are recorded in the `earliestTime` array (lines 6-9). Next, the latest event times are computed and recorded in the `latestTime` array. Notice that this is done using a `LatestTimeVisitor` in a *postorder* depth-first traversal (lines 11-14). This is because a postorder depth-first traversal is equivalent to a topological order traversal in reverse!

Once the earliest and latest event times have been found, we can compute the slack time for each edge. In the implementation shown, an edge-weighted graph is constructed that is isomorphic with the original event-node graph, but in which the edge weights are the slack times as given by Equation □ (lines 16-23). By constructing such a graph we can make use of Dijkstra's algorithm find the shortest path from start to finish since the shortest path must be the critical path (line 24).

The `DijkstrasAlgorithm` method given in Section □ returns its result in the form of a shortest-path graph. The shortest-path graph for the activity-node graph of Figure □ is shown in Figure □. By following the path in this graph from vertex 9 back to vertex 0, we find that the critical path is $\{A, C, D, E, G\}$.

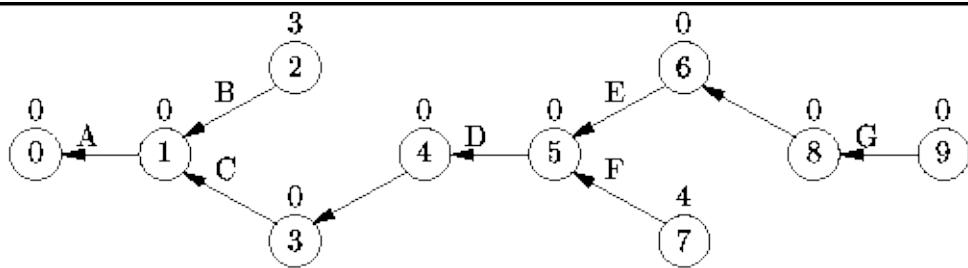


Figure: The critical path graph corresponding to Figure □.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Exercises

1. Consider the *undirected graph* G_A shown in Figure □. List the elements of \mathcal{V} and \mathcal{E} . Then, for each vertex $v \in \mathcal{V}$ do the following:
 1. Compute the in-degree of v .
 2. Compute the out-degree of v .
 3. List the elements of $\mathcal{A}(v)$.
 4. List the elements of $\mathcal{I}(v)$.

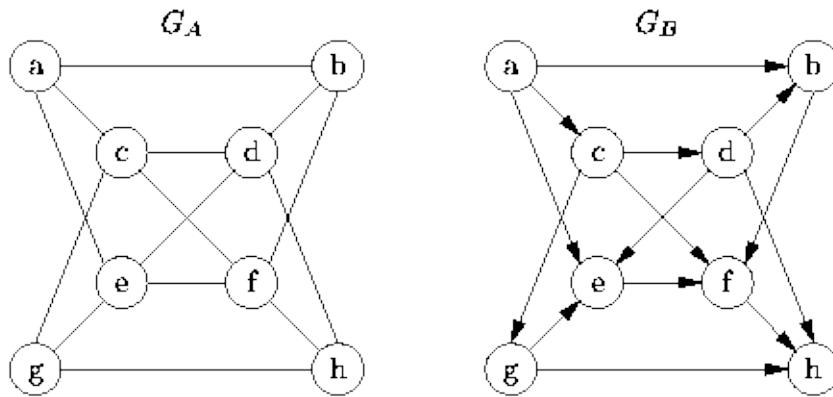
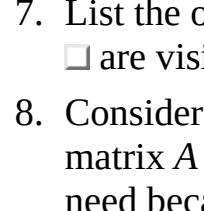


Figure: Sample graphs.

2. Consider the directed graph G_A shown in Figure □.
 1. Show how the graph is represented using an adjacency matrix.
 2. Show how the graph is represented using adjacency lists.
3. Repeat Exercises □ and □ for the *directed graph* G_B shown in Figure □.
4. Consider a *depth-first traversal* of the undirected graph G_A shown in Figure □ starting from vertex a .
 1. List the order in which the nodes are visited in a preorder traversal.
 2. List the order in which the nodes are visited in a postorder traversal.Repeat this exercise for a depth-first traversal starting from vertex d .
5. List the order in which the nodes of the undirected graph G_A shown in Figure □ are visited by a *breadth-first traversal* that starts from vertex a . Repeat this exercise for a breadth-first traversal starting from vertex d .
6. Repeat Exercises □ and □ for the *directed graph* G_B shown in Figure □.

7. List the order in which the nodes of the directed graph G_B shown in Figure  are visited by a *topological order traversal* that starts from vertex a .
8. Consider an undirected graph $G = (\mathcal{V}, \mathcal{E})$. If we use a $|\mathcal{V}| \times |\mathcal{V}|$ adjacency matrix A to represent the graph, we end up using twice as much space as we need because A contains redundant information. That is, A is symmetric about the diagonal and all the diagonal entries are zero. Show how a one-dimensional array of length $|\mathcal{V}|(|\mathcal{V}| - 1)/2$ can be used to represent G . Hint: consider just the part of A above the diagonal.
9. What is the relationship between the sum of the degrees of the vertices of a graph and the number of edges in the graph?
10. A graph with the maximum number of edges is called a *fully connected graph*. Draw fully connected, undirected graphs that contain 2, 3, 4, and 5 vertices.
11. Prove that an undirected graph with n vertices contains at most $n(n-1)/2$ edges.
12. Every tree is a directed, acyclic graph (DAG), but there exist DAGs that are not trees.
 1. How can we tell whether a given DAG is a tree?
 2. Devise an algorithm to test whether a given DAG is a tree.
13. Consider an acyclic, connected, undirected graph G that has n vertices. How many edges does G have?
14. In general, an undirected graph contains one or more *connected components*. A connected component of a graph G is a subgraph of G that is *connected* and contains the largest possible number of vertices. Each vertex of G is a member of exactly one connected component of G .
 1. Devise an algorithm to count the number of connected components in a graph.
 2. Devise an algorithm that labels the vertices of a graph in such a way that all the vertices in a given connected component get the same label and vertices in different connected components get different labels.
15. A *source* in an directed graph is a vertex with zero in-degree. Prove that every DAG has at least one source.
16. What kind of DAG has a unique topological sort?
17. Under what conditions does a *postorder* depth-first traversal of a DAG visit the vertices in *reverse topological order*.
18. Consider a pair of vertices, v and w , in a directed graph. Vertex w is said to be *reachable* from vertex v if there exists a path in G from v to w . Devise an algorithm that takes as input a graph, $G = (\mathcal{V}, \mathcal{E})$, and a pair of vertices,

$v, w \in \mathcal{V}$, and determines whether w is reachable from v .

19. An *Eulerian walk* is a path in an undirected graph that starts and ends at the same vertex *and traverses every edge* in the graph. Prove that in order for such a path to exist, all the nodes must have even degree.
20. Consider the binary relation \prec defined for the elements of the set $\{a, b, c, d\}$ as follows:

$$\{a \prec b, a \prec c, b \prec c, b \prec d, c \prec d, d \prec a\}.$$

How can we determine whether \prec is a *total order*?

21. Show how the *single-source shortest path* problem can be solved on a DAG using a topological-order traversal. What is the running time of your algorithm?
22. Consider the directed graph G_C shown in Figure □. Trace the execution of *Dijkstra's algorithm* as it solves the single-source shortest path problem starting from vertex a . Give your answer in a form similar to Table □.

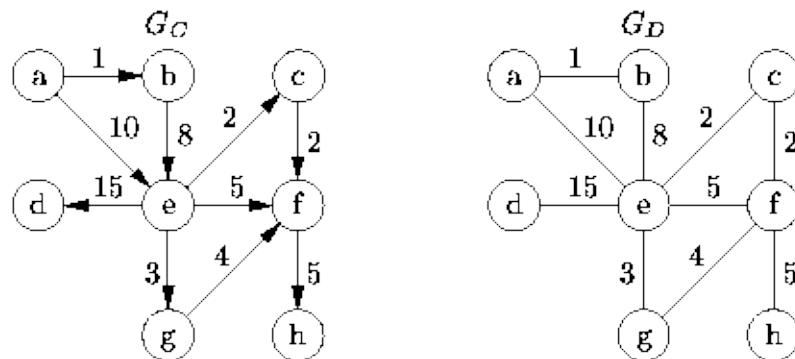


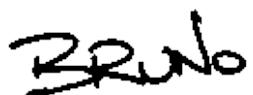
Figure: Sample weighted graphs.

23. Dijkstra's algorithm works as long as there are no negative edge weights. Given a graph that contains negative edge weights, we might be tempted to eliminate the negative weights by adding a constant weight to all of the edge weights to make them all positive. Explain why this does not work.
24. Dijkstra's algorithm can be modified to deal with negative edge weights (but not negative cost cycles) by eliminating the *known* flag k_v and by inserting a vertex back into the queue every time its *tentative distance* d_v decreases. Explain why the modified algorithm works correctly. What is the running time of the modified algorithm?

25. Consider the directed graph G_C shown in Figure □. Trace the execution of *Floyd's algorithm* as it solves the *all-pairs shortest path problem*.
 26. Prove that if the edge weights on an undirected graph are *distinct*, there is only one minimum-cost spanning tree.
 27. Consider the undirected graph G_D shown in Figure □. Trace the execution of *Prim's algorithm* as it finds the *minimum-cost spanning tree* starting from vertex a .
 28. Repeat Exercise □ using *Kruskal's algorithm*.
 29. Do Exercise □.
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Projects

1. Devise a graph description language. Implement a method that reads the description of a graph and constructs a graph object instance. Your method should be completely generic--it should not depend on the graph implementation used.
2. Extend Project □ by writing a method that prints the description of a given graph object instance.
3. Complete the implementation of the `GraphAsMatrix` class introduced in Program □ by providing suitable definitions for the following operations: `__init__`, `purge`, `addVertex`, `getVertex`, `addEdge`, `getEdge`, `isEdge`, `vertices`, `edges`, `getIncidentEdges`, and `getEmanatingEdges`. Write a test program and test your implementation.
4. Repeat Project □ for the `GraphAsLists` class.
5. The `DigraphAsMatrix` class can be implemented by extending the `GraphAsMatrix` class introduced in Program □ and the abstract `Digraph` class introduced in Program □:

```
class DigraphAsMatrix(GraphAsMatrix, Digraph):  
    # ...
```

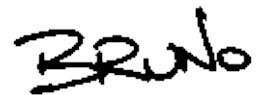
Implement the `DigraphAsMatrix` class by providing suitable definitions for the following methods: `__init__`, `purge`, `addEdge`, `getEdge`, `isEdge`, `vertices` and `edges`. You must also have a complete implementation of the base class `GraphAsMatrix` (see Project □). Write a test program and test your implementation.

6. Repeat Project □ for the `DigraphAsLists` class.
7. Add a method to the `Digraph` interface that returns the undirected graph which underlies the given digraph. Write an implementation of this method for the abstract `Graph` class introduced in Program □.
8. Devise an approach using an enumerator and a stack to perform a topological-order traversal by doing a postorder depth-first traversal in reverse.
9. The single-source shortest path problem on a DAG can be solved by visiting the vertices in topological order. Write an visitor for use with the `topologicalOrderTraversal` method that solves the single-source shortest path problem on a DAG.

10. Devise and implement a method that transforms a vertex-weighted *activity-node graph* into an edge-weighted *event-node graph*.
 11. Complete the implementation of the critical path analysis methods. In particular, you must implement the `LatestTimeVisitor` along the lines of the `EarliestTimeVisitor` defined in Program □.
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



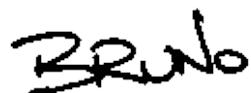
Python and Object-Oriented Programming

This appendix is a brief overview of programming in Python. It identifies and describes the features of Python that are used throughout this text. This appendix is *not* a Python tutorial--if you are not familiar with Python, you should read one of the many Python programming books.

- [Objects and Types](#)
 - [Names](#)
 - [Parameter Passing](#)
 - [Classes](#)
 - [Nested Classes](#)
 - [Inheritance and Polymorphism](#)
 - [Exceptions](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Objects and Types

An *object* is a programming language abstraction that represents a storage location. A Python object has the following *attributes*:

type

The *type* of an object determines the set of values that the object can have and the set of operations that can be performed on that object.

value

The *value* of an object is the content of the memory location(s) occupied by that variable. How the contents of the memory locations are interpreted is determined by the *type* of the variable.

lifetime

The *lifetime* of an object is the interval of time in the execution of a Python program during which the object is said to exist.

Python defines an extensive type hierarchy. This hierarchy includes the numeric types (such as `int`, `float` and `complex`), sequences (such as `tuple` and `list`), functions (type `function`), classes and methods (types `classobj` and `instancemethod`), and instances of classes (type `instance`).

Names

In order to make use of an object in a Python program, that object must have a *name*. The name of an object is the label used to identify that object in the text of a program. A given Python object may have zero, one or more names.

Consider the Python statement:

```
i = 57
```

This statement creates an object named *i* and *binds* various attributes with that object. The type of the object is *int* and its value is 57.

Some attributes of an object, such its type, are bound when the object is created and cannot be changed. This is called *static binding*. The bindings for other attributes of an object, such as its value, may be changed at run time. This is called *dynamic binding*.

Consider again the Python statement:

```
i = int(57)
```

If we follow this statement with an assignment statement such as

```
j = i
```

then the names *i* and *j* both refer to the same object!

A comparison of the the form

```
if i == j:  
    print "equal values"
```

tests whether the value of the object named *i* is the same as the value of the object named *j*. (Clearly this is true since *i* and *j* name the same object). However, it is possible for two distinct objects to have the same value. In order to test whether two names refer to the same object, it is necessary to Python *is* operator like this:

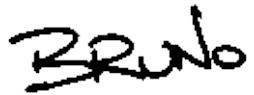
```
if i is j:
```

```
print "same object"
```

- [The Object Named None](#)
 - [Scopes and Namespaces](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



The Object Named None

In Python, a name always refers to an object. However, sometimes it is convenient to use a name to refer to ``nothing''. Python provides a special type of object for this purpose called `NoneType`. There is only ever one object of type `NoneType` and the name of that object is `None`.

We can explicitly assign a name to `None` like this:

```
f = None
```

Also, we can test explicitly whether a name refers to `None` like this:

```
if f is None:  
    # ...
```

Scopes and Namespaces

The *scope* of a name is the range of statements in the text of a program in which that name can be used to refer to an object. Python defines three scopes--*local* , *global* , and *built-in* . In Python scopes are called *namespaces* . When a name is used to refer to an object in a Python program the namespaces are searched in the following order to find the binding for that name: local namespace first, then global namespace, then built-in namespace. This is the so-called *LGB rule* .

When a Python function is called, a new *local* namespace is created. By default, name bindings created inside a function are created in the local namespace of that function. Name bindings created at the top-level of a module (or file) are created in the global namespace. The built-in namespace contains the bindings for the pre-defined names of Python.

Parameter Passing

Parameter passing methods are the ways in which parameters are transferred between methods when one method calls another. Python provides only one parameter passing method--*pass-by-reference* .

Consider the pair of Python methods defined in Program □. On line 4, the method one calls the method two. In general, every method call includes a (possibly empty) list of arguments. The arguments specified in a method call are called *actual parameters* . In this case, there is only one actual parameter--x.

```
1 def one():
2     x = 1
3     print x
4     two(x)
5     print x
6
7 def two(y):
8     print y
9     y = 2
10    print y
```

Program: Example of parameter passing.

On line 7 the method two is defined as accepting a single argument y. The arguments which appear in a method definition are called *formal parameters* .

The semantics of pass-by-reference work like this: The effect of the formal parameter definition is to create a name in the local namespace of the function and then to bind that name to the object named by the actual parameter. For example, in the method two the formal parameter is called y. When the method is called, the name y is assigned to the object named x.

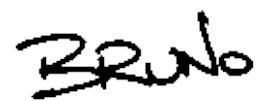
Since the formal parameters give rise to names in the local namespace, when a formal parameter is assigned a new object is bound to that name in the local namespace. The object named by the actual parameter is no longer accessible.

The output produced when the method one defined in Program □ is called is:

1
1
2
2
1

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Classes

A Python *class* defines a data structure that contains instance attributes, instance methods, and nested classes. In Python the class of an object and the type of an object are synonymous. Every Python object has a class (type) that is directly or indirectly derived from the Python built-in object class. The class (type) of an object determines what it is and how it can be manipulated. A class encapsulates data, operations, and semantics. This encapsulation is like a *contract* between the implementer of the class and the user of that class.

The `class` statement is what makes Python an *object-oriented* language. A Python class definition groups a set of values with a set of operations. Classes facilitate modularity and information hiding. The user of a class manipulates object instances of that class only through the methods provided by that class.

It is often the case that different classes possess common features. Different classes may share common values and they may perform the same operations. In Python such relationships are expressed using *derivation* and *inheritance*.

-
- [Instances, Instance Attributes and Methods](#)
 - [Example-Complex Numbers](#)
 - [`__init__` Method](#)
 - [Properties, Accessors and Mutators](#)
 - [Operator Overloading](#)
 - [Static Methods](#)
-

Instances, Instance Attributes and Methods

Python objects are *instances* of classes. Each instance (object) in a Python program has its own namespace.

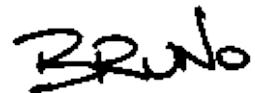
A class definition creates a class object (type `classobj`). The names in the namespace of a class object are called *class attributes*. Function definitions inside of a class statement create *methods*. Methods are functions that process instances.

When an object is created, its namespace inherits all the names in the namespace of the class of that object. The names in the namespace of an instance are called the *instance attributes* of that instance.

A method is a function created in a class definition. The first argument of a method always refers to the instance being processed. By convention, the first argument of a method is always named `self`. Therefore, the attributes of `self` are instance attributes.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Example-Complex Numbers

Suppose we wish to define a class to represent *complex numbers*. One way to do this is to define a class called `Complex` that uses two instance attributes, `_real` and `_imag`, to represent the real and imaginary parts of a complex number (respectively)◊. The `Complex` class definition shown in Program □ illustrates how this can be done.

```
1  class Complex(object):
2
3      def __init__(self, real, imag):
4          self._real = real
5          self._imag = imag
6
7      # ...
```

Program: Complex class `__init__` method.

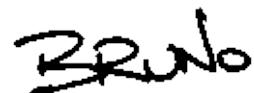
Every instance of the `Complex` class contains its own instance attributes. Consider the following statements:

```
c = Complex(1, 0)
d = Complex(2, 0)
```

Both `c` and `d` refer to distinct instances of the `Complex` class. Therefore, each of them has its own `_real` and `_imag` instance attribute. The instance attributes of an object are accessed using the *dot* operator. For example, `c._real` refers to the `_real` instance attribute of `c` and `d._imag` refers to the `_imag` instance attribute of `d`.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



__init__ Method

The `__init__ method` of a class is special. The purpose of a `__init__` method is to *initialize* an object. The `__init__` method is invoked whenever a new instance of a class is created. The `__init__` method of the `Complex` class is defined in Program □.

Consider the following statement:

```
i = Complex(0, 1)
```

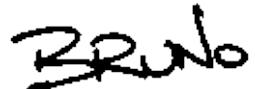
The effect of this statement is equivalent to the following sequence of statements:

```
c = object.__new__(Complex)
Complex.__init__(c, 0, 1)
```

The `__new__` method of the built-in `object` class is called to create a new object instance. Then `__init__` method of the `Complex` class is called to initialize that instance.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Properties, Accessors and Mutators

Program □ continues the definition of the `Complex` class. It defines four methods, `getReal`, `setReal`, `getImag` and `setImage`, as well as two properties, `real` and `imag`.

```
 1  class Complex(object):
 2
 3      def getReal(self):
 4          return self._real
 5      def setReal(self, value):
 6          self._real = value
 7
 8      real = property(
 9          fget = getReal,
10          fset = setReal)
11
12      def getImag(self):
13          return self._imag
14      def setImage(self, value):
15          self._imag = value
16
17      imag = property(
18          fget = getImag,
19          fset = setImage)
20
21      # ...
```

Program: Complex class `real` and `imag` properties.

The `getReal` and `setImage` methods provide the means to access the `_real` and `_imag` instance attributes, respectively. A method that accesses an instance but does not modify that instance is called an *accessor*. Therefore, `getReal` and `setReal` are accessors.

The `setReal` and `setImage` methods provide the means to modify the `_real` and `_imag` instance attributes, respectively. A method that modifies an instance is called a *mutator*. Therefore, `setReal` and `setImage` are mutators.

The *dot* operator is used to specify the object on which a method is to be invoked. For example, the sequence of statements

```
c.setReal(2)
print c.getReal()
```

is equivalent to

```
Complex.setReal(c, 2)
print Complex.getReal(c)
```

Program □ also defines two properties called `real` and `imag`.

A Python property is an class attribute that provides an accessor method and/or a mutator method. The `fget` argument of a property specifies an accessor method called the ```getter``` and the `fset` argument of a property specifies a mutator method called the ```setter```.

For example, in Program □, the `real` property getter is the `getReal` accessor method and the `real` property setter is the `setReal` mutator method. Similarly, the `imag` property getter is `getImag` and the `imag` property setter is `setReal`.

Properties let you use instance attribute notation rather than method call notation to invoke an accessor or a mutator method. Again, the `dot` operator is used to specify the object on which the operation is performed. For example, the sequence of statements

```
c.imag = 2
print c.imag
```

is equivalent to

```
Complex.setImag(c, 2)
print Complex.getImag(c)
```

Program □ defines four methods and two more properties of the `Complex` class. The `r` property uses the `getR` accessor as its getter and the `setR` mutator as its setter. Similarly, the `theta` property uses the `getTheta` accessor as its getter and the `setTheta` mutator as its setter.

```
 1 class Complex(object):
 2
 3     def getR(self):
 4         return math.sqrt(self._real * self._real
 5                         + self._imag * self._imag)
 6
 7     def setR(self, value):
 8         theta = self.theta
 9         self._real = value * math.cos(theta)
10         self._imag = value * math.sin(theta)
11
12     r = property(
13         fget = getR,
14         fset = setR)
15
16     def getTheta(self):
17         return math.atan2(self._imag, self._real)
18
19     def setTheta(self, value):
20         r = self.r
21         self._real = r * math.cos(value)
22         self._imag = r * math.sin(value)
23
24     theta = property(
25         fget = getTheta,
26         fset = setTheta)
27
28     # ...
```

Program: Complex class r and theta properties.

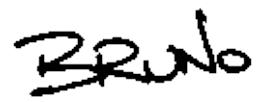
By defining suitable properties, it is possible to hide the implementation of the class from the user of that class. Consider the following statements:

```
print c._real
print c.real
```

The first statement depends on the implementation of the `Complex` class. If we change the implementation of the class from the one given (which uses rectangular coordinates) to one that uses polar coordinates, then the first statement above must also be changed. On the other hand, the second statement does not need to be modified, provided we reimplement the `real` property when we switch to polar coordinates.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Operator Overloading

Program □ illustrates operator overloading in Python. *Operator overloading* allows the programmer to use the built-in operators for user-defined types.

```
 1 class Complex(object):
 2
 3     def __add__(self, c):
 4         return Complex(self.real + c.real, self.imag + c.imag)
 5
 6     def __sub__(self, c):
 7         return Complex(self.real - c.real, self.imag - c.imag)
 8
 9     def __mul__(self, c):
10         return Complex(self.real * c.real - self.imag * c.imag,
11                         self.real * c.imag + self.imag * c.real)
12
13     # ...
```

Program: Complex class `__add__`, `__sub__` and `__mul__` methods.

To overload the built-in `+`, `-` and `*` operators so that they may be used with `Complex` operands, we define the methods called `__add__`, `__sub__` and `__mul__` (respectively). Given `Complex` variables `c`, `d` and `e`, the expression

`c + d * e`

is equivalent to

`Complex.__add__(c, Complex.__mul__(d, e))`

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Static Methods

A *static method* is a class attribute that does not require its first argument to be an instance of the class. (Normal methods require the first argument, `self`, to be a class instance).

```
 1  class Complex(object):
 2
 3      def main(*argv):
 4          "Complex test program."
 5          c = Complex(0, 0)
 6          print c
 7          c.real = 1
 8          c.imag = 2
 9          print c
10          c.theta = math.pi/2
11          c.r = 50
12          print c
13          c = Complex(1, 2)
14          d = Complex(3, 4)
15          print c, d, c+d, c-d, c*d
16          return 0
17      main = staticmethod(main)
18
19      # ...
```

Program: Complex class `__main__` method.

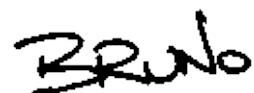
Program □ defines a `main` method for the `Complex` class. This method is a simple test program for the `Complex` class. For example, this static method can be invoked like this:

```
import sys

if __name__ == "__main__":
    sys.exit(Complex.main(*sys.argv))
```

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Nested Classes

In Python it is possible to define one class *inside* another. A class defined inside another one is called a *nested class*.

Consider the following Python code fragment:

```
class A(object):
    def __init__(self):
        self.y = 0

    class B(object):
        def __init__(self):
            self.x = 0

        def f(self):
            pass
```

This fragment defines the class `A` which contains the nested class `B`.

A nested class behaves like any ``outer'' (un-nested) class. It may contain methods and instance attributes, and it may be instantiated like this:

```
obj = A.B()
```

This statement creates a new instance of the nested class `B`. Given such an instance, we can invoke the `f` method in the usual way:

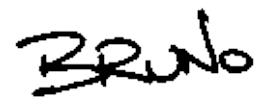
```
obj.f()
```

Note, it is not necessarily the case that an instance of the outer class `A` exists even when we have created an instance of the inner class. Similarly, instantiating the outer class `A` does not create any instances of the inner class `B`.

The methods of a nested class may access the instance attributes of the nested class instance but not of any outer class instance. Thus, method `f` can access the instance attribute `x`, but it cannot access the instance attribute `y`.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

Inheritance and Polymorphism

- [Derivation and Inheritance](#)
 - [Polymorphism](#)
 - [Multiple Inheritance](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Derivation and Inheritance

This section reviews the concept of a derived class. Derived classes are an extremely useful feature of Python because they allow the programmer to define new classes by extending existing classes. By using derived classes, the programmer can exploit the commonalities that exist among the classes in a program. Different classes can share values and operations.

Derivation is the definition of a new class by extending an existing class. The new class is called the *derived class* and the existing class from which it is derived is called the *base class*.

In Python there must be at least one base class, but there may be more than one base class (*multiple inheritance*).

Python supports so-called *classic classes* and *new-style classes*. A new-style class is one that is ultimately derived from the built-in object class. A classic class is one that does not have a base class or one that is derived only from other classic classes. Classic classes are being phased out of the Python language and are not used in this book.

Consider the Person class defined in Program □ and the Parent class defined in Program □. Because parents are people too, the Parent class is derived from the Person class. Derivation in Python is indicated by including the name(s) of the base class(es) in parentheses in the declaration of the derived class.

```
1  class Person(object):
2
3      FEMALE = 0
4      MALE = 1
5
6      def __init__(self, name, sex):
7          super(Person, self).__init__()
8          self._name = name
9          self._sex = sex
10
11     def __str__(self):
12         return str(self._name)
```

Program: Person class.

```
 1 class Parent(Person):
 2
 3     def __init__(self, name, sex, children):
 4         super(Parent, self).__init__(name, sex)
 5         self._children = children
 6
 7     def getChild(self, i):
 8         return self._children[i]
 9
10     def __str__(self):
11         # ...
```

Program: Parent class.

A derived class *inherits* all the attributes of its base class. That is, the derived class contains all the class attributes contained in the base class and the derived class supports all the same operations provided by the base class. For example, consider the following statements:

```
p = Person()
q = Parent()
```

Since p is a Person, it has the instance attributes `_name` and `_sex` and method `__str__`. Furthermore, since Parent is derived from Person, then the object q also has the instance attributes `_name` and `_sex` and method `__str__`.

A derived class can *extend* the base class in several ways: New instance attributes can be used, new methods can be defined, and existing methods can be *overridden*. For example, the Parent class adds the instance attribute `_children` and the method `getChild`.

If a method is defined in a derived class that has the same name as a method in a base class, the method in the derived class *overrides* the one in the base class. For example, the `__str__` method in the Parent class overrides the `__str__` method in the Person class. Therefore, `str(p)` invokes `Person.__str__`, whereas `str(q)` invokes `Parent.__str__`.

An instance of a derived class can be used anywhere in a program where an instance of the base class may be used. For example, this means that a Parent may be passed as an actual parameter to a method that expects to receive a Person.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno".

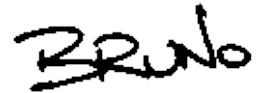
Polymorphism

Polymorphism literally means ``having many forms.'' Polymorphism arises when a set of distinct classes share a common interface. Because the derived classes are distinct, their implementations may differ. However, because the derived classes share a common interface, instances of those classes are used in exactly the same way.

- [Example-Graphical Objects](#)
 - [Method Resolution](#)
 - [Abstract Methods](#)
 - [Algorithmic Abstraction](#)
-

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.





Example-Graphical Objects

Consider a program for creating simple drawings. Suppose the program provides a set of primitive graphical objects, such as circles, rectangles, and squares. The user of the program selects the desired objects, and then invokes commands to draw, to erase, or to move them about. Ideally, all graphical objects support the same set of operations. Nevertheless, the way that the operations are implemented varies from one object to the next.

We implement this as follows: We define a class that represents the common operations provided by all graphical objects.

Program □ defines the `GraphicalObject` class. This class provides four method, `__init__`, `draw`, `erase` and `moveTo`.

```
 1  class GraphicalObject(Object):
 2
 3      def __init__(self, center):
 4          super(GraphicalObject, self).__init__()
 5          self._center = center
 6
 7      def draw(self):
 8          pass
 9      draw = abstractmethod(draw)
10
11      def erase(self):
12          self.setPenColor(self.BACKGROUND_COLOR)
13          self.draw()
14          self.setPenColor(self.FOREGROUND_COLOR)
15
16      def moveTo(self, p):
17          self.erase()
18          self._center = p
19          self.draw()
```

Program: `GraphicalObject` class `__init__`, `draw`, `erase` and `moveTo` methods.

The `draw` method is invoked in order to draw a graphical object. The `erase` method is invoked in order to erase a graphical object. The `moveTo` method is

used to move an object to a specified position in the drawing. The argument of the `moveTo` method is a `Point`. Program □ defines the `Point` class which represents a position in a drawing.

```
1  class Point(object):
2
3      def __init__(self, x, y):
4          self._x = x
5          self._y = y
6          # ...
```

Program: `Point` class `__init__` method.

The `GraphicalObject` class has a single instance attribute, `_center`, which is a `Point` that represents the position in a drawing of the center-point of the graphical object. In addition to `self`, the `__init__` method for the `GraphicalObject` class takes as its argument a `Point` and initializes the `_center` instance attribute accordingly.

Program □ shows a possible implementation for the `erase` method: In this case we assume that the image is drawn using an imaginary pen. Assuming that we know how to draw a graphical object, we can erase the object by changing the color of the pen so that it matches the background color and then redrawing the object.

Once we can erase an object as well as draw it, then moving it is easy. Just erase the object, change its `_center` point, and then draw it again. This is how the `moveTo` method shown in Program □ is implemented.

We have seen that the `GraphicalObject` class provides implementations for the `Erase` and `MoveTo` methods. However, the `GraphicalObject` class does not provide an implementation for the `Draw` method. Instead, the method is declared to be abstract. We do this because until we know what kind of object it is, we cannot possibly know how to draw it!

Consider the `Circle` class defined in Program □. The `Circle` class *extends* the `GraphicalObject` class. Therefore, it inherits the instance attribute `_center` and the methods `erase` and `moveTo`. The `Circle` class adds an additional instance attribute, `_radius`, and it overrides the `draw` method. The body of the `draw` method is not shown in Program □. However, we shall assume that it draws a circle with the given radius and center point.

```
1 class Circle(GraphicalObject):
2
3     def __init__(self, center, radius):
4         super(Circle, self).__init__(center)
5         self._radius = radius
6
7     def draw(self):
8         # ...
```

Program: Circle class.

Notice the way the `__init__` method of the `Circle` class is implemented. This method first calls the `__init__` method of the *superclass* of class `Circle`, that is `GraphicalObject`. The `GraphicalObject __init__` method initializes the `_center` attribute. Then the `Circle __init__` method initializes the `_radius` attribute.

Using the `Circle` class defined in Program □ we can write code like this:

```
c = Circle(Point(0, 0), 5)
c.draw()
c.moveTo(Point(10, 10))
c.erase()
```

This code sequence declares a circle object with its center initially at position (0,0) and radius 5. The circle is then drawn, moved to (10,10), and then erased.

Program □ defines the `Rectangle` class and Program □ defines the `Square` class. The `Rectangle` class also extends the `GraphicalObject` class. Therefore, it inherits the instance attribute `_center` and the methods `erase` and `moveTo`. The `Rectangle` class adds two additional instance attributes, `_height` and `_width`, and it overrides the `draw` method. The body of the `draw` method is not shown in Program □. However, we shall assume that it draws a rectangle with the given dimensions and center point.

```
1 class Rectangle(GraphicalObject):
2
3     def __init__(self, center, height, width):
4         super(Rectangle, self).__init__(center)
5         self._height = height
6         self._width = width
7
8     def draw(self):
9         # ...
```

Program: Rectangle class.

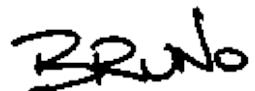
The Square class extends the Rectangle class. No new instance attributes or methods are declared--those inherited from GraphicalObject or from Rectangle are sufficient. The `__init__` method simply arranges to make sure that the `_height` and `_width` of a square are equal!

```
1 class Square(Rectangle):
2
3     def __init__(self, center, width):
4         super(Square, self).__init__(center, width, width)
```

Program: Square class.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Method Resolution

Consider the following sequence of instructions:

```
g1 = Circle(Point (0,0), 5)
g2 = Square(Point (0,0), 5)
g1.draw()
g2.draw()
```

The statement `g1.draw()` calls `Circle.draw` whereas the statement `g2.draw()` calls `Rectangle.draw`.

It is as if every object of a class ``knows'' the actual method to be invoked when a method is called on that object. E.g, a `Circle` ``knows'' to call `Circle.draw`, `GraphicalObject.erase` and `GraphicalObject.moveTo`, whereas a `Square` ``knows'' to call `Rectangle.draw`, `GraphicalObject.erase` and `GraphicalObject.moveTo`.



Abstract Methods

The `draw` method defined in Program □ should never be called. This is because it is expected that every class derived from the `GraphicalObject` class will override the `draw` method. Therefore, we define the `draw` method given in the `GraphicalObject` class to be an `abstractmethod`.

Unlike `staticmethod`, the `abstractmethod` class is not a built-in Python class. Program □ defines the `abstractmethod` class.

In order to understand what the `abstractmethod` class does, it is necessary to understand how the Python virtual machine invokes instance methods. Consider the following method call:

```
g.draw()
```

The Python interpreter performs a sequence of operations that is equivalent to the following:

```
func = GraphicalObject.draw.__get__(g, GraphicalObject)
func.__call__()
```

The purpose of the `__get__` method is to return a method object that is bound to an instance. The bound method object is then called like a normal function by invoking the `__call__` method.

The `__get__` method of the `abstractmethod` class returns an instance of the nested class called `method`. The `__call__` method of the nested `method` class raises a `TypeError` exception when called.

Algorithmic Abstraction

An *abstract class* is a class that contains one or more abstract methods. Abstract classes can be used in many interesting ways. One of the most useful paradigms is the use of an abstract class for *algorithmic abstraction*. The `erase` and `moveTo` methods defined in Program □ are examples of this.

The `erase` and `moveTo` methods are implemented in the abstract class `GraphicalObject`. The algorithms implemented are designed to work in any concrete class derived from `GraphicalObject`, be it `circle`, `Rectangle` or `Square`. In effect, we have written algorithms that work regardless of the actual class of the object. Therefore, such algorithms are called *abstract algorithms*.

Abstract algorithms typically invoke abstract methods. For example, both `moveTo` and `erase` ultimately invoke `draw` to do most of the actual work. In this case, the derived classes are expected to inherit the abstract algorithms `moveTo` and `erase` and to override the abstract method `draw`. Thus, the derived class customizes the behavior of the abstract algorithm by overriding the appropriate methods. The Python method resolution mechanism ensures that the ``correct'' method is always called.

Multiple Inheritance

In Python a class can be derived from one or more base classes. For example, consider the following class definitions:

```
class A(object):
    def f(): pass
class B(object): pass
    def f(): pass
class C(A, B): pass
```

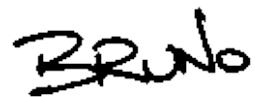
The class `C` extends both classes `A` and `B`. Therefore, the `C` class inherits class attributes from both base classes.

An interesting question arises when more than one base class defines an attribute with the same name. For example, `A` and `B` both define a method named `f`. Given an instance `c` of class `C`, what method does the expression `c.f()` call?

The method called is determined by a set of rules called the Python *method resolution order*. In order to handle the general case, the rules are quite complex and are beyond the scope of this book. (For a thorough explanation, see [44]). However, in simple cases such as this, the rules are straightforward: To find a name, first search the namespace of class `C`, and then search base classes in the order given. That is, search `A` first and then search `B`. Therefore, in this case the function invoked by the expression `c.f()` is the function `A.f`.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Exceptions

Sometimes unexpected situations arise during the execution of a program. Careful programmers write code that detects errors and deals with them appropriately. However, a simple algorithm can become unintelligible when error-checking is added because the error-checking code can obscure the normal operation of the algorithm.

Exceptions provide a clean way to detect and handle unexpected situations. When a program detects an error, it *raises* an exception. When an exception is raised, control is transferred to the appropriate *exception handler*. By defining a method that *catches* the exception, the programmer can write the code to handle the error.

In Python, an exception is an object. All exceptions in Python are ultimately derived from the built-in base class called `Exception`. For example, consider the class `A` defined in Program □. Since the `A` class extends the `Exception` class, `A` is an exception that can be *raised*.

```
1  class A(Exception):
2      pass
3
4  def f():
5      raise A
6
7  def g():
8      try:
9          f()
10     except A:
11         # ...
```

Program: Using exceptions in Python.

A method raises an exception by using the `raise` statement: The `raise` statement is similar to a `return` statement. A `return` statement represents the normal termination of a method and the object returned matches the return value of the method. A `raise` statement represents the abnormal termination of a method and

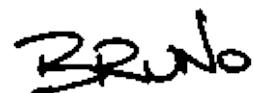
the object raised represents the type of error encountered. The `f` method in Program □ raisess an `A` exception.

Exception handlers are defined using a `try` block: The body of the `try` block is executed either until an exception is raised or until it terminates normally. One or more exception handlers follow a `try` block. Each exception handler consists of an `except` clause which specifies the exceptions to be caught, and a block of code, which is executed when the exception occurs. When the body of the `try` block raises an exception for which an exception is defined, control is transferred to the body of the exception handler.

In this example, the exception raised by the `f` method is caught by the `g` method. In general when an exception is raised, the chain of methods called is searched in reverse (from caller to callee) to find the closest matching `except` clause. When a program raises an exception that is not caught, the program terminates.

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Class Hierarchy Diagrams

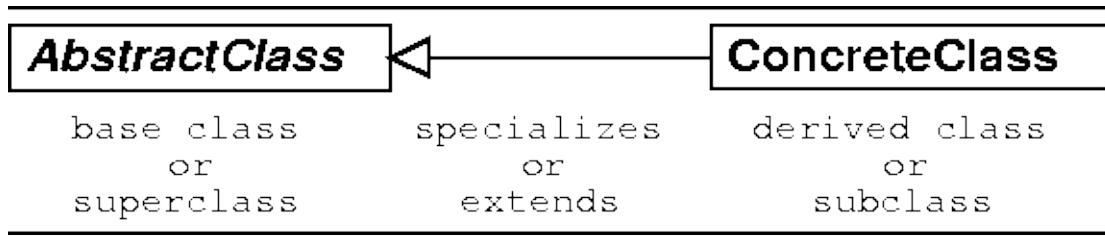


Figure: Key for the class hierarchy diagrams.

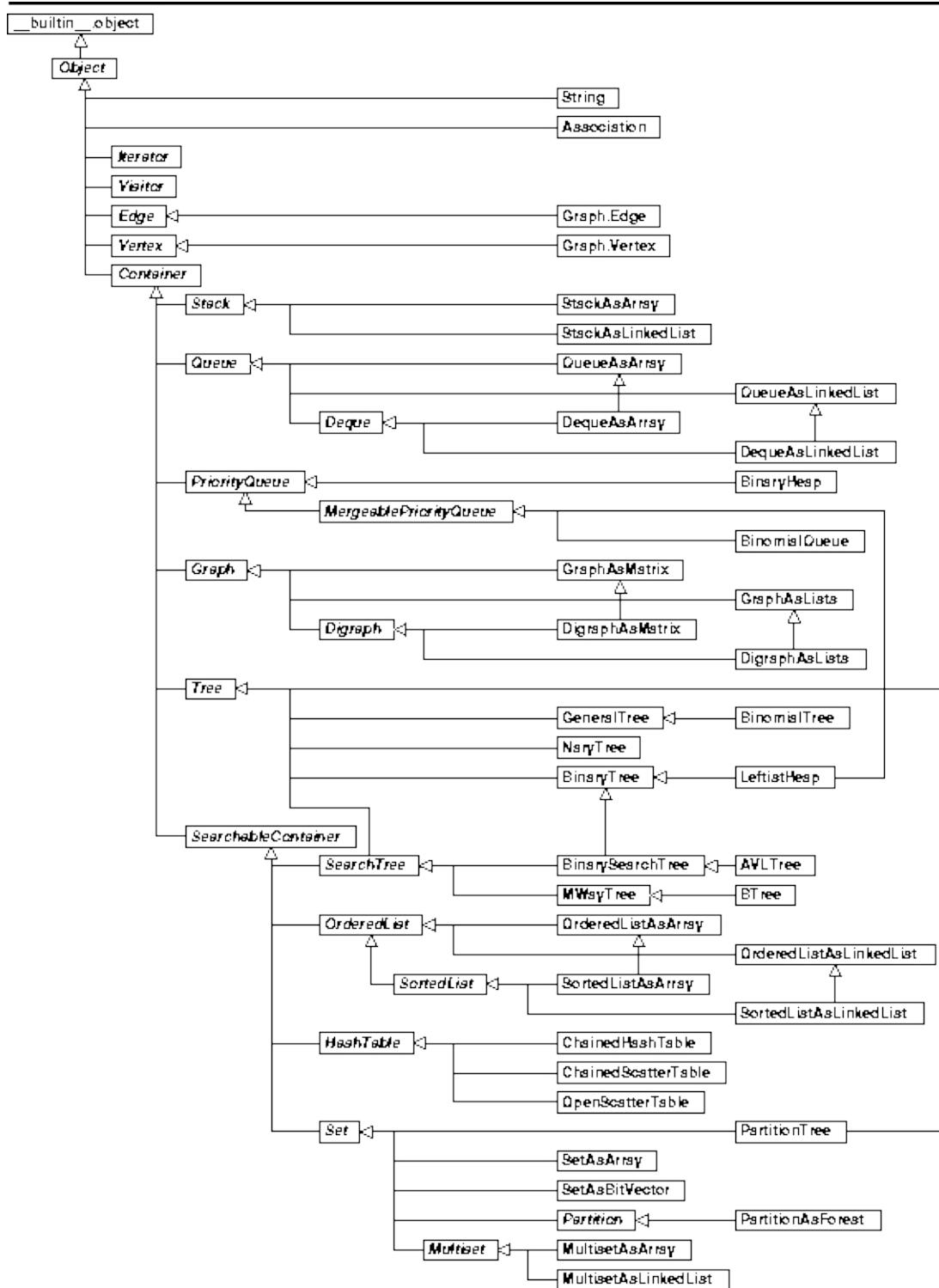


Figure: Complete class hierarchy diagram.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

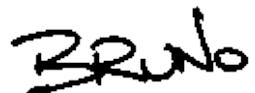
Character Codes

Table: 7-bit ASCII character set.

bits 2-0								
bits 6-3	0	1	2	3	4	5	6	7
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
1	BS	HT	NL	VT	NP	CR	SO	SI
2	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
3	CAN	EM	SUB	ESC	FS	GS	RS	US
4	SP	!	"	#	\$	%	&	'
5	()	*	+	,	-	.	/
6	0	1	2	3	4	5	6	7
7	8	9	:	;	<	=	>	?
010	@	A	B	C	D	E	F	G
011	H	I	J	K	L	M	N	O
012	P	Q	R	S	T	U	V	W
013	X	Y	Z	[\]	^	_
014	`	a	b	c	d	e	f	g
015	h	i	j	k	l	m	n	o
016	p	q	r	s	t	u	v	w
017	x	y	z	{		}	~	DEL

[Next](#) [Up](#) [Previous](#) [Contents](#) [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



References

- 1 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- 2 Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, New York, NY, 1992.
- 3 Ben Albahari, Peter Drayton, and Brad Merrill. *C# Essentials*. O'Reilly & Associates, Inc., Cambridge, MA, 2001.
- 4 ANSI Accredited Standards Committee X3, Information Processing Systems. *Working Paper for Draft Proposed International Standard for Information Systems--Programming Language C++*, December 1996. Document Number X3J16/96-0225 WG21/N1043.
- 5 Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 1996.
- 6 Borland International, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001. *Borland C++ Version 3.0 Programmer's Guide*, 1991.
- 7 Timothy A. Budd. *Classic Data Structures in C++*. Addison-Wesley, Reading, MA, 1994.
- 8 Computational Science Education Project. Mathematical optimization. Virtual book, 1995. <http://csep1.phy.ornl.gov/CSEP/MO/MO.html>.
- 9

Computational Science Education Project. Random number generators.
Virtual book, 1995. <http://csep1.phy.ornl.gov/CSEP/RN/RN.html>.

10

Gaelan Dodds de Wolf, Robert J. Gregg, Barbara P. Harris, and Matthew H. Scargill, editors. *Gage Canadian Dictionary*. Gage Educational Publishing Company, Toronto, Ontario, Canada, 1997.

11

Rick Decker and Stuart Hirshfeld. *Working Classes: Data Structures and Algorithms Using C++*. PWS Publishing Company, Boston, MA, 1996.

12

Adam Drozdek. *Data Structures and Algorithms in C++*. PWS Publishing Company, Boston, MA, 1996.

13

Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

14

James A. Field. Makegraph user's guide. Technical Report 94-04, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, 1994.

15

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

16

Michel Goosens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, MA, 1994.

17

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

18

James Gosling, Frank Yellin, and The Java Team. *The Java Application*

Programming Interface, Volume 1: Core Packages. The Java Series. Addison-Wesley, Reading, MA, 1996.

19

James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets.* The Java Series. Addison-Wesley, Reading, MA, 1996.

20

Irwin Guttman, S. S. Willks, and J. Stuart Hunter. *Introductory Engineering Statistics.* John Wiley & Sons, New York, NY, second edition, 1971.

21

Gregory L. Heileman. *Data Structures, Algorithms, and Object-Oriented Programming.* McGraw-Hill, New York, NY, 1996.

22

Anders Hejlsberg and Scott Wiltamuth. *Microsoft C# Language Specifications.* Microsoft Press, Redmond, WA, 2001.

23

Ellis Horowitz and Sartaj Sahni. *Data Structures in Pascal.* W. H. Freeman and Company, New York, NY, third edition, 1990.

24

Ellis Horowitz, Sartaj Sahni, and Dinesh Mehta. *Fundamentals of Data Structures in C++.* W. H. Freeman and Company, New York, NY, 1995.

25

Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, New York, NY, 1996.

26

Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

27

Leonard Kleinrock. *Queueing Systems, Volume I: Theory.* John Wiley & Sons, New York, NY, 1975.

28

Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition, 1973.

29

Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

30

Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition, 1981.

31

Donald E. Knuth. *The METAFONTbook*. Addison-Wesley, Reading, MA, 1986.

32

Donald E. Knuth. *The TeXbook*. Addison-Wesley, Reading, MA, 1986.

33

Elliot B. Koffman, David Stemple, and Caroline E. Wardle. Recommended curriculum for CS2, 1984. *Communications of the ACM*, 28(8):815-818, August 1985.

34

Leslie Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley, Reading, MA, second edition, 1994.

35

Yedidyah Langsam, Moshe J. Augenstein, and Aaron M. Tenenbaum. *Data Structures Using C and C++*. Prentice Hall, Upper Saddle River, NJ, second edition, 1996.

36

Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

37

Mark Lutz and David Ascher. *Learning Python*. O'Reilly & Associates, Sebastopol, CA, 1999.

38

Kenneth McAlloon and Anthony Tromba. *Calculus*, volume 1BCD. Harcourt Brace Jovanovich, Inc., New York, NY, 1972.

39

Thomas L. Naps. *Introduction to Program Design and Data Structures*. West Publishing, St. Paul, MN, 1993.

40

Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192-1201, October 1988.

41

P. J. Plauger. *The Draft Standard C++ Library*. Prentice Hall, Englewood Cliffs, NJ, 1995.

42

Stephen R. Schach. *Classical and Object-Oriented Software Engineering*. Irwin, Chicago, IL, third edition, 1996.

43

G. Michael Schneider and Steven C. Bruell. *Concepts in Data Structures and Software Development*. West Publishing, St. Paul, MN, 1991.

44

Michele Simionato. The python 2.3 method resolution order. Virtual book, 2003. <http://www.python.org/2.3/mro.html>.

45

Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, second edition, 1991.

46

Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

47

Allen B. Tucker, Bruce H. Barnes, Robert M. Aiken, Keith Barker, Kim B. Bruce, J. Thomas Cain, Susan E. Conry, Gerald L. Engel, Richard G. Epstein, Doris K. Ladtke, Michael C. Mulder, Jean B. Rogers, Eugene H. Spafford, and A. Joe Turner. *Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force*. ACM/IEEE, 1991.

48

Guido van Rossum. Python library reference; Release 2.3. Virtual book, 2003. <http://www.python.org/doc/current/lib/lib.html>.

49

Guido van Rossum. Python reference manual; Release 2.3. Virtual book, 2003. <http://www.python.org/doc/current/ref/ref.html>.

50

Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, New York, NY, 1997.

51

Larry Wall and Randal L. Schwartz. *Programming perl*. O'Reilly & Associates, Sebastopol, CA, 1991.

52

Mark Allen Weiss. *Data Structures and Algorithm Analysis*. Benjamin/Cummings, Redwood City, CA, second edition, 1995.

53

Mark Allen Weiss. *Algorithms, Data Structures and Problem Solving with C++*. Addison-Wesley, Menlo Park, CA, 1996.

54

Geoff Whale. *Data Structures and Abstraction Using C*. Thomson Nelson Australia, Melbourne, Australia, 1996.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

**Data Structures and Algorithms with Object-Oriented Design Patterns
in Python** [Next](#) [Up](#) [Previous](#) [Contents](#) 

Index

- o
 - seebig oh, seelittle oh
- γ
 - seeEuler's constant
- Ω
 - seeomega
- Θ
 - seetheta
- λ
 - seelambda
- __init__ method
 - [init Method](#)
- abstract algorithms
 - [Tree Traversals](#)
- abstract class
 - [Class Hierarchy](#), [Class Hierarchy](#), [Class Hierarchy](#), [Algorithmic Abstraction](#)
- abstract data type
 - [Foundational Data Structures](#), [Abstract Data Types](#)
- abstract method
 - [Class Hierarchy](#)
- abstract property
 - [Class Hierarchy](#)
- abstract solver
 - [Abstract Backtracking Solvers](#)
- abstract sorter
 - [Sorting and Sorters](#)
- access path
 - [Inserting Items into an](#)
- accessor
 - [Array Properties](#), [Array Properties](#), [PropertiesAccessors and Mutators](#)
- activation record
 - [The Basic Axioms](#)
- activity-node graph
 - [Application: Critical Path Analysis](#)
- actual parameter
 - [Parameter Passing](#)

acyclic

 directed graph

[Directed Acyclic Graphs](#)

adapter

[PreorderInorder, and Postorder](#), [PreorderInorder, and Postorder](#)

address

[Abstract Data Types](#)

adjacency lists

[Adjacency Lists](#)

adjacency matrix

[Adjacency Matrices](#)

adjacent

[Terminology](#)

ADT

 see abstract data type

algorithmic abstraction

[Algorithmic Abstraction](#)

ancestor

[More Terminology](#)

proper

[More Terminology](#)

and

[UnionIntersection, and Difference](#)

annealing

[Simulated Annealing](#)

annealing schedule

[Simulated Annealing](#)

arc

 directed

[Terminology](#)

 undirected

[Undirected Graphs](#)

arithmetic series

[About Arithmetic Series Summation](#)

arithmetic series summation

[An example-Geometric Series Summation](#), [About Arithmetic Series Summation](#)

arity

[N-ary Trees](#)

array
[Foundational Data Structures](#)

ASCII
[Character String Keys](#)

association
[Searchable Containers](#)

asymptotic behavior
[Asymptotic Notation](#)

attribute
 class
 [Instances](#)[Instance Attributes and](#)
 instance
 [Instances](#)[Instance Attributes and](#)

attributes
[Abstract Data Types](#)

AVL balance condition
[AVL Search Trees](#)

AVL rotation
[Balancing AVL Trees](#)

AVL tree
[Basics](#)

B-Tree
[B-Trees, B-Trees](#)

Bachmann, P.
[An Asymptotic Upper Bound-Big](#)

backtracking algorithms
[Backtracking Algorithms](#)

bag
[Projects, Multisets](#)

balance condition
[AVL Search Trees, B-Trees](#)

AVL
[AVL Search Trees](#)

base class
[Class Hierarchy, Derivation and Inheritance](#)

big oh
[An Asymptotic Upper Bound-Big](#)
tightness
[Tight Big Oh Bounds, More Notation-Theta and Little](#)

tightness

[Tight Big Oh Bounds, More Notation-Theta and Little](#)

transitive property

[Properties of Big Oh](#)

binary digit

[Binomial Queues](#)

binary heap

[Sorting with a Heap](#)

binary operator

[Applications](#)

binary search

[Locating Items in an , Example-Binary Search](#)

binary search tree

[Binary Search Trees, Binary Search Trees](#)

binary tree

[Binary Trees, Binary Trees](#)

complete

[Complete Trees](#)

binding

[Abstract Data Types, Names](#)

binomial

[Binomial Trees](#)

binomial coefficient

[Binomial Trees](#)

bit

[Binomial Queues](#)

Boolean

and

[UnionIntersection, and Difference](#)

or

[UnionIntersection, and Difference](#)

bound

[Abstract Data Types](#)

branch-and-bound

[Branch-and-Bound Solvers](#)

breadth-first spanning tree

[Constructing Spanning Trees](#)

breadth-first traversal

[Applications, Applications, Breadth-First Traversal, Example-Balancing](#)

[Scales, Breadth-First Traversal](#)

brute-force algorithms

[Brute-Force and Greedy Algorithms](#)

bubble sort

[Bubble Sort](#)

bucket sort

[Example-Bucket Sort](#)

buckets

[Example-Bucket Sort](#)

built-in scope

[Scopes and Namespaces](#)

C++ programming language

[Abstract Data Types](#)

carry

[Merging Binomial Queues](#)

ceiling function

[About Harmonic Numbers](#)

central limit theorem

[Exercises](#)

chained scatter table

[Chained Scatter Table](#)

child

[Applications, Terminology](#)

circular list

[Singly-Linked Lists, Doubly-Linked and Circular Lists](#)

class

[Classes](#)

abstract

[Algorithmic Abstraction](#)

classic

[Abstract Objects and the , Derivation and Inheritance](#)

classic

[Abstract Objects and the , Derivation and Inheritance](#)

new-style

[Abstract Objects and the , Derivation and Inheritance](#)

new-style

[Abstract Objects and the , Derivation and Inheritance](#)

class attribute

[InstancesInstance Attributes and](#)

classic class

[Abstract Objects and the , Derivation and Inheritance](#)

clock frequency

[A Simplified Model of](#)

clock period

[A Simplified Model of](#)

coalesce

[Chained Scatter Table](#)

cocktail shaker sort

[Exercises](#)

coefficient

binomial

[Binomial Trees](#)

collapsing find

[Collapsing Find](#)

column-major order

[Exercises](#)

commensurate

elements

[Sorted Lists, Basics](#)

elements

[Sorted Lists, Basics](#)

functions

[More Big Oh Fallacies , More Big Oh Fallacies](#)

functions

[More Big Oh Fallacies , More Big Oh Fallacies](#)

compact

[The Fragmentation Problem](#)

compaction

[Mark-and-Compact Garbage Collection](#)

complement

[Exercises](#)

complete N -ary tree

[Complete \$N\$ -ary Trees](#)

complete binary tree

[Complete Trees, Sorting with a Heap](#)

complex numbers

[Example-Complex Numbers](#)

component

connected
 [Connectedness of an Undirected](#)
compound statement
 [Rules For Big Oh](#)
concrete class
 [Class Hierarchy, Class Hierarchy](#)
conjunction
 [SetsMultisets, and Partitions](#)
connected
 undirected graph
 [Connectedness of an Undirected](#)
connected component
 [Connectedness of an Undirected, Exercises](#)
conquer
 seedivide
constant
 [Conventions for Writing Big](#)
ContainerEmpty
 [first and last Properties](#)
copy
 [copy Method](#)
 shallow
 [copy Method](#)
counted do loop
 [Rules For Big Oh](#)
critical activity
 [Application: Critical Path Analysis](#)
critical path
 [Application: Critical Path Analysis](#)
critical path analysis
 [Application: Critical Path Analysis](#)
cubic
 [Conventions for Writing Big](#)
cycle
 [More Terminology](#)
negative cost
 [Single-Source Shortest Path](#)
simple
 [More Terminology](#)

dangling pointer

[What is Garbage?](#)

dangling reference

[What is Garbage?](#)

data ordering property

[M-Way Search Trees](#)

database

[Associations](#)

decision tree

[A Lower Bound on](#)

defragment

[The Fragmentation Problem](#)

degree

[Applications](#)

in

[Terminology](#)

out

[Terminology](#)

dense graph

[Sparse vs. Dense Graphs](#)

depth

[More Terminology](#)

depth-first spanning tree

[Constructing Spanning Trees](#)

depth-first traversal

[Example-Balancing Scales, Depth-First Traversal](#)

deque

[StacksQueues, and Deques, Deques](#)

derivation

[Class Hierarchy, Derivation and Inheritance](#)

derivative

[Applications](#)

derived class

[Derivation and Inheritance](#)

descendant

[More Terminology](#)

proper

[More Terminology](#)

descriptor

[Abstract Methods](#)

difference

[Sets](#)[Multisets](#), and [Partitions](#), [Basics](#), [Union](#)[Intersection](#), and [Difference](#)
symmetric

[Exercises](#)

differentiation

[Applications](#)

digit

binary

[Binomial Queues](#)

digraph

seedirected graph

Dijkstra's algorithm

[Dijkstra's Algorithm](#)

directed acyclic graph

[Directed Acyclic Graphs](#)

directed arc

[Terminology](#)

directed graph

[Directed Graphs](#)

discrete event simulation

[Discrete Event Simulation](#)

disjunction

[Sets](#)[Multisets](#), and [Partitions](#)

distribution sorting

[Distribution Sorting](#)

distribution sorts

[Sorter Class Hierarchy](#)

divide and conquer

[Top-Down Algorithms: Divide-and-Conquer](#)

division method of hashing

[Division Method](#)

double hashing

[Double Hashing](#)

double rotation

[Double Rotations](#)

double-ended queue

[Deques](#)

doubly-linked list

[Doubly-Linked and Circular Lists](#)

dual

[Application: Critical Path Analysis](#)

dynamic binding

[Abstract Data Types](#)

dynamic programming

[Bottom-Up Algorithms: Dynamic Programming](#)

earliest event time

[Application: Critical Path Analysis](#)

edge

[Applications, Terminology](#)

emanate

[Terminology](#)

incident

[Terminology](#)

element

[SetsMultisets, and Partitions](#)

emanate

[Terminology](#)

enumeration

[Projects](#)

equivalence classes

[Applications](#)

equivalence of trees

[Comparing Trees](#)

equivalence relation

[Applications, Kruskal's Algorithm](#)

Euler's constant

[About Harmonic Numbers, Solving The Recurrence-Telescoping, Average Running Time](#)

Euler, Leonhard

[Binomial Trees](#)

Eulerian walk

[Exercises](#)

evaluation stack

[Postfix Notation](#)

event-node graph

[Application: Critical Path Analysis](#)

exception
 [first and last Properties](#), [extract Method](#)

exception handler
 [Exceptions](#)

exceptions
 [Exceptions](#)

exchange sorting
 [Exchange Sorting](#)

exchange sorts
 [Sorter Class Hierarchy](#)

exclusive or
 [Character String Keys](#), [Character String Keys](#)

exponent
 [Floating-Point Keys](#)

exponential
 [Conventions for Writing Big](#)

exponential cooling
 [Simulated Annealing](#)

exponential distribution
 [Exponentially Distributed Random Variables](#)

expression tree
 [Expression Trees](#)

extend
 [Example-Graphical Objects](#)

external node
 [N-ary Trees](#)

external path length
 [Unsuccessful Search](#)

factorial
 [Analyzing Recursive Methods](#)

feasible solution
 [Brute-Force Algorithm](#)

Fibonacci hashing method
 [Fibonacci Hashing](#)

Fibonacci number
 [Fibonacci Hashing](#), [AVL Search Trees](#)

Fibonacci numbers
 [Example-Fibonacci Numbers](#), [Example-Computing Fibonacci Numbers](#)

closed-form expression

[Example-Fibonacci Numbers](#)

generalized

[Example-Generalized Fibonacci Numbers](#)

FIFO

[Queues](#)

fifo-in, first-out

[Queues](#)

find

collapsing

[Collapsing Find](#)

floor function

[About Harmonic Numbers](#)

Floyd's algorithm

[Floyd's Algorithm](#)

forest

[Binomial Queues, Binomial Queues, Implementing a Partition using](#)

formal parameter

[Parameter Passing](#)

foundational data structure

[Foundational Data Structures](#)

fully connected graph

[Exercises](#)

garbage

[What is Garbage?](#)

garbage collection

[What is Garbage?](#)

mark-and-compact

[Mark-and-Compact Garbage Collection](#)

mark-and-sweep

[Mark-and-Sweep Garbage Collection](#)

reference counting

[Reference Counting Garbage Collection](#)

stop-and-copy

[Stop-and-Copy Garbage Collection](#)

Gauss, Karl Friedrich

[Binomial Trees](#)

generalized Fibonacci numbers

[Example-Generalized Fibonacci Numbers](#)

geometric series

[About Geometric Series Summation](#)

geometric series summation

[An example-Geometric Series Summation](#), [Example-Geometric Series Summation Again](#), [About Geometric Series Summation](#), [Example-Geometric Series Summation Yet](#)

getter

[PropertiesAccessors and Mutators](#)

global scope

[Scopes and Namespaces](#)

golden ratio

[Fibonacci Hashing](#)

graph

connectedness

[Connectedness of an Undirected](#)

dense

[Sparse vs. Dense Graphs](#)

directed

[Directed Graphs](#)

directed acyclic

[Directed Acyclic Graphs](#)

labeled

[Labeled Graphs](#)

sparse

[Sparse vs. Dense Graphs](#)

traversal

[Graph Traversals](#)

undirected

[Undirected Graphs](#)

graph theory

[Graphs and Graph Algorithms](#)

handle

[Handles](#)

harmonic number

[Average Running Times](#), [About Harmonic Numbers](#), [Average Case Analysis](#), [Solving The Recurrence-Telescoping](#), [Average Running Time](#)

harmonic series

[About Harmonic Numbers](#)

hash function

[Keys and Hash Functions](#), [Keys and Hash Functions](#)

hash table

[Hash Tables](#)

hashing

division method

[Division Method](#)

Fibonacci method

[Fibonacci Hashing](#)

middle-square method

[Middle Square Method](#)

multiplication method

[Multiplication Method](#)

head

[Singly-Linked Lists](#)

heap

[Basics, Garbage Collection and the](#)

heapify

[Sorting with a Heap](#)

heapsort

[Sorting with a Heap](#)

height

of a node in a tree

[More Terminology](#)

of a tree

[More Terminology](#)

heuristic

[Depth-FirstBranch-and-Bound Solver](#)

hierarchy

[Trees](#)

Horner's rule

[Another example-Horner's Rule, Example-Geometric Series Summation](#)

[Again, Character String Keys](#)

in-degree

[Terminology, Topological Sort](#)

in-place sorting

[Insertion Sorting, Selection Sorting](#)

incident

[Terminology](#)

increment

[Generating Random Numbers](#)

infix

[Applications](#)

infix notation

[Infix Notation](#)

inheritance

[Derivation and Inheritance](#)

multiple

[Derivation and Inheritance](#)

inorder traversal

[Inorder Traversal, Traversing a Search Tree](#)

M-way tree

[Traversing a Search Tree](#)

insertion sorting

[Insertion Sorting](#)

straight

[Straight Insertion Sort](#)

insertion sorts

[Sorter Class Hierarchy](#)

instance

[InstancesInstance Attributes and](#)

instance attribute

[InstancesInstance Attributes and](#)

internal node

[N-ary Trees](#)

internal path length

[Unsuccessful Search](#)

complete binary tree

[Complete Trees](#)

internal path length of a tree

[Successful Search](#)

Internet domain name

[Character String Keys](#)

intersection

[SetsMultisets, and Partitions, Basics, UnionIntersection, and Difference](#)

interval

search

[Locating Items in an](#)

inverse modulo W

[Multiplication Method](#)

inversion

[Average Running Time](#)

isomorphic

[Alternate Representations for Trees](#)

isomorphic trees

[Exercises](#)

iterative algorithm

[Example-Fibonacci Numbers](#)

iterator

[Iterators](#)

iterator protocol

[Iterators](#)

Java programming language

[Abstract Data Types](#)

key

[Associations, Keys and Hash Functions](#)

keyed data

[Using Associations](#)

KeyError

[extract Method](#)

knapsack problem

[Example-0/1 Knapsack Problem](#)

Kruskal's algorithm

[Kruskal's Algorithm](#)

L'Hôpital's rule

[About Logarithms, About Logarithms](#)

labeled graph

[Labeled Graphs](#)

lambda

seeload factor

last-in, first-out

[Stacks](#)

latest event time

[Application: Critical Path Analysis](#)

leaf

[Terminology](#)

leaf node

[N-ary Trees](#)

least-significant-digit-first radix sorting

[Radix Sort](#)

left subtree

[Binary Trees, M-Way Search Trees](#)

leftist tree

[Leftist Trees](#)

level

[More Terminology](#)

level-order

[Complete N-ary Trees](#)

level-order traversal

[Applications](#)

lexicographic order

[Array Subscript Calculations](#)

lexicographic ordering

[Radix Sort](#)

lexicographically precede

[Radix Sort](#)

LGB rule

[Scopes and Namespaces](#)

lifetime

[Abstract Data Types, Abstract Data Types, Objects and Types](#)

LIFO

[Stacks](#)

limit

[Properties of Big Oh](#)

linear

[Conventions for Writing Big](#)

linear congruential random number generator

[Generating Random Numbers](#)

linear probing

[Linear Probing](#)

linear search

[Yet Another example-Finding the](#)

linked list

[Foundational Data Structures](#)

list

[Ordered Lists and Sorted](#)

little oh

[More Notation-Theta and Little](#)

live

[Mark-and-Sweep Garbage Collection](#)

LL rotation

[Single Rotations](#)

in a B-tree

[Removing Items from a](#)

load factor

[Average Case Analysis](#)

local scope

[Scopes and Namespaces](#)

log squared

[Conventions for Writing Big](#)

logarithm

[Conventions for Writing Big](#)

long integer

[The Basic Axioms](#)

loop

[More Terminology](#)

loose asymptotic bound

[More Notation-Theta and Little](#)

LR rotation

[Double Rotations](#)

Lukasiewicz, Jan

[Applications](#)

M -way search tree

[\$M\$ -Way Search Trees](#)

mantissa

[Floating-Point Keys](#)

many-to-one mapping

[Keys and Hash Functions](#)

mark-and-compact garbage collection

[Mark-and-Compact Garbage Collection](#)

mark-and-sweep garbage collection

[Mark-and-Sweep Garbage Collection](#)

matrix

[Matrices](#)

adjacency

[Adjacency Matrices](#)

sparse

[Adjacency Matrices](#)

max-heap

[Sorting with a Heap](#)

median

[Selecting the Pivot](#)

median-of-three pivot selection

[Selecting the Pivot](#)

memory leak

[What is Garbage?](#)

merge sort

[Example-Merge Sorting](#)

merge sorting

[Merge Sorting](#)

merge sorts

[Sorter Class Hierarchy](#)

mergeable priority queue

[Basics](#)

merging nodes in a B-tree

[Removing Items from a](#)

Mersenne primes

[The Minimal Standard Random](#)

method

static

[Static Methods](#)

method resolution order

[Multiple Inheritance](#)

middle-square hashing method

[Middle Square Method](#)

min heap

[Basics](#)

minimal subgraph

[Minimum-Cost Spanning Trees](#)

minimum spanning tree

[Minimum-Cost Spanning Trees](#)

mixed linear congruential random number generator

[Generating Random Numbers](#)

modulus

[Generating Random Numbers](#)

Monte Carlo methods

[Monte Carlo Methods](#)

multi-dimensional array

[Multi-Dimensional Arrays](#)

multiple inheritance

[Derivation and Inheritance](#)

multiplication hashing method

[Multiplication Method](#)

multiplicative linear congruential random number generator

[Generating Random Numbers](#)

multiset

[Multisets](#)

mutator

[PropertiesAccessors and Mutators](#)

N-ary tree

N-ary tree

[N-ary Trees](#)

N-queens problem

N-queens problem

[Exercises](#)

name

[Abstract Data Types, Abstract Data Types, Abstract Data Types, Names](#)

namespace

[Scopes and Namespaces](#)

Nary tree

`textbf`

negative cost cycle

[Single-Source Shortest Path](#)

nested class

[Implementation, Nested Classes](#)

new-style class

[Abstract Objects and the , Derivation and Inheritance](#)

Newton, Isaac.

[Binomial Trees](#)

node

[Applications, Basics, N-ary Trees, Binary Trees, Terminology](#)

non-recursive algorithm

[Example-Fibonacci Numbers](#)

normalize

[Generating Random Numbers](#)

null path length

[Leftist Trees](#), [Leftist Trees](#)

object

[Objects and Types](#)

object-oriented programming

[Abstract Data Types](#)

object-oriented programming language

[Abstract Data Types](#)

objective function

[Brute-Force Algorithm](#)

odd-even transposition sort

[Exercises](#)

omega

[An Asymptotic Lower Bound-Omega](#)

open addressing

[Scatter Table using Open](#)

operator overloading

[Operator Overloading](#)

operator precedence

[Applications](#)

optimal binary search tree

[Exercises](#)

or

[UnionIntersection, and Difference](#)

ordered list

[Ordered Lists and Sorted](#)

ordered tree

[N-ary Trees](#), [Binary Trees](#)

ordinal number

[Positions of Items in](#)

oriented tree

[N-ary Trees](#)

out-degree

[Terminology](#)

overloading operators

[Operator Overloading](#)

override

[Class Hierarchy](#), [Derivation and Inheritance](#), [Derivation and Inheritance](#)

parameter passing

[Parameter Passing](#)

parent

[Applications, Terminology](#)

parentheses

[Applications](#)

partial order

[Comparing Sets](#)

partition

[Partitions, Kruskal's Algorithm](#)

Pascal's triangle

[Example-Computing Binomial Coefficients](#)

Pascal, Blaise

[Example-Computing Binomial Coefficients](#)

pass-by-reference

[Parameter Passing](#)

path

[Terminology](#)

access

[Inserting Items into an](#)

path length

external

[Unsuccessful Search](#)

internal

[Unsuccessful Search](#)

weighted

[Shortest-Path Algorithms](#)

perfect binary tree

[Searching a Binary Tree, AVL Search Trees](#)

period

[Generating Random Numbers](#)

pivot

[Quicksort](#)

plain integer

[The Basic Axioms](#)

Polish notation

[Applications](#)

polymorphism

[Class Hierarchy, Polymorphism](#)

polynomial

[About Polynomials](#), [About Polynomials Again](#)

postcondition

[Inserting Items in a](#)

postorder traversal

[Postorder Traversal](#)

power set

[Array and Bit-Vector Sets](#)

precede lexicographically

[Radix Sort](#)

precondition

[Inserting Items in a](#)

predecessor

[Instance Attributes](#), [More Terminology](#)

prefix notation

[Prefix Notation](#)

preorder traversal

[Preorder Traversal](#)

prepend

[prepend Method](#)

Prim's algorithm

[Prim's Algorithm](#)

primary clustering

[Linear Probing](#)

prime

relatively

[Multiplication Method](#)

priority queue

mergeable

[Basics](#)

probability density function

[Exponentially Distributed Random Variables](#)

probe sequence

[Scatter Table using Open](#)

proper subset

[Comparing Sets](#)

proper superset

[Comparing Sets](#)

pruning a solution space

[Branch-and-Bound Solvers](#)

pseudorandom

[Generating Random Numbers](#)

Python programming language

[Abstract Data Types](#)

quadratic

[Conventions for Writing Big](#)

quadratic probing

[Quadratic Probing](#)

queue

[StacksQueues, and Deques](#)

quicksort

[Quicksort](#)

radix sorting

[Radix Sort](#)

raise

[Exceptions](#)

random number generator

linear congruential

[Generating Random Numbers](#)

mixed linear congruential

[Generating Random Numbers](#)

multiplicative linear congruential

[Generating Random Numbers](#)

random numbers

[Generating Random Numbers](#)

random variable

[Random Variables](#)

rank

[Union by Height or](#)

record

[Abstract Data Types](#)

recurrence relation

[Analyzing Recursive Methods](#)

recursive algorithm

[Analyzing Recursive Methods, Example-Fibonacci Numbers](#)

reference count

[Reference Counting Garbage Collection](#)

reference counting garbage collection

[Reference Counting Garbage Collection](#)

reflexive

[Applications](#)

relation

equivalence

[Applications](#)

relatively prime

[Multiplication Method](#)

repeated substitution

[Solving Recurrence Relations-Repeated Substitution](#)

Reverse-Polish notation

[Applications](#)

right subtree

[Binary Trees](#)

RL rotation

[Double Rotations](#)

root

[Basics, Mark-and-Sweep Garbage Collection](#)

rotation

AVL

[Balancing AVL Trees](#)

double

[Double Rotations](#)

LL

[Single Rotations, Removing Items from a](#)

LL

[Single Rotations, Removing Items from a](#)

LR

[Double Rotations](#)

RL

[Double Rotations](#)

RR

[Single Rotations, Removing Items from a](#)

RR

[Single Rotations, Removing Items from a](#)

single

[Double Rotations](#)

row-major order

[Array Subscript Calculations](#)

RPN

seeReverse-Polish notation

RR rotation

[Single Rotations](#)

in a B-tree

[Removing Items from a](#)

scales

[Example-Balancing Scales](#)

scatter tables

[Scatter Tables](#)

scope

[Abstract Data Types, Abstract Data Types, Scopes and Namespaces](#)

built-in

[Scopes and Namespaces](#)

global

[Scopes and Namespaces](#)

local

[Scopes and Namespaces](#)

search interval

[Locating Items in an](#)

search tree

M-way

[M-Way Search Trees](#)

binary

[Binary Search Trees](#)

seed

[Generating Random Numbers](#)

selection sorting

[Selection Sorting](#)

selection sorts

[Sorter Class Hierarchy](#)

sentinel

[Singly-Linked Lists, Adjacency Matrices](#)

separate chaining

[Separate Chaining](#)

set

[SetsMultisets, and Partitions](#)

setter

[PropertiesAccessors and Mutators](#)

shallow copy

[copy](#) Method

sibling

[Terminology](#)

sign

[Floating-Point Keys](#)

significant

[Floating-Point Keys](#)

simple cycle

[More Terminology](#)

simulated annealing

[Simulated Annealing](#)

simulation time

[Discrete Event Simulation](#)

single rotation

[Double Rotations](#)

single-ended queue

[Queues](#)

singleton

[Exercises, Implementation](#)

singly-linked list

[Doubly-Linked and Circular Lists](#)

size

[Abstract Data Types](#)

slack time

[Application: Critical Path Analysis](#)

slide

[Handles](#)

solution space

[Example-Balancing Scales](#)

solver

[Abstract Backtracking Solvers](#)

sort

topological

[Topological Sort](#)

sorted list

[Ordered Lists and Sorted , Sorted Lists, Basics](#)

sorter

[Sorting and Sorters](#)

sorting

in place
 [Selection Sorting](#)
in-place
 [Insertion Sorting](#)
sorting algorithm
 bucket sort
 [Example-Bucket Sort](#)

sorting by distribution
 [Distribution Sorting](#)
sorting by exchanging
 [Exchange Sorting](#)
sorting by insertion
 [Insertion Sorting](#)

sorting by merging
 [Merge Sorting](#)
sorting by selection
 [Selection Sorting](#)

source
 [Exercises](#)

spanning tree
 [Minimum-Cost Spanning Trees](#)
 breadth-first
 [Constructing Spanning Trees](#)
 depth-first
 [Constructing Spanning Trees](#)
 minimum
 [Minimum-Cost Spanning Trees](#)

sparse graph
 [Sparse vs. Dense Graphs](#)
sparse matrix
 [Adjacency Matrices](#)

specializes
 [Class Hierarchy](#)

stable sorts
 [Basics](#)

stack
 [Stacks](#)

stack frame
 [The Basic Axioms](#)

state

[Discrete Event Simulation](#)

static binding

[Abstract Data Types](#)

static method

[Static Methods](#)

Stirling numbers

[Partitions, Partitions](#)

stop-and-copy garbage collection

[Stop-and-Copy Garbage Collection](#)

straight insertion sorting

[Straight Insertion Sort](#)

straight selection sorting

[Straight Selection Sorting](#)

strongly connected

[Connectedness of a Directed](#)

subgraph

[Minimum-Cost Spanning Trees](#)

minimal

[Minimum-Cost Spanning Trees](#)

subset

[Comparing Sets](#)

proper

[Comparing Sets](#)

subtraction

[SetsMultisets, and Partitions](#)

subtree

[Applications](#)

successor

[Instance Attributes, More Terminology](#)

superclass

[Example-Graphical Objects](#)

superset

[Comparing Sets](#)

proper

[Comparing Sets](#)

symbol table

[HashingHash Tables, and , Applications](#)

symmetric

Applications

symmetric difference

Exercises

tail

Singly-Linked Lists, Singly-Linked Lists

telescoping

Solving The Recurrence-Telescoping, Running Time of Divide-and-Conquer

temperature

Simulated Annealing

tertiary tree

N-ary Trees

theta

More Notation-Theta and Little

tight asymptotic bound

Tight Big Oh Bounds

time

simulation

Discrete Event Simulation

topological sort

Topological Sort

total order

Basics

binary trees

Comparing Trees

transitive

Sorted Lists, Applications, Basics

traversal

Tree Traversals, Example-Balancing Scales, Graph Traversals

breadth-first

Breadth-First Traversal, Breadth-First Traversal

breadth-first

Breadth-First Traversal, Breadth-First Traversal

depth-first

Depth-First Traversal

inorder

Inorder Traversal, Traversing a Search Tree

inorder

Inorder Traversal, Traversing a Search Tree

postorder

[Postorder Traversal](#)

preorder

[Preorder Traversal](#)

tree

[Basics](#)

N-ary

[N-ary Trees](#)

binary

[Binary Trees](#)

equivalence

[Comparing Trees](#)

expression

[Expression Trees](#)

height

[More Terminology](#)

internal path length

[Successful Search](#)

leftist

[Leftist Trees](#)

ordered

[N-ary Trees, Binary Trees](#)

ordered

[N-ary Trees, Binary Trees](#)

oriented

[N-ary Trees](#)

search

seesearch tree

tertiary

[N-ary Trees](#)

traversal

[Tree Traversals](#)

tree traversal

[Applications](#)

tuple

[Associations](#)

type

[Abstract Data Types, Objects and Types](#)

undirected arc

Undirected Graphs

undirected graph

Undirected Graphs

Unicode character set

Example

Unicode escape

Example

uniform distribution

Spreading Keys Evenly

uniform hashing model

Average Case Analysis

union

SetsMultisets, and Partitions, Basics, UnionIntersection, and Difference

union by rank

Union by Height or

union by size

Union by Size

universal set

SetsMultisets, and Partitions, Kruskal's Algorithm

unsorted list

Basics

value

Abstract Data Types, Associations, Objects and Types

Venn diagram

Alternate Representations for Trees, SetsMultisets, and Partitions

vertex

Terminology

visibility

Abstract Data Types

visitor

Visitors

weakly connected

Connectedness of a Directed

weighted path length

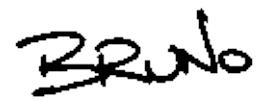
Shortest-Path Algorithms

word size

Middle Square Method

[Next](#) [Up](#) [Previous](#) [Contents](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

A handwritten signature in black ink that reads "Bruno". The signature is fluid and cursive, with the letters "B" and "R" being particularly prominent.

[Site Contents](#)

[Python Source Code](#)

[Bruno R. Preiss](#)

[C++ Version](#)

[Java Version](#)

[C# Version](#)

Data Structures and Algorithms

with Object-Oriented Design Patterns in Python

Bruno R. Preiss, B.A.Sc., M.A.Sc. Ph.D., P.Eng.

Site Contents

- [The Web Book](#)
- [Front Matter](#) (pdf)
 - Student Resources
- [Links to the Python Source Code from the book](#)
(Also available as a [zip](#) file).
 - Opus7 Package
- [Documentation](#)
- [Class Hierarchy Diagram](#)
- [Sources](#)
- [Source Distribution](#) (gzipped tar file in Python distutil format)
- [Instructor Resources](#) (Password provided to course instructors only).
- [Frequently Asked Questions](#)
- [Errata](#)

[Copyright © 2003](#) by [Bruno R. Preiss, P.Eng.](#) All rights reserved.



Contents

- [Colophon](#)
- [Dedication](#)
- [Preface](#)
 - [Goals](#)
 - [Approach](#)
 - [Outline](#)
 - [Suggested Course Outline](#)
 - [Online Course Materials](#)
- [Introduction](#)
 - [What This Book Is About](#)
 - [Object-Oriented Design](#)
 - [Abstraction](#)
 - [Encapsulation](#)
 - [Object Hierarchies and Design Patterns](#)
 - [Containers](#)
 - [Iterators](#)
 - [Visitors](#)
 - [Cursors](#)
 - [Adapters](#)
 - [Singletons](#)
 - [The Features of Python You Need to Know](#)
 - [Objects, Values and Types](#)
 - [Naming and Binding](#)
 - [Parameter Passing](#)
 - [Classes](#)
 - [Inheritance](#)
 - [New-Style Classes](#)
 - [Other Features](#)
 - [How This Book Is Organized](#)
 - [Models and Asymptotic Analysis](#)
 - [Foundational Data Structures](#)
 - [Abstract Data Types and the Class Hierarchy](#)
 - [Data Structures](#)
 - [Algorithms](#)
- [Algorithm Analysis](#)

- [A Detailed Model of the Computer](#)
 - [The Basic Axioms](#)
 - [A Simple example-Arithmetic Series Summation](#)
 - [Array Subscripting Operations](#)
 - [Another example-Horner's Rule](#)
 - [Analyzing Recursive Methods](#)
 - [Solving Recurrence Relations-Repeated Substitution](#)
 - [Yet Another example-Finding the Largest Element of an Array](#)
 - [Average Running Times](#)
 - [About Harmonic Numbers](#)
 - [Best-Case and Worst-Case Running Times](#)
 - [The Last Axiom](#)
- [A Simplified Model of the Computer](#)
 - [An example-Geometric Series Summation](#)
 - [About Arithmetic Series Summation](#)
 - [Example-Geometric Series Summation Again](#)
 - [About Geometric Series Summation](#)
 - [Example-Computing Powers](#)
 - [Example-Geometric Series Summation Yet Again](#)
- [Exercises](#)
- [Projects](#)
- [Asymptotic Notation](#)
 - [An Asymptotic Upper Bound-Big Oh](#)
 - [A Simple Example](#)
 - [Big Oh Fallacies and Pitfalls](#)
 - [Properties of Big Oh](#)
 - [About Polynomials](#)
 - [About Logarithms](#)
 - [Tight Big Oh Bounds](#)
 - [More Big Oh Fallacies and Pitfalls](#)
 - [Conventions for Writing Big Oh Expressions](#)
 - [An Asymptotic Lower Bound-Omega](#)
 - [A Simple Example](#)
 - [About Polynomials Again](#)
 - [More Notation-Theta and Little Oh](#)
 - [Asymptotic Analysis of Algorithms](#)
 - [Rules For Big Oh Analysis of Running Time](#)
 - [Example-Prefix Sums](#)
 - [Example-Fibonacci Numbers](#)

- [Example-Bucket Sort](#)
 - [Reality Check](#)
 - [Checking Your Analysis](#)
 - [Exercises](#)
 - [Projects](#)
- [Foundational Data Structures](#)
 - [Python Lists and Arrays](#)
 - [Extending Python Lists - An Array Class](#)
 - [__init__ Method](#)
 - [copy Method](#)
 - [getitem and setitem Methods](#)
 - [Array Properties](#)
 - [Resizing an Array](#)
 - [Multi-Dimensional Arrays](#)
 - [Array Subscript Calculations](#)
 - [An Implementation](#)
 - [MultiDimensionalArray class __init__ Method](#)
 - [MultiDimensionalArray class __getitem__ and __setitem__ Methods](#)
 - [Matrices](#)
 - [Dense Matrices](#)
 - [Canonical Matrix Multiplication](#)
 - [Singly-Linked Lists](#)
 - [An Implementation](#)
 - [List Elements](#)
 - [LinkedList Class __init__ Method](#)
 - [purge Method](#)
 - [LinkedList Properties](#)
 - [first and last Properties](#)
 - [prepend Method](#)
 - [append Method](#)
 - [copy Method](#)
 - [extract Method](#)
 - [insertAfter and insertBefore Methods](#)
 - [Exercises](#)
 - [Projects](#)
- [Data Types and Abstraction](#)
 - [Abstract Data Types](#)
 - [Design Patterns](#)

- [Class Hierarchy](#)
- [Abstract Objects and the `__builtin__.object` Class](#)
 - [Abstract Objects](#)
 - [Abstract Methods](#)
- [Containers](#)
 - [Container `__init__`, `hookiter` and `purge` methods](#)
 - [Container Properties](#)
- [Iterators](#)
 - [Iterators and the Python `for` statement](#)
- [Visitors](#)
 - [The `isDone` Property](#)
 - [The Container Class `str` Method](#)
- [Searchable Containers](#)
- [Associations](#)
 - [Exercises](#)
 - [Projects](#)
- [Stacks, Queues, and Deques](#)
 - [Stacks](#)
 - [Array Implementation](#)
 - [Instance Attributes](#)
 - [`__init__` and `purge` Methods](#)
 - [push, pop, and `getTop` Methods](#)
 - [accept Method](#)
 - [iter Method](#)
 - [Linked-List Implementation](#)
 - [Instance Attributes](#)
 - [`__init__` and `purge` Methods](#)
 - [push, pop and `getTop` Methods](#)
 - [accept Method](#)
 - [iter Method](#)
 - [Applications](#)
 - [Evaluating Postfix Expressions](#)
 - [Implementation](#)
 - [Queues](#)
 - [Array Implementation](#)
 - [Instance Attributes](#)
 - [`__init__` and `purge` Methods](#)
 - [enqueue, dequeue and `getHead` Methods](#)
 - [Linked-List Implementation](#)

- [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [enqueue, dequeue and getHead Methods](#)
 - [Applications](#)
 - [Implementation](#)
- [Deques](#)
 - [Array Implementation](#)
 - [enqueueHead Method](#)
 - [dequeueTail and getTail Methods](#)
 - [Linked List Implementation](#)
 - [enqueueHead Method](#)
 - [dequeueTail and getTail Methods](#)
 - [Doubly-Linked and Circular Lists](#)
- [Exercises](#)
- [Projects](#)
- [Ordered Lists and Sorted Lists](#)
 - [Ordered Lists](#)
 - [Array Implementation](#)
 - [Instance Attributes](#)
 - [Creating a List and Inserting Items](#)
 - [Finding Items in a List](#)
 - [Removing Items from a List](#)
 - [Positions of Items in a List](#)
 - [Finding the Position of an Item and Accessing by Position](#)
 - [Inserting an Item at an Arbitrary Position](#)
 - [Removing Arbitrary Items by Position](#)
 - [Linked-List Implementation](#)
 - [Instance Attributes](#)
 - [Inserting and Accessing Items in a List](#)
 - [Finding Items in a List](#)
 - [Removing Items from a List](#)
 - [Positions of Items in a List](#)
 - [Finding the Position of an Item and Accessing by Position](#)
 - [Inserting an Item at an Arbitrary Position](#)
 - [Removing Arbitrary Items by Position](#)
 - [Performance Comparison: orderedListAsArray vs. ListAsLinkedList](#)
 - [Applications](#)
 - [Sorted Lists](#)

- [Array Implementation](#)
 - [Inserting Items in a Sorted List](#)
 - [Locating Items in an Array-Binary Search](#)
 - [Finding Items in a Sorted List](#)
 - [Removing Items from a List](#)
- [Linked-List Implementation](#)
 - [Inserting Items in a Sorted List](#)
 - [Other Operations on Sorted Lists](#)
- [Performance Comparison: SortedListAsArray vs. SortedListAsList](#)
- [Applications](#)
 - [Implementation](#)
 - [Analysis](#)
- [Exercises](#)
- [Projects](#)
- [Hashing, Hash Tables, and Scatter Tables](#)
 - [Hashing-The Basic Idea](#)
 - [Example](#)
 - [Keys and Hash Functions](#)
 - [Avoiding Collisions](#)
 - [Spreading Keys Evenly](#)
 - [Ease of Computation](#)
 - [Hashing Methods](#)
 - [Division Method](#)
 - [Middle Square Method](#)
 - [Multiplication Method](#)
 - [Fibonacci Hashing](#)
 - [Hash Function Implementations](#)
 - [Integer Keys](#)
 - [Floating-Point Keys](#)
 - [Character String Keys](#)
 - [Hashing Containers](#)
 - [Using Associations](#)
 - [Hash Tables](#)
 - [Separate Chaining](#)
 - [Implementation](#)
 - [__init__, __len__ and __purge__ Methods](#)
 - [Inserting and Removing Items](#)
 - [Finding an Item](#)

- [Average Case Analysis](#)
 - [Scatter Tables](#)
 - [Chained Scatter Table](#)
 - [Implementation](#)
 - [__init__, len and purge Methods](#)
 - [Inserting and Finding an Item](#)
 - [Removing Items](#)
 - [Worst-Case Running Time](#)
 - [Average Case Analysis](#)
 - [Scatter Table using Open Addressing](#)
 - [Linear Probing](#)
 - [Quadratic Probing](#)
 - [Double Hashing](#)
 - [Implementation](#)
 - [__init__, len and purge Methods](#)
 - [Inserting Items](#)
 - [Finding Items](#)
 - [Removing Items](#)
 - [Average Case Analysis](#)
 - [Applications](#)
 - [Exercises](#)
 - [Projects](#)
- [Trees](#)
 - [Basics](#)
 - [Terminology](#)
 - [More Terminology](#)
 - [Alternate Representations for Trees](#)
 - [N-ary Trees](#)
 - [Binary Trees](#)
 - [Tree Traversals](#)
 - [Preorder Traversal](#)
 - [Postorder Traversal](#)
 - [Inorder Traversal](#)
 - [Breadth-First Traversal](#)
 - [Expression Trees](#)
 - [Infix Notation](#)
 - [Prefix Notation](#)
 - [Postfix Notation](#)
 - [Implementing Trees](#)

- [Tree Traversals](#)
 - [Depth-First Traversal](#)
 - [Preorder, Inorder, and Postorder Traversals](#)
 - [Breadth-First Traversal](#)
 - [accept Method](#)
- [Tree Iterators](#)
 - [__init__ Method](#)
 - [next Method](#)
- [General Trees](#)
 - [Instance Attributes](#)
 - [__init__ and Purge Methods](#)
 - [getKey and GetSubtree Methods](#)
 - [attachSubtree and detachSubtree Methods](#)
- [N-ary Trees](#)
 - [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [getIsEmpty Method](#)
 - [getKey, AttachKey and DetachKey Methods](#)
 - [getSubtree, attachSubtree and detachSubtree Methods](#)
- [Binary Trees](#)
 - [Instance Attributes](#)
 - [__init__ method](#)
 - [purge Method](#)
 - [right and left Properties](#)
- [Binary Tree Traversals](#)
- [Comparing Trees](#)
- [Applications](#)
 - [Implementation](#)
- [Exercises](#)
- [Projects](#)
- [Search Trees](#)
 - [Basics](#)
 - [M-Way Search Trees](#)
 - [Binary Search Trees](#)
 - [Searching a Search Tree](#)
 - [Searching an M-way Tree](#)
 - [Searching a Binary Tree](#)
 - [Average Case Analysis](#)
 - [Successful Search](#)

- [Solving The Recurrence-Telescoping](#)
 - [Unsuccessful Search](#)
 - [Traversing a Search Tree](#)
- [Implementing Search Trees](#)
 - [Binary Search Trees](#)
 - [Instance Attributes](#)
 - [find Method](#)
 - [getMin Method](#)
 - [Inserting Items in a Binary Search Tree](#)
 - [insert and attachKey Methods](#)
 - [Removing Items from a Binary Search Tree](#)
 - [withdraw Method](#)
- [AVL Search Trees](#)
 - [Implementing AVL Trees](#)
 - [Instance Attributes](#)
 - [__init__ Method](#)
 - [adjustHeight and getHeight Methods and balanceFactor Property](#)
 - [Inserting Items into an AVL Tree](#)
 - [Balancing AVL Trees](#)
 - [Single Rotations](#)
 - [Double Rotations](#)
 - [Implementation](#)
 - [Removing Items from an AVL Tree](#)
- [M-Way Search Trees](#)
 - [Implementing M-Way Search Trees](#)
 - [Implementation](#)
 - [__init__ Method and m Property](#)
 - [Inorder Traversal](#)
 - [Finding Items in an M-Way Search Tree](#)
 - [Linear Search](#)
 - [Binary Search](#)
 - [Inserting Items into an M-Way Search Tree](#)
 - [Removing Items from an M-Way Search Tree](#)
- [B-Trees](#)
 - [Implementing B-Trees](#)
 - [Instance Attributes](#)
 - [__init__ and attachSubtree Methods](#)
 - [Inserting Items into a B-Tree](#)

- [Implementation](#)
 - [Running Time Analysis](#)
 - [Removing Items from a B-Tree](#)
- [Applications](#)
- [Exercises](#)
- [Projects](#)
- [Heaps and Priority Queues](#)
 - [Basics](#)
 - [Binary Heaps](#)
 - [Complete Trees](#)
 - [Complete N-ary Trees](#)
 - [Implementation](#)
 - [Instance Attributes](#)
 - [__init__ and purge Methods](#)
 - [Putting Items into a Binary Heap](#)
 - [Removing Items from a Binary Heap](#)
 - [Leftist Heaps](#)
 - [Leftist Trees](#)
 - [Implementation](#)
 - [Instance Attributes](#)
 - [Merging Leftist Heaps](#)
 - [Putting Items into a Leftist Heap](#)
 - [Removing Items from a Leftist Heap](#)
 - [Binomial Queues](#)
 - [Binomial Trees](#)
 - [Binomial Queues](#)
 - [Implementation](#)
 - [Heap-Ordered Binomial Trees](#)
 - [Adding Binomial Trees](#)
 - [Binomial Queues](#)
 - [Instance Attributes](#)
 - [addTree and removeTree](#)
 - [getMinTree and getMin Methods](#)
 - [Merging Binomial Queues](#)
 - [Putting Items into a Binomial Queue](#)
 - [Removing an Item from a Binomial Queue](#)
 - [Applications](#)
 - [Discrete Event Simulation](#)
 - [Implementation](#)

- [Exercises](#)
- [Projects](#)
- [Sets, Multisets, and Partitions](#)
 - [Basics](#)
 - [Implementing Sets](#)
 - [Array and Bit-Vector Sets](#)
 - [Basic Operations](#)
 - [Union, Intersection, and Difference](#)
 - [Comparing Sets](#)
 - [Bit-Vector Sets](#)
 - [Basic Operations](#)
 - [Union, Intersection, and Difference](#)
 - [Multisets](#)
 - [Array Implementation](#)
 - [Basic Operations](#)
 - [Union, Intersection, and Difference](#)
 - [Linked-List Implementation](#)
 - [Union](#)
 - [Intersection](#)
 - [Partitions](#)
 - [Representing Partitions](#)
 - [Implementing a Partition using a Forest](#)
 - [Implementation](#)
 - [init Method](#)
 - [find and join Methods](#)
 - [Collapsing Find](#)
 - [Union by Size](#)
 - [Union by Height or Rank](#)
 - [Applications](#)
 - [Exercises](#)
 - [Projects](#)
- [Garbage Collection and the Other Kind of Heap](#)
 - [What is Garbage?](#)
 - [Reduce, Reuse, Recycle](#)
 - [Reduce](#)
 - [Reuse](#)
 - [Recycle](#)
 - [Helping the Garbage Collector](#)
 - [Reference Counting Garbage Collection](#)

- [When Objects Refer to Other Objects](#)
 - [Why Reference Counting Does Not Work](#)
- [Mark-and-Sweep Garbage Collection](#)
 - [The Fragmentation Problem](#)
- [Stop-and-Copy Garbage Collection](#)
 - [The Copy Algorithm](#)
- [Mark-and-Compact Garbage Collection](#)
 - [Handles](#)
- [Exercises](#)
- [Projects](#)
- [Algorithmic Patterns and Problem Solvers](#)
 - [Brute-Force and Greedy Algorithms](#)
 - [Example-Counting Change](#)
 - [Brute-Force Algorithm](#)
 - [Greedy Algorithm](#)
 - [Example-0/1 Knapsack Problem](#)
 - [Backtracking Algorithms](#)
 - [Example-Balancing Scales](#)
 - [Representing the Solution Space](#)
 - [Abstract Backtracking Solvers](#)
 - [Depth-First Solver](#)
 - [Breadth-First Solver](#)
 - [Branch-and-Bound Solvers](#)
 - [Depth-First, Branch-and-Bound Solver](#)
 - [Example-0/1 Knapsack Problem Again](#)
 - [Top-Down Algorithms: Divide-and-Conquer](#)
 - [Example-Binary Search](#)
 - [Example-Computing Fibonacci Numbers](#)
 - [Example-Merge Sorting](#)
 - [Running Time of Divide-and-Conquer Algorithms](#)
 - [Case 1 \(\$a > b^k\$ \)](#)
 - [Case 2 \(\$a = b^k\$ \)](#)
 - [Case 3 \(\$a < b^k\$ \)](#)
 - [Summary](#)
 - [Example-Matrix Multiplication](#)
 - [Bottom-Up Algorithms: Dynamic Programming](#)
 - [Example-Generalized Fibonacci Numbers](#)

- [Example-Computing Binomial Coefficients](#)
 - [Application: Typesetting Problem](#)
 - [Example](#)
 - [Implementation](#)
- [Randomized Algorithms](#)
 - [Generating Random Numbers](#)
 - [The Minimal Standard Random Number Generator](#)
 - [Implementation](#)
 - [Random Variables](#)
 - [A Simple Random Variable](#)
 - [Uniformly Distributed Random Variables](#)
 - [Exponentially Distributed Random Variables](#)
 - [Monte Carlo Methods](#)
 - [Example-Computing \$\pi\$](#)
 - [Simulated Annealing](#)
 - [Example-Balancing Scales](#)
- [Exercises](#)
- [Projects](#)
- [Sorting Algorithms and Sorters](#)
 - [Basics](#)
 - [Sorting and Sorters](#)
 - [Abstract Sorters](#)
 - [Sorter Class Hierarchy](#)
 - [Insertion Sorting](#)
 - [Straight Insertion Sort](#)
 - [Implementation](#)
 - [Average Running Time](#)
 - [Binary Insertion Sort](#)
 - [Exchange Sorting](#)
 - [Bubble Sort](#)
 - [Quicksort](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
 - [Worst-Case Running Time](#)
 - [Best-Case Running Time](#)
 - [Average Running Time](#)
 - [Selecting the Pivot](#)
 - [Selection Sorting](#)
 - [Straight Selection Sorting](#)

- [Implementation](#)
- [Sorting with a Heap](#)
 - [Implementation](#)
- [Building the Heap](#)
 - [Running Time Analysis](#)
 - [The Sorting Phase](#)
- [Merge Sorting](#)
 - [Implementation](#)
 - [Merging](#)
 - [Two-Way Merge Sorting](#)
 - [Running Time Analysis](#)
- [A Lower Bound on Sorting](#)
- [Distribution Sorting](#)
 - [Bucket Sort](#)
 - [Implementation](#)
 - [Radix Sort](#)
 - [Implementation](#)
- [Performance Data](#)
- [Exercises](#)
- [Projects](#)
- [Graphs and Graph Algorithms](#)
 - [Basics](#)
 - [Directed Graphs](#)
 - [Terminology](#)
 - [More Terminology](#)
 - [Directed Acyclic Graphs](#)
 - [Undirected Graphs](#)
 - [Terminology](#)
 - [Labeled Graphs](#)
 - [Representing Graphs](#)
 - [Adjacency Matrices](#)
 - [Sparse vs. Dense Graphs](#)
 - [Adjacency Lists](#)
 - [Implementing Graphs](#)
 - [Vertices](#)
 - [Iterators](#)
 - [Edges](#)
 - [Graphs and Digraphs](#)
 - [Abstract Graphs](#)

- [Properties](#)
 - [Accessors and Mutators](#)
 - [Graph Traversals](#)
 - [Directed Graphs](#)
 - [Implementing Undirected Graphs](#)
 - [Using Adjacency Matrices](#)
 - [Using Adjacency Lists](#)
 - [Comparison of Graph Representations](#)
 - [Space Comparison](#)
 - [Time Comparison](#)
- [Graph Traversals](#)
 - [Depth-First Traversal](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
 - [Breadth-First Traversal](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
 - [Topological Sort](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
 - [Graph Traversal Applications:](#)
[Testing for Cycles and Connectedness](#)
 - [Connectedness of an Undirected Graph](#)
 - [Connectedness of a Directed Graph](#)
 - [Testing Strong Connectedness](#)
 - [Testing for Cycles in a Directed Graph](#)
- [Shortest-Path Algorithms](#)
 - [Single-Source Shortest Path](#)
 - [Dijkstra's Algorithm](#)
 - [Data Structures for Dijkstra's Algorithm](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
 - [All-Pairs Source Shortest Path](#)
 - [Floyd's Algorithm](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
- [Minimum-Cost Spanning Trees](#)
 - [Constructing Spanning Trees](#)
 - [Minimum-Cost Spanning Trees](#)

- [Prim's Algorithm](#)
 - [Implementation](#)
- [Kruskal's Algorithm](#)
 - [Implementation](#)
 - [Running Time Analysis](#)
- [Application: Critical Path Analysis](#)
 - [Implementation](#)
- [Exercises](#)
- [Projects](#)
- [Python and Object-Oriented Programming](#)
 - [Objects and Types](#)
 - [Names](#)
 - [The Object Named None](#)
 - [Scopes and Namespaces](#)
 - [Parameter Passing](#)
 - [Classes](#)
 - [Instances, Instance Attributes and Methods](#)
 - [Example-Complex Numbers](#)
 - [`__init__` Method](#)
 - [Properties, Accessors and Mutators](#)
 - [Operator Overloading](#)
 - [Static Methods](#)
 - [Nested Classes](#)
 - [Inheritance and Polymorphism](#)
 - [Derivation and Inheritance](#)
 - [Polymorphism](#)
 - [Example-Graphical Objects](#)
 - [Method Resolution](#)
 - [Abstract Methods](#)
 - [Algorithmic Abstraction](#)
 - [Multiple Inheritance](#)
 - [Exceptions](#)
- [Class Hierarchy Diagrams](#)
- [Character Codes](#)
- [References](#)
- [Index](#)

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno

Copyright Notice

Copyright © 2003 by Bruno R. Preiss.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

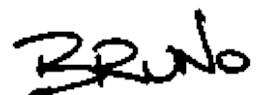


Bruno R. Preiss-Signature Page

Bruno R. Preiss,
B.A.Sc., M.A.Sc., Ph.D., P.Eng.

Email: brpreiss@brpreiss.com
 URL: <http://www.brpreiss.com>

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.



Provides the [`abstractmethod`](#) descriptor.

Modules

[inspect](#)

[sys](#)

[types](#)

Classes

[__builtin__.object](#)
[abstractmethod](#)

class `abstractmethod`([__builtin__.object](#))

Descriptor for an abstract method.

Methods defined here:

[__get__\(self, obj, type\)](#)
([abstractmethod](#), [object](#), [type](#)) -> [abstractmethod.method](#)

Returns an [abstractmethod.method](#) that represents
the binding of this abstract method to the given [obj](#).

[__init__\(self, func\)](#)
([abstractmethod](#), [function](#)) -> None

Constructor.

Static methods defined here:

[main\(*argv\)](#)
[abstractmethod](#) test program.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'abstractmethod' object>
list of weak references to the [object](#) (if defined)

method = <class 'opus7.abstractmethod.method'>
Abstract method.

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:00:48 \$'
__version__ = '\$Revision: 1.19 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Algorithms](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Algorithms](#)

class Algorithms([__builtin__.object](#))

Contains a bunch of algorithms.

Static methods defined here:

DijkstrasAlgorithm(g, s)

(Digraph, int) -> DigraphAsLists
Dijkstra's algorithm to solve the single-source, shortest path problem for the given edge-weighted, directed graph.

FloydsAlgorithm(g)

(Digraph) -> DigraphAsMatrix
Floyd's algorithm to solve the all-pairs, shortest for the given edge-weighted, directed graph.

KruskalsAlgorithm(g)

(Graph) -> GraphAsLists
Kruskal's algorithm to find a minimum-cost spanning for the given edge-weighted, undirected graph.

PrimsAlgorithm(g, s)

(Graph, int) -> GraphAsLists

Prim's algorithm to find a minimum-cost spanning tree for the given edge-weighted, undirected graph.

breadthFirstTraversal(tree)

(Tree) -> None

Does a breadth-first traversal of a tree and prints

calculator(input, output)

(File, File) -> None

A very simple reverse-Polish calculator.

criticalPathAnalysis(g)

(Digraph) -> DigraphAsLists

Computes the critical path in an event-node graph.

equivalenceClasses(input, output)

(File, File) -> None

Computes equivalence classes using a partition.

First reads an integer from the input stream that specifies the size of the universal set.

Then reads pairs of integers from the input stream that denote equivalent items in the universal set.

Prints the partition on end-of-file.

translate(dictionary, input, output)

(File, File, File) -> None

Reads all the word pairs from the dictionary file

and then reads words from the input file,

translates the words (if possible),

and writes them to the output file.

wordCounter(input, output)

(File, File) -> None

Counts the number of occurrences of each word in the input file.

Data and other attributes defined here:

Counter = <class 'opus7.algorithms.Counter'>

A counter.

EarliestTimeVisitor = <class 'opus7.algorithms.EarliestTimeVisit

Used by the critical path analysis program

to compute the earliest completion time for each event.

Entry = <class 'opus7.algorithms.Entry'>

Data structure used in Dijkstra's and Prim's algorithm

LatestTimeVisitor = <class 'opus7.algorithms.LatestTimeVisitor'>
Used by the critical path analysis program
to compute the latest completion time for each event

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Algorithms' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/27 00:43:09 \$'
__version__ = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application1](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application1](#)

class Application1([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 1. (calculator)

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application1'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.7 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application11](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application11](#)

class Application11([__builtin__.object](#))

Static methods defined here:

main(*argv)
[Application11](#) test program.

weightedDigraphTest(g)
Weighted digraph test program.

weightedGraphTest(g)
Weighted graph test program.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application11'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the Application12 class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application12](#)

class Application12([__builtin__.object](#))

Static methods defined here:

main(*argv)
[Application12](#) test program.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application12'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.8 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application2](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application2](#)

class Application2([__builtin__.object](#))

Static methods defined here:

buildTree(lo, hi)
(char, char) -> NaryTree
Builds an N-
ary tree that contains keys in the xrange from lo to hi.

main(*argv)
Application program number 2.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application2' objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.7 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application3](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application3](#)

class Application3([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 3.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application3'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application4](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application4](#)

class Application4([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 4.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application4'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application5](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application5](#)

class Application5([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 5. (word counter)

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application5'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
 date = '\$Date: 2003/09/06 16:35:15 \$'
 version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application6](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application6](#)

class Application6([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 6. (expression tree)

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application6'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application7](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application7](#)

class Application7([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 7. (translator)

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application7'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application8](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application8](#)

class Application8([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 8.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application8'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Application9](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Application9](#)

class Application9([__builtin__.object](#))

Static methods defined here:

main(*argv)
Application program number 9. (equivalence classes)

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Application9'
objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
 date = '\$Date: 2003/09/06 16:35:15 \$'
 version = '\$Revision: 1.6 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Array](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Array](#)

class Array([__builtin__.object](#))

[Array](#) class.

Methods defined here:

__copy__(self)
([Array](#)) -> [Array](#)

Returns a shallow copy of this array.

__getitem__(self, index)
([Array](#), int) -> Object

Returns the item in this array at the given index.

__init__(self, length=0, baseIndex=0)
([Array](#), int) -> None

Constructs an array of the given length.

__len__(self)
([Array](#)) -> int

Returns the length of this array.

__setitem__(self, index, value)

__setitem__(self, index, value)
Sets the item in this array at the given index to the given value.

__str__(self)
`(Array, int, Object) -> None`
Returns a string representation of this array.

getBaseIndex(self)

getData(self)

getOffset(self, index)
`(Array, int) -> int`
Returns the offset for the given index.

setBaseIndex(self, baseIndex)

setLength(self, value)

Static methods defined here:

main(*argv)
`Array test program.`

Properties defined here:

baseIndex

get lambda self
set lambda self, value

data

get lambda self

length

get lambda self
set lambda self, value

Data and other attributes defined here:

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Array' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:00:48 \$'
__version__ = '\$Revision: 1.31 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Association](#) class.

Modules

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Association](#)

class Association([opus7.object.Object](#))

Represents a (key, value) pair using a tuple.

Method resolution order:

[Association](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__hash__\(self\)](#)
([Association](#)) -> int
Hashes the key of this association.

[__init__\(self, *args\)](#)
([Association](#), [Object](#) [, [Object](#)]) -> None
Constructs an association with the given key and op-

[__str__\(self\)](#)
([Association](#)) -> string

Returns a string representation of this association

getKey(self)

[\(Association\)](#) -> [Object](#)

Returns the key of this association.

getValue(self)

[\(Association\)](#) -> [Object](#)

Returns the value of this association.

Static methods defined here:

__new__ = new(*args, **kwargs)

[\(Metaclass, ...\)](#) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)

[Association](#) test program.

Properties defined here:

key

get lambda self

value

get lambda self

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

[\(Object, Object\)](#) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.MetaClass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

opus7.avlTree (version 1.17, \$Date: 2003/09/23
23:01:18 \$)

[index](#)
[opus7/avlTree.py](#)

Provides the [AVLTree](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.binarySearchTree.BinarySearchTree](#)([opus7.binaryTree.Bir](#)
[opus7.searchTree.SearchTree](#))
[AVLTree](#)

class **AVLTree**([opus7.binarySearchTree.BinarySearchTree](#))

AVL tree class.

Method resolution order:

[AVLTree](#)
[opus7.binarySearchTree.BinarySearchTree](#)
[opus7.binaryTree.BinaryTree](#)
[opus7.searchTree.SearchTree](#)
[opus7.tree.Tree](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self\)](#)
([AVLTree](#)) -> None

Constructs an empty AVL tree.

adjustHeight(self)

([AVLTree](#)) -> None

Adjusts the height of this AVL tree.

attachKey(self, obj)

([AVLTree](#), Object) -> None

Attaches the given object to this AVL tree node.

balance(self)

([AVLTree](#)) -> None

Balances this AVL tree node.

detachKey(self)

([AVLTree](#)) -> Object

Detaches and returns the key in this AVL tree node.

doLLRotation(self)

([AVLTree](#)) -> None

Does an LL rotation at this AVL tree node.

doLRRotation(self)

([AVLTree](#)) -> None

Does an LR rotation at this AVL tree node.

doRLRotation(self)

([AVLTree](#)) -> None

Does an RL rotation at this AVL tree node.

doRRRotation(self)

([AVLTree](#)) -> None

Does an RR rotation at this AVL tree node.

getBalanceFactor(self)

([AVLTree](#)) -> int

Returns the balance factor of this AVL tree node.

getHeight(self)

([AVLTree](#)) -> int

Returns the height of this AVL tree.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[AVLTree](#) test program.

Properties defined here:

balanceFactor
get lambda self

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.binarySearchTree.BinarySearchTree](#)

__contains__(self, obj)
([BinarySearchTree](#), Object) -> bool
Returns true if the given object is in this binary search tree.

find(self, obj)
([BinarySearchTree](#), Object) -> Object
Returns the object in this binary search tree that matches the given object.

getMax(self)
([BinarySearchTree](#)) -> Object
Returns the object in this binary search tree with the maximum value.

getMin(self)
([BinarySearchTree](#)) -> Object
Returns the object in this binary search tree with the minimum value.

insert(self, obj)
([BinarySearchTree](#), Object) -> None
Inserts the given object into this binary search tree.

withdraw(self, obj)
 ([BinarySearchTree](#), Object) -> None
 Withdraws the given object from this binary search tree.

Methods inherited from [opus7.binaryTree.BinaryTree](#):

attachLeft(self, t)
 (BinaryTree, Binary Tree) -> None
 Attaches the given binary tree as the left subtree of this binary tree.

attachRight(self, t)
 (BinaryTree, Binary Tree) -> None
 Attaches the given binary tree as the right subtree of this binary tree.

depthFirstGenerator(self, mode)
 (BinaryTree) -> generator
 Yields the keys in this tree in depth-first traversal mode.

depthFirstTraversal(self, visitor)
 (BinaryTree, PrePostVisitor) -> None
 Makes the given visitor do a depth-first traversal of this binary tree.

detachLeft(self)
 (BinaryTree) -> BinaryTree
 Detaches and returns the left subtree of this binary tree.

detachRight(self)
 (BinaryTree) -> BinaryTree
 Detaches and returns the right subtree of this binary tree.

getDegree(self)
 (BinaryTree) -> int
 Returns the degree of this binary tree node.

getIsEmpty(self)
 (BinaryTree) -> bool
 Returns true if this binary tree is empty.

getIsLeaf(self)
 (BinaryTree) -> bool
 Returns true if this binary tree node is a leaf.

getKey(self)
 (BinaryTree) -> Object

Returns the key in this binary tree node.

getLeft(self)

(BinaryTree) -> BinaryTree

Returns the left subtree of this binary tree.

getRight(self)

(BinaryTree) -> BinaryTree

Returns the right subtree of this binary tree.

getSubtree(self, i)

(BinaryTree, int) -> Object

Returns the specified subtree of this binary tree.

purge(self)

(BinaryTree) -> None

Purges this binary tree.

Properties inherited from [opus7.binaryTree.BinaryTree](#):

left

get lambda self

right

get lambda self

Static methods inherited from [opus7.searchTree.SearchTree](#):

test(tree)

SearchTree test program.

Properties inherited from [opus7.searchTree.SearchTree](#):

max

get lambda self

min

get lambda self

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)
(Tree) -> Tree.Iterator
Returns an interator for this tree.

accept(self, visitor)
(Tree) -> Visitor
Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)
(Tree) -> generator
Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)
(Tree, Visitor) -> None
Makes the given visitor do a breadth-first traversal.

getCount(self)
(Tree) -> int
Returns the number of nodes in this tree.

Properties inherited from [opus7.tree.Tree](#):

degree
get lambda self

height
get lambda self

isLeaf
get lambda self

key
get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 23:01:18 \$'
__version__ = '\$Revision: 1.17 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BTree](#) class.

Modules

[sys](#)

Classes

[opus7.mWayTree.MWayTree](#)([opus7.searchTree.SearchTree](#))
[BTree](#)

class **BTree**([opus7.mWayTree.MWayTree](#))

B-tree class.

Method resolution order:

[BTree](#)
[opus7.mWayTree.MWayTree](#)
[opus7.searchTree.SearchTree](#)
[opus7.tree.Tree](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin_object](#)

Methods defined here:

__init__(self, m)
([BTree](#), int) -> None
Constructs an empty B-tree with the given value of |

attachLeftHalfOf(self, btree)
`(BTree, Btree) -> None`
Attaches the left half of this B-tree to the given B-tree.

attachRightHalfOf(self, btree)
`(BTree, Btree) -> None`
Attaches the right half of this B-tree to the given B-tree.

attachSubtree(self, i, t)
`(BTree, int, Btree):`
Attaches the given B-tree as the specified subtree at index `i`.

insert(self, obj)
`(Btree, Object) -> None`
Inserts the given object into this B-tree.

insertPair(self, index, obj, child)
`(BTree, int, Object, BTree) -> (Object, BTree)`
Inserts the given Object at the specified index in the key array of this B-tree node and returns any leftover object.

insertUp(self, obj, child)
`(BTree, Object, Btree) -> None`
Inserts the given (Object, Btree) pair into this B-tree.

withdraw(self, obj)
`(Btree, Object) -> Object`
Withdraws the given object from this B-tree.

Static methods defined here:

__new__ = new(*args, **kwargs)
`(Metaclass, ...) -> object`
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
`BTree test program.`

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.mWayTree.MWayTree](#):

__contains__(self, obj)

([MWayTree](#), Object) -> bool

Returns true if the given object is in this M-way tree.

__iter__(self)

([MWayTree](#)) -> [MWayTree.Iterator](#)

Returns an iterator for this M-way tree.

breadthFirstGenerator(self)

(BinaryTree) -> generator

Yields the keys in this tree in breadth-first traversal order.

breadthFirstTraversal(self, visitor)

([MWayTree](#), Visitor) -> None

Makes the given visitor do a breadth-first traversal of this M-way tree.

depthFirstGenerator(self, mode)

(BinaryTree) -> generator

Yields the keys in this tree in depth-first traversal order.

depthFirstTraversal(self, visitor)

([MWayTree](#), PrePostVisitor) -> None

Makes the given visitor do a depth-first traversal of this M-way tree.

find(self, obj)

([MWayTree](#), Object):

Returns the object in this M-way tree that matches the given object.

findIndex(self, obj)

([MWayTree](#), Object) -> int

Returns the position of the specified object in the array of keys contained in this M-way tree node.

Uses a binary search.

getCount(self)

([MWayTree](#)) -> int
Returns the number of keys in this M-way tree node.

getDegree(self)
([MWayTree](#)) -> int
Returns the degree of this M-way tree node.

getIsEmpty(self)
([MWayTree](#)) -> bool
Returns true if this M-way tree is empty.

getIsFull(self)
([MWayTree](#)) -> bool
Returns true if this M-way tree is full.

getIsLeaf(self)
([MWayTree](#)) -> bool
Returns true if this M-way tree is a leaf.

getKey(self, *args)
([MWayTree](#), ...) -> Object
Returns the specified key of this M-way tree node.

getM(self)
([MWayTree](#)) -> int
Returns the value of M for this M-way tree.

getMax(self)
([MWayTree](#)) -> Object
Returns the object in this M-way tree with the largest key.

getMin(self)
([MWayTree](#)) -> Object
Returns the object in this M-way tree with the smallest key.

getSubtree(self, i)
([MWayTree](#), int) -> [MWayTree](#)
Returns the specified subtree of this M-way tree node.

purge(self)
([MWayTree](#)) -> None
Purges this M-way tree.

Properties inherited from [opus7.mWayTree.MWayTree](#):

m

get lambda self

Data and other attributes inherited from [opus7.mWayTree.MWayT](#)

Iterator = <class 'opus7.mWayTree.Iterator'>
Enumerates the objects in an M-way tree.

Static methods inherited from [opus7.searchTree.SearchTree](#):

test(tree)
SearchTree test program.

Properties inherited from [opus7.searchTree.SearchTree](#):

max
get lambda self

min
get lambda self

Methods inherited from [opus7.tree.Tree](#):

accept(self, visitor)
(Tree) -> Visitor
Makes the given visitor visit the nodes of this tree.

getHeight(self)
(Tree) -> int
Returns the height of this tree.

Properties inherited from [opus7.tree.Tree](#):

degree
get lambda self

height
get lambda self

isLeaf

get lambda self

key

get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 23:12:00 \$'
__version__ = '\$Revision: 1.19 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BinaryHeap](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.priorityQueue.PriorityQueue](#)([opus7.container.Container](#))
[BinaryHeap](#)

class **BinaryHeap**([opus7.priorityQueue.PriorityQueue](#))

Binary heap class implemented using an array.

Method resolution order:

[BinaryHeap](#)
[opus7.priorityQueue.PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, length=0)
([BinaryHeap](#) [,int]) -> None
Constructs a binary heap of the given length.

__iter__(self)
([BinaryHeap](#)) -> [BinaryHeap](#).Iterator.
Returns an interator for this binary heap.

accept(self, visitor)
 ([BinaryHeap](#), Visitor) -> None
 Makes the given visitor visit all the objects in this binary heap.

dequeueMin(self)
 ([BinaryHeap](#)) -> Object
 Dequeues and returns the object in this binary heap with the smallest value.

enqueue(self, obj)
 ([BinaryHeap](#), Object) -> None
 Enqueues the given object in this binary heap.

getIsFull(self)
 ([BinaryHeap](#)) -> bool
 Returns true if this binary heap is full.

getMin(self)
 ([BinaryHeap](#)) -> Object
 Returns the object in this binary heap with the smallest value.

purge(self)
 ([BinaryHeap](#)) -> None
 Purges this binary heap.

Static methods defined here:

__new__ = new(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
 [BinaryHeap](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.binaryHeap.Iterator'>
 Enumerates the elements of a binary heap.

__abstractmethods__ = []

Static methods inherited from [opus7.priorityQueue.PriorityQueue](#):

test(pqueue)

[PriorityQueue](#) test program.

Properties inherited from [opus7.priorityQueue.PriorityQueue](#):

min

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.19 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BinaryInsertionSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[BinaryInsertionSorter](#)

class BinaryInsertionSorter([opus7.sorter.Sorter](#))

Binary insertion sorter.

Method resolution order:

[BinaryInsertionSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

[**__init__\(self\)**](#)
([BinaryInsertionSorter](#)) -> None
Constructor.

Static methods defined here:

[**__new__** = new\(*args, **kwargs\)](#)

`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)

[BinaryInsertionSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)

[\(Sorter, Array\)](#) -> None
Sorts the given array.

swap(self, i, j)

[\(Sorter, int, int\)](#) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

[\(Object, Object\)](#) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>

Metaclass of the `Object` class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BinarySearchTree](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.binaryTree.BinaryTree\(opus7.tree.Tree\)](#)
[BinarySearchTree\(opus7.binaryTree.BinaryTree, opus7.searchTree.SearchTree, opus7.searchableContainer.SearchableContainer\)](#)
[SearchTree\(opus7.tree.Tree, opus7.searchableContainer.SearchableContainer\)](#)
[BinarySearchTree\(opus7.binaryTree.BinaryTree, opus7.searchableContainer.SearchableContainer\)](#)

class **BinarySearchTree**([opus7.binaryTree.BinaryTree](#), [opus7.searchableContainer.SearchableContainer](#))

Binary search tree class.

Method resolution order:

[BinarySearchTree](#)
[opus7.binaryTree.BinaryTree](#)
[opus7.searchTree.SearchTree](#)
[opus7.tree.Tree](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

```
__contains__(self, obj)
    (BinarySearchTree, Object) -> bool
    Returns true if the given object is in this binary search tree.

__init__(self)
    (BinarySearchTree) -> None
    Constructs an empty binary search tree.

attachKey(self, obj)
    (BinarySearchTree, Object) -> None
    Attaches the given object to the root node of this binary search tree.

balance(self)
    (BinarySearchTree) -> None
    Balances this binary search tree.

find(self, obj)
    (BinarySearchTree, Object) -> Object
    Returns the object in this binary search tree that matches the given object.

getMax(self)
    (BinarySearchTree) -> Object
    Returns the object in this binary search tree with the maximum value.

getMin(self)
    (BinarySearchTree) -> Object
    Returns the object in this binary search tree with the minimum value.

insert(self, obj)
    (BinarySearchTree, Object) -> None
    Inserts the given object into this binary search tree.

withdraw(self, obj)
    (BinarySearchTree, Object) -> None
    Withdraws the given object from this binary search tree.
```

Static methods defined here:

```
__new__ = new(*args, **kwargs)
    (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a TypeError exception if the class is abstract.
This method is inserted as the method __new__.
```

in classes instances derived from Metaclass.

```
main(*argv)
    BinarySearchTree test program.
```

Data and other attributes defined here:

```
__abstractmethods__ = []
```

Methods inherited from [opus7.binaryTree.BinaryTree](#):

```
attachLeft(self, t)
    (BinaryTree, Binary Tree) -> None
    Attaches the given binary tree as the left subtree

attachRight(self, t)
    (BinaryTree, Binary Tree) -> None
    Attaches the given binary tree as the right subtree

depthFirstGenerator(self, mode)
    (BinaryTree) -> generator
    Yields the keys in this tree in depth-first traversal

depthFirstTraversal(self, visitor)
    (BinaryTree, PrePostVisitor) -> None
    Makes the given visitor do a depth-first traversal

detachKey(self)
    (BinaryTree) -> None
    Detaches and returns the key in this binary tree node

detachLeft(self)
    (BinaryTree) -> BinaryTree
    Detaches and returns the left subtree of this binary tree

detachRight(self)
    (BinaryTree) -> BinaryTree
    Detaches and returns the right subtree of this binary tree

getDegree(self)
    (BinaryTree) -> int
    Returns the degree of this binary tree node.
```

getIsEmpty(self)
`(BinaryTree) -> bool`
Returns true if this binary tree is empty.

getIsLeaf(self)
`(BinaryTree) -> bool`
Returns true if this binary tree node is a leaf.

getKey(self)
`(BinaryTree) -> Object`
Returns the key in this binary tree node.

getLeft(self)
`(BinaryTree) -> BinaryTree`
Returns the left subtree of this binary tree.

getRight(self)
`(BinaryTree) -> BinaryTree`
Returns the right subtree of this binary tree.

getSubtree(self, i)
`(BinaryTree, int) -> Object`
Returns the specified subtree of this binary tree.

purge(self)
`(BinaryTree) -> None`
Purges this binary tree.

Properties inherited from [opus7.binaryTree.BinaryTree](#):

left
get lambda self

right
get lambda self

Static methods inherited from [opus7.searchTree.SearchTree](#):

test(tree)
`SearchTree test program.`

Properties inherited from [opus7.searchTree.SearchTree](#):

max

get lambda self

min

get lambda self

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)

(Tree) -> Tree.Iterator

Returns an interator for this tree.

accept(self, visitor)

(Tree) -> Visitor

Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)

(Tree) -> generator

Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)

(Tree, Visitor) -> None

Makes the given visitor do a breadth-first traversal.

getCount(self)

(Tree) -> int

Returns the number of nodes in this tree.

getHeight(self)

(Tree) -> int

Returns the height of this tree.

Properties inherited from [opus7.tree.Tree](#):

degree

get lambda self

height

get lambda self

isLeaf

get lambda self

key

get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 23:01:18 \$'
__version__ = '\$Revision: 1.18 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BinaryTree](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.tree.Tree](#)([opus7.container.Container](#))
[BinaryTree](#)

class **BinaryTree**([opus7.tree.Tree](#))

Binary tree class.

Method resolution order:

[BinaryTree](#)
[opus7.tree.Tree](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

__init__(self, *args)
([BinaryTree](#) [, Object [, [BinaryTree](#), [BinaryTree](#)]])
Constructs a binary tree.

attachKey(self, obj)
([BinaryTree](#), Object) -> None
Makes the given object the key of this binary tree |

attachLeft(self, t)
`(BinaryTree, Binary Tree) -> None`
Attaches the given binary tree as the left subtree.

attachRight(self, t)
`(BinaryTree, Binary Tree) -> None`
Attaches the given binary tree as the right subtree.

depthFirstGenerator(self, mode)
`(BinaryTree) -> generator`
Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)
`(BinaryTree, PrePostVisitor) -> None`
Makes the given visitor do a depth-first traversal.

detachKey(self)
`(BinaryTree) -> None`
Detaches and returns the key in this binary tree node.

detachLeft(self)
`(BinaryTree) -> BinaryTree`
Detaches and returns the left subtree of this binary tree node.

detachRight(self)
`(BinaryTree) -> BinaryTree`
Detaches and returns the right subtree of this binary tree node.

getDegree(self)
`(BinaryTree) -> int`
Returns the degree of this binary tree node.

getIsEmpty(self)
`(BinaryTree) -> bool`
Returns true if this binary tree is empty.

getIsLeaf(self)
`(BinaryTree) -> bool`
Returns true if this binary tree node is a leaf.

getKey(self)
`(BinaryTree) -> Object`
Returns the key in this binary tree node.

getLeft(self)

getLeft(self)
([BinaryTree](#)) -> [BinaryTree](#)
Returns the left subtree of this binary tree.

getRight(self)
([BinaryTree](#)) -> [BinaryTree](#)
Returns the right subtree of this binary tree.

getSubtree(self, i)
([BinaryTree](#), int) -> Object
Returns the specified subtree of this binary tree.

purge(self)
([BinaryTree](#)) -> None
Purges this binary tree.

Static methods defined here:

__new__(self, *args, **kwargs)
([MetaClass](#), ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `MetaClass`.

main(*argv)
[BinaryTree](#) test program.

Properties defined here:

left
get lambda self

right
get lambda self

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)
`(Tree) -> Tree.Iterator`
Returns an interator for this tree.

accept(self, visitor)
`(Tree) -> Visitor`
Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)
`(Tree) -> generator`
Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)
`(Tree, Visitor) -> None`
Makes the given visitor do a breadth-first traversal.

getCount(self)
`(Tree) -> int`
Returns the number of nodes in this tree.

getHeight(self)
`(Tree) -> int`
Returns the height of this tree.

Static methods inherited from [opus7.tree.Tree](#):

test(tree)
`Tree` test program.

Properties inherited from [opus7.tree.Tree](#):

degree
get lambda self

height
get lambda self

isLeaf
get lambda self

key
get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.26 \$'

Author

Bruno R. Preiss, P.Eng.

Credits



Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BinomialQueue](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#)([opus7.p
BinomialQueue](#))

class **BinomialQueue**([opus7.mergeablePriorityQueue.MergeablePriorityQueue](#))

Mergeable priority queue implemented as a binomial queue

Method resolution order:

[BinomialQueue](#)
[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#)
[opus7.priorityQueue.PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__init__(self, *args)
([BinomialQueue](#), ...) -> None
Constructor.

__iter__(self)
([BinomialQueue](#)) -> iterator

Returns an iterator that enumerates the elements of

__str__(self)
`(BinomialQueue) -> str`
Returns a string representation of this binomial queue.

accept(self, visitor)
`(BinomialQueue, Visitor) -> None`
Makes the given visitor visit the elements of this binomial queue.

addTree(self, tree)
`(BinomialQueue, BinomialQueue.BinomialTree) -> None`
Adds the given binomial tree to this binomial queue.

dequeueMin(self)
`(BinomialQueue) -> Object`
Dequeues and returns the object in this binomial queue with the smallest value.

enqueue(self, obj)
`(BinomialQueue, Object) -> None`
Enqueues the given object in this binomial queue.

getMin(self)
`(BinomialQueue) -> Object`
Returns the object in this binomial queue with the smallest value.

getMinTree(self)
`(BinomialQueue) -> BinomialQueue.BinomialTree`
Returns the binomial tree in this binomial queue with the smallest root.

merge(self, queue)
`(BinomialQueue, BinomialQueue) -> None`
Merges the contents of the given binomial queue with this binomial queue.

purge(self)
`(BinomialQueue) -> None`
Purges this binomial queue.

removeTree(self, tree)
`(BinomialQueue, BinomialQueue.BinomialTree) -> None`
Removes the given binomial tree from this binomial queue.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

fullAdder(a, b, c)
(BinomialTree, BinomialTree, BinomialTree) ->
(BinomialTree, BinomialTree)
Returns the (sum, carry) of the given binomial tree.

main(*argv)
[BinomialQueue](#) test program.

Properties defined here:

minTree
get lambda self

Data and other attributes defined here:

BinomialTree = <class 'opus7.binomialQueue.BinomialTree'>
A binomial tree implemented as a general tree.

__abstractmethods__ = []

Methods inherited from [opus7.mergeablePriorityQueue.MergeablePriorityQueue](#):

test(pqueue)
[MergeablePriorityQueue](#) test program.

Properties inherited from [opus7.priorityQueue.PriorityQueue](#):

min
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.17 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.breadthFirstBranchAndBoundSolver](#)

(version 1.3, \$Date: 2003/07/23 17:54:18 \$)

[opus7/breadthFirst](#)

Provides the BreadthFirstBranchAndBoundSolver class.

Classes

[opus7.solver.Solver](#)([opus7.object.Object](#))
[BreadthFirstBranchAndBoundSolver](#)

class BreadthFirstBranchAndBoundSolver([opus7.solver.Solver](#))

Breadth-first brand-and-bound solver.

Method resolution order:

[BreadthFirstBranchAndBoundSolver](#)
[opus7.solver.Solver](#)
[opus7.object.Object](#)
[_builtin__object](#)

Methods defined here:

__init__(self)
([BreadthFirstBranchAndBoundSolver](#)) -> None
Constructor.

search(self, initial)
([BreadthFirstBranchAndBoundSolver](#), Solution) -> Solution
Does a breadth-first traversal of the solution space starting from the given node.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

Data and other attributes defined here:

`__abstractmethods__` = []

Methods inherited from [opus7.solver.Solver](#):

`solve(self, initial)`
`(Solver, Solution) -> Solution`
Solves a problem by searching the solution space
starting from the given node.

`updateBest(self, solution)`
`(Solver, Solution) -> None`
Records the given solution if it is complete, feasible
and has a lower objective function value than the best
solution seen so far.

Methods inherited from [opus7.object.Object](#):

`__cmp__(self, obj)`
`(Object, Object) -> int`
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

`main(*argv)`
Object test program.

Data and other attributes inherited from [opus7.object.Object](#):

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__metaclass__` = <class 'opus7.metaclass.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.3 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BreadthFirstSolver](#) class.

Classes

[opus7.solver.Solver](#)([opus7.object.Object](#))
[BreadthFirstSolver](#)

class BreadthFirstSolver([opus7.solver.Solver](#))

Breadth-first solver.

Method resolution order:

[BreadthFirstSolver](#)
[opus7.solver.Solver](#)
[opus7.object.Object](#)
[_builtin__object](#)

Methods defined here:

[__init__\(self\)](#)
([BreadthFirstSolver](#)) -> None
Constructor.

[search\(self, initial\)](#)
([BreadthFirstSolver](#), Solution) -> Solution
Does a breadth-first traversal of the solution space
starting from the given node.

Static methods defined here:

[__new__ = new\(*args, **kwargs\)](#)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

Data and other attributes defined here:

`__abstractmethods__` = []

Methods inherited from [opus7.solver.Solver](#):

`solve(self, initial)`
`(Solver, Solution) -> Solution`
Solves a problem by searching the solution space
starting from the given node.

`updateBest(self, solution)`
`(Solver, Solution) -> None`
Records the given solution if it is complete, feasible
and has a lower objective function value than the best
solution seen so far.

Methods inherited from [opus7.object.Object](#):

`__cmp__(self, obj)`
`(Object, Object) -> int`
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

`main(*argv)`
Object test program.

Data and other attributes inherited from [opus7.object.Object](#):

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__metaclass__` = <class 'opus7.metamodel.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.3 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BubbleSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[BubbleSorter](#)

class **BubbleSorter**([opus7.sorter.Sorter](#))

Bubble sorter.

Method resolution order:

[BubbleSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self)
([BubbleSorter](#)) -> None
Constructor.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
[BubbleSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)
([Sorter](#), Array) -> None
Sorts the given array.

swap(self, i, j)
([Sorter](#), int, int) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abstract methods.

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [BucketSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[BucketSorter](#)

class **BucketSorter**([opus7.sorter.Sorter](#))

Bucket sorter.

Method resolution order:

[BucketSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, m)
([BucketSorter](#), int) -> None
Constructs a bucket sorter with the given number of

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
[BucketSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)
([Sorter](#), Array) -> None
Sorts the given array.

swap(self, i, j)
([Sorter](#), int, int) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abstract methods.

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

opus7.chainedHashTable (version
1.27, \$Date: 2003/09/25 01:07:38 \$)

[index](#)

[opus7/chainedHashTable.py](#)

Provides the [ChainedHashTable](#) class.

Modules

[sys](#)

Classes

[opus7.hashTable.HashTable](#)([opus7.searchableContainer.SearchableContainer](#))
[ChainedHashTable](#)

class ChainedHashTable(opus7.hashTable.HashTable)

Hash table implemented using an array of linked lists.

Method resolution order:

[ChainedHashTable](#)
[opus7.hashTable.HashTable](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__contains__(self, obj)
([ChainedHashTable](#), Object) -> bool
Returns true if the given object is in this chained

__init__(self, length)
([ChainedHashTable](#), int) -> None

Constructs a chained hash table with the given length.

__iter__(self)

([ChainedHashTable](#)) -> [ChainedHashTable](#).Iterator

Returns an iterator for this hash table.

__len__(self)

([ChainedHashTable](#)) -> int

Returns the length of this chained hash table.

accept(self, visitor)

([ChainedHashTable](#), Visitor) -> None

Makes the given visitor visit all the objects in this hash table.

find(self, obj)

([ChainedHashTable](#), Object) -> Object

Returns the obj in this hash table
that matches the given object.

insert(self, obj)

([ChainedHashTable](#), Object) -> None

Inserts the given object into this chained hash table.

purge(self)

([ChainedHashTable](#)) -> None

Purges this chained hash table.

withdraw(self, obj)

([ChainedHashTable](#), Object) -> None

Withdraws the given object from this chained hash table.

Static methods defined here:

__new__ = new(*args, **kwargs)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.

Raises a `TypeError` exception if the class is abstract.

This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)

[ChainedHashTable](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.chainedHashTable.Iterator'>
Enumerates the._elements of a chained hash table.

__abstractmethods__ = []

Methods inherited from [opus7.hashTable.HashTable](#):

f(self, obj)
([HashTable](#), Object) -> int
Returns the hash of the given object.

g(self, x)
([HashTable](#), int) -> int
Hashes an integer using the division method of hash.

getLoadFactor(self)
([HashTable](#)) -> double
Returns the load factor of this hash table.

h(self, obj)
([HashTable](#), Object) -> int
Hashes the specified object
using the composition of the methods f and g.

Static methods inherited from [opus7.hashTable.HashTable](#):

test(hashTable)
[HashTable](#) test program.

Properties inherited from [opus7.hashTable.HashTable](#):

loadFactor
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
 Metaclass of the Object class.
 Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
 list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.27 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.chainedScatterTable](#)

(version 1.33, \$Date: 2003/09/25
01:07:38 \$)

[index](#)

[opus7/chainedScatterTable.py](#)

Provides the [ChainedScatterTable](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.hashTable.HashTable](#)([opus7.searchableContainer.SearchableContainer](#))
[ChainedScatterTable](#)

class ChainedScatterTable([opus7.hashTable.HashTable](#))

Hash table implemented as a chained scatter table using a linked list of buckets.

Method resolution order:

[ChainedScatterTable](#)
[opus7.hashTable.HashTable](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

__contains__(self, obj)
([ChainedScatterTable](#), Object) -> bool
Returns true if the given object is in this chained

__init__(self, length)

`(ChainedScatterTable, int) -> None`
Constructs a chained scatter table with the given length.

`__iter__(self)`
`(ChainedScatterTable) -> ChainedScatterTable.Iterator`
Returns an iterator for this chained scatter table.

`__len__(self)`
`(ChainedScatterTable) -> int`
Returns the length of this chained scatter table.

`accept(self, visitor)`
`(ChainedScatterTable, Visitor) -> None`
Makes the given visitor visit all the objects in this chained scatter table.

`find(self, obj)`
`(ChainedScatterTable, Object) -> Object`
Returns the object in this chained scatter table that matches the given object.

`getIsFull(self)`
`(ChainedScatterTable) -> bool`
Returns true if this chained scatter table is full.

`insert(self, obj)`
`(ChainedScatterTable, Object) -> None`
Inserts the given object into this chained scatter table.

`purge(self)`
`(ChainedScatterTable) -> None`
Purges this chained scatter table.

`withdraw(self, obj)`
`(ChainedScatterTable, Object) -> None`
Withdraws the given object from this chained scatter table.

Static methods defined here:

`__new__ = new(*args, **kwargs)`
`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`.

in classes instances derived from Metaclass.

main(*argv)
[ChainedScatterTable](#) test program.

Data and other attributes defined here:

Entry = <class 'opus7.chainedScatterTable.Entry'>
An entry in a chained scatter table.

Iterator = <class 'opus7.chainedScatterTable.Iterator'>
Enumerates the elements of a chained scatter table.

NULL = -1

__abstractmethods__ = []

Methods inherited from [opus7.hashTable.HashTable](#):

f(self, obj)
([HashTable](#), Object) -> int
Returns the hash of the given object.

g(self, x)
([HashTable](#), int) -> int
Hashes an integer using the division method of hash.

getLoadFactor(self)
([HashTable](#)) -> double
Returns the load factor of this hash table.

h(self, obj)
([HashTable](#), Object) -> int
Hashes the specified object
using the composition of the methods f and g.

Static methods inherited from [opus7.hashTable.HashTable](#):

test(hashTable)
[HashTable](#) test program.

Properties inherited from [opus7.hashTable.HashTable](#):

loadFactor

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.33 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [circle](#) class.

Modules

[sys](#)

Classes

[opus7.graphicalObject.GraphicalObject](#)([opus7.object.Object](#))
[Circle](#)

class Circle([opus7.graphicalObject.GraphicalObject](#))

A circle.

Method resolution order:

[Circle](#)
[opus7.graphicalObject.GraphicalObject](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__init__](#)(self, center, radius)

[\(Circle, Point, int\)](#) -> None

Constructs a circle with the given center and radius.

[draw](#)(self)

[\(Circle\)](#) -> None

Draws this circle.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
Circle test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.graphicalObject.GraphicalObject](#):

erase(self)
([GraphicalObject](#)) -> None
Erases this graphical object.

moveTo(self, p)
([GraphicalObject](#)) -> None
Moves the center of this graphical object to the given point.

setPenColor(self, color)

Static methods inherited from [opus7.graphicalObject.GraphicalObject](#):

test(go)
[GraphicalObject](#) test program.

Data and other attributes inherited from [opus7.graphicalObject.GraphicalObject](#):

BACKGROUND_COLOR = 0

FOREGROUND_COLOR = 1

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/28 00:06:11 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Complex](#) class.

Modules

[math](#)

[sys](#)

Classes

[__builtin__.object](#)
[Complex](#)

class Complex([__builtin__.object](#))

Sample complex number class.

Methods defined here:

[__add__\(self, c\)](#)

[\(Complex, Complex\) -> Complex](#)

Returns a complex number equal to the sum of this complex number and the given complex number.

[__init__\(self, real, imag\)](#)

[\(Complex, double, double\) -> None](#)

Constructs a complex number with the given real and

[__mul__\(self, c\)](#)

[\(Complex, Complex\) -> Complex](#)

Returns a complex number equal to the product of this complex number and the given complex number.

[__str__\(self\)](#)

[\(Complex\) -> str](#)

Returns a textual representation of this complex number.

__sub__(self, c)
`(Complex, Complex) -> Complex`
Returns a complex number equal to the difference of
number and the given complex number.

getImag(self)
`(Complex) -> double`
Returns the imaginary part of this complex number.

getR(self)
`(Complex) -> double`
Returns the magnitude of this complex number.

getReal(self)
`(Complex) -> double`
Returns the real part of this complex number.

getTheta(self)
`(Complex) -> double`
Returns the phase of this complex number.

setImag(self, value)
`(Complex, double) -> double`
Sets the imaginary part of this complex number to t

setR(self, value)
`(Complex, double) -> double`
Sets the magnitude of this complex number to the gi

setReal(self, value)
`(Complex, double) -> double`
Sets the real part of this complex number to the gi

setTheta(self, value)
`(Complex, double) -> double`
Sets the phase of this complex number to the given

Static methods defined here:

main(*argv)
`Complex test program.`

Properties defined here:

imag

get">**get** = [getImag](#)(self)
set">**set** = [setImag](#)(self, value)

r

get">**get** = [getR](#)(self)
set">**set** = [setR](#)(self, value)

real

get">**get** = [getReal](#)(self)
set">**set** = [setReal](#)(self, value)

theta

get">**get** = [getTheta](#)(self)
set">**set** = [setTheta](#)(self, value)

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Complex' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/28 00:06:11 \$'
__version__ = '\$Revision: 1.12 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Container](#) class.

Modules

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Container](#)

class Container([opus7.object.Object](#))

Base class from which all containers are derived.

Method resolution order:

[Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__hash__\(self\)](#)
[\(Container\)](#) -> int
Returns the hash of this container.

[__init__\(self\)](#)
[\(Container\)](#) -> None
Constructs this container.

[__iter__\(...\)](#)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
Container test program.

Properties defined here:

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes defined here:

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

__abstractmethods__ = ['__iter__', '_compareTo', 'purge']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.38 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [cursor](#) class.

Modules

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Cursor](#)

class Cursor([opus7.object.Object](#))

Base class from which all cursor classes are derived.

Method resolution order:

[Cursor](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self, list\)](#)
([Cursor](#)) -> None
Constructor.

[getDatum\(...\)](#)
Abstract method.

[insertAfter\(...\)](#)
Abstract method.

insertBefore(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[Cursor](#) test program.

Properties defined here:

datum
get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['getDatum', 'insertAfter', 'insertBefore', '']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.28 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Deap](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.doubleEndedPriorityQueue.DoubleEndedPriorityQueue](#)([op](#)
[Deap](#))

class **Deap**([opus7.doubleEndedPriorityQueue.DoubleEndedPrior](#)

Double-ended heap.

Method resolution order:

[Deap](#)
[opus7.doubleEndedPriorityQueue.DoubleEndedPriorityQueue](#)
[opus7.priorityQueue.PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__init__(self, length=0)
([Deap](#), int) -> None
Constructs a deap with the given length.

__iter__(self)
([Deap](#)) -> [Deap.Iterator](#)

Returns an iterator for this deap.

accept(self, visitor)
`(Deap, Visitor) -> None`
Makes the given visitor visit all the objects in th.

dequeueMax(self)
`(Deap) -> Object`
Dequeues and returns the object in this deap with the largest value.

dequeueMin(self)
`(Deap) -> Object`
Dequeues and returns the object in this deap with the smallest value.

dual(self, i)
`(Deap, i) -> int`
Returns the position of the dual of the given posit.

enqueue(self, obj)
`(Deap, Object) -> None`
Enqueues the given object in this deap.

getIsFull(self)
`(Deap) -> bool`
Returns true if this deap is full

getMax(self)
`(Deap) -> Object`
Returns the object in this deap with the largest va.

getMin(self)
`(Deap) -> Object`
Returns the object in this deap with the smallest va.

insertMax(self, pos, obj)
`(Deap, int, Object) -> None`
Inserts the given object into the max heap of this deap starting from the given position.

insertMin(self, pos, obj)
`(Deap, int, Object) -> None`
Inserts the given object into the min heap of this deap starting from the given position.

purge(self)
 ([Deap](#)) -> None
 Purges this deap.

Static methods defined here:

**__new__(*
 (Metaclass, ...) -> object**

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

log2(i)
 (int) -> int
 Returns `ceil(log_2(i))`.

main(*argv)
 [Deap](#) test program.

mask(i)
 (int) -> int
 Returns $2^{\lceil \log_2(i) \rceil - 1}$.

Data and other attributes defined here:

Iterator = <class 'opus7.deap.Iterator'>
 Enumerates the objects in a deap.

__abstractmethods__ = []

Static methods inherited from [opus7.doubleEndedPriorityQueue.D](#)

test(pqueue)
 [DoubleEndedPriorityQueue](#) test program.

Properties inherited from [opus7.doubleEndedPriorityQueue.Doubl](#)

max
 get lambda self

Properties inherited from [opus7.priorityQueue.PriorityQueue](#):

min

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.20 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo1](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo1](#)

class Demo1([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 1.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo1' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.7 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo10](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo10](#)

class Demo10([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 10.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo10' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.12 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo2](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo2](#)

class Demo2([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 2.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo2' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.14 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo3](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo3](#)

class Demo3([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 3.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo3' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.14 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo4](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo4](#)

class Demo4([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 4.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo4' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.15 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo5](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo5](#)

class Demo5([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 5.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo5' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.16 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo6](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo6](#)

class Demo6([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demonstration program number 6.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo6' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.9 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo7](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo7](#)

class Demo7([__builtin__.object](#))

Static methods defined here:

main(*argv)
 Demostration program number 7.

Data and other attributes defined here:

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo7' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.11 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Demo9](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Demo9](#)

class Demo9([__builtin__.object](#))

Static methods defined here:

main(*argv)
Demonstration program number 9.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Demo9' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DenseMatrix](#) class.

Modules

[sys](#)

Classes

[opus7.matrix.Matrix\(__builtin__.object\)](#)
[DenseMatrix](#)

class DenseMatrix([opus7.matrix.Matrix](#))

Dense matrix.

Method resolution order:

[DenseMatrix](#)
[opus7.matrix.Matrix](#)
[__builtin__.object](#)

Methods defined here:

__add__(self, mat)
([DenseMatrix](#), [DenseMatrix](#)) -> [DenseMatrix](#)
Returns the sum of this matrix and the given matrix

getitem__(self, (i, j))
([DenseMatrix](#), (int, int)) -> Object
Returns the specified element of this matrix.

init__(self, rows, cols)
([DenseMatrix](#), int, int) -> None

Constructor.

__mul__(self, mat)
`(DenseMatrix, DenseMatrix) -> DenseMatrix`
Returns the product of this matrix and the given ma

__setitem__(self, (i, j), value)
`(DenseMatrix, (int, int), Object) -> Object`
Sets the specified element of this matrix to the gi

getTranspose(self)
`(DenseMatrix) -> DenseMatrix`
Returns the transpose of this matrix.

Static methods defined here:

main(*argv)
`DenseMatrix test program`

Methods inherited from [opus7.matrix.Matrix](#):

getNumberOfColumns(self)

getNumberOfRows(self)

Static methods inherited from [opus7.matrix.Matrix](#):

test(mat)
`Matrix test program.`

testTimes(mat1, mat2)
`Matrix multiply test program.`

testTranspose(mat)
`Matrix transpose test program.`

Properties inherited from [opus7.matrix.Matrix](#):

numberOfColumns
get lambda self

numberOfRows
get lambda self

transpose
get lambda self

Data and other attributes inherited from [opus7.matrix.Matrix](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Matrix' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:00:48 \$'
__version__ = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DepthFirstBranchAndBoundSolver](#) class.

Classes

[opus7.solver.Solver](#)([opus7.object.Object](#))
[DepthFirstBranchAndBoundSolver](#)

class **DepthFirstBranchAndBoundSolver**([opus7.solver.Solver](#))

Depth-first branch-and-bound solver.

Method resolution order:

[DepthFirstBranchAndBoundSolver](#)
[opus7.solver.Solver](#)
[opus7.object.Object](#)
[_builtin__object](#)

Methods defined here:

__init__(self)
([DepthFirstBranchAndBoundSolver](#)) -> None
Constructor.

search(self, current)
([DepthFirstBranchAndBoundSolver](#), Solution) -> Solution
Does a depth-first traversal of the solution space
starting from the given node.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

Data and other attributes defined here:

`__abstractmethods__` = []

Methods inherited from [opus7.solver.Solver](#):

`solve(self, initial)`
`(Solver, Solution) -> Solution`
Solves a problem by searching the solution space
starting from the given node.

`updateBest(self, solution)`
`(Solver, Solution) -> None`
Records the given solution if it is complete, feasible
and has a lower objective function value than the best
solution seen so far.

Methods inherited from [opus7.object.Object](#):

`__cmp__(self, obj)`
`(Object, Object) -> int`
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

`main(*argv)`
Object test program.

Data and other attributes inherited from [opus7.object.Object](#):

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__metaclass__` = <class 'opus7.metaclass.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.3 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DepthFirstSolver](#) class.

Classes

[opus7.solver.Solver](#)([opus7.object.Object](#))
[DepthFirstSolver](#)

class DepthFirstSolver([opus7.solver.Solver](#))

Depth-first solver.

Method resolution order:

[DepthFirstSolver](#)
[opus7.solver.Solver](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([DepthFirstSolver](#)) -> None
Constructor.

search(self, current)
([DepthFirstSolver](#), Solution) -> Solution
Does a depth-first traversal of the solution space
starting from the given node.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

Data and other attributes defined here:

`__abstractmethods__` = []

Methods inherited from [opus7.solver.Solver](#):

`solve(self, initial)`
`(Solver, Solution) -> Solution`
Solves a problem by searching the solution space
starting from the given node.

`updateBest(self, solution)`
`(Solver, Solution) -> None`
Records the given solution if it is complete, feasible
and has a lower objective function value than the best
solution seen so far.

Methods inherited from [opus7.object.Object](#):

`__cmp__(self, obj)`
`(Object, Object) -> int`
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

`main(*argv)`
Object test program.

Data and other attributes inherited from [opus7.object.Object](#):

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__metaclass__` = <class 'opus7.metamodel.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.3 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Deque](#) class.

Classes

[opus7.queue.Queue](#)([opus7.container.Container](#))
[Deque](#)

class Deque([opus7.queue.Queue](#))

Represents a doubled-ended queue.

Method resolution order:

[Deque](#)
[opus7.queue.Queue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self)
([Deque](#)) -> None
Constructor.

dequeueHead(self)
([Deque](#)) -> Object
Dequeues the head of this deque.

dequeueTail(...)
Abstract method.

enqueueHead(...)
Abstract method.

enqueueTail(self, object)
([Deque](#), Object) -> None
Enqueues the given object at the tail of this deque

getHead(...)
Abstract method.

getTail(...)
Abstract method.

Static methods defined here:

__new__(*args, **kwargs)
(Metaclass, ...) -> object
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(deque)
[Deque](#) test program.

Properties defined here:

head
get lambda self

tail
get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__iter__', '_compareTo', 'dequeue', 'dequeueAll', 'enqueue', 'enqueueHead', 'getHead', 'getTail', 'purge']

Methods inherited from [opus7.queue.Queue](#):

dequeue(...)
Abstract method.

enqueue(...)
Abstract method.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

Container test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/23 12:23:21 \$'
version = '\$Revision: 1.24 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DequeAsArray](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.deque.Deque](#)([opus7.queue.Queue](#))
[DequeAsArray](#)([opus7.queueAsArray.QueueAsArray](#), [opus7.queueAsArray.QueueAsArray](#))
[opus7.queueAsArray.QueueAsArray](#)([opus7.queue.Queue](#))
[DequeAsArray](#)([opus7.queueAsArray.QueueAsArray](#), [opus7.queueAsArray.QueueAsArray](#))

**class DequeAsArray([opus7.queueAsArray.QueueAsArray](#),
[opus7.deque.Deque](#))**

[Deque](#) implemented using an array.

Method resolution order:

[DequeAsArray](#)
[opus7.queueAsArray.QueueAsArray](#)
[opus7.deque.Deque](#)
[opus7.queue.Queue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__](#)(self, size=0)
 ([DequeAsArray](#), int) -> None

Constructs a deque of the given size.

dequeueTail(self)

([DequeAsArray](#)) -> Object

Dequeues the objectc at the tail of this deque.

enqueueHead(self, obj)

([DequeAsArray](#), Object) -> None

Enqueues the given object at the head of this deque

getTail(self)

([DequeAsArray](#)) -> Object

Returns the object at the tail of this deque.

Static methods defined here:

__new__(*args, **kwargs)

(Metaclass, ...) -> object

Creates an instance of the class using the given ar

Raises a TypeError exception if the class is abstrac

This method is inserted as the method __new__

in classes instances derived from Metaclass.

main(*argv)

[DequeAsArray](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.queueAsArray.QueueAsArray](#):

__iter__(self)

([QueueAsArray](#)) -> [QueueAsArray](#).Iterator

Returns an iterator for this queue.

accept(self, visitor)

([QueueAsArray](#), Visitor) -> None

Makes the given visitor visit all the objects in th

dequeue(self)

([QueueAsArray](#)) -> Object

Dequeues the object at the head of this queue.

enqueue(self, obj)

([QueueAsArray](#), Object) -> None

Enqueues the given object to the tail of this queue

getHead(self)

([QueueAsArray](#)) -> Object

Returns the object at the head of this queue.

getIsFull(self)

([QueueAsArray](#)) -> bool

Returns true if this queue is full.

purge(self)

([QueueAsArray](#)) -> None

Purges this queue.

Data and other attributes inherited from [opus7.queueAsArray.Queue](#):

Iterator = <class 'opus7.queueAsArray.Iterator'>

Enumerates the elements of a [QueueAsArray](#).

Methods inherited from [opus7.deque.Deque](#):

dequeueHead(self)

([Deque](#)) -> Object

Dequeues the head of this deque.

enqueueTail(self, object)

([Deque](#), Object) -> None

Enqueues the given object at the tail of this deque

Static methods inherited from [opus7.deque.Deque](#):

test(deque)

[Deque](#) test program.

Properties inherited from [opus7.deque.Deque](#):

head

get *lambda self*

tail

get *lambda self*

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count

get *lambda self*

isEmpty

get *lambda self*

isFull

get *lambda self*

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.20 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DequeAsLinkedList](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.deque.Deque](#)([opus7.queue.Queue](#))
 [DequeAsLinkedList](#)([opus7.queueAsLinkedList.QueueAsLinkedList](#),
 [opus7.deque.Deque](#))
[opus7.queueAsLinkedList.QueueAsLinkedList](#)([opus7.queue.Queue](#))
 [DequeAsLinkedList](#)([opus7.queueAsLinkedList.QueueAsLinkedList](#),
 [opus7.deque.Deque](#))

**class DequeAsLinkedList([opus7.queueAsLinkedList.QueueAsLinkedList](#),
 [opus7.deque.Deque](#))**

[Deque](#) implemented using a linked list.

Method resolution order:

[DequeAsLinkedList](#)
 [opus7.queueAsLinkedList.QueueAsLinkedList](#)
 [opus7.deque.Deque](#)
 [opus7.queue.Queue](#)
 [opus7.container.Container](#)
 [opus7.object.Object](#)
 [builtin_object](#)

Methods defined here:

__init__(self)
[\(DequeAsLinkedList\)](#) -> None
Constructs a deque.

dequeueTail(self)
[\(DequeAsLinkedList\)](#) -> Object
Dequeues the object at the tail of this deque.

enqueueHead(self, obj)
[\(DequeAsLinkedList, Object\)](#) -> None
Enqueues the given object at the head of this deque

getTail(self)
[\(DequeAsLinkedList\)](#) -> Object
Returns the object at the tail of this deque.

Static methods defined here:

__new__ = new(*args, **kwargs)
[\(Metaclass, ...\)](#) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
[DequeAsLinkedList](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.queueAsLinkedList.QueueAsLinkedList](#)

__iter__(self)
[\(QueueAsLinkedList\)](#) -> [QueueAsLinkedList](#).Iterator
Returns an iterator for this queue.

accept(self, visitor)
[\(QueueAsLinkedList, Visitor\)](#) -> None
Makes the given visitor visit all the objects in this queue.

dequeue(self)
[\(QueueAsLinkedList\)](#) -> Object
Dequeues the object at the head of this queue.

enqueue(self, obj)
[\(QueueAsLinkedList, Object\)](#) -> None
Enqueues the given object to the tail of this queue

getHead(self)
[\(QueueAsLinkedList\)](#) -> Object
Returns the object at the head of this queue.

purge(self)
[\(QueueAsLinkedList\)](#) -> None
Purges this queue.

Data and other attributes inherited from
[opus7.queueAsLinkedList.QueueAsLinkedList](#):

Iterator = <class 'opus7.queueAsLinkedList.Iterator'>
Enumerates the elements of a [QueueAsLinkedList](#).

Methods inherited from [opus7.deque.Deque](#):

dequeueHead(self)
[\(Deque\)](#) -> Object
Dequeues the head of this deque.

enqueueTail(self, object)
[\(Deque, Object\)](#) -> None
Enqueues the given object at the tail of this deque

Static methods inherited from [opus7.deque.Deque](#):

test(deque)
[Deque](#) test program.

Properties inherited from [opus7.deque.Deque](#):

head
get lambda self

tail

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.18 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Digraph](#) class.

Classes

[opus7.graph.Graph](#)([opus7.container.Container](#))
[Digraph](#)

class Digraph([opus7.graph.Graph](#))

Base class from which all directed graph classes are derived.

Method resolution order:

[Digraph](#)
[opus7.graph.Graph](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__init__\(self, size\)](#)
([Digraph](#), int) -> None
Constructs a digraph with the specified maximum number of nodes.

[getIsCyclic\(self\)](#)
([Graph](#)) -> bool
Returns true if the graph is cyclic.

[getIsStronglyConnected\(self\)](#)
([Graph](#)) -> bool
Returns true if the graph is strongly connected.

[topologicalOrderTraversal\(self, visitor\)](#)

`(Graph, Visitor) -> None`
Makes the given visitor visit the vertices of this graph in topological order.

Static methods defined here:

`__new__ = new(*args, **kwargs)`
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

test(g)
`Digraph` test program.

testWeighted(g)
Weighted digraph test program.

Properties defined here:

isStronglyConnected
`get lambda self`

Data and other attributes defined here:

`__abstractmethods__ = ['_compareTo', 'addEdge', 'getEdge', 'getEmanatingEdges', 'getIncidentEdges', 'isEdge']`

Methods inherited from `opus7.graph.Graph`:

`__getitem__(self, v)`

`__iter__(self)`
(`Graph`) -> `Graph.Iterator`
Returns an iterator that enumerates the vertices of

`__len__(self)`

`__str__(self)`

([Graph](#)) -> str
Returns the string representation of this graph.

accept(self, visitor)
([Graph](#), Visitor) -> None
Makes the given visitor visit all the vertices in this graph.

addEdge(...)
Abstract method.

addVertex(self, *args)
([Graph](#), int [, Object]) -> None
Adds a vertex (with optional weight) to this graph.

breadthFirstTraversal(self, visitor, start)
([Graph](#), Visitor, Vertex) -> None
Makes the given visitor visit the vertices of this graph in breadth-first traversal order starting from the given vertex.

depthFirstTraversal(self, visitor, start)
([Graph](#), PrePostVisitor, Vertex) -> None
Makes the given visitor visit the vertices of this graph in depth-first traversal order starting from the given vertex.

getEdge(...)
Abstract method.

getEdges(...)
Abstract method.

getEmanatingEdges(...)
Abstract method.

getIncidentEdges(...)
Abstract method.

getIsConnected(self)
([Graph](#)) -> bool
Returns true if the graph is connected.

getIsDirected(self)
([Graph](#)) -> bool
Returns true if this graph is a directed graph.

getNumberOfEdges(self)

(Graph) -> int
Returns the number of edges in this graph.

getNumberOfVertices(self)
(Graph) -> int
Returns the number of vertices in this graph.

getVertex(self, v)
(Graph, int) -> Vertex
Returns the specified vertex of this graph.

getVertices(self)
(Graph) -> Graph.Iterator
Returns an iterator for this graph.

isEdge(...)
Abstract method.

purge(self)
(Graph) -> None
Purges this graph.

Properties inherited from [opus7.graph.Graph](#):

edges
get lambda self

isConnected
get lambda self

isCyclic
get lambda self

isDirected
get lambda self

numberOfEdges
get lambda self

numberOfVertices
get lambda self

vertices

get lambda self

Data and other attributes inherited from [opus7.graph.Graph](#):

CountingVisitor = <class 'opus7.graph.CountingVisitor'>
A visitor that counts the objects it visits.

Edge = <class 'opus7.graph.Edge'>
Represents an edge in a graph.

Iterator = <class 'opus7.graph.Iterator'>
Enumerates the vertices of a graph.

StrVisitor = <class 'opus7.graph.StrVisitor'>
Visitor that accumulates visited items in a string.

Vertex = <class 'opus7.graph.Vertex'>
Represents a vertex in a graph.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

Container test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/27 00:43:09 \$'
version = '\$Revision: 1.18 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DigraphAsLists](#) class.

Modules

[sys](#)

Classes

[opus7.digraph.Digraph](#)([opus7.graph.Graph](#))
[DigraphAsLists](#)([opus7.digraph.Digraph](#), [opus7.graphAsLists.GraphAsLists](#))
[opus7.graphAsLists.GraphAsLists](#)([opus7.graph.Graph](#))
[DigraphAsLists](#)([opus7.digraph.Digraph](#), [opus7.graphAsLists.GraphAsLists](#))

**class DigraphAsLists([opus7.digraph.Digraph](#),
[opus7.graphAsLists.GraphAsLists](#))**

[Digraph](#) implemented using adjacency lists.

Method resolution order:

[DigraphAsLists](#)
[opus7.digraph.Digraph](#)
[opus7.graphAsLists.GraphAsLists](#)
[opus7.graph.Graph](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__](#)(self, size)

Constructs a digraph with the given maximum number

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[DigraphAsLists](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.digraph.Digraph](#):

getIsCyclic(self)
(Graph) -> bool
Returns true if the graph is cyclic.

getIsStronglyConnected(self)
(Graph) -> bool
Returns true if the graph is strongly connected.

topologicalOrderTraversal(self, visitor)
(Graph, Visitor) -> None
Makes the given visitor visit the vertices of this graph in topological order.

Static methods inherited from [opus7.digraph.Digraph](#):

test(g)
[Digraph](#) test program.

testWeighted(g)
Weighted digraph test program.

Properties inherited from [opus7.digraph.Digraph](#):

isStronglyConnected

get lambda self

Methods inherited from [opus7.graphAsLists.GraphAsLists](#):

addEdge(self, *args)

[\(GraphAsLists\)](#), int, int [, Object]) -> None
Adds an edge with the (optional) given weight connecting the given vertices in this graph.

getEdge(self, v, w)

[\(GraphAsLists\)](#), int, int) -> Edge
Returns the edge connecting the specified vertices .

getEdges(self)

[\(GraphAsLists\)](#) -> [GraphAsLists](#).EdgeIterator
Returns an iterator that enumerates the edges in th.

getEmanatingEdges(self, v)

[\(GraphAsLists\)](#), int) -> [GraphAsLists](#).EmanatingEdgesI
Returns an iterator that enumerates the edges emanating from the given vertex in this graph.

getIncidentEdges(self, v)

[\(GraphAsLists\)](#), int) -> [GraphAsLists](#).EmanatingEdgesI
Returns an iterator that enumerates the edges incident to the given vertex in this graph.

isEdge(self, v, w)

[\(GraphAsLists\)](#), int, int) -> bool
Returns true if there is an edge connecting the specified vertices in this graph.

purge(self)

[\(GraphAsLists\)](#) -> None
Purges this graph.

Data and other attributes inherited from [opus7.graphAsLists.GraphAsLists](#):

EdgeIterator = <class 'opus7.graphAsLists.EdgeIterator'>
Enumerates the edges of a [GraphAsLists](#).

EmanatingEdgeIterator = <class 'opus7.graphAsLists.EmanatingEdgeIterator'>

Enumerates the edges emanating from a given vertex :

Methods inherited from [opus7.graph.Graph](#):

__getitem__(self, v)

__iter__(self)

(Graph) -> Graph.Iterator

Returns an iterator that enumerates the vertices of

__len__(self)

__str__(self)

(Graph) -> str

Returns the string representation of this graph.

accept(self, visitor)

(Graph, Visitor) -> None

Makes the given visitor visit all the vertices in t

addVertex(self, *args)

(Graph, int [, Object]) -> None

Adds a vertex (with optional weight) to this graph.

breadthFirstTraversal(self, visitor, start)

(Graph, Visitor, Vertex) -> None

Makes the given visitor visit the vertices of this graph in breadth-first traversal order starting from the given vertex.

depthFirstTraversal(self, visitor, start)

(Graph, PrePostVisitor, Vertex) -> None

Makes the given visitor visit the vertices of this graph in depth-first traversal order starting from the given vertex.

getIsConnected(self)

(Graph) -> bool

Returns true if the graph is connected.

getIsDirected(self)

(Graph) -> bool

Returns true if this graph is a directed graph.

getNumberOfEdges(self)

(Graph) -> int

Returns the number of edges in this graph.

getNumberOfVertices(self)

(Graph) -> int

Returns the number of vertices in this graph.

getVertex(self, v)

(Graph, int) -> Vertex

Returns the specified vertex of this graph.

getVertices(self)

(Graph) -> Graph.Iterator

Returns an iterator for this graph.

Properties inherited from [opus7.graph.Graph](#):

edges

get lambda self

isConnected

get lambda self

isCyclic

get lambda self

isDirected

get lambda self

numberOfEdges

get lambda self

numberOfVertices

get lambda self

vertices

get lambda self

Data and other attributes inherited from [opus7.graph.Graph](#):

CountingVisitor = <class 'opus7.graph.CountingVisitor'>

A visitor that counts the objects it visits.

Edge = <class 'opus7.graph.Edge'>
Represents an edge in a graph.

Iterator = <class 'opus7.graph.Iterator'>
Enumerates the vertices of a graph.

StrVisitor = <class 'opus7.graph.StrVisitor'>
Visitor that accumulates visited items in a string.

Vertex = <class 'opus7.graph.Vertex'>
Represents a vertex in a graph.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull

get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'

__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [DigraphAsMatrix](#) class.

Modules

[sys](#)

Classes

[opus7.digraph.Digraph\(opus7.graph.Graph\)](#)
[DigraphAsMatrix\(opus7.digraph.Digraph,](#)
[opus7.graphAsMatrix.GraphAsMatrix\)](#)
[opus7.graphAsMatrix.GraphAsMatrix\(opus7.graph.Graph\)](#)
[DigraphAsMatrix\(opus7.digraph.Digraph,](#)
[opus7.graphAsMatrix.GraphAsMatrix\)](#)

class DigraphAsMatrix([opus7.digraph.Digraph](#),
[opus7.graphAsMatrix.GraphAsMatrix](#))

Directed graph implemented using an adjacency matrix.

Method resolution order:

[DigraphAsMatrix](#)
[opus7.digraph.Digraph](#)
[opus7.graphAsMatrix.GraphAsMatrix](#)
[opus7.graph.Graph](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, size)
([DigraphAsMatrix](#), int) -> None
Constructs a digraph with the specified maximum number of vertices.

addEdge(self, *args)
(DigraphAsMatrix, int, int [, Object]) -> None
Adds an edge with the (optional) weight connecting the given vertices in this digraph.

getEdges(self)
([DigraphAsMatrix](#)) -> [DigraphAsMatrix](#).EdgeIterator
Returns an iterator that enumerates the edges in this digraph.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[DigraphAsMatrix](#) test program.

Data and other attributes defined here:

EdgeIterator = <class 'opus7.digraphAsMatrix.EdgeIterator'>
Enumerates the edges of a [DigraphAsMatrix](#).

__abstractmethods__ = []

Methods inherited from [opus7.digraph.Digraph](#):

getIsCyclic(self)
(Graph) -> bool
Returns true if the graph is cyclic.

getIsStronglyConnected(self)
(Graph) -> bool
Returns true if the graph is strongly connected.

topologicalOrderTraversal(self, visitor)
`(Graph, Visitor) -> None`
Makes the given visitor visit the vertices of this graph in topological order.

Static methods inherited from [opus7.digraph.Digraph](#):

test(g)
`Digraph` test program.

testWeighted(g)
Weighted digraph test program.

Properties inherited from [opus7.digraph.Digraph](#):

isStronglyConnected
get lambda self

Methods inherited from [opus7.graphAsMatrix.GraphAsMatrix](#):

getEdge(self, v, w)
`(GraphAsMatrix, int, int) -> Edge`
Returns the edge connecting the specified vertices.

getEmanatingEdges(self, v)
`(GraphAsMatrix, int) -> GraphAsMatrix.EmanatingEdges`
Returns an iterator that enumerates the edges emanating from the given vertex in this graph.

getIncidentEdges(self, w)
`(GraphAsMatrix, int) -> GraphAsMatrix.IncidentEdges`
Returns an iterator that enumerates the edges incident to the given vertex in this graph.

isEdge(self, v, w)
`(GraphAsMatrix, int, int) -> bool`
Returns true if there is an edge connecting the specified vertices in this graph.

purge(self)
`(GraphAsMatrix) -> None`
Purges this graph.

Data and other attributes inherited from [opus7.graphAsMatrix.Graph](#)

EmanatingEdgesIterator = <class 'opus7.graphAsMatrix.EmanatingEdgesIterator'>
Enumerates the edges emanating from a given vertex.

IncidentEdgesIterator = <class 'opus7.graphAsMatrix.IncidentEdgesIterator'>
Enumerates the edges incident upon a given vertex.

Methods inherited from [opus7.graph.Graph](#):

__getitem__(self, v)

__iter__(self)

(Graph) -> Graph.Iterator
Returns an iterator that enumerates the vertices of

__len__(self)

__str__(self)

(Graph) -> str
Returns the string representation of this graph.

accept(self, visitor)

(Graph, Visitor) -> None
Makes the given visitor visit all the vertices in this graph.

addVertex(self, *args)

(Graph, int [, Object]) -> None
Adds a vertex (with optional weight) to this graph.

breadthFirstTraversal(self, visitor, start)

(Graph, Visitor, Vertex) -> None
Makes the given visitor visit the vertices of this graph in breadth-first traversal order starting from the given vertex.

depthFirstTraversal(self, visitor, start)

(Graph, PrePostVisitor, Vertex) -> None
Makes the given visitor visit the vertices of this graph in depth-first traversal order starting from the given vertex.

getIsConnected(self)

(Graph) -> bool
Returns true if the graph is connected.

getIsDirected(self)
(Graph) -> bool
Returns true if this graph is a directed graph.

getNumberOfEdges(self)
(Graph) -> int
Returns the number of edges in this graph.

getNumberOfVertices(self)
(Graph) -> int
Returns the number of vertices in this graph.

getVertex(self, v)
(Graph, int) -> Vertex
Returns the specified vertex of this graph.

getVertices(self)
(Graph) -> Graph.Iterator
Returns an iterator for this graph.

Properties inherited from [opus7.graph.Graph](#):

edges
get lambda self

isConnected
get lambda self

isCyclic
get lambda self

isDirected
get lambda self

numberOfEdges
get lambda self

numberOfVertices
get lambda self

vertices
get lambda self

Data and other attributes inherited from [opus7.graph.Graph](#):

CountingVisitor = <class 'opus7.graph.CountingVisitor'>
A visitor that counts the objects it visits.

Edge = <class 'opus7.graph.Edge'>
Represents an edge in a graph.

Iterator = <class 'opus7.graph.Iterator'>
Enumerates the vertices of a graph.

StrVisitor = <class 'opus7.graph.StrVisitor'>
Visitor that accumulates visited items in a string.

Vertex = <class 'opus7.graph.Vertex'>
Represents a vertex in a graph.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.doubleEndedPriorityQueue](#)

(version 1.12, \$Date: 2003/09/23
12:23:21 \$)

[opus7/doubleEndedPriorityQueue](#)

Provides the [DoubleEndedPriorityQueue](#) class.

Classes

[opus7.priorityQueue.PriorityQueue](#)([opus7.container.Container](#))
[DoubleEndedPriorityQueue](#)

class **DoubleEndedPriorityQueue**([opus7.priorityQueue.PriorityQueue](#))

Base class from which all double-ended priority queues are derived.

Method resolution order:

[DoubleEndedPriorityQueue](#)
[opus7.priorityQueue.PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self)
([DoubleEndedPriorityQueue](#)) -> None
Constructor.

dequeueMax(...)
Abstract method.

getMax(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

test(pqueue)

[DoubleEndedPriorityQueue](#) test program.

Properties defined here:

max

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__iter__', '_compareTo', 'dequeueMax',
'enqueue', 'getMax', 'getMin', 'purge']

Methods inherited from [opus7.priorityQueue.PriorityQueue](#):

dequeueMin(...)

Abstract method.

enqueue(...)

Abstract method.

getMin(...)

Abstract method.

Properties inherited from [opus7.priorityQueue.PriorityQueue](#):

min

get lambda self

Methods inherited from [opus7.container.Container](#):

`__hash__(self)`
`(Container) -> int`
Returns the hash of this container.

`__iter__(...)`
Abstract method.

`__str__(self)`
`(Container) -> string`
Returns a string representation of this container.

`accept(self, visitor)`
`(Container, Visitor) -> None`
Makes the given visitor visit all the items in this

`elements(self)`
`(Container) -> Object`
Generator that yields the objects in this container

`getCount(self)`
`(Container) -> int`
Returns the number of items in this container.

`getIsEmpty(self)`
`(Container) -> bool`
Returns true if this container is empty.

`getIsFull(self)`
`(Container) -> bool`
Returns true if this container is full.

`purge(...)`
Abstract method.

Static methods inherited from [opus7.container.Container](#):

`main(*argv)`
Container test program.

Properties inherited from [opus7.container.Container](#):

`count`

get *lambda self*

isEmpty

get *lambda self*

isFull

get *lambda self*

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/23 12:23:21 \$'

__version__ = '\$Revision: 1.12 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Edge](#) class.

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Edge](#)

class Edge([opus7.object.Object](#))

Base class from which all graph edge classes are derived

Method resolution order:

[Edge](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self\)](#)
([Edge](#)) -> None
Constructor.

[getIsDirected\(...\)](#)
Abstract method.

[getV0\(...\)](#)
Abstract method.

[getV1\(...\)](#)
Abstract method.

[getWeight\(...\)](#)
Abstract method.

mateOf(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Properties defined here:

isDirected
get lambda self

v0
get lambda self

v1
get lambda self

weight
get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['_compareTo', 'getIsDirected', 'getV0', 'getWeight', 'mateOf']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)
Object test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/27 00:43:09 \$'
__version__ = '\$Revision: 1.12 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides a number of example functions.

Modules

[exceptions](#) [math](#) [sys](#)

Classes

[exceptions.Exception](#)

A

class A([exceptions.Exception](#))

#{

Methods inherited from [exceptions.Exception](#):

[__getitem__](#)(...)

[__init__](#)(...)

[__str__](#)(...)

Functions

Fibonacci1(n)

```
#{
#!def Fibonacci(n):
#[
```

Fibonacci2(n)

```
#{
#!def Fibonacci(n):
#[
```

Fibonacci3(n)

```
#{
#!def Fibonacci(n):
#[
```

Fibonacci4(n, k)

```
#{
#!def Fibonacci(n, k):
#[
```

Horner1(a, n, x)

```
#{
#!def Horner(a, n, x):
#[
```

Horner2(a, n, x)

```
#{
#!def Horner(a, n, x):
#[
```

binarySearch(array, target, i, n)

```
#{
```

binom(n, m)

```
#{
```

bucketSort(a, n, buckets, m)

```
#{
```

f()

factorial(n)

```
#{
```

findMaximum(a, n)

```
#{
```

g()

gamma()

```
#{

geometricSeriesSum1(x, n)
#{  
#!def geometricSeriesSum(x, n):
#[

geometricSeriesSum2(x, n)
#{  
#!def geometricSeriesSum(x, n):
#[

geometricSeriesSum3(x, n)
#{  
#!def geometricSeriesSum(x, n):
#[

merge(array, pos, m, n)

mergeSort(array, i, n)
#{

one()
#{

pi(trials)
#{

power(x, n)
#{

prefixSums(a, n)
#{

sum(n)
#{

two(y)

typeset(l, D, s)
#{}
```

Data

```
RandomNumberGenerator =  
<opus7.randomNumberGenerator.RandomNumberGenerator object>  
  __author__ = 'Bruno R. Preiss, P.Eng.'  
  __credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
  __date__ = '$Date: 2003/09/28 00:06:11 $'  
  __version__ = '$Revision: 1.11 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [InternalError](#), [StateError](#),
[ContainerEmpty](#) and [ContainerFull](#) exception classes.

Modules

[exceptions](#) [sys](#)

Classes

[exceptions.Exception](#)
[InternalError](#)
[StateError](#)
[ContainerEmpty](#)
[ContainerFull](#)

class **ContainerEmpty**([StateError](#))

Raised when a container operation fails because the container is empty.

Method resolution order:

[ContainerEmpty](#)
[StateError](#)
[exceptions.Exception](#)

Methods inherited from [exceptions.Exception](#):

[__getitem__\(...\)](#)

[__init__\(...\)](#)

[__str__\(...\)](#)

class ContainerFull([StateError](#))

Raised when a container operation fails because the container is full.

Method resolution order:

[ContainerFull](#)

[StateError](#)

[exceptions.Exception](#)

Methods inherited from [exceptions.Exception](#):

`__getitem__(...)`

`__init__(...)`

`__str__(...)`

class InternalError([exceptions.Exception](#))

Raised when situation arises that should never occur.

Methods inherited from [exceptions.Exception](#):

`__getitem__(...)`

`__init__(...)`

`__str__(...)`

class StateError([exceptions.Exception](#))

Raised when an operation on an object is not allowed due to the state of that object.

Methods inherited from [exceptions.Exception](#):

__getitem__(...)

__init__(...)

__str__(...)

Functions

main(*argv)

Exceptions test program.

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/03 00:52:42 \$'

__version__ = '\$Revision: 1.13 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Experiment1](#) class.

Modules

[opus7.example](#) [sys](#)

Classes

[__builtin__.object](#)
[Experiment1](#)

class Experiment1([__builtin__.object](#))

Program tha measures the running times of both a recursive and a non-recursive method to compute the Fibonacci numbers.

Static methods defined here:

main(*argv)
[Experiment1](#) test program.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Experiment1' objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Experiment2](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Experiment2](#)

class Experiment2([__builtin__.object](#))

Program tha measures the running times of both a recursive and a non-recursive method to compute the Fibonacci numbers.

Static methods defined here:

main(*argv)
[Experiment2](#) test program.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Experiment2'
objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.7 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [ExponentialRV](#) class.

Modules

[math](#)

[sys](#)

Classes

[opus7.randomVariable.RandomVariable](#)([opus7.object.Object](#))
[ExponentialRV](#)

class **ExponentialRV**([opus7.randomVariable.RandomVariable](#))

Exponentially distributed random variable.

Method resolution order:

[ExponentialRV](#)
[opus7.randomVariable.RandomVariable](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, mu)
([ExponentialRV](#), double) -> None
Constructs an exponentially distributed random variable with the given mean.

getNext(self)
([ExponentialRV](#)) -> double
Returns the next sample.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[ExponentialRV](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Properties inherited from [opus7.randomVariable.RandomVariable](#):

next

get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

```
RandomNumberGenerator =  
<opus7.randomNumberGenerator.RandomNumberGenerator object>  
__author__ = 'Bruno R. Preiss, P.Eng.'  
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
__date__ = '$Date: 2003/09/26 14:46:27 $'  
__version__ = '$Revision: 1.6 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [ExpressionTree](#) class.

Classes

[opus7.binaryTree.BinaryTree](#)([opus7.tree.Tree](#))
[ExpressionTree](#)

class ExpressionTree([opus7.binaryTree.BinaryTree](#))

Represents expressions comprised of binary operators.

Method resolution order:

[ExpressionTree](#)
[opus7.binaryTree.BinaryTree](#)
[opus7.tree.Tree](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, word)

Constructs an expression tree with the given word.
([ExpressionTree](#), str) -> None

__str__(self)

([ExpressionTree](#)) -> str
Returns a string containing the infix representation
of the expression represented by this expression tree

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

parsePostfix(input)
(File) -> ExpressionTree
 Parses the given file into an expression tree.

Data and other attributes defined here:

InfixVisitor = <class 'opus7.expressionTree.InfixVisitor'>
Visits the nodes of an expression tree and constructs a string that contains the infix representation of the tree.

__abstractmethods__ = []

Methods inherited from [opus7.binaryTree.BinaryTree](#):

attachKey(self, obj)
([BinaryTree](#), Object) -> None
Makes the given object the key of this binary tree.

attachLeft(self, t)
([BinaryTree](#), Binary Tree) -> None
Attaches the given binary tree as the left subtree.

attachRight(self, t)
([BinaryTree](#), Binary Tree) -> None
Attaches the given binary tree as the right subtree.

depthFirstGenerator(self, mode)
([BinaryTree](#)) -> generator
Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)
([BinaryTree](#), PrePostVisitor) -> None
Makes the given visitor do a depth-first traversal.

detachKey(self)

detach(self)
`(BinaryTree) -> None`
Detaches and returns the key in this binary tree node.

detachLeft(self)
`(BinaryTree) -> BinaryTree`
Detaches and returns the left subtree of this binary tree node.

detachRight(self)
`(BinaryTree) -> BinaryTree`
Detaches and returns the right subtree of this binary tree node.

getDegree(self)
`(BinaryTree) -> int`
Returns the degree of this binary tree node.

getIsEmpty(self)
`(BinaryTree) -> bool`
Returns true if this binary tree is empty.

getIsLeaf(self)
`(BinaryTree) -> bool`
Returns true if this binary tree node is a leaf.

getKey(self)
`(BinaryTree) -> Object`
Returns the key in this binary tree node.

getLeft(self)
`(BinaryTree) -> BinaryTree`
Returns the left subtree of this binary tree.

getRight(self)
`(BinaryTree) -> BinaryTree`
Returns the right subtree of this binary tree.

getSubtree(self, i)
`(BinaryTree, int) -> Object`
Returns the specified subtree of this binary tree.

purge(self)
`(BinaryTree) -> None`
Purges this binary tree.

Static methods inherited from [opus7.binaryTree.BinaryTree](#):

main(*argv)
[BinaryTree](#) test program.

Properties inherited from [opus7.binaryTree.BinaryTree](#):

left

get lambda self

right

get lambda self

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)

(Tree) -> Tree.Iterator

Returns an interator for this tree.

accept(self, visitor)

(Tree) -> Visitor

Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)

(Tree) -> generator

Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)

(Tree, Visitor) -> None

Makes the given visitor do a breadth-first traversal.

getCount(self)

(Tree) -> int

Returns the number of nodes in this tree.

getHeight(self)

(Tree) -> int

Returns the height of this tree.

Static methods inherited from [opus7.tree.Tree](#):

test(tree)

Tree test program.

Properties inherited from [opus7.tree.Tree](#):

degree

get lambda self

height

get lambda self

isLeaf

get lambda self

key

get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/07/23 18:18:06 \$'
version = '\$Revision: 1.4 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[index](#)

[opus7.float](#) (version 1.4, \$Date: 2003/09/25 01:07:38 \$) [opus7/float.py](#)

Provides the [Float](#) class.

Modules

[math](#)

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
 [Float](#)([__builtin__.float](#), [opus7.object.Object](#))
[__builtin__.float](#)([__builtin__.object](#))
 [Float](#)([__builtin__.float](#), [opus7.object.Object](#))

class [Float](#)([__builtin__.float](#), [opus7.object.Object](#))

[Float](#) class.

Method resolution order:

[Float](#)
 [__builtin__.float](#)
[opus7.object.Object](#)
 [__builtin__.object](#)

Methods defined here:

[__hash__\(self\)](#)
 ([Float](#)) -> [float](#)
Hashes this string.

[__init__\(self, obj\)](#)
 ([Float](#), object) -> None
Constructs a string with the string representation.
The [Object](#) metaclass provides a [__new__](#) method

that initializes the str instance, so none is defined.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a TypeError exception if the class is abstract.
This method is inserted as the method __new__
in classes instances derived from Metaclass.

main(*argv)
[Float](#) test program.

testHash()
[Float](#) hash test program.

Data and other attributes defined here:

__abstractmethods__ = []

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Float' objects>
list of weak references to the object (if defined)

Methods inherited from [__builtin__.float](#):

__abs__(...)
x. [abs](#)() <==> abs(x)

__add__(...)
x. [add](#)(y) <==> x+y

__cmp__(...)
x. [cmp](#)(y) <==> cmp(x, y)

__coerce__(...)
x. [coerce](#)(y) <==> coerce(x, y)

`__div__(...)`
 $x \cdot \underline{\text{div}}(y) \iff x/y$

`__divmod__(...)`
 $x \cdot \underline{\text{divmod}}(y) \iff x \text{divmod}(x, y)y$

`__float__(...)`
 $x \cdot \underline{\text{float}}() \iff \underline{\text{float}}(x)$

`__floordiv__(...)`
 $x \cdot \underline{\text{floordiv}}(y) \iff x//y$

`__getattribute__(...)`
 $x \cdot \underline{\text{getattribute}}('name') \iff x.name$

`__getnewargs__(...)`

`__int__(...)`
 $x \cdot \underline{\text{int}}() \iff \text{int}(x)$

`__long__(...)`
 $x \cdot \underline{\text{long}}() \iff \text{long}(x)$

`__mod__(...)`
 $x \cdot \underline{\text{mod}}(y) \iff x \% y$

`__mul__(...)`
 $x \cdot \underline{\text{mul}}(y) \iff x * y$

`__neg__(...)`
 $x \cdot \underline{\text{neg}}() \iff -x$

`__nonzero__(...)`
 $x \cdot \underline{\text{nonzero}}() \iff x != 0$

`__pos__(...)`
 $x \cdot \underline{\text{pos}}() \iff +x$

`__pow__(...)`
 $x \cdot \underline{\text{pow}}(y[, z]) \iff \text{pow}(x, y[, z])$

`__radd__(...)`
 $x \cdot \underline{\text{radd}}(y) \iff y + x$

__rdiv__(...)
x.rdiv(y) <==> y/x

__rdivmod__(...)
x.rdivmod(y) <==> ydivmod(y, x)x

__repr__(...)
x.repr() <==> repr(x)

__rfloordiv__(...)
x.rfloordiv(y) <==> y//x

__rmod__(...)
x.rmod(y) <==> y%x

__rmul__(...)
x.rmul(y) <==> y*x

__rpow__(...)
y.rpow(x[, z]) <==> pow(x, y[, z])

__rsub__(...)
x.rsub(y) <==> y-x

__rtruediv__(...)
x.rtruediv(y) <==> y/x

__str__(...)
x.str() <==> str(x)

__sub__(...)
x.sub(y) <==> x-y

__truediv__(...)
x.truediv(y) <==> x/y

Data and other attributes inherited from [opus7.object.Object](#):

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [GeneralTree](#) class.

Modules

[sys](#)

Classes

[opus7.tree.Tree](#)([opus7.container.Container](#))
[GeneralTree](#)

class **GeneralTree**([opus7.tree.Tree](#))

A general tree implemented using a linked list of subtrees.

Method resolution order:

[GeneralTree](#)
[opus7.tree.Tree](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, key)
([GeneralTree](#), Object) -> None
Constructs a general tree with the given object at .

attachSubtree(self, t)
([GeneralTree](#), [GeneralTree](#)) -> None
Attaches the given general tree as a subtree
of this general tree node.

detachSubtree(self, t)
[\(GeneralTree, GeneralTree\)](#) -> [GeneralTree](#)
Detaches and returns specified general tree from this general tree node.

getDegree(self)
[\(GeneralTree\)](#) -> int
Returns the degree of this general tree node.

getIsEmpty(self)
[\(GeneralTree\)](#) -> bool
Returns false always.

getIsLeaf(self)
[\(GeneralTree\)](#) -> bool
Returns true if this general tree is a leaf.

getKey(self)
[\(GeneralTree\)](#) -> Object
Returns the key in this general tree node.

getSubtree(self, i)
[\(GeneralTree\)](#) -> Object
Returns the specified subtree of this general tree |

purge(self)
[\(GeneralTree\)](#) -> None
Purges this general tree.

Static methods defined here:

__new__ = new(*args, **kwargs)
[\(Metaclass, ...\)](#) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
[GeneralTree](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)
([Tree](#)) -> [Tree](#).Iterator
Returns an interator for this tree.

accept(self, visitor)
([Tree](#)) -> Visitor
Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)
([Tree](#)) -> generator
Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)
([Tree](#), Visitor) -> None
Makes the given visitor do a breadth-first traversal of this tree.

depthFirstGenerator(self, mode)
([Tree](#)) -> generator
Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)
([Tree](#), PrePostVisitor) -> None
Makes the given visitor do a depth-first traversal.

getCount(self)
([Tree](#)) -> int
Returns the number of nodes in this tree.

getHeight(self)
([Tree](#)) -> int
Returns the height of this tree.

Static methods inherited from [opus7.tree.Tree](#):

test(tree)
[Tree](#) test program.

Properties inherited from [opus7.tree.Tree](#):

degree

get lambda self

height

get lambda self

isLeaf

get lambda self

key

get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>

Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getIsFull(self)

`(Container) -> bool`
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

`(Object, Object) -> int`

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.22 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Graph](#) class.

Modules

[exceptions](#) [sys](#) [types](#)

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[Graph](#)

class Graph(opus7.container.Container)

Base class from which all graph classes are derived.

Method resolution order:

[Graph](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__getitem__\(self, v\)](#)

[__init__\(self, size\)](#)

[\(Graph, int\)](#) -> None

Constructs a graph with the given maximum number of

[__iter__\(self\)](#)

[\(Graph\)](#) -> [Graph.Iterator](#)

Returns an iterator that enumerates the vertices of

__len__(self)

__str__(self)
`(Graph) -> str`
Returns the string representation of this graph.

accept(self, visitor)
`(Graph, Visitor) -> None`
Makes the given visitor visit all the vertices in this graph.

addEdge(...)
Abstract method.

addVertex(self, *args)
`(Graph, int [, Object]) -> None`
Adds a vertex (with optional weight) to this graph.

breadthFirstTraversal(self, visitor, start)
`(Graph, Visitor, Vertex) -> None`
Makes the given visitor visit the vertices of this graph in breadth-first traversal order starting from the given vertex.

depthFirstTraversal(self, visitor, start)
`(Graph, PrePostVisitor, Vertex) -> None`
Makes the given visitor visit the vertices of this graph in depth-first traversal order starting from the given vertex.

getEdge(...)
Abstract method.

getEdges(...)
Abstract method.

getEmanatingEdges(...)
Abstract method.

getIncidentEdges(...)
Abstract method.

getIsConnected(self)
`(Graph) -> bool`
Returns true if the graph is connected.

getIsCyclic(...)
Abstract method.

getIsDirected(self)
`(Graph) -> bool`
Returns true if this graph is a directed graph.

getNumberOfEdges(self)
`(Graph) -> int`
Returns the number of edges in this graph.

getNumberOfVertices(self)
`(Graph) -> int`
Returns the number of vertices in this graph.

getVertex(self, v)
`(Graph, int) -> Vertex`
Returns the specified vertex of this graph.

getVertices(self)
`(Graph) -> Graph.Iterator`
Returns an iterator for this graph.

isEdge(...)
Abstract method.

purge(self)
`(Graph) -> None`
Purges this graph.

test(g)
`Graph` test program.

testWeighted(g)
Weighted graph test program.

Static methods defined here:

__new__ = new(*args, **kwargs)
`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Properties defined here:

edges

get lambda self

isConnected

get lambda self

isCyclic

get lambda self

isDirected

get lambda self

numberOfEdges

get lambda self

numberOfVertices

get lambda self

vertices

get lambda self

Data and other attributes defined here:

CountingVisitor = <class 'opus7.graph.CountingVisitor'>

A visitor that counts the objects it visits.

Edge = <class 'opus7.graph.Edge'>

Represents an edge in a graph.

Iterator = <class 'opus7.graph.Iterator'>

Enumerates the vertices of a graph.

StrVisitor = <class 'opus7.graph.StrVisitor'>

Visitor that accumulates visited items in a string.

Vertex = <class 'opus7.graph.Vertex'>

Represents a vertex in a graph.

__abstractmethods__ = ['_compareTo', 'addEdge', 'getEdge', 'getE

'getEmanatingEdges', 'getIncidentEdges', 'getIsCyclic', 'isEdge']

Methods inherited from [opus7.container.Container](#):

__hash__(self)

([Container](#)) -> int

Returns the hash of this container.

elements(self)

([Container](#)) -> Object

Generator that yields the objects in this container

getCount(self)

([Container](#)) -> int

Returns the number of items in this container.

getIsEmpty(self)

([Container](#)) -> bool

Returns true if this container is empty.

getIsFull(self)

([Container](#)) -> bool

Returns true if this container is full.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

[Container](#) test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/27 00:43:09 \$'
__version__ = '\$Revision: 1.27 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [GraphAsLists](#) class.

Modules

[sys](#)

Classes

[opus7.graph.Graph](#)([opus7.container.Container](#))
[GraphAsLists](#)

class GraphAsLists(opus7.graph.Graph)

[Graph](#) implemented using adjacency lists.

Method resolution order:

[GraphAsLists](#)
[opus7.graph.Graph](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, size)
([GraphAsLists](#), int) -> None
Constructs a graph with the given maximum number of

addEdge(self, *args)
([GraphAsLists](#), int, int [, Object]) -> None
Adds an edge with the (optional) given weight
connecting the given vertices in this graph.

getEdge(self, v, w)
([GraphAsLists](#), int, int) -> Edge
Returns the edge connecting the specified vertices .

getEdges(self)
([GraphAsLists](#)) -> [GraphAsLists](#).EdgeIterator
Returns an iterator that enumerates the edges in th.

getEmanatingEdges(self, v)
([GraphAsLists](#), int) -> [GraphAsLists](#).EmanatingEdgesI
Returns an iterator that enumerates the edges emanating from the given vertex in this graph.

getIncidentEdges(self, v)
([GraphAsLists](#), int) -> [GraphAsLists](#).EmanatingEdgesI
Returns an iterator that enumerates the edges incident to the given vertex in this graph.

getIsCyclic(self)
([GraphAsLists](#)) -> bool

Returns true if this graph is cyclic.

isEdge(self, v, w)
([GraphAsLists](#), int, int) -> bool
Returns true if there is an edge connecting the specified vertices in this graph.

purge(self)
([GraphAsLists](#)) -> None
Purges this graph.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[GraphAsLists](#) test program.

Data and other attributes defined here:

EdgeIterator = <class 'opus7.graphAsLists.EdgeIterator'>
Enumerates the edges of a [GraphAsLists](#).

EmanatingEdgeIterator = <class 'opus7.graphAsLists.EmanatingEdgeIterator'>
Enumerates the edges emanating from a given vertex.

__abstractmethods__ = []

Methods inherited from [opus7.graph.Graph](#):

__getitem__(self, v)

__iter__(self)
([Graph](#)) -> [Graph](#).Iterator
Returns an iterator that enumerates the vertices of

__len__(self)

__str__(self)
([Graph](#)) -> str
Returns the string representation of this graph.

accept(self, visitor)

([Graph](#), Visitor) -> None
Makes the given visitor visit all the vertices in this graph.

addVertex(self, *args)

([Graph](#), int [, Object]) -> None
Adds a vertex (with optional weight) to this graph.

breadthFirstTraversal(self, visitor, start)

([Graph](#), Visitor, Vertex) -> None
Makes the given visitor visit the vertices of this graph in breadth-first traversal order starting from the given vertex.

depthFirstTraversal(self, visitor, start)

([Graph](#), PrePostVisitor, Vertex) -> None
Makes the given visitor visit the vertices of this graph in depth-first traversal order starting from the given vertex.

getIsConnected(self)

getIsConnected(self)
`(Graph) -> bool`
Returns true if the graph is connected.

getIsDirected(self)
`(Graph) -> bool`
Returns true if this graph is a directed graph.

getNumberOfEdges(self)
`(Graph) -> int`
Returns the number of edges in this graph.

getNumberOfVertices(self)
`(Graph) -> int`
Returns the number of vertices in this graph.

getVertex(self, v)
`(Graph, int) -> Vertex`
Returns the specified vertex of this graph.

getVertices(self)
`(Graph) -> Graph.Iterator`
Returns an iterator for this graph.

test(g)
`Graph` test program.

testWeighted(g)
Weighted graph test program.

Properties inherited from [opus7.graph.Graph](#):

edges
`get lambda self`

isConnected
`get lambda self`

isCyclic
`get lambda self`

isDirected
`get lambda self`

numberOfEdges
get lambda self

numberOfVertices
get lambda self

vertices
get lambda self

Data and other attributes inherited from [opus7.graph.Graph](#):

CountingVisitor = <class 'opus7.graph.CountingVisitor'>
A visitor that counts the objects it visits.

Edge = <class 'opus7.graph.Edge'>
Represents an edge in a graph.

Iterator = <class 'opus7.graph.Iterator'>
Enumerates the vertices of a graph.

StrVisitor = <class 'opus7.graph.StrVisitor'>
Visitor that accumulates visited items in a string.

Vertex = <class 'opus7.graph.Vertex'>
Represents a vertex in a graph.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)

`(Container) -> bool`
Returns true if this container is empty.

getIsFull(self)
`(Container) -> bool`
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
`(Object, Object) -> int`
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)



```
__author__ = 'Bruno R. Preiss, P.Eng.'  
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
__date__ = '$Date: 2003/09/25 01:07:38 $'  
__version__ = '$Revision: 1.16 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [GraphAsMatrix](#) class.

Modules

[sys](#)

Classes

[opus7.graph.Graph](#)([opus7.container.Container](#))
[GraphAsMatrix](#)

class GraphAsMatrix([opus7.graph.Graph](#))

[Graph](#) implemented using an adjacency matrix.

Method resolution order:

[GraphAsMatrix](#)
[opus7.graph.Graph](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self, size\)](#)
([GraphAsMatrix](#), int) -> None
Constructs a graph with the given maximum number of

[addEdge\(self, *args\)](#)
([GraphAsMatrix](#), int, int [, Object]) -> None
Adds an edge with the (optional) given weight
connecting the given vertices in this graph.

getEdge(self, v, w)
[\(GraphAsMatrix\)](#), int, int) -> Edge
Returns the edge connecting the specified vertices .

getEdges(self)
[\(GraphAsMatrix\)](#) -> [GraphAsMatrix](#).EdgeIterator
Returns an iterator that enumerates the edges in th.

getEmanatingEdges(self, v)
[\(GraphAsMatrix\)](#), int) -> [GraphAsMatrix](#).EmanatingEdge
Returns an iterator that enumerates the edges emanat the given vertex in this graph.

getIncidentEdges(self, w)
[\(GraphAsMatrix\)](#), int) -> [GraphAsMatrix](#).IncidentEdges
Returns an iterator that enumerates the edges incident to the given vertex in this graph.

getIsCyclic(self)
[\(GraphAsMatrix\)](#) -> bool
Returns true if this graph is cyclic.

isEdge(self, v, w)
[\(GraphAsMatrix\)](#), int, int) -> bool
Returns true if there is an edge connecting the speci in this graph.

purge(self)
[\(GraphAsMatrix\)](#) -> None
Purges this graph.

Static methods defined here:

__new__ = new(*args, **kwargs)
[\(Metaclass, ...\)](#) -> object
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
[GraphAsMatrix](#) test program.

Data and other attributes defined here:

EdgeIterator = <class 'opus7.graphAsMatrix.EdgeIterator'>
 Enumerates the edges of a [GraphAsMatrix](#).

EmanatingEdgesIterator = <class
'opus7.graphAsMatrix.EmanatingEdgesIterator'>
 Enumerates the edges emanating from a given vertex .

IncidentEdgesIterator = <class 'opus7.graphAsMatrix.IncidentE
 Enumerates the edges incident upon a given vertex i

__abstractmethods__ = []

Methods inherited from [opus7.graph.Graph](#):

__getitem__(self, v)

__iter__(self)
 ([Graph](#)) -> [Graph](#).Iterator
 Returns an iterator that enumerates the vertices of

__len__(self)

__str__(self)
 ([Graph](#)) -> str
 Returns the string representation of this graph.

accept(self, visitor)
 ([Graph](#), Visitor) -> None
 Makes the given visitor visit all the vertices in t

addVertex(self, *args)
 ([Graph](#), int [, Object]) -> None
 Adds a vertex (with optional weight) to this graph.

breadthFirstTraversal(self, visitor, start)
 ([Graph](#), Visitor, Vertex) -> None
 Makes the given visitor visit the vertices of this
 in breadth-first traversal order starting from the

depthFirstTraversal(self, visitor, start)
 ([Graph](#), PrePostVisitor, Vertex) -> None

Makes the given visitor visit the vertices of this graph in depth-first traversal order starting from the given vertex.

getIsConnected(self)
`(Graph) -> bool`
Returns true if the graph is connected.

getIsDirected(self)
`(Graph) -> bool`
Returns true if this graph is a directed graph.

getNumberOfEdges(self)
`(Graph) -> int`
Returns the number of edges in this graph.

getNumberOfVertices(self)
`(Graph) -> int`
Returns the number of vertices in this graph.

getVertex(self, v)
`(Graph, int) -> Vertex`
Returns the specified vertex of this graph.

getVertices(self)
`(Graph) -> Graph.Iterator`
Returns an iterator for this graph.

test(g)
`Graph` test program.

testWeighted(g)
Weighted graph test program.

Properties inherited from [opus7.graph.Graph](#):

edges
`get lambda self`

isConnected
`get lambda self`

isCyclic
`get lambda self`

isDirected
get lambda self

numberOfEdges
get lambda self

numberOfVertices
get lambda self

vertices
get lambda self

Data and other attributes inherited from [opus7.graph.Graph](#):

CountingVisitor = <class 'opus7.graph.CountingVisitor'>
A visitor that counts the objects it visits.

Edge = <class 'opus7.graph.Edge'>
Represents an edge in a graph.

Iterator = <class 'opus7.graph.Iterator'>
Enumerates the vertices of a graph.

StrVisitor = <class 'opus7.graph.StrVisitor'>
Visitor that accumulates visited items in a string.

Vertex = <class 'opus7.graph.Vertex'>
Represents a vertex in a graph.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getCount(self)
(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.16 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [GraphicalObject](#) class.

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[GraphicalObject](#)

class GraphicalObject([opus7.object.Object](#))

Base class from which all graphical objects are derived.

Method resolution order:

[GraphicalObject](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self, center\)](#)
([GraphicalObject](#), Point) -> None
Constructs a graphical object with the given point .

[draw\(...\)](#)
Abstract method.

[erase\(self\)](#)
([GraphicalObject](#)) -> None
Erases this graphical object.

[moveTo\(self, p\)](#)
([GraphicalObject](#)) -> None
Moves the center of this graphical object to the gi

setPenColor(self, color)

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

test(go)

[GraphicalObject](#) test program.

Data and other attributes defined here:

BACKGROUND_COLOR = 0

FOREGROUND_COLOR = 1

__abstractmethods__ = ['_compareTo', 'draw']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

([Object](#), [Object](#)) -> int

Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)

[Object](#) test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the [Object](#) class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/28 00:06:11 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [HashTable](#) class.

Classes

[opus7.searchableContainer.SearchableContainer](#)([opus7.container.HashTable](#))

class HashTable([opus7.searchableContainer.SearchableContainer](#))

Base class from which all hash tables are derived.

Method resolution order:

[HashTable](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[builtin .object](#)

Methods defined here:

__init__(self)

([HashTable](#)) -> None

Constructs this hash table.

__len__(...)

Abstract method.

f(self, obj)

([HashTable](#), Object) -> int

Returns the hash of the given object.

g(self, x)

([HashTable](#), int) -> int

Hashes an integer using the division method of hash.

getLoadFactor(self)

([HashTable](#)) -> double

Returns the load factor of this hash table.

h(self, obj)

([HashTable](#), Object) -> int

Hashes the specified object

using the composition of the methods f and g.

Static methods defined here:

__new__(*args, **kwargs)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.

Raises a `TypeError` exception if the class is abstract.

This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

test(hashTable)

[HashTable](#) test program.

Properties defined here:

loadFactor

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__contains__', '__iter__', '__len__', '__coerce__', 'find', 'insert', 'purge', 'withdraw']

Methods inherited from [opus7.searchableContainer.SearchableContainer](#)

__contains__(...)

Abstract method.

find(...)

Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods inherited from [opus7.searchableContainer.SearchableContainer](#):

main(*argv)
[SearchableContainer](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool

Returns true if this container is full.

purge(...)

Abstract method.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/23 12:23:21 \$'
version = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [HeapSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[HeapSorter](#)

class **HeapSorter**([opus7.sorter.Sorter](#))

 Heap sorter.

Method resolution order:

[HeapSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self)
 ([HeapSorter](#)) -> None
 Constructor.

buildHeap(self)
 ([HeapSorter](#)) -> None
 Builds the heap.

percolateDown(self, i, length)

__init__(self, array, pos=0)
([HeapSorter](#), int, int) -> None
Percolates the elements in the array with the given
and starting at the given position.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
[HeapSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)
([Sorter](#), Array) -> None
Sorts the given array.

swap(self, i, j)
([Sorter](#), int, int) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.MetaClass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:37:26 \$'
__version__ = '\$Revision: 1.9 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [InOrder](#) class.

Modules

[sys](#)

Classes

[opus7.prePostVisitor.PrePostVisitor](#)([opus7.visitor.Visitor](#))
[InOrder](#)

class [InOrder](#)([opus7.prePostVisitor.PrePostVisitor](#))

Adapter to convert a Visitor to a [PrePostVisitor](#) for in-order traversal.

Method resolution order:

[InOrder](#)
[opus7.prePostVisitor.PrePostVisitor](#)
[opus7.visitor.Visitor](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

[__init__\(self, visitor\)](#)
([InOrder](#), Visitor) -> None
Constructs a in-order visitor from the given visitor.

[getIsDone\(self\)](#)
([InOrder](#)) -> bool
Returns true if the visitor is done.

inVisit(self, obj)
([InOrder](#), Object) -> None
In-visits the given object.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)
[InOrder](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.prePostVisitor.PrePostVisitor](#):

postVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default post-visit method does nothing.

preVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default pre-visit method does nothing.

visit = inVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default in-visit method does nothing.

Properties inherited from [opus7.visitor.Visitor](#):

isDone
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.14 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Integer](#) class.

Modules

[sys](#)

[warnings](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Integer](#)([__builtin__.int](#), [opus7.object.Object](#))
[__builtin__.int](#)([__builtin__.object](#))
[Integer](#)([__builtin__.int](#), [opus7.object.Object](#))

class [Integer](#)([__builtin__.int](#), [opus7.object.Object](#))

[Integer](#) class.

Method resolution order:

[Integer](#)
[__builtin__.int](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__hash__\(self\)](#)
([Integer](#)) -> [int](#)
Hashes this string.

[__init__\(self, obj\)](#)
([Integer](#), [object](#)) -> None

Constructs a string with the string representation.
The [Object](#) metaclass provides a `__new__` method
that initializes the str instance, so none is defined.

Static methods defined here:

`__new__` = `new(*args, **kwargs)`
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

`main(*argv)`
[Integer](#) test program.

`testHash()`
[Integer](#) hash test program.

Data and other attributes defined here:

`__abstractmethods__` = []

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__weakref__` = <attribute '__weakref__' of 'Integer' objects>
list of weak references to the object (if defined)

Methods inherited from [builtin .int](#):

`__abs__(...)`
`x.__abs__()` <==> `abs(x)`

`__add__(...)`
`x.__add__(y)` <==> `x+y`

`__and__(...)`
`x.__and__(y)` <==> `x&y`

`__cmp__(...)`

`x.cmp(y) <==> cmp(x, y)`

`coerce(...)`
`x.coerce(y) <==> coerce(x, y)`

`div(...)`
`x.div(y) <==> x/y`

`divmod(...)`
`x.divmod(y) <==> xdivmod(x, y)y`

`float(...)`
`x.float() <==> float(x)`

`floordiv(...)`
`x.floordiv(y) <==> x//y`

`getattribute(...)`
`x.getattribute('name') <==> x.name`

`getnewargs(...)`

`hex(...)`
`x.hex() <==> hex(x)`

`int(...)`
`x.int() <==> int(x)`

`invert(...)`
`x.invert() <==> ~x`

`long(...)`
`x.long() <==> long(x)`

`lshift(...)`
`x.lshift(y) <==> x<<y`

`mod(...)`
`x.mod(y) <==> x%y`

`mul(...)`
`x.mul(y) <==> x*y`

`neg(...)`

`x.neg() <==> -x`

`nonzero(...)`
`x.nonzero() <==> x != 0`

`oct(...)`
`x.oct() <==> oct(x)`

`or(...)`
`x.or(y) <==> x|y`

`pos(...)`
`x.pos() <==> +x`

`pow(...)`
`x.pow(y[, z]) <==> pow(x, y[, z])`

`radd(...)`
`x.radd(y) <==> y+x`

`rand(...)`
`x.rand(y) <==> y&x`

`rdiv(...)`
`x.rdiv(y) <==> y/x`

`rdivmod(...)`
`x.rdivmod(y) <==> ydivmod(y, x)x`

`repr(...)`
`x.repr() <==> repr(x)`

`rfloordiv(...)`
`x.rfloordiv(y) <==> y//x`

`rlshift(...)`
`x.rlshift(y) <==> y<<x`

`rmod(...)`
`x.rmod(y) <==> y%x`

`rmul(...)`
`x.rmul(y) <==> y*x`

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

Data and other attributes inherited from [opus7.object.Object](#):

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Base class from which all container iterators are derived.

Modules

[sys](#)

Classes

[opus7.object.Object\(__builtin__.object\)](#)
[Iterator](#)

class Iterator(opus7.object.Object)

Base class from which all container iterators are derived

Method resolution order:

[Iterator](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self, container\)](#)
([Iterator](#), Container) -> None

Constructs an iterator for the given container.

[__iter__\(self\)](#)
([Iterator](#)) -> [Iterator](#)

Returns this iterator.

next(...)

Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)

[Iterator](#) test program.

Data and other attributes defined here:

__abstractmethods__ = ['next']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

([Object](#), [Object](#)) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the [Object](#) class.

Prevents instantiation of classes that contain abstract methods.

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:15 \$'
version = '\$Revision: 1.3 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [LeftistHeap](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.binaryTree.BinaryTree](#)([opus7.tree.Tree](#))
[LeftistHeap](#)([opus7.binaryTree.BinaryTree](#),
[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#))
[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#)([opus7.p](#)
[LeftistHeap](#)([opus7.binaryTree.BinaryTree](#),
[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#))

class **LeftistHeap**([opus7.binaryTree.BinaryTree](#),
[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#))

Mergeable priority queue implemented as a leftist heap b:

Method resolution order:

[LeftistHeap](#)
[opus7.binaryTree.BinaryTree](#)
[opus7.tree.Tree](#)
[opus7.mergeablePriorityQueue.MergeablePriorityQueue](#)
[opus7.priorityQueue.PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

`__init__(self, *args)`
`(LeftistHeap [, key]) -> None`
Constructor.

`dequeueMin(self)`
`(LeftistHeap) -> Object`
Dequeues and returns the object in this leftist heap with the smallest value.

`enqueue(self, obj)`
`(LeftistHeap, Object) -> None`
Enqueues the given object in this leftist heap.

`getMin(self)`
`(LeftistHeap) -> Object`
Returns the object in this leftist heap with the smallest value.

`merge(self, queue)`
`(LeftistHeap, LeftistHeap) -> None`
Merges the contents of the given leftist heap with this leftist heap.

`swapContentsWith(self, heap)`
`(LeftistHeap, LeftistHeap) -> None`
Swaps the contents of this leftist heap with the given leftist heap.

`swapSubtrees(self)`
`(LeftistHeap) -> None`
Swaps the subtrees of this leftist heap.

Static methods defined here:

`__new__ = new(*args, **kwargs)`
`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

`main(*argv)`
`LeftistHeap test program.`

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.binaryTree.BinaryTree](#):

attachKey(self, obj)

([BinaryTree](#), Object) -> None

Makes the given object the key of this binary tree

attachLeft(self, t)

([BinaryTree](#), Binary Tree) -> None

Attaches the given binary tree as the left subtree

attachRight(self, t)

([BinaryTree](#), Binary Tree) -> None

Attaches the given binary tree as the right subtree

depthFirstGenerator(self, mode)

([BinaryTree](#)) -> generator

Yields the keys in this tree in depth-first traversal

depthFirstTraversal(self, visitor)

([BinaryTree](#), PrePostVisitor) -> None

Makes the given visitor do a depth-first traversal

detachKey(self)

([BinaryTree](#)) -> None

Detaches and returns the key in this binary tree node

detachLeft(self)

([BinaryTree](#)) -> [BinaryTree](#)

Detaches and returns the left subtree of this binary tree

detachRight(self)

([BinaryTree](#)) -> [BinaryTree](#)

Detaches and returns the right subtree of this binary tree

getDegree(self)

([BinaryTree](#)) -> int

Returns the degree of this binary tree node.

getIsEmpty(self)

([BinaryTree](#)) -> bool

Returns true if this binary tree is empty.

getIsLeaf(self)
`(BinaryTree) -> bool`
Returns true if this binary tree node is a leaf.

getKey(self)
`(BinaryTree) -> Object`
Returns the key in this binary tree node.

getLeft(self)
`(BinaryTree) -> BinaryTree`
Returns the left subtree of this binary tree.

getRight(self)
`(BinaryTree) -> BinaryTree`
Returns the right subtree of this binary tree.

getSubtree(self, i)
`(BinaryTree, int) -> Object`
Returns the specified subtree of this binary tree.

purge(self)
`(BinaryTree) -> None`
Purges this binary tree.

Properties inherited from [opus7.binaryTree.BinaryTree](#):

left
get lambda self

right
get lambda self

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)
`(Tree) -> Tree.Iterator`
Returns an interator for this tree.

accept(self, visitor)
`(Tree) -> Visitor`
Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)
(Tree) -> generator
Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)
(Tree, Visitor) -> None
Makes the given visitor do a breadth-first traversal.

getCount(self)
(Tree) -> int
Returns the number of nodes in this tree.

getHeight(self)
(Tree) -> int
Returns the height of this tree.

Static methods inherited from [opus7.tree.Tree](#):

test(tree)
Tree test program.

Properties inherited from [opus7.tree.Tree](#):

degree
get lambda self

height
get lambda self

isLeaf
get lambda self

key
get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Properties inherited from [opus7.priorityQueue.PriorityQueue](#):

min

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [LinkedList](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[__builtin__.object](#)
[LinkedList](#)

class LinkedList([__builtin__.object](#))

Linked list class.

Methods defined here:

__copy__(self)
[\(LinkedList\)](#) -> [LinkedList](#)
Returns a shallow copy of this linked list.

__init__(self)
[\(LinkedList\)](#) -> None
Constructs an empty linked list.

__str__(self)
[\(LinkedList\)](#) -> string
Returns a string representation of this list.

append(self, item)
[\(LinkedList, Object\)](#) -> None
Appends the given item to this list.

extract(self, item)

remove(self, item)
`(LinkedList, Object) -> None`
Extracts the given item from this list.

getFirst(self)
`(LinkedList) -> Object`
Returns the first item in this list.

getHead(self)
`(LinkedList) -> LinkedList.Element`
Returns the list element at the head of this list.

getIsEmpty(self)
`(LinkedList) -> bool`
Returns true if this list is empty.

getLast(self)
`(LinkedList) -> Object`
Returns the last item in this list.

getTail(self)
`(LinkedList) -> LinkedList.Element`
Returns the list element at the tail of this list.

prepend(self, item)
`(LinkedList, Object) -> None`
Prepends the given item to this list.

purge(self)
`(LinkedList) -> None`
Purges this linked list.

Static methods defined here:

main(*argv)
`LinkedList test program.`

Properties defined here:

first
`get lambda self`

head
`get lambda self`

isEmpty
get lambda self

last
get lambda self

tail
get lambda self

Data and other attributes defined here:

Element = <class 'opus7.linkedList.Element'>
An element of a linked list.

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'LinkedList' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.long](#)

(version 1.1, \$Date:
2003/09/19
22:24:51 \$)

[index](#)

</home/brpreiss/books/opus7/sources/opus7/long.py>

Provides the [Long](#) class.

Modules

[sys](#)

[warnings](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Long](#)([__builtin__.long](#), [opus7.object.Object](#))
[__builtin__.long](#)([__builtin__.object](#))
[Long](#)([__builtin__.long](#), [opus7.object.Object](#))

class [Long](#)([__builtin__.long](#), [opus7.object.Object](#))

[Long](#) class.

Method resolution order:

[Long](#)
[__builtin__.long](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__hash__\(self\)](#)
([Long](#)) -> [long](#)
Hashes this string.

__init__(self, obj)
([Long](#), object) -> None
Constructs a string with the string representation.
The [Object](#) metaclass provides a __new__ method
that initializes the str instance, so none is defined.

__compareTo(self, obj)
([Long](#), [Long](#)) -> [long](#)

Compares this string with the given string.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a [TypeError](#) exception if the class is abstract.
This method is inserted as the method __new__
in classes instances derived from Metaclass.

main(*argv)
[Long](#) test program.

testHash()
[Long](#) hash test program.

Data and other attributes defined here:

__abstractmethods__ = []

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

Methods inherited from [builtin.long](#):

__abs__(...)
x.[__abs__](#)() <==> abs(x)

__add__(...)
x.[__add__](#)(y) <==> x+y

- __and__(...)**
x.and(y) \iff x&y
- __cmp__(...)**
x.cmp(y) \iff cmp(x, y)
- __coerce__(...)**
x.coerce(y) \iff coerce(x, y)
- __div__(...)**
x.div(y) \iff x/y
- __divmod__(...)**
x.divmod(y) \iff xdivmod(x, y)y
- __float__(...)**
x.float() \iff float(x)
- __floordiv__(...)**
x.floordiv(y) \iff x//y
- __getattribute__(...)**
x.getattribute('name') \iff x.name
- __getnewargs__(...)**
- __hex__(...)**
x.hex() \iff hex(x)
- __int__(...)**
x.int() \iff int(x)
- __invert__(...)**
x.invert() \iff ~x
- __long__(...)**
x.long() \iff long(x)
- __lshift__(...)**
x.lshift(y) \iff x<<y
- __mod__(...)**
x.mod(y) \iff x%y

- __mul__**(...)
x.__mul__(y) \iff x^*y
- __neg__**(...)
x.__neg__() \iff $-x$
- __nonzero__**(...)
x.__nonzero__() \iff $x \neq 0$
- __oct__**(...)
x.__oct__() \iff $\text{oct}(x)$
- __or__**(...)
x.__or__(y) \iff $x|y$
- __pos__**(...)
x.__pos__() \iff $+x$
- __pow__**(...)
x.__pow__(y[, z]) \iff $\text{pow}(x, y[, z])$
- __radd__**(...)
x.__radd__(y) \iff $y+x$
- __rand__**(...)
x.__rand__(y) \iff $y\&x$
- __rdiv__**(...)
x.__rdiv__(y) \iff y/x
- __rdivmod__**(...)
x.__rdivmod__(y) \iff $y\text{divmod}(y, x)x$
- __repr__**(...)
x.__repr__() \iff $\text{repr}(x)$
- __rfloordiv__**(...)
x.__rfloordiv__(y) \iff $y//x$
- __rlshift__**(...)
x.__rlshift__(y) \iff $y<<x$
- __rmod__**(...)
x.__rmod__(y) \iff $y\%x$

__rmul__(...)
x.__rmul__(y) <==> y*x

__ror__(...)
x.__ror__(y) <==> y|x

__rpow__(...)
y.__rpow__(x[, z]) <==> pow(x, y[, z])

__rrshift__(...)
x.__rrshift__(y) <==> y>>x

__rshift__(...)
x.__rshift__(y) <==> x>>y

__rsub__(...)
x.__rsub__(y) <==> y-x

__rtruediv__(...)
x.__rtruediv__(y) <==> y/x

__rxor__(...)
x.__rxor__(y) <==> y^x

__str__(...)
x.__str__() <==> str(x)

__sub__(...)
x.__sub__(y) <==> x-y

__truediv__(...)
x.__truediv__(y) <==> x/y

__xor__(...)
x.__xor__(y) <==> x^y

Data and other attributes inherited from [opus7.object.Object](#):

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>



list of weak references to the object (if defined)

Data

```
__author__ = 'Bruno R. Preiss, P.Eng.'  
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
__date__ = '$Date: 2003/09/19 22:24:51 $'  
__version__ = '$Revision: 1.1 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [MWayTree](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.searchTree.SearchTree](#)([opus7.tree.Tree](#),
[opus7.searchableContainer.SearchableContainer](#))
[MWayTree](#)

class **MWayTree**([opus7.searchTree.SearchTree](#))

M-way tree implemented using arrays.

Method resolution order:

[MWayTree](#)
[opus7.searchTree.SearchTree](#)
[opus7.tree.Tree](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin_object](#)

Methods defined here:

__contains__(self, obj)
([MWayTree](#), [Object](#)) -> [bool](#)
Returns true if the given object is in this M-way t

__init__(self, m)
`(MWayTree, int) -> None`
Constructs an empty M-way tree.

__iter__(self)
`(MWayTree) -> MWayTree.Iterator`
Returns an iterator for this M-way tree.

breadthFirstGenerator(self)
`(BinaryTree) -> generator`
Yields the keys in this tree in breadth-first traversal.

breadthFirstTraversal(self, visitor)
`(MWayTree, Visitor) -> None`
Makes the given visitor do a breadth-first traversal of this M-way tree.

depthFirstGenerator(self, mode)
`(BinaryTree) -> generator`
Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)
`(MWayTree, PrePostVisitor) -> None`
Makes the given visitor do a depth-first traversal of this M-way tree.

find(self, obj)
`(MWayTree, Object):`
Returns the object in this M-way tree that matches the given object.

findIndex(self, obj)
`(MWayTree, Object) -> int`
Returns the position of the specified object in the array of keys contained in this M-way tree node.
Uses a binary search.

getCount(self)
`(MWayTree) -> int`
Returns the number of keys in this M-way tree node.

getDegree(self)
`(MWayTree) -> int`
Returns the degree of this M-way tree node.

getIsEmpty(self)
`(MWayTree) -> bool`
Returns true if this M-way tree is empty.

getIsFull(self)
`(MWayTree) -> bool`
Returns true if this M-way tree is full.

getIsLeaf(self)
`(MWayTree) -> bool`
Returns true if this M-way tree is a leaf.

getKey(self, *args)
`(MWayTree, ...) -> Object`
Returns the specified key of this M-way tree node.

getM(self)
`(MWayTree) -> int`
Returns the value of M for this M-way tree.

getMax(self)
`(MWayTree) -> Object`
Returns the object in this M-way tree with the largest key.

getMin(self)
`(MWayTree) -> Object`
Returns the object in this M-way tree with the smallest key.

getSubtree(self, i)
`(MWayTree, int) -> MWayTree`
Returns the specified subtree of this M-way tree node.

insert(self, obj)
`(MWayTree, Object) -> None`
Inserts the given object into this M-way tree.

purge(self)
`(MWayTree) -> None`
Purges this M-way tree.

withdraw(self, obj)
`(MWayTree, Object) -> None`
Withdraws the given object from this M-way tree.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[MWayTree](#) test program.

Properties defined here:

m

get lambda self

Data and other attributes defined here:

Iterator = <class 'opus7.mWayTree.Iterator'>
Enumerates the objects in an M-way tree.

__abstractmethods__ = []

Static methods inherited from [opus7.searchTree.SearchTree](#):

test(tree)
[SearchTree](#) test program.

Properties inherited from [opus7.searchTree.SearchTree](#):

max

get lambda self

min

get lambda self

Methods inherited from [opus7.tree.Tree](#):

accept(self, visitor)
(Tree) -> Visitor
Makes the given visitor visit the nodes of this tree.

getHeight(self)
(Tree) -> int
Returns the height of this tree.

Properties inherited from [opus7.tree.Tree](#):

degree
get lambda self

height
get lambda self

isLeaf
get lambda self

key
get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/26 15:00:48 \$'
version = '\$Revision: 1.33 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Matrix](#) class.

Classes

[__builtin__.object](#) [Matrix](#)

class Matrix([__builtin__.object](#))

Base class from which all matrices are derived.

Methods defined here:

__init__(self, numberOfRows, numberOfColumns)
([Matrix](#), int, int) -> None
Constructs this matrix.

getNumberOfColumns(self)

getNumberOfRows(self)

Static methods defined here:

test(mat)
[Matrix](#) test program.

testTimes(mat1, mat2)
[Matrix](#) multiply test program.

testTranspose(mat)
[Matrix](#) transpose test program.

Properties defined here:

numberOfColumns
get lambda self

numberOfRows
get lambda self

transpose
get lambda self

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Matrix' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:00:48 \$'
__version__ = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.medianOfThreeQuickSorter](#)

(version 1.7, \$Date: 2003/09/06 16:35:15

\$)

[opus7/medianOfThreeQuickSorter](#)

Provides the [MedianOfThreeQuickSorter](#) class.

Modules

[sys](#)

Classes

[opus7.quickSorter.QuickSorter](#)([opus7.sorter.Sorter](#)) [MedianOfThreeQuickSorter](#)

class **MedianOfThreeQuickSorter**([opus7.quickSorter.QuickSorter](#))

Quick sorter that uses median-of-three pivot selection.

Method resolution order:

[MedianOfThreeQuickSorter](#)
[opus7.quickSorter.QuickSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__init__(self)
([MedianOfThreeQuickSorter](#)) -> None
Constructor.

selectPivot(self, left, right)
([MedianOfThreeQuickSorter](#), int, int) -> int

Sorts the left, middle, and right array element
and returns the position of the middle element as t

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)

[MedianOfThreeQuickSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.quickSorter.QuickSorter](#):

quicksort(self, left, right)

([QuickSorter](#), left, right) -> None

Recursively sorts the elements of the array between

Data and other attributes inherited from [opus7.quickSorter.QuickSorter](#):

CUTOFF = 2

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)

(Sorter, Array) -> None

Sorts the given array.

swap(self, i, j)

(Sorter, int, int) -> None

Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'

__date__ = '\$Date: 2003/09/06 16:35:15 \$'

__version__ = '\$Revision: 1.7 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.mergeablePriorityQueue](#)

(version 1.9, \$Date: 2003/09/06
16:35:15 \$)

[index](#)

[opus7/mergeablePriorityQueue.py](#)

Provides the [MergeablePriorityQueue](#) class.

Classes

[opus7.priorityQueue.PriorityQueue](#)([opus7.container.Container](#))
[MergeablePriorityQueue](#)

class **MergeablePriorityQueue**([opus7.priorityQueue.PriorityQueue](#))

A mergeable priority queue.

Method resolution order:

[MergeablePriorityQueue](#)
[opus7.priorityQueue.PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self)
([MergeablePriorityQueue](#)) -> None
Constructor.

merge(...)
Abstract method.

test(pqueue)
[MergeablePriorityQueue](#) test program.

Static methods defined here:

```
__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object
```

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Data and other attributes defined here:

```
__abstractmethods__ = ['__iter__', '_compareTo', 'dequeueMin', 'getMin', 'merge', 'purge']
```

Methods inherited from [opus7.priorityQueue.PriorityQueue](#):

dequeueMin(...)
Abstract method.

enqueue(...)
Abstract method.

getMin(...)
Abstract method.

Properties inherited from [opus7.priorityQueue.PriorityQueue](#):

min
get lambda self

Methods inherited from [opus7.container.Container](#):

```
__hash__(self)
(Container) -> int
Returns the hash of this container.
```

__iter__(...)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)
Container test program.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.9 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides a metaclass for the Object class.

Modules

[sets](#)

[string](#)

[sys](#)

Classes

[__builtin__.type\(__builtin__.object\)](#)
[MetaClass](#)

class MetaClass(__builtin__.type)

[MetaClass](#) of the Object class.
Prevents instantiation of classes that contain abstract methods.

Method resolution order:

[MetaClass](#)
[__builtin__.type](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self, name, bases, dict\)](#)
([MetaClass](#), str, tuple, mapping) -> None

Initializes this metaclass instance.

Static methods defined here:

[**main\(*argv\)**](#)

[Metaclass](#) test program.

new(*args, **kwargs)
([Metaclass](#), ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from [Metaclass](#).

Methods inherited from [builtin .type](#):

__call__(...)
x.[call](#)(...) <==> x(...)

__cmp__(...)
x.[cmp](#)(y) <==> cmp(x, y)

__delattr__(...)
x.[delattr](#)('name') <==> del x.name

__getattribute__(...)
x.[getattribute](#)('name') <==> x.name

__hash__(...)
x.[hash](#)() <==> hash(x)

__repr__(...)
x.[repr](#)() <==> repr(x)

__setattr__(...)
x.[setattr](#)('name', value) <==> x.name = value

__subclasses__(...)
[subclasses](#)() -> list of immediate subclasses

mro(...)
[mro](#)() -> list
return a [type](#)'s method resolution order

Data and other attributes inherited from [builtin .type](#):

__base__ = <type 'type'>

```
__bases__ = (<type 'type'>,)

__basicsize__ = 420

__dict__ = <dictproxy object>

__dictoffset__ = 132

__flags__ = 22523

__itemsize__ = 20

__mro__ = (<class 'opus7.metaclass.Metaclass'>, <type 'type'>, <t  
'object'>)

__new__ = <built-in method __new__ of type object>
    T.new(S, ...) -
        > a new object with type S, a subtype of T

__weakrefoffset__ = 184
```

Data

```
__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '$Date: 2003/09/06 16:35:15 $'
__version__ = '$Revision: 1.15 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.multiDimensionalArray](#)

(version 1.11, \$Date: 2003/09/06
16:35:15 \$)

[index](#)

[opus7/multiDimensionalArray.py](#)

Provides the [MultiDimensionalArray](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[MultiDimensionalArray](#)

class MultiDimensionalArray([__builtin__.object](#))

Multi-dimensional array implemented using a one-dimensional array.

Methods defined here:

[__getitem__\(self, indices\)](#)
([MultiDimensionalArray](#), Array) -> Object
Returns the [object](#) in this multi-dimensional array at the given indices.

[__init__\(self, *dimensions\)](#)
([MultiDimensionalArray](#), Array) -> None
Constructs a multi-dimensional array with the given dimensions.

[__setitem__\(self, indices, value\)](#)
([MultiDimensionalArray](#), Array) -> Object
Sets the [object](#) in this multi-dimensional array at the given indices to the given value.

__str__(self)
([MultiDimensionalArray](#)) -> str
Returns a string representation of this multi-dimensional array.

getOffset(self, indices)
([MultiDimensionalArray](#), Array) -> int
Maps the given indices of this multi-dimensional array into a one-dimensional array index.

Static methods defined here:

main(*argv)
[MultiDimensionalArray](#) test program.

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of
'MultiDimensionalArray' objects>
list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.11 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Multiset](#) class.

Classes

[opus7.set.Set](#)([opus7.searchableContainer.SearchableContainer](#))
[Multiset](#)

class Multiset([opus7.set.Set](#))

Base class from which all multiset classes are derived.

Method resolution order:

[Multiset](#)
[opus7.set.Set](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, universeSize)
([Multiset](#), int) -> None
Constructs a multiset with the given universe size.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given args.
Raises a TypeError exception if the class is abstract.

This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(s1, s2, s3)
[Multiset](#) test program.

Data and other attributes defined here:

`__abstractmethods__` = ['`__and__`', '`__contains__`', '`__eq__`', '`__it__`', '`__or__`', '`__sub__`', '`_compareTo`', '`insert`', '`purge`', '`withdraw`']

Methods inherited from [opus7.set.Set](#):

`__and__(...)`
Abstract method.

`__eq__(...)`
Abstract method.

`__lt__(...)`
Abstract method.

`__or__(...)`
Abstract method.

`__sub__(...)`
Abstract method.

find(self, i)
([Set](#), int) -> int

Returns the given integer if it is in this set.
Returns None otherwise.

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize
get lambda self

Methods inherited from [opus7.searchableContainer.SearchableCor](#)

__contains__(...)
Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods inherited from [opus7.searchableContainer.Searchat](#)

main(*argv)
SearchableContainer test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.6 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [MultisetAsArray](#) class.

Modules

[sys](#)

Classes

[opus7.multiset.Multiset](#)([opus7.set.Set](#))
[MultisetAsArray](#)

class **MultisetAsArray**([opus7.multiset.Multiset](#))

[Multiset](#) implemented using an array of counters.

Method resolution order:

[MultisetAsArray](#)
[opus7.multiset.Multiset](#)
[opus7.set.Set](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__and__(self, set)
([MultisetAsArray](#), [MultisetAsArray](#)) -> [MultisetAsArray](#)
Returns the intersection of this multiset and the given multiset.

__contains__(self, item)

contains(self, item)
([MultisetAsArray](#), Object) -> bool
Returns true if the given element is in this multiset.

eq(set)
([MultisetAsArray](#), [MultisetAsArray](#)) -> bool
Returns true if this equals the given multiset.

init(self, n)
([MultisetAsArray](#), int) -> None
Constructs a multiset with the given universe size.

iter(self)
([MultisetAsArray](#)) -> [MultisetAsArray](#).Iterator
Returns an iterator for the given multiset.

lt(self, set)
([MultisetAsArray](#), [MultisetAsArray](#)) -> bool
Returns true if this multiset is a proper subset of the given multiset.

or(self, set)
([MultisetAsArray](#), [MultisetAsArray](#)) -> [MultisetAsArray](#)
Returns the union of this multiset and the given multiset.

sub(self, set)
([MultisetAsArray](#), [MultisetAsArray](#)) -> [MultisetAsArray](#)
Returns the difference of this multiset and the given multiset.

accept(self, visitor)
([MultisetAsArray](#), Visitor) -> None
Makes the given visitor visit all the elements in this multiset.

getCount(self)
([MultisetAsArray](#)) -> int
Returns the number of elements in this multiset.

insert(self, item)
([MultisetAsArray](#), Object) -> None
Inserts the given element into this multiset.

purge(self)
([MultisetAsArray](#)) -> None
Purges this multiset.

withdraw(self, item)
([MultisetAsArray](#), Object) -> None
Removes the given element from this multiset.

Withdraws the given element from this multiset.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)
[MultisetAsArray](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.multisetAsArray.Iterator'>
Enumerates the elements of a [MultisetAsArray](#).

__abstractmethods__ = []

Static methods inherited from [opus7.multiset.Multiset](#):

test(s1, s2, s3)
[Multiset](#) test program.

Methods inherited from [opus7.set.Set](#):

find(self, i)
(Set, int) -> int

Returns the given integer if it is in this set.
Returns None otherwise.

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize

get *lambda self*

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.21 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.multisetAsLinkedList](#)

(version 1.20, \$Date: 2003/09/26
14:21:18 \$)

[index](#)

[opus7/multisetAsLinkedList.py](#)

Provides the [MultisetAsLinkedList](#) class.

Modules

[sys](#)

Classes

[opus7.multiset.Multiset](#)([opus7.set.Set](#))
[MultisetAsLinkedList](#)

class **MultisetAsLinkedList**([opus7.multiset.Multiset](#))

[Multiset](#) implemented using a linked list of elements.

Method resolution order:

[MultisetAsLinkedList](#)
[opus7.multiset.Multiset](#)
[opus7.set.Set](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin_object](#)

Methods defined here:

[__and__\(self, set\)](#)

([MultisetAsLinkedList](#), [MultisetAsLinkedList](#)) -> [MultisetAsLinkedList](#)

Returns the intersection of this multiset and the given multiset.

__contains__(self, item)
[\(MultisetAsLinkedList, Object\)](#) -> bool
Returns true if the given elements is in this multiset.

__eq__(self, set)
[\(MultisetAsLinkedList, MultisetAsLinkedList\)](#) -> bool
Returns true if this multiset is equal to the given multiset.

__init__(self, n)
[\(MultisetAsLinkedList, int\)](#) -> None
Constructs a multiset with the given universe size.

__iter__(self)
[\(MultisetAsLinkedList\)](#) -> [MultisetAsLinkedList.Iterator](#)
Returns an iterator for this multiset.

__lt__(self, set)
[\(MultisetAsLinkedList, MultisetAsLinkedList\)](#) -> bool
Returns true if this multiset is a proper subset of the given multiset.

__or__(self, set)
[\(MultisetAsLinkedList, MultisetAsLinkedList\)](#) -> [MultisetAsLinkedList](#)
Returns the union of this multiset and the given multiset.

__sub__(self, set)
[\(MultisetAsLinkedList, MultisetAsLinkedList\)](#) -> [MultisetAsLinkedList](#)
Returns the difference of this multiset and the given multiset.

accept(self, visitor)
[\(MultisetAsLinkedList, Visitor\)](#) -> None
Makes the given visitor visit all the elements in this multiset.

getCount(self)
[\(MultisetAsLinkedList\)](#) -> int
Returns the number of elements in this multiset.

insert(self, item)
[\(MultisetAsLinkedList, Object\)](#) -> None
Inserts the given element into this multiset.

purge(self)
[\(MultisetAsLinkedList\)](#) -> None
Purges this multiset.

withdraw(self, item)

[MultisetAsLinkedList](#), Object) -> None
Withdraws the given element from this multiset.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)
[MultisetAsLinkedList](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.multisetAsLinkedList.Iterator'>
Enumerates the elements of a [MultisetAsLinkedList](#).

__abstractmethods__ = []

Static methods inherited from [opus7.multiset.Multiset](#):

test(s1, s2, s3)
[Multiset](#) test program.

Methods inherited from [opus7.set.Set](#):

find(self, i)
(Set, int) -> int

Returns the given integer if it is in this set.
Returns None otherwise.

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.20 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [NaryTree](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.tree.Tree](#)([opus7.container.Container](#))
[NaryTree](#)

class **NaryTree**([opus7.tree.Tree](#))

N-ary tree implemented using an array of._subtrees.

Method resolution order:

[NaryTree](#)
[opus7.tree.Tree](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, *args)
([NaryTree](#), int [, Object]) -> None
Constructs an N-ary tree.

attachKey(self, obj)
([NaryTree](#), Object) -> None
Makes the given object the._key of this N-ary tree |

attachSubtree(self, i, t)
`(NaryTree, int, NaryTree) -> None`
Attaches the given tree as the specified._subtree of this N-ary tree.

detachKey(self)
`(NaryTree) -> Object`
Detaches and returns the._key of this N-ary tree node.

detachSubtree(self, i)
`(NaryTree, int) -> NaryTree`
Detaches and returns the specified._subtree of this N-ary tree.

getDegree(self)
`(NaryTree) -> int`
Returns the degree of this N-ary tree.

getIsEmpty(self)
`(NaryTree) -> bool`
Returns true if this N-ary tree is empty.

getIsLeaf(self)
`(NaryTree) -> bool`
Returns true if this N-ary tree is a leaf.

getKey(self)
`(NaryTree) -> Object`
Returns the._key of this N-ary tree node.

getSubtree(self, i)
`(NaryTree, int) -> NaryTree`
Returns the specified._subtree of this N-ary tree.

purge(self)
`(NaryTree) -> None`
Purges this N-ary tree.

Static methods defined here:

__new__ = new(*args, **kwargs)
`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`.

in classes instances derived from Metaclass.

main(*argv)
[NaryTree](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)
([Tree](#)) -> [Tree](#).Iterator
Returns an interator for this tree.

accept(self, visitor)
([Tree](#)) -> Visitor
Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)
([Tree](#)) -> generator
Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)
([Tree](#), Visitor) -> None
Makes the given visitor do a breadth-first traversal of this tree.

depthFirstGenerator(self, mode)
([Tree](#)) -> generator
Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)
([Tree](#), PrePostVisitor) -> None
Makes the given visitor do a depth-first traversal of this tree.

getCount(self)
([Tree](#)) -> int
Returns the number of nodes in this tree.

getHeight(self)
([Tree](#)) -> int
Returns the height of this tree.

Static methods inherited from [opus7.tree.Tree](#):

test(tree)
 [Tree](#) test program.

Properties inherited from [opus7.tree.Tree](#):

degree
 get lambda self

height
 get lambda self

isLeaf
 get lambda self

key
 get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
 Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.container.Container](#):

__hash__(self)
 (Container) -> int
 Returns the hash of this container.

__str__(self)
 (Container) -> string
 Returns a string representation of this container.

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.21 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Object](#) class.

Modules

[sys](#)

Classes

[__builtin__.object](#)
[Object](#)

class Object([__builtin__.object](#))

Base class from which all objects are derived.

Methods defined here:

[__cmp__\(self, obj\)](#)
[\(Object, Object\)](#) -> int
Compares this [object](#) with the given [object](#).

[__init__\(self\)](#)
[\(Object\)](#) -> None
Constructor.

Static methods defined here:

[__new__ = new\(*args, **kwargs\)](#)
[\(Metaclass, ...\)](#) -> [object](#)

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
Object test program.

Data and other attributes defined here:

__abstractmethods__ = ['_compareTo']

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.25 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[index](#)
[opus7/openScatterTable.py](#)

Provides the [OpenScatterTable](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.hashTable.HashTable](#)([opus7.searchableContainer.SearchableContainer](#))
[OpenScatterTable](#)

class **OpenScatterTable**([opus7.hashTable.HashTable](#))

Hash table implemented as an open scatter table using an array of buckets.

Method resolution order:

[OpenScatterTable](#)
[opus7.hashTable.HashTable](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__contains__(self, obj)
([OpenScatterTable](#), Object) -> bool
Returns true if the given object is in this open scatter table.

__init__(self, length)
([OpenScatterTable](#), int) -> None

Constructs an open scatter table of the given length.

__iter__(self)
`(OpenScatterTable) -> OpenScatterTable.Iterator`
Returns an interator for this open scatter table.

__len__(self)
`(OpenScatterTable) -> None`
Returns the length of this open scatter table.

accept(self, visitor)
`(OpenScatterTable, Visitor) -> None`
Makes the given visitor visit all the objects in this open scatter table.

c(self, i)
`(OpenScatterTable, int) -> int`
Linear probing function.

find(self, obj)
`(OpenScatterTable, Object) -> Object`
Returns the object in this open scatter table that matches the given object.

findInstance(self, obj)
`(OpenScatterTable, Object) -> int`
Finds the index of the given object in this open scatter table.

findMatch(self, obj)
`(OpenScatterTable, Object) -> int`
Finds the index of an object in this open scatter table that matches the given object.

findUnoccupied(self, obj)
`(OpenScatterTable, Object) -> int`
Returns the index of an unoccupied entry in this open scatter table.

getIsFull(self)
`(OpenScatterTable) -> bool`
Returns true if this open scatter table is full.

insert(self, obj)
`(OpenScatterTable, Object) -> None`
Inserts the given object into this open scatter table.

purge(self)
([OpenScatterTable](#)) -> None
Purges this open scatter table.

withdraw(self, obj)
([OpenScatterTable](#), Object) -> None
Withdraws the given object from this open scatter t

Static methods defined here:

__new__(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[OpenScatterTable](#) test program.

Data and other attributes defined here:

DELETED = 2

EMPTY = 0

Entry = <class 'opus7.openScatterTable.Entry'>
An entry in an open scatter table.

Iterator = <class 'opus7.openScatterTable.Iterator'>
Enumerates the elements of an open scatter table.

OCCUPIED = 1

__abstractmethods__ = []

Methods inherited from [opus7.hashTable.HashTable](#):

f(self, obj)
([HashTable](#), Object) -> int
Returns the hash of the given object.

g(self, x)
 ([HashTable](#), int) -> int
 Hashes an integer using the division method of hash.

getLoadFactor(self)
 ([HashTable](#)) -> double
 Returns the load factor of this hash table.

h(self, obj)
 ([HashTable](#), Object) -> int
 Hashes the specified object
 using the composition of the methods f and g.

Static methods inherited from [opus7.hashTable.HashTable](#):

test(hashTable)
 [HashTable](#) test program.

Properties inherited from [opus7.hashTable.HashTable](#):

loadFactor
 get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
 (Container) -> int
 Returns the hash of this container.

__str__(self)
 (Container) -> string
 Returns a string representation of this container.

elements(self)
 (Container) -> Object

 Generator that yields the objects in this container

getCount(self)
 (Container) -> int
 Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.27 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.openScatterTableV2](#) (version
1.15, \$Date: 2003/09/21 14:11:02 \$)

[index](#)
[opus7/openScatterTableV2.py](#)

Provides the [OpenScatterTableV2](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.openScatterTable.OpenScatterTable](#)([opus7.hashTable.HashTable](#).[OpenScatterTableV2](#))

class **OpenScatterTableV2**([opus7.openScatterTable.OpenScatterTable](#))

Hash Table implemented as an open scatter table using an array of pointers to objects.

Method resolution order:

[OpenScatterTableV2](#)
[opus7.openScatterTable.OpenScatterTable](#)
[opus7.hashTable.HashTable](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__init__(self, length=0)
([OpenScatterTableV2](#) [, int]) -> None
Constructs an open scatter table with the given length.

withdraw(self, obj)

[\(OpenScatterTableV2, Object\)](#) -> None
Withdraws the given object from this open scatter table.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[OpenScatterTableV2](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.openScatterTable.OpenScatterTable](#)

__contains__(self, obj)
([OpenScatterTable](#), Object) -> bool
Returns true if the given object is in this open scatter table.

__iter__(self)
([OpenScatterTable](#)) -> [OpenScatterTable](#).Iterator
Returns an interator for this open scatter table.

__len__(self)
([OpenScatterTable](#)) -> None
Returns the length of this open scatter table.

accept(self, visitor)
([OpenScatterTable](#), Visitor) -> None
Makes the given visitor visit all the objects in this open scatter table.

c(self, i)
([OpenScatterTable](#), int) -> int
Linear probing function.

find(self, obj)
[\(OpenScatterTable, Object\) -> Object](#)
Returns the object in this open scatter table
that matches the given object.

findInstance(self, obj)
[\(OpenScatterTable, Object\) -> int](#)
Finds the index of the given object in this open scatter table.

findMatch(self, obj)
[\(OpenScatterTable, Object\) -> int](#)
Finds the index of an object in this open scatter table
that matches the given object.

findUnoccupied(self, obj)
[\(OpenScatterTable, Object\) -> int](#)
Returns the index of an unoccupied entry in this open scatter table.

getIsFull(self)
[\(OpenScatterTable\) -> bool](#)
Returns true if this open scatter table is full.

insert(self, obj)
[\(OpenScatterTable, Object\) -> None](#)
Inserts the given object into this open scatter table.

purge(self)
[\(OpenScatterTable\) -> None](#)
Purges this open scatter table.

Data and other attributes inherited from [opus7.openScatterTable.OpenScatterTable](#)

DELETED = 2

EMPTY = 0

Entry = <class 'opus7.openScatterTable.Entry'>
An entry in an open scatter table.

Iterator = <class 'opus7.openScatterTable.Iterator'>
Enumerates the elements of an open scatter table.

OCCUPIED = 1

Methods inherited from [opus7.hashTable.HashTable](#):

f(self, obj)

(HashTable, Object) -> int

Returns the hash of the given object.

g(self, x)

(HashTable, int) -> int

Hashes an integer using the division method of hash.

getLoadFactor(self)

(HashTable) -> double

Returns the load factor of this hash table.

h(self, obj)

(HashTable, Object) -> int

Hashes the specified object

using the composition of the methods f and g.

Static methods inherited from [opus7.hashTable.HashTable](#):

test(hashTable)

HashTable test program.

Properties inherited from [opus7.hashTable.HashTable](#):

loadFactor

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/21 14:11:02 \$'
__version__ = '\$Revision: 1.15 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [orderedList](#) class.

Classes

[opus7.searchableContainer.SearchableContainer](#)([opus7.container.OrderedList](#))

class **OrderedList**([opus7.searchableContainer.SearchableContainer](#))

Base class from which all ordered list classes are derived.

Method resolution order:

[OrderedList](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[builtin.object](#)

Methods defined here:

__getitem__(...)

Abstract method.

__init__(self)

([OrderedList](#)) -> None

Constructor.

findPosition(...)

Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

test(list)
[OrderedList](#) test program.

Data and other attributes defined here:

__abstractmethods__ = ['`__contains__`', '`__getitem__`', '`__iter__`',
'`find`', '`findPosition`', '`insert`', '`purge`', '`withdraw`']

Methods inherited from [opus7.searchableContainer.SearchableContainer](#):

__contains__(...)
Abstract method.

find(...)
Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods inherited from [opus7.searchableContainer.SearchableContainer](#):

main(*argv)
[SearchableContainer](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

`__iter__(...)`
Abstract method.

`__str__(self)`
`(Container) -> string`
Returns a string representation of this container.

`accept(self, visitor)`
`(Container, Visitor) -> None`
Makes the given visitor visit all the items in this

`elements(self)`
`(Container) -> Object`
Generator that yields the objects in this container

`getCount(self)`
`(Container) -> int`
Returns the number of items in this container.

`getIsEmpty(self)`
`(Container) -> bool`
Returns true if this container is empty.

`getIsFull(self)`
`(Container) -> bool`
Returns true if this container is full.

`purge(...)`
Abstract method.

Properties inherited from [opus7.container.Container](#):

`count`
get lambda self

`isEmpty`
get lambda self

`isFull`
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:15 \$'
__version__ = '\$Revision: 1.16 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.orderedListAsArray](#) (version 1.34, \$Date: 2003/09/25 01:07:38 \$)

[index](#)

[opus7/orderedListAsArray.py](#)

Provides the [OrderedListAsArray](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.orderedList.OrderedList](#)([opus7.searchableContainer.SearchableContainer](#))

class OrderedDictAsArray([opus7.orderedList.OrderedList](#))

Ordered list implemented using an array.

Method resolution order:

[OrderedListAsArray](#)

[opus7.orderedList.OrderedList](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[builtin.object](#)

Methods defined here:

[__contains__\(self, obj\)](#)

[\(OrderedListAsArray, Object\) -> bool](#)

Returns true if the given object instance is in this list.

[__getitem__\(self, offset\)](#)

[\(OrderedListAsArray, int\) -> Object](#)

Returns the object in this list at the given `_offset`.

`__init__(self, size=0)`
`(OrderedListAsArray [, int]) -> None`
Constructs an ordered list of the given size.

`__iter__(self)`
`(OrderedListAsArray) -> OrderedDictAsArray.Iterator`
Returns an iterator for this ordered list.

`accept(self, visitor)`
`(OrderedListAsArray, Visitor) -> bool`
Makes the given visitor visit the objects in this ordered list.

`find(self, obj)`
`(OrderedListAsArray, Object) -> Object`
Finds an object in this ordered list that equals the given object.

`findPosition(self, obj)`
`(OrderedListAsArray, Object) -> OrderedDictAsArray.Cursor`
Finds the position of an object in this list that equals the given object and returns a cursor that refers to that object.

`getIsFull(self)`
`(OrderedListAsArray) -> bool`
Returns true if this ordered list is full.

`insert(self, obj)`
`(OrderedListAsArray, Object) -> None`
Inserts the given object at the end of this list.

`purge(self)`
`(OrderedListAsArray) -> None`
Purges this ordered list.

`withdraw(self, obj)`
`(OrderedListAsArray, Object) -> None`
Withdraws the given object instance from this ordered list.

Static methods defined here:

`__new__ = new(*args, **kwargs)`
`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
[OrderedListAsArray](#) test program.

Data and other attributes defined here:

Cursor = <class 'opus7.orderedListAsArray.Cursor'>
A cursor that refers to an object in an ordered list.

Iterator = <class 'opus7.orderedListAsArray.Iterator'>
Enumerates the items in an ordered list.

__abstractmethods__ = []

Static methods inherited from [opus7.orderedList.OrderedList](#):

test(list)
[OrderedList](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.34 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.orderedListAsLinkedList](#)

(version 1.29, \$Date: 2003/09/25
01:07:38 \$)

[index](#)

[opus7/orderedListAsLinkedList.py](#)

Provides the [OrderedListAsLinkedList](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.orderedList.OrderedList](#)([opus7.searchableContainer.SearchableContainer](#).[OrderedListAsLinkedList](#))

class **OrderedListAsLinkedList**([opus7.orderedList.OrderedList](#))

Ordered list implemented using a linked list.

Method resolution order:

[OrderedListAsLinkedList](#)
[opus7.orderedList.OrderedList](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__contains__(self, obj)
([OrderedListAsLinkedList](#), Object) -> bool
Returns true if the given object instance is in this list.

__getitem__(self, offset)

([OrderedListAsLinkedList](#), int) -> Object
Returns the object in this list at the given offset

__init__(self)
([OrderedListAsLinkedList](#)) -> None
Constructs an ordered list.

__iter__(self)
([OrderedListAsLinkedList](#)) -> [OrderedListAsLinkedList](#)
Returns an iterator for this ordered list.

accept(self, visitor)
([OrderedListAsLinkedList](#), Visitor) -> None
Makes the given visitor visit all the objects in th

find(self, arg)
([OrderedListAsLinkedList](#), Object) -> Object
Finds an object in this ordered list that equals the

findPosition(self, obj)
([OrderedListAsLinkedList](#), Object) -> [OrderedListAsLinkedList](#)
Finds the position of an object in this list
that equals the given object and returns a cursor
that refers to that object.

insert(self, obj)
([OrderedListAsLinkedList](#), Object) -> None
Inserts the given object at the end of this list.

purge(self)
([OrderedListAsLinkedList](#)) -> None
Purges this ordered list.

withdraw(self, obj)
([OrderedListAsLinkedList](#), Object) -> None
Withdraws the given object instance from this ordered

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`.

in classes instances derived from Metaclass.

main(*argv)

[OrderedListAsLinkedList](#) test program.

Data and other attributes defined here:

Cursor = <class 'opus7.orderedListAsLinkedList.Cursor'>
A cursor that refers to an object in an ordered list.

Iterator = <class 'opus7.orderedListAsLinkedList.Iterator'>
Enumerates the items in an ordered list.

__abstractmethods__ = []

Static methods inherited from [opus7.orderedList.OrderedList](#):

test(list)

[OrderedList](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.29 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Parent](#) class.

Classes

[opus7.person.Person](#)([__builtin__.object](#))
[Parent](#)

class Parent(opus7.person.Person)

Represents a parent.

Method resolution order:

[Parent](#)
[opus7.person.Person](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, name, sex, children)
([Parent](#), str, int, ([Person](#), [Person](#), ...)) -> None
Constructs a parent with the given name and sex
and collection of children.

__str__(self)
([Parent](#)) -> str
Returns a string representation of this parent.

getChild(self, i)
([Parent](#), int) -> [Person](#)
Returns the specified child of this parent.

Data and other attributes inherited from [opus7.person.Person](#):

FEMALE = 0

MALE = 1

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Person' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Partition](#) class.

Classes

[opus7.set.Set](#)([opus7.searchableContainer.SearchableContainer](#))
[Partition](#)

class Partition([opus7.set.Set](#))

Base class from which all partitions are derived.

Method resolution order:

[Partition](#)
[opus7.set.Set](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__init__\(self, n\)](#)

[find\(...\)](#)
Abstract method.

[join\(...\)](#)
Abstract method.

Static methods defined here:

[__new__ = new\(*args, **kwargs\)](#)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(p)

[Partition](#) test program.

Data and other attributes defined here:

`__abstractmethods__` = ['`and`', '`contains`', '`eq`', '`it`', '`or`', '`sub`', '`compareTo`', 'find', 'insert', 'join', 'purge', 'with'

Methods inherited from [opus7.set.Set](#):

and(...)

Abstract method.

eq(...)

Abstract method.

lt(...)

Abstract method.

or(...)

Abstract method.

sub(...)

Abstract method.

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize

get lambda self

Methods inherited from [opus7.searchableContainer.SearchableContainer](#):

__contains__(...)
Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods inherited from [opus7.searchableContainer.Searchat](#)

main(*argv)
SearchableContainer test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)



Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/06 16:35:16 \$'
version = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PartitionAsForest](#) class.

Modules

[sys](#)

Classes

[opus7.partition.Partition](#)([opus7.set.Set](#))
[PartitionAsForest](#)

class PartitionAsForest([opus7.partition.Partition](#))

A partition (set of sets) implemented as a forest of trees.

Method resolution order:

[PartitionAsForest](#)
[opus7.partition.Partition](#)
[opus7.set.Set](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__and__](#)(self, partition)

[__contains__](#)(self, obj)
([PartitionAsForest](#), Set) -> bool
Returns true if the given set is an element of this

`__eq__(self, partition)`

`__init__(self, n)`
`(PartitionAsForest, int) -> None`
Constructs a partition with the given universe size

`__iter__(self)`

`__lt__(self, partition)`

`__or__(self, partition)`

`__sub__(self, partition)`

`accept(self, visitor)`
`(PartitionAsForest, Visitor) -> None`
Makes the given visitor visit the elements of this |

`find(self, item)`
`(PartitionAsForest, int) -> Set`
Finds the set in this partition that contains the g.

`insert(self, obj)`

`join(self, s, t)`
`(PartitionAsForest, Set, Set) -> None`
Joins the given sets in this partition.

`purge(self)`
`(PartitionAsForest) -> None`
Purges this partition.

`withdraw(self, obj)`

Static methods defined here:

`__new__ = new(*args, **kwargs)`
`(Metaclass, ...) -> object`
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`

in classes instances derived from Metaclass.

main(*argv)
[PartitionAsForest](#) test program.

Data and other attributes defined here:

PartitionTree = <class 'opus7.partitionAsForest.PartitionTree'>
Represents a element of a partition.

__abstractmethods__ = []

Static methods inherited from [opus7.partition.Partition](#):

test(p)
[Partition](#) test program.

Methods inherited from [opus7.set.Set](#):

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.21 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the PartitionAsForestV2 class.

Modules

[sys](#)

Classes

[opus7.partitionAsForest.PartitionAsForest](#)([opus7.partition.PartitionAsForestV2](#))

class PartitionAsForestV2([opus7.partitionAsForest.PartitionAsForest](#))

Partition (set of sets) implemented as a forest of trees

Method resolution order:

[PartitionAsForestV2](#)
[opus7.partitionAsForest.PartitionAsForest](#)
[opus7.partition.Partition](#)
[opus7.set.Set](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin_object](#)

Methods defined here:

__init__(self, n)
([PartitionAsForestV2](#), int) -> None
Constructs a partition with the given universe size

find(self, item)
 ([PartitionAsForestV2](#), item) -> Set
 Returns the element of this partition that contains
 (Collapsing find).

join(self, s, t)
 ([PartitionAsForestV2](#), Set, Set) -> None
 Joins the given elements of this partition.
 (Union by size).

Static methods defined here:

__new__ = new(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
 [PartitionAsForestV2](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.partitionAsForest.PartitionAsForest](#)

__and__(self, partition)

__contains__(self, obj)
 ([PartitionAsForest](#), Set) -> bool
 Returns true if the given set is an element of this

__eq__(self, partition)

__iter__(self)

__lt__(self, partition)

__or__(self, partition)

__sub__(self, partition)

accept(self, visitor)

([PartitionAsForest](#), Visitor) -> None

Makes the given visitor visit the elements of this |

insert(self, obj)

purge(self)

([PartitionAsForest](#)) -> None

Purges this partition.

withdraw(self, obj)

Data and other attributes inherited from [opus7.partitionAsForest.P](#)

PartitionTree = <class 'opus7.partitionAsForest.PartitionTree'>

Represents a element of a partition.

Static methods inherited from [opus7.partition.Partition](#):

test(p)

Partition test program.

Methods inherited from [opus7.set.Set](#):

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
 Metaclass of the Object class.
 Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
 list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.9 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PartitionAsForestV2](#) class.

Modules

[sys](#)

Classes

[opus7.partitionAsForestV2.PartitionAsForestV2](#)([opus7.partitionAsForestV3](#))

class PartitionAsForestV3([opus7.partitionAsForestV2.PartitionA](#)

Partition (set of sets) implemented as a forest of trees

Method resolution order:

[PartitionAsForestV3](#)

[opus7.partitionAsForestV2.PartitionAsForestV2](#)

[opus7.partitionAsForest.PartitionAsForest](#)

[opus7.partition.Partition](#)

[opus7.set.Set](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[__builtin__.object](#)

Methods defined here:

[__init__](#)(self, n)

([PartitionAsForestV3](#), int) -> None

Constructs a partition with the given universe size

join(self, s, t)
([PartitionAsForestV3](#), Set, Set) -> None
Joins the given sets of this partition.
(Union by rank).

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
[PartitionAsForestV3](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.partitionAsForestV2.PartitionAsForestV2](#)

find(self, item)
([PartitionAsForestV2](#), item) -> Set
Returns the element of this partition that contains the item.
(Collapsing find).

Methods inherited from [opus7.partitionAsForest.PartitionAsForest](#)

__and__(self, partition)

__contains__(self, obj)
(PartitionAsForest, Set) -> bool
Returns true if the given set is an element of this partition.

__eq__(self, partition)

__iter__(self)

__lt__(self, partition)
__or__(self, partition)
__sub__(self, partition)

accept(self, visitor)
(PartitionAsForest, Visitor) -> None
Makes the given visitor visit the elements of this |

insert(self, obj)

purge(self)
(PartitionAsForest) -> None
Purges this partition.

withdraw(self, obj)

Data and other attributes inherited from [opus7.partitionAsForest.P](#)

PartitionTree = <class 'opus7.partitionAsForest.PartitionTree'>
Represents a element of a partition.

Static methods inherited from [opus7.partition.Partition](#):

test(p)
Partition test program.

Methods inherited from [opus7.set.Set](#):

getUniverseSize(self)

Properties inherited from [opus7.set.Set](#):

universeSize
get lambda self

Methods inherited from [opus7.container.Container](#):

`__hash__(self)`
`(Container) -> int`
Returns the hash of this container.

`__str__(self)`
`(Container) -> string`
Returns a string representation of this container.

`elements(self)`
`(Container) -> Object`
Generator that yields the objects in this container

`getCount(self)`
`(Container) -> int`
Returns the number of items in this container.

`getIsEmpty(self)`
`(Container) -> bool`
Returns true if this container is empty.

`getIsFull(self)`
`(Container) -> bool`
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

`count`
get lambda self

`isEmpty`
get lambda self

`isFull`
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

`StrVisitor` = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Person](#) class.

Classes

[__builtin__.object](#) [Person](#)

class Person([__builtin__.object](#))

Represents a person.

Methods defined here:

__init__(self, name, sex)
([Person](#), str, int) -> None
Constructs a person with the given name and sex.

__str__(self)
([Person](#)) -> str
Returns a string representation of this person.

Data and other attributes defined here:

FEMALE = 0

MALE = 1

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Person' objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/07/23 17:54:18 \$'
version = '\$Revision: 1.6 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Point](#) class.

Classes

[__builtin__.object](#) [Point](#)

class Point([__builtin__.object](#))

Represents a point in an image.

Methods defined here:

[__init__](#)(self, x, y)
[\(Point, int, int\)](#) -> None
Constructs a point with the given coordinates.

[__str__](#)(self)
[\(Point\)](#) -> str
Returns a textual representation of this point.

Data and other attributes defined here:

[__dict__](#) = <dictproxy object>
dictionary for instance variables (if defined)

[__weakref__](#) = <attribute '__weakref__' of 'Point' objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/28 00:06:11 \$'
version = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Polynomial](#) class.

Modules

[sys](#)

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[Polynomial](#)

class Polynomial([opus7.container.Container](#))

Base class from which all polynomial classes are derived

Method resolution order:

[Polynomial](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__add__\(...\)](#)
Abstract method.

[__init__\(self\)](#)
(Polynomial) -> None
Constructor.

[addTerm\(...\)](#)
Abstract method.

differentiate(self)
([Polynomial](#)) -> None
Differentiates this polynomial.

Static methods defined here:

__new__(self, *args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

Data and other attributes defined here:

DifferentiatingVisitor = <class 'opus7.polynomial.DifferentiatingVisitor' that differentiates the terms it visits.

Term = <class 'opus7.polynomial.Term'>
Represents a term in a polynomial.

__abstractmethods__ = ['__add__', '__iter__', '_compareTo', 'add'

Methods inherited from [opus7.container.Container](#):

__hash__(self)
([Container](#)) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
([Container](#)) -> string
Returns a string representation of this container.

accept(self, visitor)
([Container](#), Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)

(Container) -> Object

Generator that yields the objects in this container

getCount(self)

(Container) -> int

Returns the number of items in this container.

getIsEmpty(self)

(Container) -> bool

Returns true if this container is empty.

getIsFull(self)

(Container) -> bool

Returns true if this container is full.

purge(...)

Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

[Container](#) test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
 (Object, Object) -> int
 Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
 Metaclass of the Object class.
 Prevents instantiation of classes that contain abst...

__weakref__ = <attribute '__weakref__' of 'Object' objects>
 list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:00:48 \$'
__version__ = '\$Revision: 1.20 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.polynomialAsOrderedList](#)

(version 1.9, \$Date: 2003/09/25
01:07:38 \$)

ind
[opus7/polynomialAsOrderedList.](#)

Provides the [PolynomialAsOrderedList](#) class.

Classes

[opus7.polynomial.Polynomial](#)([opus7.container.Container](#))
[PolynomialAsOrderedList](#)

class [PolynomialAsOrderedList](#)([opus7.polynomial.Polynomial](#))

[Polynomial](#) implemented as an ordered list of terms.

Method resolution order:

[PolynomialAsOrderedList](#)
[opus7.polynomial.Polynomial](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__add__\(self, polynomial\)](#)

[__init__\(self\)](#)
([PolynomialAsOrderedList](#)) -> None
Constructor.

[__iter__\(self\)](#)

[__str__\(self\)](#)
([PolynomialAsOrderedList](#)) -> str
Returns a string representation of this polynomial.

accept(self, visitor)
`(PolynomialAsOrderedList, visitor) -> None`
Makes the given visitor visit all the terms in this

addTerm(self, term)
`(PolynomialAsOrderedList) -> None`
Adds the given term to this polynomial.

find(self, term)
`(PolynomialAsOrderedList, Polynomial.Term) -> Polynomial`
Finds a term in this polynomial that matches the given term.

purge(self)
`(PolynomialAsOrderedList) -> None`
Purges this polynomial.

withdraw(self, term)
`(PolynomialAsOrderedList, Polynomial.Term) -> None`
Withdraws the given term from this polynomial.

Static methods defined here:

__new__ = new(*args, **kwargs)
`(Metaclass, ...) -> object`
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.polynomial.Polynomial](#):

differentiate(self)
`(Polynomial) -> None`
Differentiates this polynomial.

Data and other attributes inherited from [opus7.polynomial.Polynomial](#):

DifferentiatingVisitor = <class 'opus7.polynomial.DifferentiatingVisitor' that differentiates the terms it visits.

Term = <class 'opus7.polynomial.Term'>
Represents a term in a polynomial.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Static methods inherited from [opus7.container.Container](#):

main(*argv)
Container test program.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.9 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.polynomialAsSortedList](#)

(version 1.11, \$Date: 2003/09/25
01:07:38 \$)

[index](#)

[opus7/polynomialAsSortedList.py](#)

Provides the [PolynomialAsSortedList](#) class.

Classes

[opus7.polynomial.Polynomial](#)([opus7.container.Container](#))
[PolynomialAsSortedList](#)

class **PolynomialAsSortedList**([opus7.polynomial.Polynomial](#))

[Polynomial](#) implemented as a sorted list of terms.

Method resolution order:

[PolynomialAsSortedList](#)
[opus7.polynomial.Polynomial](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__add__(self, poly)
([PolynomialAsSortedList](#), [PolynomialAsSortedList](#))
-> [PolynomialAsSortedList](#).

Returns the sum of this polynomial and the given po.

__init__(self)
([PolynomialAsSortedList](#)) -> None
Constructor.

__iter__(self)

__str__(self)

([PolynomialAsSortedList](#)) -> str
Returns the string representation of this polynomial.

accept(self, visitor)
[\(PolynomialAsSortedList, Visitor\) -> None](#)
Makes the given visitor visit all the terms in this polynomial.

addTerm(self, term)
[\(PolynomialAsSortedList, Polynomial.Term\) -> None](#)
Adds the given term to this polynomial.

find(self, term)
[\(PolynomialAsSortedList, Polynomial.Term\) -> Polynomial](#)
Finds a term in this polynomial that matches the given term.

nextTerm(self, iter)
[\(PolynomialAsSortedList, Iterator\) -> Polynomial.Term](#)
Returns the next term or None if there are no more terms.

purge(self)
[\(PolynomialAsSortedList\) -> None](#)
Purges this polynomial.

withdraw(self, term)
[\(PolynomialAsSortedList, Polynomial.Term\) -> Polynomial](#)
Withdraws the given term from this polynomial.

Static methods defined here:

__new__ = new(*args, **kwargs)
[\(Metaclass, ...\) -> object](#)
Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.polynomial.Polynomial](#):

differentiate(self)
 ([Polynomial](#)) -> None
 Differentiates this polynomial.

Data and other attributes inherited from [opus7.polynomial.Poly](#)

DifferentiatingVisitor = <class 'opus7.polynomial.DifferentiatingVisitor' that differentiates the terms it visits.

Term = <class 'opus7.polynomial.Term'>
 Represents a term in a polynomial.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
 (Container) -> int
 Returns the hash of this container.

elements(self)
 (Container) -> Object

Generator that yields the objects in this container

getCount(self)
 (Container) -> int
 Returns the number of items in this container.

getIsEmpty(self)
 (Container) -> bool
 Returns true if this container is empty.

getIsFull(self)
 (Container) -> bool
 Returns true if this container is full.

Static methods inherited from [opus7.container.Container](#):

main(*argv)
 Container test program.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'

 **__version__** = '\$Revision: 1.11 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PostOrder](#) class.

Modules

[sys](#)

Classes

[opus7.prePostVisitor.PrePostVisitor](#)([opus7.visitor.Visitor](#))
[PostOrder](#)

class **PostOrder**([opus7.prePostVisitor.PrePostVisitor](#))

Adapter to convert a Visitor to a [PrePostVisitor](#) for post-order traversal.

Method resolution order:

[PostOrder](#)
[opus7.prePostVisitor.PrePostVisitor](#)
[opus7.visitor.Visitor](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__init__(self, visitor)
([PostOrder](#), Visitor) -> None
Constructs a post-order visitor from the given visitor.

getIsDone(self)
([PostOrder](#)) -> bool
Returns true if the visitor is done.

postVisit(self, obj)
([PostOrder](#), Object) -> None
Post-visits the given object.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)
[PostOrder](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.prePostVisitor.PrePostVisitor](#):

inVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default in-visit method does nothing.

preVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default pre-visit method does nothing.

visit = inVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default in-visit method does nothing.

Properties inherited from [opus7.visitor.Visitor](#):

isDone
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.14 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PreOrder](#) class.

Modules

[sys](#)

Classes

[opus7.prePostVisitor.PrePostVisitor](#)([opus7.visitor.Visitor](#))
[PreOrder](#)

class PreOrder([opus7.prePostVisitor.PrePostVisitor](#))

Adapter to convert a Visitor to a [PrePostVisitor](#) for pre-order traversal.

Method resolution order:

[PreOrder](#)
[opus7.prePostVisitor.PrePostVisitor](#)
[opus7.visitor.Visitor](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

[__init__\(self, visitor\)](#)
([PreOrder](#), Visitor) -> None
Constructs a pre-order visitor from the given visitor.

[getIsDone\(self\)](#)
([PreOrder](#)) -> bool
Returns true if the visitor is done.

preVisit(self, obj)
([PreOrder](#), Object) -> None
Pre-visits the given object.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)
[PreOrder](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.prePostVisitor.PrePostVisitor](#):

inVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default in-visit method does nothing.

postVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default post-visit method does nothing.

visit = inVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default in-visit method does nothing.

Properties inherited from [opus7.visitor.Visitor](#):

isDone
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.14 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PrePostVisitor](#) class.

Modules

[sys](#)

Classes

[opus7.visitor.Visitor](#)([opus7.object.Object](#))
[PrePostVisitor](#)

class PrePostVisitor([opus7.visitor.Visitor](#))

Pre/Post visitor class.

Method resolution order:

[PrePostVisitor](#)
[opus7.visitor.Visitor](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__init__\(self\)](#)
([PrePostVisitor](#)) -> None
Constructor.

[inVisit\(self, obj\)](#)
([PrePostVisitor](#), Object) -> None
Default in-visit method does nothing.

[postVisit\(self, obj\)](#)

([PrePostVisitor](#), Object) -> None
Default post-visit method does nothing.

preVisit(self, obj)
([PrePostVisitor](#), Object) -> None
Default pre-visit method does nothing.

visit = [inVisit](#)(self, obj)

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[PrePostVisitor](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.visitor.Visitor](#):

getIsDone(self)
([Visitor](#)) -> bool
Default `isDone_get` method returns false always.

Properties inherited from [opus7.visitor.Visitor](#):

isDone
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PrintingVisitor](#) class.

Classes

[opus7.visitor.Visitor](#)([opus7.object.Object](#))
[PrintingVisitor](#)

class **PrintingVisitor**([opus7.visitor.Visitor](#))

[Visitor](#) that prints the objects it visits.

Method resolution order:

[PrintingVisitor](#)
[opus7.visitor.Visitor](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([PrintingVisitor](#)) -> None
Constructor.

finish(self)
([PrintingVisitor](#)) -> None
Finishes the line.

visit(self, obj)
([PrintingVisitor](#), Object) -> None
Prints the object.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.visitor.Visitor](#):

getIsDone(self)
([Visitor](#)) -> bool
Default `isDone_get` method returns false always.

Static methods inherited from [opus7.visitor.Visitor](#):

main(*argv)
[Visitor](#) test program.

Properties inherited from [opus7.visitor.Visitor](#):

isDone
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.MetaClass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/08/28 01:56:13 \$'
__version__ = '\$Revision: 1.5 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [PriorityQueue](#) class.

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[PriorityQueue](#)

class PriorityQueue([opus7.container.Container](#))

Base class from which all priority queue classes are derived.

Method resolution order:

[PriorityQueue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([PriorityQueue](#)) -> None
Constructor.

dequeueMin(...)
Abstract method.

enqueue(...)
Abstract method.

getMin(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(pqueue)
[PriorityQueue](#) test program.

Properties defined here:

min

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__iter__', '_compareTo', 'dequeueMin', '
'getMin', 'purge']

Methods inherited from [opus7.container.Container](#):

__hash__(self)
([Container](#)) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
([Container](#)) -> string
Returns a string representation of this container.

accept(self, visitor)
([Container](#), Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
([Container](#)) -> Object

Generator that yields the objects in this container

getCount(self)

([Container](#)) -> int

Returns the number of items in this container.

getIsEmpty(self)

([Container](#)) -> bool

Returns true if this container is empty.

getIsFull(self)

([Container](#)) -> bool

Returns true if this container is full.

purge(...)

Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

[Container](#) test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst...

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.13 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Queue](#) class.

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[Queue](#)

class Queue(opus7.container.Container)

Base class from which all queue classes are derived.

Method resolution order:

[Queue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([Queue](#)) -> None
Constructor.

dequeue(...)
Abstract method.

enqueue(...)
Abstract method.

getHead(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(queue)
[Queue](#) test program.

Properties defined here:

head

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__iter__', '_compareTo', 'dequeue', 'enqueue',
'getHead', 'purge']

Methods inherited from [opus7.container.Container](#):

__hash__(self)
([Container](#)) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
([Container](#)) -> string
Returns a string representation of this container.

accept(self, visitor)
([Container](#), Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
([Container](#)) -> Object

Generator that yields the objects in this container

getCount(self)

([Container](#)) -> int

Returns the number of items in this container.

getIsEmpty(self)

([Container](#)) -> bool

Returns true if this container is empty.

getIsFull(self)

([Container](#)) -> bool

Returns true if this container is full.

purge(...)

Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

[Container](#) test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst...

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.23 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [QueueAsArray](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.queue.Queue](#)([opus7.container.Container](#))
[QueueAsArray](#)

class **QueueAsArray**([opus7.queue.Queue](#))

[Queue](#) implemented using an array.

Method resolution order:

[QueueAsArray](#)
[opus7.queue.Queue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, size=0)
([QueueAsArray](#) [, int]) -> None
Constructs a queue of the given size.

__iter__(self)
([QueueAsArray](#)) -> [QueueAsArray](#).Iterator
Returns an iterator for this queue.

accept(self, visitor)
 ([QueueAsArray](#), Visitor) -> None
 Makes the given visitor visit all the objects in th.

dequeue(self)
 ([QueueAsArray](#)) -> Object
 Dequeues the object at the head of this queue.

enqueue(self, obj)
 ([QueueAsArray](#), Object) -> None
 Enqueues the given object to the tail of this queue

getHead(self)
 ([QueueAsArray](#)) -> Object
 Returns the object at the head of this queue.

getIsFull(self)
 ([QueueAsArray](#)) -> bool
 Returns true if this queue is full.

purge(self)
 ([QueueAsArray](#)) -> None
 Purges this queue.

Static methods defined here:

__new__ = new(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
 [QueueAsArray](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.queueAsArray.Iterator'>
 Enumerates the elements of a [QueueAsArray](#).

__abstractmethods__ = []

Static methods inherited from [opus7.queue.Queue](#):

test(queue)
Queue test program.

Properties inherited from [opus7.queue.Queue](#):

head
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.27 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [QueueAsLinkedList](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.queue.Queue](#)([opus7.container.Container](#))
[QueueAsLinkedList](#)

class QueueAsLinkedList([opus7.queue.Queue](#))

[Queue](#) implemented using a linked list.

Method resolution order:

[QueueAsLinkedList](#)
[opus7.queue.Queue](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__init__\(self\)](#)
([QueueAsLinkedList](#)) -> None
Constructs a queue.

[__iter__\(self\)](#)
([QueueAsLinkedList](#)) -> [QueueAsLinkedList.Iterator](#)
Returns an iterator for this queue.

accept(self, visitor)
 ([QueueAsLinkedList](#), Visitor) -> None
 Makes the given visitor visit all the objects in th.

dequeue(self)
 ([QueueAsLinkedList](#)) -> Object
 Dequeues the object at the head of this queue.

enqueue(self, obj)
 ([QueueAsLinkedList](#), Object) -> None
 Enqueues the given object to the tail of this queue

getHead(self)
 ([QueueAsLinkedList](#)) -> Object
 Returns the object at the head of this queue.

purge(self)
 ([QueueAsLinkedList](#)) -> None
 Purges this queue.

Static methods defined here:

__new__ = new(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
 [QueueAsLinkedList](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.queueAsLinkedList.Iterator'>
 Enumerates the elements of a [QueueAsLinkedList](#).

__abstractmethods__ = []

Static methods inherited from [opus7.queue.Queue](#):

test(queue)
[Queue](#) test program.

Properties inherited from [opus7.queue.Queue](#):

head
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.26 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [QuickSorter](#) class.

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[QuickSorter](#)

class QuickSorter([opus7.sorter.Sorter](#))

Base class from which all [QuickSorter](#) classes are derived

Method resolution order:

[QuickSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([QuickSorter](#)) -> None
Constructor.

quicksort(self, left, right)
([QuickSorter](#), left, right) -> None
Recursively sorts the elements of the array between

selectPivot(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Data and other attributes defined here:

CUTOFF = 2

__abstractmethods__ = ['selectPivot']

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)
([Sorter](#), Array) -> None
Sorts the given array.

swap(self, i, j)
([Sorter](#), int, int) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)
Object test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
 Metaclass of the Object class.
 Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
 list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 20:11:05 \$'
__version__ = '\$Revision: 1.11 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [RadixSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[RadixSorter](#)

class RadixSorter([opus7.sorter.Sorter](#))

Radix sorter.

Method resolution order:

[RadixSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

[__init__\(self\)](#)
([RadixSorter](#)) -> None
Constructor.

Static methods defined here:

[__new__ = new\(*args, **kwargs\)](#)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
[RadixSorter](#) test program.

Data and other attributes defined here:

R = 256

__abstractmethods__ = []

p = 4

r = 8

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)
([Sorter](#), Array) -> None
Sorts the given array.

swap(self, i, j)
([Sorter](#), int, int) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.9 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.randomNumberGenerator](#)

(version 1.14, \$Date: 2003/09/27
13:03:42 \$)

inc

[opus7/randomNumberGenerator](#)

Provides the RandomNumberGenerator class
and the RandomNumberGenerator singleton.

Modules

[sys](#)

Classes

[__builtin__.object](#)

RandomNumberGenerator

_RandomNumberGenerator = class
RandomNumberGenerator([__builtin__.object](#))

A multiplicative linear congruential pseudo-random number generator.

Adapted from the minimal standard pseudo-random number generator described in Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones Are Hard To Find," Communications of the ACM, Vol. 31, No. 10, Oct. 1988, pp. 124-130.

Methods defined here:

__init__(self, seed=1)
(RandomNumberGenerator [, int]) -> None
Constructs a random number generator with the given seed.

getNext(self)
(RandomNumberGenerator) -> double
Returns the next random number.

getSeed(self)

`(RandomNumberGenerator) -> int`
Returns the seed of this random number generator.

setSeed(self, seed)

`(RandomNumberGenerator, int) -> None`
Sets the seed of this random number generator to the specified value.

Static methods defined here:

main(*argv)

RandomNumberGenerator test program.

Properties defined here:

next

get lambda self

seed

get lambda self

set lambda self, value

Data and other attributes defined here:

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'RandomNumberGenerator' objects>
list of weak references to the [object](#) (if defined)

a = 16807

m = 2147483647

q = 127773

r = 2836

Data

```
RandomNumberGenerator =  
<opus7.randomNumberGenerator.RandomNumberGenerator object>  
  __author__ = 'Bruno R. Preiss, P.Eng.'  
  __credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
  __date__ = '$Date: 2003/09/27 13:03:42 $'  
  __version__ = '$Revision: 1.14 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [RandomVariable](#) class.

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[RandomVariable](#)

class RandomVariable([opus7.object.Object](#))

Base class from which all random variables are derived.

Method resolution order:

[RandomVariable](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([RandomVariable](#)) -> None
Constructor.

getNext(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given args.
Raises a TypeError exception if the class is abstract.
This method is inserted as the method __new__

in classes instances derived from Metaclass.

Properties defined here:

next

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['getNext']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

[\(Object, Object\)](#) -> int

Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)

[Object](#) test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the [Object](#) class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'

 **_credits_** = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
 date = '\$Date: 2003/09/25 01:07:38 \$'
 version = '\$Revision: 1.15 \$'

Author

 Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Rectangle](#) class.

Modules

[sys](#)

Classes

[opus7.graphicalObject.GraphicalObject](#)([opus7.object.Object](#))
[Rectangle](#)

class **Rectangle**([opus7.graphicalObject.GraphicalObject](#))

A rectangle.

Method resolution order:

[Rectangle](#)
[opus7.graphicalObject.GraphicalObject](#)
[opus7.object.Object](#)
[builtin__object](#)

Methods defined here:

__init__(self, center, height, width)

([Rectangle](#), Point, int, int) -> None

Constructs a rectangle with the given center, height and width.

draw(self)

([Rectangle](#)) -> None

Draws this rectangle.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[Rectangle](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.graphicalObject.GraphicalObject](#):

erase(self)
([GraphicalObject](#)) -> None
Erases this graphical object.

moveTo(self, p)
([GraphicalObject](#)) -> None
Moves the center of this graphical object to the given point.

setPenColor(self, color)

Static methods inherited from [opus7.graphicalObject.GraphicalObject](#):

test(go)
[GraphicalObject](#) test program.

Data and other attributes inherited from [opus7.graphicalObject.GraphicalObject](#):

BACKGROUND_COLOR = 0

FOREGROUND_COLOR = 1

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/28 00:06:11 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

opus7.scalesBalancingProblem

(version 1.17, \$Date: 2003/09/25
01:07:38 \$)

[inde](#)

[opus7/scalesBalancingProblem.p](#)

Provides the [ScalesBalancingProblem](#) class.

Classes

[__builtin__.object](#)
[ScalesBalancingProblem](#)

class ScalesBalancingProblem([__builtin__.object](#))

Represents a scale-balancing problem.

Methods defined here:

__init__(self, weight)
[\(ScalesBalancingProblem, Array\)](#) -> None
Constructs a scales balancing problem with the given weight.

solve(self, solver)
[\(ScalesBalancingProblem, Solver\)](#) -> Solution
Solves this problem using the given solver.

Data and other attributes defined here:

Node = <class 'opus7.scalesBalancingProblem.Node'>
Represents a node in the solution space
of this scales balancing problem.

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'ScalesBalancingProblem' objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.17 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [SearchTree](#) class.

Classes

[opus7.searchableContainer.SearchableContainer](#)([opus7.container.Container](#))
 [SearchTree](#)([opus7.tree.Tree](#),
 [opus7.searchableContainer.SearchableContainer](#))
[opus7.tree.Tree](#)([opus7.container.Container](#))
 [SearchTree](#)([opus7.tree.Tree](#),
 [opus7.searchableContainer.SearchableContainer](#))

class SearchTree([opus7.tree.Tree](#),
[opus7.searchableContainer.SearchableContainer](#))

Base class from which all search tree classes are derived.

Method resolution order:

[SearchTree](#)
[opus7.tree.Tree](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([SearchTree](#)) -> None
Constructor.

getMax(...)
Abstract method.

getMin(...)

Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

test(tree)

[SearchTree](#) test program.

Properties defined here:

max

get lambda self

min

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__contains__', '__compareTo', 'find', 'get'
'getIsLeaf', 'getKey', 'getMax', 'getMin', 'getSubtree', 'insert', 'purge']

Methods inherited from [opus7.tree.Tree](#):

__iter__(self)
([Tree](#)) -> [Tree](#).Iterator
Returns an iterator for this tree.

accept(self, visitor)

([Tree](#)) -> Visitor

Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)

([Tree](#)) -> generator

Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)

([Tree](#), Visitor) -> None

Makes the given visitor do a breadth-first traversal of this tree.

depthFirstGenerator(self, mode)

([Tree](#)) -> generator

Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)

([Tree](#), PrePostVisitor) -> None

Makes the given visitor do a depth-first traversal of this tree.

getCount(self)

([Tree](#)) -> int

Returns the number of nodes in this tree.

getDegree(...)

Abstract method.

getHeight(self)

([Tree](#)) -> int

Returns the height of this tree.

getIsLeaf(...)

Abstract method.

getKey(...)

Abstract method.

getSubtree(...)

Abstract method.

Properties inherited from [opus7.tree.Tree](#):

degree

get lambda self

height

get lambda self

isLeaf
get lambda self

key
get lambda self

Data and other attributes inherited from [opus7.tree.Tree](#):

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

Methods inherited from [opus7.searchableContainer.SearchableCor](#)

__contains__(...)
Abstract method.

find(...)
Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods inherited from [opus7.searchableContainer.Searchat](#)

main(*argv)
[SearchableContainer](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int

Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object
Generator that yields the objects in this container

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

`(Object, Object) -> int`
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__metaclass__` = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

`__weakref__` = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

`__author__` = 'Bruno R. Preiss, P.Eng.'
`__credits__` = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
`__date__` = '\$Date: 2003/09/23 23:01:18 \$'
`__version__` = '\$Revision: 1.21 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.searchableContainer](#)

(version 1.19, \$Date: 2003/09/06
16:35:16 \$)

[index](#)

[opus7/searchableContainer.py](#)

Provides the [SearchableContainer](#) class.

Modules

[sys](#)

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[SearchableContainer](#)

class SearchableContainer([opus7.container.Container](#))

Base class from which all searchable container classes are derived.

Method resolution order:

[SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

[__contains__\(...\)](#)
Abstract method.

[__init__\(self\)](#)
([SearchableContainer](#)) -> None
Constructor.

find(...)
Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[SearchableContainer](#) test program.

Data and other attributes defined here:

__abstractmethods__ = ['__contains__', '__iter__', '_compareTo',
'purge', 'withdraw']

Methods inherited from [opus7.container.Container](#):

__hash__(self)
([Container](#)) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
([Container](#)) -> string
Returns a string representation of this container.

accept(self, visitor)
([Container](#), Visitor) -> None

Makes the given visitor visit all the items in this

elements(self)
[\(Container\)](#) -> Object

Generator that yields the objects in this container

getCount(self)
[\(Container\)](#) -> int
Returns the number of items in this container.

getIsEmpty(self)
[\(Container\)](#) -> bool
Returns true if this container is empty.

getIsFull(self)
[\(Container\)](#) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.19 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Set](#) class.

Classes

[opus7.searchableContainer.SearchableContainer](#)([opus7.container.Container](#)) [Set](#)

class **Set**([opus7.searchableContainer.SearchableContainer](#))

Base class from which all [Set](#) classes are derived.

Method resolution order:

[Set](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[__builtin__.object](#)

Methods defined here:

[__and__\(...\)](#)

Abstract method.

[__eq__\(...\)](#)

Abstract method.

[__init__\(self, universeSize\)](#)

([Set](#), int) -> None

Constructs a set with the given universe size.

[__lt__\(...\)](#)

Abstract method.

[__or__\(...\)](#)

Abstract method.

__sub__(...)

Abstract method.

find(self, i)

([Set](#), int) -> int

Returns the given integer if it is in this set.
Returns None otherwise.

getUniverseSize(self)

Static methods defined here:

__new__ = new(*args, **kwargs)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(s1, s2, s3)

[Set](#) test program.

Properties defined here:

universeSize

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__and__', '__contains__', '__eq__', '__invert__', '__or__', '__sub__', '_compareTo', 'insert', 'purge', 'withdraw']

Methods inherited from [opus7.searchableContainer.SearchableContainer](#)

__contains__(...)

Abstract method.

insert(...)
Abstract method.

withdraw(...)
Abstract method.

Static methods inherited from [opus7.searchableContainer.SearchableContainer](#):

main(*argv)
[SearchableContainer](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
(Container) -> string
Returns a string representation of this container.

accept(self, visitor)
(Container, Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool

Returns true if this container is full.

purge(...)

Abstract method.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the Object class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>

list of weak references to the object (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/26 14:21:18 \$'
version = '\$Revision: 1.16 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [SetAsArray](#) class.

Modules

[sys](#)

Classes

[opus7.set.Set](#)([opus7.searchableContainer.SearchableContainer](#))
[SetAsArray](#)

class **SetAsArray**([opus7.set.Set](#))

[Set](#) implemented using an array of boolean values.

Method resolution order:

[SetAsArray](#)

[opus7.set.Set](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[builtin.object](#)

Methods defined here:

[__and__](#)(self, set)

([SetAsArray](#), [SetAsArray](#)) -> None

Returns the intersection of this set and the given set.

[__contains__](#)(self, item)

([SetAsArray](#), int) -> bool

Returns true if the given element is in this set.

__eq__(self, set)

([SetAsArray](#), [SetAsArray](#)) -> bool

Returns true if this set is equal to the given set.

__init__(self, n)

([SetAsArray](#), int) -> None

Constructs a set with the given universe size.

__iter__(self)

([SetAsArray](#)) -> [SetAsArray](#).Iterator

Returns an interator for this set.

__lt__(self, set)

([SetAsArray](#), [SetAsArray](#)) -> bool

Returns true if this set is a proper subset to the given set.

__or__(self, set)

([SetAsArray](#), [SetAsArray](#)) -> None

Returns the union of this set and the given set.

__sub__(self, set)

([SetAsArray](#), [SetAsArray](#)) -> None

Returns the difference of this set and the given set.

accept(self, visitor)

([SetAsArray](#), Visitor) -> None

Makes the given visitor visit all the elements of this set.

getCount(self)

([SetAsArray](#)) -> None

Returns the number of elements in this set.

getIsEmpty(self)

([SetAsArray](#)) -> bool

Returns true if this set is empty.

getIsFull(self)

([SetAsArray](#)) -> bool

Returns true if this set is full.

insert(self, item)

([SetAsArray](#), int) -> None

Inserts the given element into this set.

purge(self)
 ([SetAsArray](#)) -> None
 Purges this set.

withdraw(self, item)
 ([SetAsArray](#), int) -> None
 Withdraws the given element from this set.

Static methods defined here:

__new__(self, *args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `MetaClass`.

main(*argv)
 [SetAsArray](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.setAsArray.Iterator'>
 Enumerates the elements of a [SetAsArray](#).

__abstractmethods__ = []

Methods inherited from [opus7.set.Set](#):

find(self, i)
 ([Set](#), int) -> int

Returns the given integer if it is in this set.
Returns None otherwise.

getUniverseSize(self)

Static methods inherited from [opus7.set.Set](#):

test(s1, s2, s3)

[Set](#) test program.

Properties inherited from [opus7.set.Set](#):

universeSize

get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int

Returns the hash of this container.

__str__(self)

(Container) -> string

Returns a string representation of this container.

elements(self)

(Container) -> Object

Generator that yields the objects in this container

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.25 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

opus7.setAsBitVector (version 1.27,
\$Date: 2003/09/26 14:21:18 \$)

[index](#)

[opus7/setAsBitVector.py](#)

Provides the [SetAsBitVector](#) class.

Modules

[sys](#)

[warnings](#)

Classes

[opus7.set.Set](#)([opus7.searchableContainer.SearchableContainer](#))
[SetAsBitVector](#)

class **SetAsBitVector**([opus7.set.Set](#))

[Set](#) implemented using a bit vector.

Method resolution order:

[SetAsBitVector](#)

[opus7.set.Set](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[builtin.object](#)

Methods defined here:

__and__(self, set)

([SetAsBitVector](#), [SetAsBitVector](#)) -> [SetAsBitVector](#)
Returns the intersection of this set an the given s

__contains__(self, item)

([SetAsBitVector](#), int) -> bool

Returns true if the given element is in this set.

__eq__(self, set)

([SetAsBitVector](#), [SetAsBitVector](#)) -> bool

Returns true if this set equals the given set.

__init__(self, n)

([SetAsBitVector](#), int) -> None

Constructs a set with the given universe size.

__iter__(self)

([SetAsBitVector](#)) -> [SetAsBitVector](#).Iterator

Returns an interator for this set.

__lt__(self, set)

([SetAsBitVector](#), [SetAsBitVector](#)) -> bool

Returns true if this set is a subset of the given set.

__or__(self, set)

([SetAsBitVector](#), [SetAsBitVector](#)) -> [SetAsBitVector](#)

Returns the union of this set an the given set.

__sub__(self, set)

([SetAsBitVector](#), [SetAsBitVector](#)) -> [SetAsBitVector](#)

Returns the difference of this set an the given set

accept(self, visitor)

([SetAsBitVector](#), Visitor) -> None

Makes the given visitor visit all the elements in this set.

getCount(self)

([SetAsBitVector](#)) -> int

Returns the number of elements in this set.

getIsEmpty(self)

([SetAsBitVector](#)) -> bool

Returns true if this set is empty.

getIsFull(self)

([SetAsBitVector](#)) -> bool

Returns true if this set is full.

insert(self, item)

([SetAsBitVector](#), int) -> None

Inserts the given element into this set.

purge(self)
 ([SetAsBitVector](#)) -> None
 Purges this set.

withdraw(self, item)
 ([SetAsBitVector](#), int) -> None
 Withdraws the given element from this set.

Static methods defined here:

__new__(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `MetaClass`.

main(*argv)
 [SetAsBitVector](#) test program.

Data and other attributes defined here:

BITS = 32

Iterator = <class 'opus7.setAsBitVector.Iterator'>
 Enumerates the elements of a [SetAsBitVector](#).

__abstractmethods__ = []

Methods inherited from [opus7.set.Set](#):

find(self, i)
 ([Set](#), int) -> int

Returns the given integer if it is in this set.
Returns None otherwise.

getUniverseSize(self)

Static methods inherited from [opus7.set.Set](#):

test(s1, s2, s3)
 Set test program.

Properties inherited from [opus7.set.Set](#):

universeSize
 get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
 (Container) -> int
 Returns the hash of this container.

__str__(self)
 (Container) -> string
 Returns a string representation of this container.

elements(self)
 (Container) -> Object

 Generator that yields the objects in this container

Properties inherited from [opus7.container.Container](#):

count
 get lambda self

isEmpty
 get lambda self

isFull
 get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
 Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 14:21:18 \$'
__version__ = '\$Revision: 1.27 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [SimpleRV](#) class.

Modules

[sys](#)

Classes

[opus7.randomVariable.RandomVariable](#)([opus7.object.Object](#))
[SimpleRV](#)

class **SimpleRV**([opus7.randomVariable.RandomVariable](#))

A random variable uniformly distributed on the interval

Method resolution order:

[SimpleRV](#)
[opus7.randomVariable.RandomVariable](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

getNext(self)
([SimpleRV](#)) -> double
Returns the next sample.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)
[SimpleRV](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.randomVariable.RandomVariable](#):

__init__(self)
([RandomVariable](#)) -> None
Constructor.

Properties inherited from [opus7.randomVariable.RandomVariable](#):

next
get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>



list of weak references to the object (if defined)

Data

```
RandomNumberGenerator =  
<opus7.randomNumberGenerator.RandomNumberGenerator object>  
  __author__ = 'Bruno R. Preiss, P.Eng.'  
  __credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
  __date__ = '$Date: 2003/09/26 14:46:27 $'  
  __version__ = '$Revision: 1.6 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Simulation](#) class.

Classes

[__builtin__.object](#) [Simulation](#)

class **Simulation**([__builtin__.object](#))

A discrete-event simulation of an M/M/1 queue.

Methods defined here:

__init__(self)

run(self, timeLimit)

 ([Simulation](#), double) -> None

 Runs the simulation up to the given time limit.

Data and other attributes defined here:

ARRIVAL = 0

DEPARTURE = 1

Event = <class 'opus7.simulation.Event'>

 Represents an event in the simulation.

__dict__ = <dictproxy object>

 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Simulation' objects>

 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.11 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Solution](#) class.

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Solution](#)

class Solution([opus7.object.Object](#))

Base class from which all solution space nodes are derived.

Method resolution order:

[Solution](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self\)](#)
([Solution](#)) -> None
Constructor.

[getBound\(...\)](#)
Abstract method.

[getIsComplete\(...\)](#)
Abstract method.

[getIsFeasible\(...\)](#)
Abstract method.

[getObjective\(...\)](#)
Abstract method.

getSuccessors(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Properties defined here:

bound

get lambda self

isComplete

get lambda self

isFeasible

get lambda self

objective

get lambda self

successors

get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['_compareTo', 'getBound', 'getIsComplete',
'getIsFeasible', 'getObjective', 'getSuccessors']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)
[Object](#) test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.11 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Solver](#) class.

Modules

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Solver](#)

class Solver(opus7.object.Object)

Base class from which all problem solvers are derived.

Method resolution order:

[Solver](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([Solver](#)) -> None
Constructor.

search(...)
Abstract method.

solve(self, initial)
([Solver](#), Solution) -> Solution
Solves a problem by searching the solution space

starting from the given node.

updateBest(self, solution)

([Solver](#), Solution) -> None

Records the given solution if it is complete, feasible and has a lower objective function value than the best solution seen so far.

Static methods defined here:

__new__(self, *args, **kwargs)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.

Raises a `TypeError` exception if the class is abstract.

This method is inserted as the method `__new__`

in classes instances derived from Metaclass.

Data and other attributes defined here:

__abstractmethods__ = ['search']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

([Object](#), [Object](#)) -> int

Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)

[Object](#) test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>

Metaclass of the [Object](#) class.

Prevents instantiation of classes that contain abstract methods.

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 17:41:12 \$'
__version__ = '\$Revision: 1.14 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [SortedList](#) class.

Classes

[opus7.orderedList.OrderedList](#)([opus7.searchableContainer.SearchableContainer](#))

[SortedList](#)

class SortedList([opus7.orderedList.OrderedList](#))

Base class from which all sorted list classes are derived.

Method resolution order:

[SortedList](#)

[opus7.orderedList.OrderedList](#)

[opus7.searchableContainer.SearchableContainer](#)

[opus7.container.Container](#)

[opus7.object.Object](#)

[__builtin__.object](#)

Methods defined here:

[__init__\(self\)](#)

([SortedList](#)) -> None

Constructor.

Static methods defined here:

[__new__\(self, *args, **kwargs\)](#)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.

Raises a `TypeError` exception if the class is abstract.

This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(list)
 [SortedList](#) test program.

Data and other attributes defined here:

`__abstractmethods__` = ['`__contains__`', '`__getitem__`', '`__iter__`', 'find', 'findPosition', 'insert', 'purge', 'withdraw']

Methods inherited from [opus7.orderedList.OrderedList](#):

__getitem__(...)
 Abstract method.

findPosition(...)
 Abstract method.

Methods inherited from [opus7.searchableContainer.SearchableCor](#)

__contains__(...)
 Abstract method.

find(...)
 Abstract method.

insert(...)
 Abstract method.

withdraw(...)
 Abstract method.

Static methods inherited from [opus7.searchableContainer.Searchat](#)

main(*argv)
 SearchableContainer test program.

Methods inherited from [opus7.container.Container](#):

`__hash__(self)`
`(Container) -> int`
Returns the hash of this container.

`__iter__(...)`
Abstract method.

`__str__(self)`
`(Container) -> string`
Returns a string representation of this container.

`accept(self, visitor)`
`(Container, Visitor) -> None`
Makes the given visitor visit all the items in this

`elements(self)`
`(Container) -> Object`
Generator that yields the objects in this container

`getCount(self)`
`(Container) -> int`
Returns the number of items in this container.

`getIsEmpty(self)`
`(Container) -> bool`
Returns true if this container is empty.

`getIsFull(self)`
`(Container) -> bool`
Returns true if this container is full.

`purge(...)`
Abstract method.

Properties inherited from [opus7.container.Container](#):

`count`
get lambda self

`isEmpty`
get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.sortedListAsArray](#) (version
1.17, \$Date: 2003/09/06 16:35:16 \$)

[index](#)

[opus7/sortedListAsArray.py](#)

Provides the [SortedListAsArray](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.orderedListAsArray.OrderedListAsArray](#)([opus7.orderedListAsArray.SortedListAsArray](#)([opus7.orderedListAsArray.OrderedListAsArray](#),
[opus7.sortedList.SortedList](#))
[opus7.sortedList.SortedList](#)([opus7.orderedList.OrderedList](#),
[SortedListAsArray](#)([opus7.orderedListAsArray.OrderedListAsArray](#),
[opus7.sortedList.SortedList](#))

class **SortedListAsArray**([opus7.orderedListAsArray.OrderedListAsArray](#),
[opus7.sortedList.SortedList](#))

A sorted list implemented using an array.

Method resolution order:

[SortedListAsArray](#)
[opus7.orderedListAsArray.OrderedListAsArray](#)
[opus7.sortedList.SortedList](#)
[opus7.orderedList.OrderedList](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin.object](#)

Methods defined here:

__init__(self, size=0)
 ([SortedListAsArray](#) [, int]) -> None
 Constructs a sorted list of the given size.

find(self, obj)
 ([SortedListAsArray](#), Object) -> Object
 Finds the object in this list that equals the given

findOffset(self, obj)
 ([SortedListAsArray](#), Object) -> int
 Finds the offset in this list
 of the object that equals the given object.

findPosition(self, obj)
 ([SortedListAsArray](#), Object) -> [SortedListAsArray](#).Cu
 Finds the position of an object in this sorted list
 that equals the given object and returns a cursor
 that refers to that object.

insert(self, obj)
 ([SortedListAsArray](#), Object) -> None
 Inserts the given object into this sorted list.

withdraw(self, obj)
 ([SortedListAsArray](#), Object) -> None
 Withdraws the given object from this sorted list.

Static methods defined here:

__new__ = new(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given args.
Raises a TypeError exception if the class is abstract.
This method is inserted as the method __new__
in classes instances derived from Metaclass.

main(*argv)
 [SortedListAsArray](#) test program.

Data and other attributes defined here:

Cursor = <class 'opus7.sortedListAsArray.Cursor'>
A cursor that refers to an object in a sorted list.

__abstractmethods__ = []

Methods inherited from [opus7.orderedListAsArray.OrderedListAs](#)

__contains__(self, obj)
([OrderedListAsArray](#), Object) -> bool
Returns true if the given object instance is in this list.

__getitem__(self, offset)
([OrderedListAsArray](#), int) -> Object
Returns the object in this list at the given __offset__.

__iter__(self)
([OrderedListAsArray](#)) -> [OrderedListAsArray](#).Iterator
Returns an iterator for this ordered list.

accept(self, visitor)
([OrderedListAsArray](#), Visitor) -> bool
Makes the given visitor visit the objects in this ordered list.

getIsFull(self)
([OrderedListAsArray](#)) -> bool
Returns true if this ordered list is full.

purge(self)
([OrderedListAsArray](#)) -> None
Purges this ordered list.

Data and other attributes inherited from
[opus7.orderedListAsArray.OrderedListAsArray](#):

Iterator = <class 'opus7.orderedListAsArray.Iterator'>
Enumerates the items in an ordered list.

Static methods inherited from [opus7.sortedList.SortedList](#):

test(list)
[SortedList](#) test program.

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.17 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.sortedListAsLinkedList](#)

(version 1.18, \$Date: 2003/09/14
22:24:08 \$)

[index](#)

[opus7/sortedListAsLinkedList.py](#)

Provides the [SortedListAsLinkedList](#) class.

Modules

[sys](#)

Classes

[opus7.orderedListAsLinkedList.OrderedListAsLinkedList](#)([opus7.o](#)
[SortedListAsLinkedList](#)([opus7.orderedListAsLinkedList.Orde](#)
[opus7.sortedList.SortedList](#))
[opus7.sortedList.SortedList](#)([opus7.orderedList.OrderedList](#))
[SortedListAsLinkedList](#)([opus7.orderedListAsLinkedList.Orde](#)
[opus7.sortedList.SortedList](#))

class

[SortedListAsLinkedList](#)([opus7.orderedListAsLinkedList.OrderedListAsLinkedList](#),
[opus7.sortedList.SortedList](#))

A sorted list implemented using a linked list.

Method resolution order:

[SortedListAsLinkedList](#)
[opus7.orderedListAsLinkedList.OrderedListAsLinkedList](#)
[opus7.sortedList.SortedList](#)
[opus7.orderedList.OrderedList](#)
[opus7.searchableContainer.SearchableContainer](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([SortedlistAsLinkedList](#)) -> None
Constructs a sorted list.

findElement(self, obj)
([SortedlistAsLinkedList](#), Object) -> [LinkedList.Element](#)
Finds the list element that contains an object
that equals the given object.

findPosition(self, obj)
([SortedlistAsLinkedList](#), Object) -> [SortedlistAsLinkedList.Cursor](#)
Finds the position of an object in this sorted list
that equals the given object and returns a cursor
that refers to that object.

insert(self, obj)
([SortedlistAsLinkedList](#), Object) -> None
Inserts the given object into this sorted list.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

main(*argv)
[SortedlistAsLinkedList](#) test program.

Data and other attributes defined here:

Cursor = <class 'opus7.sortedlistAsLinkedList.Cursor'>
A cursor that refers to an object in a sorted list.

__abstractmethods__ = []

Methods inherited from [opus7.orderedListAsLinkedList.OrderedList](#):

__contains__(self, obj)
`(OrderedListAsLinkedList, Object) -> bool`
Returns true if the given object instance is in this list.

__getitem__(self, offset)
`(OrderedListAsLinkedList, int) -> Object`
Returns the object in this list at the given offset.

__iter__(self)
`(OrderedListAsLinkedList) -> OrderedListAsLinkedList`
Returns an iterator for this ordered list.

accept(self, visitor)
`(OrderedListAsLinkedList, Visitor) -> None`
Makes the given visitor visit all the objects in this list.

find(self, arg)
`(OrderedListAsLinkedList, Object) -> Object`
Finds an object in this ordered list that equals the given argument.

purge(self)
`(OrderedListAsLinkedList) -> None`
Purges this ordered list.

withdraw(self, obj)
`(OrderedListAsLinkedList, Object) -> None`
Withdraws the given object instance from this ordered list.

Data and other attributes inherited from
[opus7.orderedListAsLinkedList.OrderedListAsLinkedList](#):

Iterator = <class 'opus7.orderedListAsLinkedList.Iterator'>
Enumerates the items in an ordered list.

Static methods inherited from [opus7.sortedList.SortedList](#):

test(list)
`SortedList test program.`

Methods inherited from [opus7.container.Container](#):

__hash__(self)

(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/14 22:24:08 \$'
__version__ = '\$Revision: 1.18 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Sorter](#) class.

Modules

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Sorter](#)

class Sorter(opus7.object.Object)

Base class from which all sorters are derived.

Method resolution order:

[Sorter](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self\)](#)
([Sorter](#)) -> None
Constructor.

[sort\(self, array\)](#)
([Sorter](#), Array) -> None
Sorts the given array.

[swap\(self, i, j\)](#)
([Sorter](#), int, int) -> None

Swaps the specified element of the array being sorted.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

test(sorter, n, seed, *args)

Data and other attributes defined here:

__abstractmethods__ = ['_sort']

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)
[Object](#) test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

```
RandomNumberGenerator =  
<opus7.randomNumberGenerator.RandomNumberGenerator object>  
__author__ = 'Bruno R. Preiss, P.Eng.'  
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
__date__ = '$Date: 2003/09/26 17:53:38 $'  
__version__ = '$Revision: 1.19 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [`SparseMatrix`](#) class.

Classes

[`opus7.matrix.Matrix`](#)([`__builtin__.object`](#))
[`SparseMatrix`](#)

class `SparseMatrix`([`opus7.matrix.Matrix`](#))

Base class from which all sparse matrix classes are derived.

Method resolution order:

[`SparseMatrix`](#)
[`opus7.matrix.Matrix`](#)
[`__builtin__.object`](#)

Methods defined here:

`__init__(self, numberOfRows, numberOfColumns)`
([`SparseMatrix`](#), int, int) -> None
Constructs a sparse matrix with the given number of

Methods inherited from [`opus7.matrix.Matrix`](#):

`getNumberOfColumns(self)`

`getNumberOfRows(self)`

Static methods inherited from [`opus7.matrix.Matrix`](#):

`test(mat)`

[Matrix](#) test program.

testTimes(mat1, mat2)
[Matrix](#) multiply test program.

testTranspose(mat)
[Matrix](#) transpose test program.

Properties inherited from [opus7.matrix.Matrix](#):

numberOfColumns

get lambda self

numberOfRows

get lambda self

transpose

get lambda self

Data and other attributes inherited from [opus7.matrix.Matrix](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Matrix' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/07/23 17:54:18 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.sparseMatrixAsArray](#)

(version 1.10, \$Date: 2003/09/23
12:23:21 \$)

[index](#)

[opus7/sparseMatrixAsArray.py](#)

Provides the [SparseMatrixAsArray](#) class.

Modules

[sys](#)

Classes

[opus7.sparseMatrix.SparseMatrix](#)([opus7.matrix.Matrix](#))
[SparseMatrixAsArray](#)

class SparseMatrixAsArray([opus7.sparseMatrix.SparseMatrix](#))

Sparse matrix implemented using a two-dimensional array.

Method resolution order:

[SparseMatrixAsArray](#)
[opus7.sparseMatrix.SparseMatrix](#)
[opus7.matrix.Matrix](#)
[__builtin__.object](#)

Methods defined here:

[**__getitem__\(self, indices\)**](#)
([SparseMatrixAsArray](#), (int, int)) -> Object
Returns the object at the given indices in this arr

[**__init__\(self, numberOfRows, numberOfColumns, fill\)**](#)
([SparseMatrixAsArray](#), int, int, int) -> None
Constructs a sparse matrix with the given number of
and with the given row fill (maximum number of item

__setitem__(self, indices, value)
[\(SparseMatrixAsArray, \(int, int\)\) -> Object](#)
Sets the object at the given indices in this array

findPosition(self, i, j)
[\(SparseMatrixAsArray, int, int\) -> int](#)
Returns the column in which the (i,j) element
of this matrix is stored.

getNumberOfColumns(self)
[\(SparseMatrixAsArray\) -> int](#)
Returns the number of columns in this matrix.

getNumberOfRows(self)
[\(SparseMatrixAsArray\) -> int](#)
Returns the number of rows in this matrix.

putZero(self, i, j)
[\(SparseMatrixAsArray, \(int, int\)\) -> None](#)
Zeroes the element at the given indices in this arr

Static methods defined here:

main(*argv)
[SparseMatrixAsArray](#) test program.

Data and other attributes defined here:

END_OF_ROW = -1

Static methods inherited from [opus7.matrix.Matrix](#):

test(mat)
Matrix test program.

testTimes(mat1, mat2)
Matrix multiply test program.

testTranspose(mat)
Matrix transpose test program.

Properties inherited from [opus7.matrix.Matrix](#):

numberOfColumns

get lambda self

numberOfRows

get lambda self

transpose

get lambda self

Data and other attributes inherited from [opus7.matrix.Matrix](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Matrix' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.sparseMatrixAsLinkedList](#)

(version 1.10, \$Date: 2003/09/23
12:23:21 \$)

in
[opus7/sparseMatrixAsLinkedList](#)

Provides the [SparseMatrixAsLinkedList](#) class.

Modules

[sys](#)

Classes

[opus7.sparseMatrix.SparseMatrix](#)([opus7.matrix.Matrix](#))
[SparseMatrixAsLinkedList](#)

class **SparseMatrixAsLinkedList**([opus7.sparseMatrix.SparseMatrix](#))

Sparse matrix implemented using a linked list of matrix entries.

Method resolution order:

[SparseMatrixAsLinkedList](#)
[opus7.sparseMatrix.SparseMatrix](#)
[opus7.matrix.Matrix](#)
[__builtin__.object](#)

Methods defined here:

__getitem__(self, indices)
([SparseMatrixAsLinkedList](#), (int, int)) -> Object
Returns the element of this matrix at the given index.

__init__(self, numberOfRows, numberOfColumns)
([SparseMatrixAsLinkedList](#), int, int) -> None
Constructs a sparse matrix with the given number of

__setitem__(self, indices, value)
[\(SparseMatrixAsLinkedList, \(int, int\), Object\) -> None](#)
Sets the element of this matrix at the given indices.

getNumberOfColumns(self)
[\(SparseMatrixAsLinkedList\) -> int](#)
Returns the number of columns in this matrix.

getNumberOfRows(self)
[\(SparseMatrixAsLinkedList\) -> int](#)
Returns the number of rows in this matrix.

getTranspose(self)
[\(SparseMatrixAsLinkedList\) -> SparseMatrixAsLinkedList](#)
Returns the transpose of this matrix.

putZero(self, i, j)
[\(SparseMatrixAsLinkedList, \(int, int\), Object\) -> None](#)
Sets the element of this matrix at the given indices.

Static methods defined here:

main(*argv)
[SparseMatrixAsLinkedList test program.](#)

Data and other attributes defined here:

Entry = <class 'opus7.sparseMatrixAsLinkedList.Entry'>
Represents a element of a sparse matrix.

Static methods inherited from [opus7.matrix.Matrix](#):

test(mat)
Matrix test program.

testTimes(mat1, mat2)
Matrix multiply test program.

testTranspose(mat)
Matrix transpose test program.

Properties inherited from [opus7.matrix.Matrix](#):

numberOfColumns

get lambda self

numberOfRows

get lambda self

transpose

get lambda self

Data and other attributes inherited from [opus7.matrix.Matrix](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Matrix' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 12:23:21 \$'
__version__ = '\$Revision: 1.10 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.sparseMatrixAsVector](#)

(version 1.11, \$Date: 2003/09/26
15:00:48 \$)

[index](#)

[opus7/sparseMatrixAsVector.py](#)

Provides the [SparseMatrixAsVector](#) class.

Modules

[sys](#)

Classes

[opus7.sparseMatrix.SparseMatrix](#)([opus7.matrix.Matrix](#))
[SparseMatrixAsVector](#)

class **SparseMatrixAsVector**([opus7.sparseMatrix.SparseMatrix](#))

Sparse matrix implemented as a vector of non-zero entries.

Method resolution order:

[SparseMatrixAsVector](#)
[opus7.sparseMatrix.SparseMatrix](#)
[opus7.matrix.Matrix](#)
[__builtin__.object](#)

Methods defined here:

__getitem__(self, indices)
([SparseMatrixAsVector](#), (int, int)) -> Object
Returns the element of this matrix at the given index.

__init__(self, numberOfRows, numberOfColumns, numberOfRowsElements,
([SparseMatrixAsVector](#), int, int, int) -> None
Constructs a sparse matrix with the given number of rows and columns and the given number of non-zero entries.

__mul__(self, mat)
([SparseMatrixAsVector](#), [SparseMatrixAsVector](#)) -> [SparseMatrixAsVector](#)
Returns the product of this matrix and the given matrix.

__setitem__(self, indices, value)
([SparseMatrixAsVector](#), (int, int), Object) -> None
Sets the element of this matrix at the given index.

findPosition(self, i, j)
([SparseMatrixAsVector](#), int, int) -> int
Returns the position in the vector of the entry with index (i,j).

getNumberOfColumns(self)
([SparseMatrixAsVector](#)) -> int
Returns the number of columns in this matrix.

getNumberOfRows(self)
([SparseMatrixAsVector](#)) -> int
Returns the number of rows in this matrix.

getTranspose(self)
([SparseMatrixAsVector](#)) -> [SparseMatrixAsVector](#)
Returns the transpose of this matrix.

putZero(self, i, j)
([SparseMatrixAsVector](#), int, int) -> None
Sets the element of this matrix at the given index.

Static methods defined here:

main(*argv)
[SparseMatrixAsVector](#) test program.

Data and other attributes defined here:

Entry = <class 'opus7.sparseMatrixAsVector.Entry'>
Represents an entry in this sparse matrix.

Static methods inherited from [opus7.matrix.Matrix](#):

test(mat)
Matrix test program.

testTimes(mat1, mat2)
Matrix multiply test program.

testTranspose(mat)
Matrix transpose test program.

Properties inherited from [opus7.matrix.Matrix](#):

numberOfColumns
get lambda self

numberOfRows
get lambda self

transpose
get lambda self

Data and other attributes inherited from [opus7.matrix.Matrix](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Matrix' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/26 15:00:48 \$'
__version__ = '\$Revision: 1.11 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Square](#) class.

Modules

[sys](#)

Classes

[opus7.rectangle.Rectangle](#)([opus7.graphicalObject.GraphicalObject](#))
[Square](#)

class **Square**([opus7.rectangle.Rectangle](#))

A square.

Method resolution order:

[Square](#)
[opus7.rectangle.Rectangle](#)
[opus7.graphicalObject.GraphicalObject](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self, center, width)
([Square](#), Point, width) -> None
Constructs a square with the given width (and height).

Static methods defined here:

__new__ = new(*args, **kwargs)

`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)

Square test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.rectangle.Rectangle](#):

draw(self)

Rectangle) -> None

Draws this rectangle.

Methods inherited from [opus7.graphicalObject.GraphicalObject](#):

erase(self)

GraphicalObject) -> None

Erases this graphical object.

moveTo(self, p)

GraphicalObject) -> None

Moves the center of this graphical object to the given point.

setPenColor(self, color)

Static methods inherited from [opus7.graphicalObject.GraphicalObject](#):

test(go)

GraphicalObject test program.

Data and other attributes inherited from [opus7.graphicalObject.GraphicalObject](#):

BACKGROUND_COLOR = 0

BACKGROUND_COLOR = 1

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/28 00:06:12 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Stack](#) class.

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[Stack](#)

class Stack([opus7.container.Container](#))

Base class from which all stack classes are derived.

Method resolution order:

[Stack](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([Stack](#)) -> None
Constructor

getTop(...)
Abstract method.

pop(...)
Abstract method.

push(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(stack)
Stack test program.

Properties defined here:

top
get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['__iter__', '_compareTo', 'getTop', 'pop',

Methods inherited from [opus7.container.Container](#):

__hash__(self)
([Container](#)) -> int
Returns the hash of this container.

__iter__(...)
Abstract method.

__str__(self)
([Container](#)) -> string
Returns a string representation of this container.

accept(self, visitor)
([Container](#), Visitor) -> None
Makes the given visitor visit all the items in this

elements(self)
([Container](#)) -> Object

Generator that yields the objects in this container

getCount(self)

([Container](#)) -> int

Returns the number of items in this container.

getIsEmpty(self)

([Container](#)) -> bool

Returns true if this container is empty.

getIsFull(self)

([Container](#)) -> bool

Returns true if this container is full.

purge(...)

Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)

[Container](#) test program.

Properties inherited from [opus7.container.Container](#):

count

get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>

Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst...

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/23 17:24:17 \$'
__version__ = '\$Revision: 1.24 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [StackAsArray](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.stack.Stack](#)([opus7.container.Container](#))
[StackAsArray](#)

class **StackAsArray**([opus7.stack.Stack](#))

[Stack](#) implemented using an array.

Method resolution order:

[StackAsArray](#)
[opus7.stack.Stack](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, size=0)
([StackAsArray](#) [, int]) -> None
Constructs a stack of the given size.

__iter__(self)
([StackAsArray](#)) -> [StackAsArray](#).Iterator
Returns an iterator for this stack.

accept(self, visitor)
 ([StackAsArray](#), Visitor) -> None
 Makes the given visitor visit all the objects in th.

getIsFull(self)
 ([StackAsArray](#)) -> bool
 Returns true if this stack is full.

getTop(self)
 ([StackAsArray](#)) -> Object
 Returns the object at the top of this stack.

pop(self)
 ([StackAsArray](#)) -> Object
 Pops the top object off this stack.

purge(self)
 ([StackAsArray](#)) -> None
 Purges this stack.

push(self, obj)
 ([StackAsArray](#), Object) -> None
 Pushes the given object on to this stack.

Static methods defined here:

__new__ = new(*args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
 [StackAsArray](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.stackAsArray.Iterator'>
 Enumerates the elements of a [StackAsArray](#).

__abstractmethods__ = []

Static methods inherited from [opus7.stack.Stack](#):

test(stack)
Stack test program.

Properties inherited from [opus7.stack.Stack](#):

top
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.28 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [StackAsLinkedList](#) class.

Modules

[exceptions](#) [sys](#)

Classes

[opus7.stack.Stack](#)([opus7.container.Container](#))
[StackAsLinkedList](#)

class **StackAsLinkedList**([opus7.stack.Stack](#))

[Stack](#) implemented using a linked list.

Method resolution order:

[StackAsLinkedList](#)
[opus7.stack.Stack](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

__init__(self)
([StackAsLinkedList](#))
Constructs a stack.

__iter__(self)
([StackAsLinkedList](#)) -> [StackAsLinkedList](#).Iterator
Returns an iterator for this stack.

accept(self, visitor)
 ([StackAsLinkedList](#), Visitor) -> None
 Makes the given visitor visit all the objects in th.

getTop(self)
 ([StackAsLinkedList](#)) -> None
 Returns the object at the top of this stack.

pop(self)
 ([StackAsLinkedList](#)) -> None
 Pops the top object off this stack.

purge(self)
 ([StackAsLinkedList](#)) -> None
 Purges this stack.

push(self, obj)
 ([StackAsLinkedList](#), Object) -> None
 Pushes the given object on to this stack.

Static methods defined here:

__new__(self, *args, **kwargs)
 (Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from `Metaclass`.

main(*argv)
 [StackAsLinkedList](#) test program.

Data and other attributes defined here:

Iterator = <class 'opus7.stackAsLinkedList.Iterator'>
 Enumerates the elements of a [StackAsLinkedList](#).

__abstractmethods__ = []

Static methods inherited from [opus7.stack.Stack](#):

test(stack)
Stack test program.

Properties inherited from [opus7.stack.Stack](#):

top
get lambda self

Methods inherited from [opus7.container.Container](#):

__hash__(self)
(Container) -> int
Returns the hash of this container.

__str__(self)
(Container) -> string
Returns a string representation of this container.

elements(self)
(Container) -> Object

Generator that yields the objects in this container

getCount(self)
(Container) -> int
Returns the number of items in this container.

getIsEmpty(self)
(Container) -> bool
Returns true if this container is empty.

getIsFull(self)
(Container) -> bool
Returns true if this container is full.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty

get lambda self

isFull

get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.30 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.straightInsertionSorter](#)

(version 1.8, \$Date: 2003/09/06
16:35:16 \$)

[index](#)

[opus7/straightInsertionSorter.py](#)

Provides the [StraightInsertionSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[StraightInsertionSorter](#)

class **StraightInsertionSorter**([opus7.sorter.Sorter](#))

Straight insertion sorter.

Method resolution order:

[StraightInsertionSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

[**__init__**](#)(self)
([StraightInsertionSorter](#)) -> None
Sorts the elements of the array.

Static methods defined here:

[**__new__**](#) = new(*args, **kwargs)

`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)

[StraightInsertionSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)

[\(Sorter, Array\)](#) -> None
Sorts the given array.

swap(self, i, j)

[\(Sorter, int, int\)](#) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

[\(Object, Object\)](#) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>

Metaclass of the `Object` class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.straightSelectionSorter](#)

(version 1.8, \$Date: 2003/09/06
16:35:16 \$)

[index](#)

[opus7/straightSelectionSorter.py](#)

Provides the [StraightSelectionSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[StraightSelectionSorter](#)

class **StraightSelectionSorter**([opus7.sorter.Sorter](#))

Straight selection sorter.

Method resolution order:

[StraightSelectionSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[_builtin_.object](#)

Methods defined here:

[**__init__**](#)(self)
([StraightSelectionSorter](#)) -> None
Constructor.

Static methods defined here:

[**__new__**](#) = new(*args, **kwargs)

`(Metaclass, ...) -> object`

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from `Metaclass`.

main(*argv)

[StraightSelectionSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)

[\(Sorter, Array\)](#) -> None
Sorts the given array.

swap(self, i, j)

[\(Sorter, int, int\)](#) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

[\(Object, Object\)](#) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>

dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>

Metaclass of the `Object` class.

Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [String](#) class.

Modules

[sys](#)

[warnings](#)

Classes

[opus7.object.Object\(__builtin__.object\)](#)
[String\(__builtin__.str, \[opus7.object.Object\]\(#\)\)](#)
[__builtin__.str\(__builtin__.basestring\)](#)
[String\(__builtin__.str, \[opus7.object.Object\]\(#\)\)](#)

class String(__builtin__.str, [opus7.object.Object](#))

[String](#) class.

Method resolution order:

[String](#)
[__builtin__.str](#)
[__builtin__.basestring](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__hash__\(self\)](#)
([String](#)) -> int
Hashes this string.

[__init__\(self, obj\)](#)

`(String, object) -> None`
Constructs a string with the string representation.
The [Object](#) metaclass provides a `__new__` method
that initializes the [str](#) instance, so none is defined.

Static methods defined here:

`__new__ = new(*args, **kwargs)`
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__`
in classes instances derived from Metaclass.

`main(*argv)`
[String](#) test program.

`testHash()`
[String](#) hash test program.

Data and other attributes defined here:

`__abstractmethods__ = []`

`__dict__ = <dictproxy object>`
dictionary for instance variables (if defined)

`mask = -33554432`

`shift = 6`

Methods inherited from [builtin .str](#):

`__add__(...)`
x.[__add__](#)(y) <==> x+y

`__contains__(...)`
x.[__contains__](#)(y) <==> y in x

`__eq__(...)`
x.[__eq__](#)(y) <==> x==y

__ge__(...)
x.__ge__(y) <==> x>=y

__getattribute__(...)
x.__getattribute__('name') <==> x.name

__getitem__(...)
x.__getitem__(y) <==> x[y]

__getnewargs__(...)

__getslice__(...)
x.__getslice__(i, j) <==> x[i:j]

Use of negative indices is not supported.

__gt__(...)
x.__gt__(y) <==> x>y

__le__(...)
x.__le__(y) <==> x<=y

__len__(...)
x.__len__() <==> len(x)

__lt__(...)
x.__lt__(y) <==> x<y

__mod__(...)
x.__mod__(y) <==> x%y

__mul__(...)
x.__mul__(n) <==> x*n

__ne__(...)
x.__ne__(y) <==> x!=y

__repr__(...)
x.__repr__() <==> repr(x)

__rmod__(...)
x.__rmod__(y) <==> y%x

__rmul__(...)

x.[rmul](#)(n) <==> n*x

—[str](#)(...)
x.[str](#)() <==> [str](#)(x)

capitalize(...)
S.[capitalize](#)() -> string

Return a copy of the string S with only its first character capitalized.

center(...)
S.[center](#)(width) -> string

Return S centered in a string of length width. Pads with spaces on both sides if necessary.

count(...)
S.[count](#)(sub[, start[, end]]) -> int

Return the number of occurrences of substring sub in S[start:end]. Optional arguments start and end are interpreted as in slice notation.

decode(...)
S.[decode](#)([encoding[,errors]]) -> object

Decodes S using the codec registered for encoding. If no encoding is given, it uses the default encoding. errors may be given to set the error handling scheme. Default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' as well as any other name registered with codecs.register_error that is able to handle UnicodeDecodeErrors.

encode(...)
S.[encode](#)([encoding[,errors]]) -> object

Encodes S using the codec registered for encoding. If no encoding is given, it uses the default encoding. errors may be given to set the error handling scheme. Default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that is able to handle UnicodeEncodeErrors.

endswith(...)
S.[endswith](#)(suffix[, start[, end]]) -> bool

Return True if S ends with the specified suffix, False otherwise.
With optional start, test S beginning at that position.
With optional end, stop comparing S at that position.

expandtabs(...)

S.[expandtabs](#)([tabsize]) -> string

Return a copy of S where all tab characters are expanded to spaces.
If tabsize is not given, a tab size of 8 characters is assumed.

find(...)

S.[find](#)(sub [,start [,end]]) -> int

Return the lowest index in S where substring sub is found.
Return -1 if sub is not found.
Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

index(...)

S.[index](#)(sub [,start [,end]]) -> int

Like S.[find\(\)](#) but raise ValueError when the substring is not found.

isalnum(...)

S.[isalnum](#)() -> bool

Return True if all characters in S are alphanumeric,
and there is at least one character in S, False otherwise.

isalpha(...)

S.[isalpha](#)() -> bool

Return True if all characters in S are alphabetic,
and there is at least one character in S, False otherwise.

isdigit(...)

S.[isdigit](#)() -> bool

Return True if there are only digit characters in S,
False otherwise.

islower(...)

S.[islower](#)() -> bool

Return True if all cased characters in S are lowercased,
and there is at least one cased character in S, False otherwise.

isspace(...)S.[isspace\(\)](#) -> bool

Return True if there are only whitespace characters
False otherwise.

istitle(...)S.[istitle\(\)](#) -> bool

Return True if S is a titlecased string, i.e. uppercase
may only follow uncased characters and lowercase char-
ones. Return False otherwise.

isupper(...)S.[isupper\(\)](#) -> bool

Return True if all cased characters in S are uppercase
at least one cased character in S, False otherwise.

join(...)S.[join](#)(sequence) -> string

Return a string which is the concatenation of the s-
sequence. The separator between elements is S.

ljust(...)S.[ljust](#)(width) -> string

Return S left justified in a string of length width
done using spaces.

lower(...)S.[lower\(\)](#) -> string

Return a copy of the string S converted to lowercase.

lstrip(...)S.[lstrip](#)([chars]) -> string or unicode

Return a copy of the string S with leading whitespace
removed. If chars is given and not None, remove characters in
chars. If chars is unicode, S will be converted to unicode.

replace(...)

S.replace (old, new[, maxsplit]) -> string

Return a copy of string S with all occurrences of s-

old replaced by new. If the optional argument maxsplit is given, only the first maxsplit occurrences are replaced.

rfind(...)

S.[rfind](#)(sub [,start [,end]]) -> int

Return the highest index in S where substring sub is found such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(...)

S.[rindex](#)(sub [,start [,end]]) -> int

Like S.[rfind\(\)](#) but raise ValueError when the substring is not found.

rjust(...)

S.[rjust](#)(width) -> string

Return S right justified in a string of length width, padded on the left with spaces if necessary. The padding is done using spaces.

rstrip(...)

S.[rstrip](#)([chars]) -> string or unicode

Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. If chars is unicode, S will be converted to unicode first.

split(...)

S.[split](#)([sep [,maxsplit]]) -> list of strings

Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, whitespace string is a separator.

splitlines(...)

S.[splitlines](#)([keepends]) -> list of strings

Return a list of the lines in S, breaking at line breaks. Line breaks are not included in the resulting list unless keepends is given and true.

startswith(...)

S.[startswith](#)(prefix[, start[, end]]) -> bool

Return True if S starts with the specified prefix, i.e. if S[:len(prefix)] == prefix. With optional start, test S beginning at that position. With optional end, stop comparing S at that position.

strip(...)

S.[strip](#)([chars]) -> string or unicode

Return a copy of the string S with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars from both ends. If chars is unicode, S will be converted to unicode first.

swapcase(...)

S.[swapcase](#)() -> string

Return a copy of the string S with uppercase characters converted to lowercase and vice versa.

title(...)

S.[title](#)() -> string

Return a titlecased version of S, i.e. words start with uppercase characters, all remaining cased characters have lowercase.

translate(...)

S.[translate](#)(table [,deletechars]) -> string

Return a copy of the string S, where all characters in the optional argument deletechars are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length len(deletechars).

upper(...)

S.[upper](#)() -> string

Return a copy of the string S converted to uppercase.

zfill(...)

S.[zfill](#)(width) -> string

Pad a numeric string S with zeros on the left, to form a string of the specified width. The string S is never truncated.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
 ([Object](#), [Object](#)) -> int
 Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
 Metaclass of the [Object](#) class.
 Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
 list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/25 01:07:38 \$'
__version__ = '\$Revision: 1.17 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides a class for measuring execution time.

Modules

[exceptions](#)

[sys](#)

[time](#)

Classes

[__builtin__.object](#)
[Timer](#)

class Timer([__builtin__.object](#))

A timer for measuring execution time.

Methods defined here:

__init__(self)
([Timer](#)) -> None

Constructs a timer.

getElapsedTime(self)
([Timer](#)) -> double

Returns the elapsed time.

start(self)
([Timer](#)) -> None

Starts this timer.

stop(self)
([Timer](#)) -> None

Stops this timer.

Static methods defined here:

main(*argv)
 [Timer](#) test program.

Data and other attributes defined here:

RUNNING = 2

STOPPED = 1

TOLERANCE = 100

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__weakref__ = <attribute '__weakref__' of 'Timer' objects>
 list of weak references to the [object](#) (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.4 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Tree](#) class.

Classes

[opus7.container.Container](#)([opus7.object.Object](#))
[Tree](#)

class Tree([opus7.container.Container](#))

Base class from which all tree classes are derived.

Method resolution order:

[Tree](#)
[opus7.container.Container](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([Tree](#)) -> None
Constructor.

__iter__(self)
([Tree](#)) -> [Tree.Iterator](#)
Returns an interator for this tree.

accept(self, visitor)
([Tree](#)) -> Visitor
Makes the given visitor visit the nodes of this tree.

breadthFirstGenerator(self)
([Tree](#)) -> generator

Yields the keys in this tree in depth-first traversal.

breadthFirstTraversal(self, visitor)

([Tree](#), Visitor) -> None

Makes the given visitor do a breadth-first traversal of this tree.

depthFirstGenerator(self, mode)

([Tree](#)) -> generator

Yields the keys in this tree in depth-first traversal.

depthFirstTraversal(self, visitor)

([Tree](#), PrePostVisitor) -> None

Makes the given visitor do a depth-first traversal of this tree.

getCount(self)

([Tree](#)) -> int

Returns the number of nodes in this tree.

getDegree(...)

Abstract method.

getHeight(self)

([Tree](#)) -> int

Returns the height of this tree.

getIsLeaf(...)

Abstract method.

getKey(...)

Abstract method.

getSubtree(...)

Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)

(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.

Raises a `TypeError` exception if the class is abstract.

This method is inserted as the method `__new__` in classes instances derived from Metaclass.

test(tree)
 [Tree](#) test program.

Properties defined here:

degree
 get lambda self

height
 get lambda self

isLeaf
 get lambda self

key
 get lambda self

Data and other attributes defined here:

INORDER = 0

Iterator = <class 'opus7.tree.Iterator'>
 Enumerates the nodes of a tree.

POSTORDER = 1

PREORDER = -1

__abstractmethods__ = ['_compareTo', 'getDegree', 'getIsLeaf', 'g
'getSubtree', 'purge']

Methods inherited from [opus7.container.Container](#):

__hash__(self)
 ([Container](#)) -> int
 Returns the hash of this container.

__str__(self)
 ([Container](#)) -> string
 Returns a string representation of this container.

elements(self)
[\(Container\)](#) -> Object
Generator that yields the objects in this container

getIsEmpty(self)
[\(Container\)](#) -> bool
Returns true if this container is empty.

getIsFull(self)
[\(Container\)](#) -> bool
Returns true if this container is full.

purge(...)
Abstract method.

Static methods inherited from [opus7.container.Container](#):

main(*argv)
[Container](#) test program.

Properties inherited from [opus7.container.Container](#):

count
get lambda self

isEmpty
get lambda self

isFull
get lambda self

Data and other attributes inherited from [opus7.container.Container](#)

StrVisitor = <class 'opus7.container.StrVisitor'>
Visitor that accumulates visited items in a string.

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)

`(Object, Object) -> int`
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

`__dict__` = <dictproxy object>
dictionary for instance variables (if defined)

`__metaclass__` = <class 'opus7.metaclass.Metaclass'>
Metaclass of the Object class.
Prevents instantiation of classes that contain abst

`__weakref__` = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

`__author__` = 'Bruno R. Preiss, P.Eng.'
`__credits__` = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
`__date__` = '\$Date: 2003/09/25 01:07:38 \$'
`__version__` = '\$Revision: 1.31 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [TwoWayMergeSorter](#) class.

Modules

[sys](#)

Classes

[opus7.sorter.Sorter](#)([opus7.object.Object](#))
[TwoWayMergeSorter](#)

class **TwoWayMergeSorter**([opus7.sorter.Sorter](#))

Two-way merge sorter.

Method resolution order:

[TwoWayMergeSorter](#)
[opus7.sorter.Sorter](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self)
([TwoWayMergeSorter](#)) -> None
Constructor.

merge(self, left, middle, right)
([TwoWayMergeSorter](#), int, int, int) -> None
Merges two sorted sub-arrays,
array[left] ... array[middle] and
array[middle + 1] ... array[right]

using the temporary array.

mergesort(self, left, right)
([TwoWayMergeSorter](#), int, int) -> None
Sorts the elements of the array array[left] ... arr

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[TwoWayMergeSorter](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.sorter.Sorter](#):

sort(self, array)
([Sorter](#), Array) -> None
Sorts the given array.

swap(self, i, j)
([Sorter](#), int, int) -> None
Swaps the specified element of the array being sorted.

Static methods inherited from [opus7.sorter.Sorter](#):

test(sorter, n, seed, *args)

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int

Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
 dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.Metaclass'>
 Metaclass of the Object class.
 Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
 list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/06 16:35:16 \$'
__version__ = '\$Revision: 1.8 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [UniformRV](#) class.

Modules

[sys](#)

Classes

[opus7.randomVariable.RandomVariable](#)([opus7.object.Object](#))
[UniformRV](#)

class **UniformRV**([opus7.randomVariable.RandomVariable](#))

A random variable uniformly distributed on the interval

Method resolution order:

[UniformRV](#)
[opus7.randomVariable.RandomVariable](#)
[opus7.object.Object](#)
[builtin .object](#)

Methods defined here:

__init__(self, u, v)
([UniformRV](#), double, double) -> None
Constructs a uniform random variable on the given interval.

getNext(self)
([UniformRV](#)) -> double
Returns the next sample.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[UniformRV](#) test program.

Data and other attributes defined here:

__abstractmethods__ = []

Properties inherited from [opus7.randomVariable.RandomVariable](#):

next

get lambda self

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
(Object, Object) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the `Object` class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

```
RandomNumberGenerator =  
<opus7.randomNumberGenerator.RandomNumberGenerator object>  
__author__ = 'Bruno R. Preiss, P.Eng.'  
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
__date__ = '$Date: 2003/09/26 15:02:08 $'  
__version__ = '$Revision: 1.7 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Vertex](#) class.

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Vertex](#)

class Vertex(opus7.object.Object)

Base class from which all graph vertex classes are derived.

Method resolution order:

[Vertex](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

__init__(self)
([Vertex](#)) -> None
Constructor.

getEmanatingEdges(...)
Abstract method.

getIncidentEdges(...)
Abstract method.

getNumber(...)
Abstract method.

getPredecessors(...)
Abstract method.

getSuccessors(...)
Abstract method.

getWeight(...)
Abstract method.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

Properties defined here:

emanatingEdges
get lambda self

incidentEdges
get lambda self

number
get lambda self

predecessors
get lambda self

successors
get lambda self

weight
get lambda self

Data and other attributes defined here:

__abstractmethods__ = ['_compareTo', 'getEmanatingEdges',
'getIncidentEdges', 'getNumber', 'getPredecessors', 'getSuccessors',

Methods inherited from [opus7.object.Object](#):

__cmp__ (self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Static methods inherited from [opus7.object.Object](#):

main(*argv)
[Object](#) test program.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metaclass.Metaclass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>
list of weak references to the object (if defined)

Data

__author__ = 'Bruno R. Preiss, P.Eng.'
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
__date__ = '\$Date: 2003/09/27 00:43:09 \$'
__version__ = '\$Revision: 1.12 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

 Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

Provides the [Visitor](#) class.

Modules

[sys](#)

Classes

[opus7.object.Object](#)([__builtin__.object](#))
[Visitor](#)

class Visitor([opus7.object.Object](#))
[Visitor](#) class.

Method resolution order:

[Visitor](#)
[opus7.object.Object](#)
[__builtin__.object](#)

Methods defined here:

[__init__\(self\)](#)
([Visitor](#)) -> None
Constructs this visitor.

[getIsDone\(self\)](#)
([Visitor](#)) -> bool
Default isDone_get method returns false always.

[visit\(self, obj\)](#)
([Visitor](#), [Object](#)) -> None

Default visit method does nothing.

Static methods defined here:

__new__ = new(*args, **kwargs)
(Metaclass, ...) -> object

Creates an instance of the class using the given arguments.
Raises a `TypeError` exception if the class is abstract.
This method is inserted as the method `__new__` in classes instances derived from Metaclass.

main(*argv)
[Visitor](#) test program.

Properties defined here:

isDone
get lambda self

Data and other attributes defined here:

__abstractmethods__ = []

Methods inherited from [opus7.object.Object](#):

__cmp__(self, obj)
([Object](#), [Object](#)) -> int
Compares this object with the given object.

Data and other attributes inherited from [opus7.object.Object](#):

__dict__ = <dictproxy object>
dictionary for instance variables (if defined)

__metaclass__ = <class 'opus7.metamodel.MetaClass'>
Metaclass of the [Object](#) class.
Prevents instantiation of classes that contain abst

__weakref__ = <attribute '__weakref__' of 'Object' objects>



list of weak references to the object (if defined)

Data

```
__author__ = 'Bruno R. Preiss, P.Eng.'  
__credits__ = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'  
__date__ = '$Date: 2003/09/25 01:07:38 $'  
__version__ = '$Revision: 1.24 $'
```

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

[opus7.zeroOneKnapsackProblem](#)

(version 1.16, \$Date: 2003/09/25

01:07:38 \$)

[opus7/zeroOneKnapsackProblem](#)

Provides the [ZeroOneKnapsackProblem](#) class.

Classes

[__builtin__.object](#)
[ZeroOneKnapsackProblem](#)

class ZeroOneKnapsackProblem([__builtin__.object](#))

Represents a zero-one knapsack problem.

Methods defined here:

[__init__\(self, weight, profit, capacity\)](#)
([ZeroOneKnapsackProblem](#), Array, Array, int) -
> None
Constructs a zero-
one knapsack problem with the given
array of weights, profits, and capacity.

[solve\(self, solver\)](#)
([ZeroOneKnapsackProblem](#), Solver) -> Solution
Solves this problem using the given solver.

Data and other attributes defined here:

[Node](#) = <class 'opus7.zeroOneKnapsackProblem.Node'>
Represents a node in the solution space
of this zero-one knapsack problem.

[__dict__](#) = <dictproxy object>
dictionary for instance variables (if defined)

[__weakref__](#) = <attribute '__weakref__' of

'ZeroOneKnapsackProblem' objects>
list of weak references to the [object](#) (if defined)

Data

author = 'Bruno R. Preiss, P.Eng.'
credits = 'Copyright (c) 2003 by Bruno R. Preiss, P.Eng.'
date = '\$Date: 2003/09/25 01:07:38 \$'
version = '\$Revision: 1.16 \$'

Author

Bruno R. Preiss, P.Eng.

Credits

Copyright (c) 2003 by Bruno R. Preiss, P.Eng.

...monotonicity

Don't worry if you don't know what that means. Essentially it says that Python's new-style classes ``do the right thing.''

...102#102.

The notation 103#103 denotes the *floor function* , which is defined as follows: For any real number x , 104#104 is the greatest integer less than or equal to x . While we are on the subject, there is a related function, the *ceiling function* , written 105#105. For any real number x , 106#106 is the smallest integer greater than or equal to x .

...119#119.

In fact, we would normally write 120#120, but we have not yet seen the 1#1 notation which is introduced in Chapter □.

...rule

Guillaume François Antoine de L'Hôpital, marquis de Sainte-Mesme, is known for his rule for computing limits which states that if $\lim_{x \rightarrow a} f(x) = \lim_{x \rightarrow a} g(x) = 0$ and $\lim_{x \rightarrow a} \frac{f'(x)}{g'(x)} = L$, then

359#359

where $f(n)$ and $g'(n)$ are the first derivatives with respect to n of $f(n)$ and $g(n)$, respectively. The rule is also effective if 360#360 and 361#361.

...commensurate.

Functions which are commensurate are functions which can be compared one with the other.

...436#436.

This notion of the looseness (tightness) of an asymptotic bound is related to but not exactly the same as that given in Definition \square .

...numbers.

Fibonacci numbers are named in honor of Leonardo Pisano (Leonardo of Pisa), the son of Bonaccio (in Latin, *Filius Bonacci*), who discovered the series in 1202.

...numbers.

These running times were measured on an Intel Pentium III, which has a 1 GHz clock and 256MB RAM under the Red Hat Linux 7.1 operating system. The programs were executed using the Python version 2.2.3 interpreter.

```
...object
```

Strictly speaking, this is true only for the so-called ``new-style'' Python classes which were introduced in Python version 2.2. Support for the so-called ``classic'' classes , which are not ultimately derived from the object class, is being phased out of the Python language. In this book we will use only Python new-style classes.

```
...class
```

For a complete list of the methods defined in the `__builtin__.object` class, you should consult the *Python Reference Manual*[[49](#)].

...NAME=7372>.

The word *deque* is usually pronounced like ``deck" and sometimes like ``deek."

...order

A *total order* is a relation, say $<$, defined on a set of elements, say 795#795, with the following properties:

1. For all pairs of elements 796#796, such that 797#797, exactly one of either $i < j$ or $j < i$ holds. (All elements are commensurate).
2. For all triples 798#798, 799#799. (The relation 394#394 is transitive).

(See also Definition □).

822#822

This is the Swedish word for the number two. The symbol å in the *Unicode character set* can be represented in a Python program using the *Unicode escape* ``\u00E5''.

...

822#822

I have been advised that a book with out sex will never be a best seller.
``Sex" is the Swedish word for the number six.

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...i.

What else would it be?

...NAME=14664> .

Isomorphic is a fancy word that means being of identical or similar form or shape or structure.

...Landis

Russian mathematicians G. M. Adel'son-Vel'skiı and E. M. Landis

published this result in 1962.

...trees.

Obviously since B-Trees are M -way trees, the ``B'' in *B-Tree* does not stand for *binary*. B-Trees were invented by R. Bayer and E. McCreight in 1972, so the ``B'' either stands for *balanced* or *Bayer*-take your pick.

...[HREF="page384.html#exercisepqueuesbinom">□](#)).

Isaac Newton discovered the binomial theorem in 1676 but did not publish a proof. Leonhard Euler attempted a proof in 1774. Karl Friedrich Gauss produced the first correct proof in 1812.

.
. .
. .
. .
. .

...subsets.

Stirling numbers of the second kind are given by the formula

1609#1609

where $n > 0$ and 1610#1610.

...explicitly

Note: The Python `del` operator does not destroy objects-it removes the

binding for a name from a namespace. That is, `del` disassociates a name from an object but leave the object itself intact.

...structures.

Mark-and-sweep garbage collection is described by John McCarthy in a paper on the LISP language published in 1960.

...space.

The reader may find it instructive to compare Program □ with Program □ and Program □.

...space.

The reader may find it instructive to compare Program □ with Program □ and Program □.

...NAME=33205> .

The table is named in honor of *Blaise Pascal* who published a treatise on the subject in 1653.

...number!

Prime numbers of the form $2^p - 1$ are known as *Mersenne primes*.

...1908#1908.

For convenience, we use the notation 1909#1909 to denote 1910#1910.

...NAME=35168> .

Unfortunately, the fame of bubble sort exceeds by far its practical value.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

.

<

...(respectively)

In this book, the names of instance attributes typically begin with an underscore ``_''.

Copyright © 2003 by Bruno R. Preiss, P.Eng. All rights reserved.

Bruno