2nd Edition

DATA STRUCTURE AND ALGORITHM

Simply In Depth

Ajit Singh

2nd Edition

DATA STRUCTURE AND ALGORITHM

Simply In Depth

Ajit Singh

Data Structure and Algorithm

Simply In Depth

Copyrighted Material

For information about this title or to order other books and/or electronic media, contact the publisher:

Ajit Singh
ajit_singh24@yahoo.com
http://www.ajitvoice.wordpress.in

Acknowledgement

This piece of study of Data Structure & Algorithm is an outcome of

the encouragement, guidance, help and assistance provided to me by my colleagues, Sr. faculties, Tech-friends and my family members.

Here, I am taking this opportunity to express my deep sense of gratitude to everybody whose roles were crucial in the successful completion this book, especially to my sr. students. This book benefited from discussions with many IT professionals (Ex-students) over the three years it took me to write it.

My primary goal is to provide a sufficient introduction and details of the Data Structure so that the students can have an efficient knowledge about Data Structure & Algorithm. Moreover, it presupposes knowledge of the principles and concepts. While reading this textbook, students may find the going somewhat tough in spite of my treating the topics in depth but in simple manner. In that case, reread the topic, if still it doesnt do the trick, then only you may need the help of expert. Any remaining errors and inaccuracies are my responsibility and any suggestions in this regard are warmly welcomed !

Special thanks go to my respected elder brother ***Er. Ranbir Singh,*** who encouraged me to utilize my skills in timely manner. He was my test audience representing non-technical readers; to the extent this book is accessible to people who aren't already programmers, that's largely his doing.

Last but not the least, I pay my sincere respect and regards to my Mother-In-Law ***Late Smt. Jayashree Singh*** and my Daughter whore ever obliged for bearing with me from time to time and insist me to share my knowledge across the world.

I would also like to thank those who provided the odd suggestion via email to us. All feedback was listened to and you will no doubt see some content influenced by your suggestions.

Finally, I would like to thank the **Kindle Direct Publishing** team and **Amazon** team for its enthusiastic online support and guidance in bringing out this book.

I hope that the reader likes this book and finds it useful in learning the concepts of Data Structure & Algorithm.

Thank You !!

Preface

Share the knowledge!!

Strong the surroundings.....

The study/learning of Data Struture & Algorithm is an essential part of any computer science education and of course for the B.Tech / MCA / M.Tech courses of several Universities across the world. This textbook is intended as a text for an explanatory course of Data Structure & Algorithm for Graduate and Post Graduate students of several universities across the world.

To The Student

This text is an introduction to the complex world of the Data Structure & Algorithm. A key factor of this book and its associated implementations is that all algorithms (unless otherwise stated) were designed by me, using the theory of the algorithm in question as a guideline (for which we are eternally grateful to their original creators). Therefore they may sometimes turn out to be worse than the normal implementationsn and sometimes not.

The book is the result of several decades of teaching experience in data structures and algorithms. The book is self-contained and does not assume any prior knowledge of data structures, just a

comprehension of basic programming and mathematics tools generally learned at the very beginning of computer science or other related studies. Through this book I hope that you will see the absolute necessity of understanding which data structure or algorithm to use for a certain scenario. In all projects, especially those that are concerned with performance (here we apply an even greater emphasis on real-time systems) the selection of the wrong data structure or algorithm can be the cause of a great deal of performance pain. The chapters cover: Models of Computation, Lists, Induction and Recursion, Trees, Algorithm Design, Hashing, Heaps, Balanced Trees, Sets Over a Small Universe, Graphs, Strings.

Therefore, it is absolutely key that you think about the run time complexity and space requirements of your selected approach. In this book, I only explain the theoretical implications to consider, but this is for a good reason: Compilers are very diifferent in how they work. One C++ compiler may have some amazing optimisation phases specifically targeted at recursion, another may not, for example. Of course this is just an example but you would be surprised by how many subtle diifferences there are between compilers. These diifferences which may make a fast algorithm slow, and vice versa. We could also factor in the same concerns about languages that target virtual machines, leaving all the actual various implementation issues to you given that you will know your language's compiler much better than me...well in most cases. This has resulted in a more concise book that focuses on what we think are the key issues.

Algorithms in the book are presented in a way that readers can easily understand the method of solving problems. Concepts are illustrated through examples. All programs in the text are tested.

All the algorithms in the book are provided in pseudocode, so that students can implement the algorithms in a programming language of their choice.

One final note: never take the words of others as gospel; verify all that can be feasibly verified and make up your own mind. I hope you enjoy reading this book as much as I have enjoyed writing it.

Feedback

I *have attempted to wash out every error in my first edition of this book after being reviewed by lots of bachelor of Computer Science, but as happens with Algorithm A few difficult to understand bugs shall remain - and I shall highly welcome the suggestions from the part of students that may lead to improvement of next edition in shortcoming future.*

Constructive suggestions and criticism always go a long way in enhancing any endeavour. I request all readers to email me their valuable comments / views / feedback for the betterment of the book at ***ajit_singh24@yahoo.com*** *mentioning the title and author name in the subject line. Please report any piracy spotted by you as well.* .*I* would be glad to hear suggestions from you.

Contents

Chapter 1
Introduction

1.1 What this book is, and what it isn't

This book provides implementations of common and uncommon algorithms in pseudocode which is language independent and provides for easy porting to most imperative programming languages.
You should use this book alongside another on the same subject, but one that contains formal proofs of the algorithms in question. In this book I use the abstract big Oh notation to depict the run time complexity of algorithms so that the book appeals to a larger audience.

1.2 Assumed knowledge

I have written this book with few assumptions of the reader, but some have been necessary in order to keep the book as concise and approachable as possible. I assume that the reader is familiar with the following:

Big Oh notation

An imperative programming language

Object oriented concepts

1.2.1 Big Oh notation

For run time complexity analysis we use big Oh notation extensively so it is vital that you are familiar with the general concepts to determine which is the best algorithm for you in certain scenarios. I have chosen to use big Oh notation for a few reasons, the most important of which is that it provides an abstract measurement by which we can judge the performance of algorithms without using mathematical proofs.

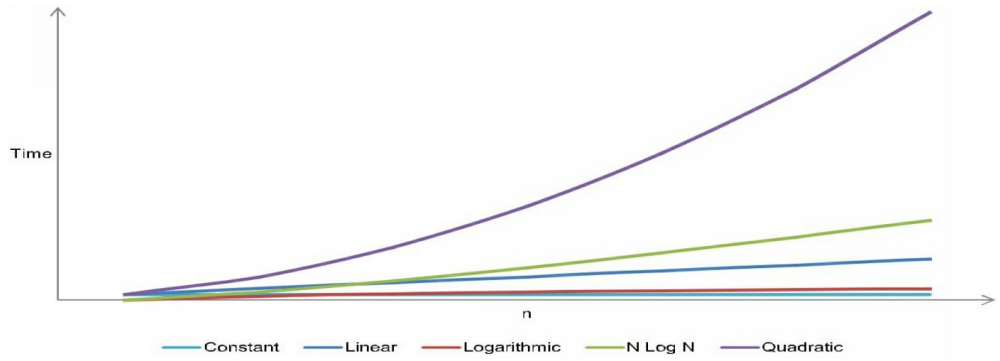


Figure 1.1: Algorithmic run time expansion

Figure 1.1 shows some of the run times to demonstrate how important it is to

choose an e±cient algorithm. For the sanity of our graph we have omitted cubic O(n3), and exponential O(2n) run times. Cubic and exponential algorithms should only ever be used for very small problems (if ever!); avoid them if feasibly possible.

Time complexity

Use of time complexity makes it easy to estimate the running time of a program. Performing an accurate calculation of a programs operation time is a very labour-intensive process (it depends on the compiler and the type of computer or speed of the processor). Therefore, we will not make an accurate measurement; just a measurement of a certain order of magnitude.

Complexity can be viewed as the maximum number of primitive operations that a program may execute. Regular operations are single additions, multiplications, assignments etc. We may leave some operations uncounted and concentrate on those that are performed the largest number of times. Such operations are referred to as dominant.

The number of dominant operations depends on the specific input data. We usually want to know how the performance time depends on a particular aspect of the data. This is most frequently the data size, but it can also be the size of a square matrix or the value of some input variable.

```
def dominant(n):
result = 0
for i in xrange(n):
result += 1
```

return result

The operation in line 4 is dominant and will be executed n times. The complexity is described in Big-O notation: in this case $O(n)$ *linear* complexity.

The complexity specifies the order of magnitude within which the program will perform its operations. More precisely, in the case of $O(n)$, the program may perform $c \cdot n$ operations, where c is a constant; however, it may not perform n^2 operations, since this involves a different order of magnitude of data. In other words, when calculating the complexity we omit constants: i.e. regardless of whether the loop is executed $20 \cdot n$ times or $\frac{n}{5}$ times, we still have a complexity of $O(n)$, even though the running time of the program may vary. When analyzing the complexity we must look for specific, worst-case examples of data that the program will take a long time to process.

Comparison of different time complexities

Lets compare some basic time complexities.

- **def** constant(n):
- result = n * n
- **return** result

There is always a fixed number of operations.

def logarithmic(n):

result = 0

hile n > 1: n //= 2

result += 1 **return** result

The value of n is halved on each iteration of the loop. If $n = 2^x$ then $\log n = x$. How long would the program below take to execute, depending on the input data?

def linear(n, A):

for i **in** xrange(n):

```
    if A[i] == 0:
            return 0
    return 1
```

Lets note that if the first value of array *A* is 0 then the program will end immediately. But remember, when analyzing time complexity we should check for worst cases. The program will take the longest time to execute if array *A* does not contain any 0.

```
def quadratic(n):
    result = 0
    for i in xrange(n):
        for j in xrange(i, n):
            result += 1
    return result
```

The result of the function equals $\frac{1}{2} \cdot (n \cdot (n + 1)) = \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n$ (the explanation is in the exercises). When calculating the complexity we are interested in a term that grows fastest, so we not only omit constants, but also other terms ($\frac{1}{2} \cdot n$ in this case). Thus we get quadratic time complexity. Sometimes the complexity depends on more variables (see example below).

```
def linear2(n, m):
    result = 0
    for i in xrange(n):
```

```
    result += i
    for j in xrange(m):
    result += j
    return result
```

2

Exponential and Factorial Time

It is worth knowing that there are other types of time complexity such as factorial time $O(n!)$ and exponential time $O(2^n)$. Algorithms with such complexities can solve problems only for very small values of n, because they would take too long to execute for large values of n.

Time limit

Nowadays, an average computer can perform 10^8 operations in less than a second. Sometimes we have the information we need about the expected time complexity (for example, Codility specifies the expected time complexity), but sometimes we do not.

The time limit set for online tests is usually from 1 to 10 seconds. We can therefore estimate the expected complexity. During contests, we are often given a limit on the size of data, and therefore we can guess the time complexity within which the task should be solved. This is usually a great convenience because we can look for a solution that works in a specific complexity instead of worrying about a faster solution. For example, if:

- $n \leq 1\,000\,000$, the expected time complexity is $O(n)$ or $O(n \log n)$,

- $n \lesssim 10\,000$, the expected time complexity is $O(n^2)$,
- $n \lesssim 500$, the expected time complexity is $O(n^3)$.

Of course, these limits are not precise. They are just approximations, and will vary depending on the specific task.

Space complexity

Memory limits provide information about the expected space complexity. You can estimate the number of variables that you can declare in your programs.

In short, if you have constant numbers of variables, you also have constant space com-plexity: in Big-O notation this is $O(1)$. If you need to declare an array with n elements, you have linear space complexity $O(n)$.

More specifically, space complexity is the amount of memory needed to perform the com-putation. It includes all the variables, both global and local, dynamic pointer data-structures and, in the case of recursion, the contents of the stack. Depending on the convention, input data may also be included. Space complexity is more tricky to calculate than time complexity because not all of these variables and data-structures may be needed at the same time. Global variables exist and occupy memory all the time; local variables (and additional information kept on the stack) will exist only during invocation of the function.

```
def slow_solution(n):
result = 0
for i in xrange(n):
    for j in xrange(i + 1): result += 1
 return result
```

Another person may increment the result respectively by 1, 2, . . . , n. This algorithm is much faster:

```
def fast_solution(n):
result = 0
for i in xrange(n):
result += (i + 1)
return result
```

But the third persons solution is even quicker. Let us write the sequence 1, 2, . . . , n and repeat the same sequence underneath it, but in reverse order. Then just add the numbers from the same columns:

1	2	3	. . .	$n \neq 1$	n
n	$n \neq 1$	$n \neq 2$	. . .	2	1
$n + 1$	$n + 1$	$n + 1$	. . .	$n + 1$	$n + 1$

The result in each column is $n + 1$, so we can easily count the final result:

The following list explains some of the most common big Oh notations:

$O(1)$ constant: the operation doesn't depend on the size of its input, e.g. adding a node to the tail of a linked list where we always maintain a pointer to the tail node.

$O(n)$ linear: the run time complexity is proportionate to the size of n.

$O(log\ n)$ logarithmic: normally associated with algorithms that break the problem into smaller chunks per each invocation, e.g. searching a binary search tree.

$O(n\ log\ n)$ just $n\ log\ n$: usually associated with an algorithm that breaks the problem into smaller chunks per each invocation, and then takes the results of these smaller chunks and stitches them back together, e.g. quick sort.

$O(n^2)$ quadratic: e.g. bubble sort.

$O(n^3)$ cubic: very rare.

$O(2^n)$ exponential: incredibly rare.

If you encounter either of the latter two items (cubic and exponential) this is really a signal for you to review the design of your algorithm. While prototyp-ing algorithm designs you may just have the intention of solving the problem irrespective of how fast it works. I would strongly advise that you always review your algorithm design and optimise where possible|particularly loops and recursive calls|so that you can get the most e ±cient run times for your algorithms.

The biggest asset that big Oh notation gives us is that it allows us to essentially discard things like hardware. If you have two sorting algorithms, one with a quadratic run time, and the other with a logarithmic run time then the logarithmic algorithm will always be faster than the quadratic one when the data set becomes suitably large. This applies even if the former is ran on a machine that is far faster than the latter. Why? Because big Oh notation isolates a key factor in algorithm analysis: growth. An algorithm with a quadratic run time grows faster than one with a logarithmic run time.

Big Oh notation also acts as a communication tool. Picture the scene: you are having a meeting with some fellow developers within your product group. You are discussing prototype algorithms for node discovery in massive networks. Several minutes elapse after you and two others have discussed your respective algorithms and how they work. Does this give you a good idea of how fast each respective algorithm is? No. The result of such a discussion will tell you more about the high level algorithm design rather than its e±ciency. Replay the scene back in your head, but this time as well as talking about algorithm design each respective developer states the asymptotic run time of their algorithm. Using the latter approach you not only get a good general idea about the algorithm design, but also key e±ciency data which allows you to make better choices when it comes to selecting an algorithm fit for purpose.

Some readers may actually work in a product group where they are given budgets per feature. Each feature holds with it a budget that represents its uppermost time bound. If you save some time in one feature it doesn't necessarily give you a buffer for the remaining features. Imagine you are

working on an application, and you are in the team that is developing the routines that will essentially spin up everything that is required when the application is started. Everything is great until your boss comes in and tells you that the start up time should not exceed *n* ms. The effciency of every algorithm that is invoked during start up in this example is absolutely key to a successful product. Even if you don't have these budgets you should still strive for optimal solutions.

Taking a quantitative approach for many software development properties will make you a far superior programmer - measuring one's work is critical to success.

1.2.2 Imperative programming language

All examples are given in a pseudo-imperative coding format and so the reader must know the basics of some imperative mainstream programming language to port the examples effectively, I have written this book with the following target languages in mind:

C++

C#

Java

The reason that we are explicit in this requirement is simple|all our implementations are based on an imperative thinking style. If you are a functional programmer you will need to apply various aspects from the functional paradigm to produce e±cient solutions with respect to your functional language whether it be Haskell, F#, OCaml, etc.

Two of the languages that we have listed (C# and Java) target virtual machines which provide various things like security sand boxing, and memory management via garbage collection algorithms. It is trivial to port our implementations to these languages. When porting to C++ you must remember to use pointers for certain things. For example, when we describe a linked list node as having a reference to the next node, this description is in the context of a managed environment. In C++ you should interpret the reference as a pointer to the next node and so on. For programmers who have a fair amount of experience with their respective language these subtleties will present no is-sue, which is why we really do emphasise that the reader must be comfortable with at least one imperative language in order to successfully port the pseudo-implementations in this book.

It is essential that the user is familiar with primitive imperative language constructs before reading this book otherwise you will just get lost. Some algorithms presented in this book can be confusing to follow even for experienced programmers !

1.2.3 Object oriented concepts

For the most part this book does not use features that are specific to any one language. In particular, I never provide data structures or algorithms that work on generic types|this is in order to make the samples as easy to follow as possible. However, to appreciate the designs of our data structures you will need to be familiar with the following object oriented (OO) concepts:

Inheritance

Encapsulation

Polymorphism

This is especially important if you are planning on looking at the C/C++/C# target that we have implemented (more on that in *x*1.7) which makes extensive use of the OO concepts listed above. As a final note it is also desirable that the reader is familiar with interfaces as the C++/C# target uses interfaces throughout the sorting algorithms.

1.3 Pseudocode

Throughout this book we use pseudocode to describe our solutions. For the most part interpreting the pseudocode is trivial as it looks very much like a more abstract C++, or C#, but there are a few things to point out:

1. Pre-conditions should always be enforced
2. Post-conditions represent the result of applying algorithm *a* to data structure*d.*
3. The type of parameters is inferred
4. All primitive language constructs are explicitly begun and ended

If an algorithm has a return type it will often be presented in the post-condition, but where the return type is su±ciently obvious it may be omitted for the sake of brevity.

Most algorithms in this book require parameters, and because we assign no explicit type to those parameters the type is inferred from the contexts in which it is used, and the operations performed upon it. Additionally, the name of the parameter usually acts as the biggest clue to its type. For instance *n* is a pseudoname for a number and so you can assume unless otherwise stated that *n* translates to an integer that has the same number of bits as a WORD on a 32 bit machine, similarly *l* is a pseudoname for a list where a list is a resizeable array (e.g. a vector).

The last major point of reference is that we always explicitly end a language construct. For instance if we wish to close the scope of a for loop we will explicitly state end for rather than leaving the interpretation of when scopes are closed to the reader. While implicit scope closure works well in simple code, in complex cases it can lead to ambiguity.

The pseudocode style that we use within this book is rather straightforward. All algorithms start with a simple algorithm signature, e.g.

algorithm AlgorithmName(*arg*1, *arg*2, ..., *argN*)
...
end AlgorithmName

Immediately after the algorithm signature we list any Pre or Post conditions.

algorithm AlgorithmName(*n*)
    Pre: *n* is the value to compute the factorial of *n*>=
    Post: the factorial of *n* has been computed
    // ...
end AlgorithmName

The example above describes an algorithm by the name of *AlgorithmName*, which takes a single numeric parameter *n*. The pre and post conditions follow the algorithm signature; you should always enforce the pre-conditions of an algorithm when porting them to your language of choice.

Normally what is listed as a pre-conidition is critical to the algorithms operation. This may cover things like the actual parameter not being null, or that the collection passed in must contain at least *n* items. The post-condition mainly describes the effect of the algorithms operation. An example of a

post-condition might be - The list has been sorted in ascending order.

Because everything we describe is language independent you will need to make your own mind up on how to best handle pre-conditions. For example, in the C++ / C# target I have implemented, I consider non-conformance to pre-conditions to be exceptional cases. I provide a message in the exception to tell the caller why the algorithm has failed to execute normally.

1.4 Tips for working through the examples

As with most books you get out what you put in and so I recommend that in order to get the most out of this book you work through each algorithm with a pen and paper to track things like variable names, recursive calls etc.

The best way to work through algorithms is to set up a table, and in that table give each variable its own column and continuously update these columns. This will help you keep track of and visualise the mutations that are occurring throughout the algorithm. Often while working through algorithms in such a way you can intuitively map relationships between data structures rather than trying to work out a few values on paper and the rest in your head. We suggest you put everything on paper irrespective of how trivial some variables and calculations may be so that you always have a point of reference.

When dealing with recursive algorithm traces we recommend you do the same as the above, but also have a table that records function calls and who they return to. This approach is a far cleaner way than drawing out an elaborate map of function calls with arrows to one another, which gets large quickly and simply makes things more complex to follow. Track everything

in a simple and systematic way to make your time studying the implementations far easier.

1.5 Book outline

I have split this book into two parts:

Part 1: Provides discussion and pseudo-implementations of common and uncommon data structures; and

Part 2: Provides algorithms of varying purposes from sorting to string operations.

The reader doesn't have to read the book sequentially from beginning to end: chapters can be read independently from one another. I suggest that in part 1 you read each chapter in its entirety, but in part 2 you can get away with just reading the section of a chapter that describes the algorithm you are interested in.

Each of the chapters on data structures present initially the algorithms concerned with:
Insertion
Deletion
Searching

The previous list represents what we believe in the vast majority of cases to be the most important for each respective data structure.

For all readers we recommend that before looking at any algorithm you

quickly look at Appendix D which contains a table listing the various symbols used within our algorithms and their meaning. One keyword that we would like to point out here is yield. You can think of yield in the same light as return. The return keyword causes the method to exit and returns control to the caller, whereas yield returns each value to the caller. With yield control only returns to the caller when all values to return to the caller have been exhausted.

1.6 Where can I get the code?

This book doesn't provide any code specifically aligned with it i any language

.

1.7 Final messages

I have just a few final messages to the reader that we hope you digest before you embark on reading this book:

1. Understand how the algorithm works first in an abstract sense; and
2. Always work through the algorithms on paper to understand how they achieve their outcome

If you always follow these key points, you will get the most out of this book.

Part I

Data Structures

Chapter 2

Data Structure

Data structure is an organization of data in computer's me mory. It makes the data quickly available to the processor for required operations.
It is a software artifact which allows data to be stored, organized and accessed.
It is a structure program used to store ordered data, so that various operations can be performed on it easily.

For example, if we have an employee's data like name 'ABC' and salary 10000. Here, 'Ajit' is of String data type and 10000 is of Float data type.
We can organize this data as a record like Employee record and collect & store employee's records in a file or database as a data structure like 'Ajit' 10000'.

Data structure is about providing data elements in terms of some relationship for better organization and storage.

It is a specialized format for organizing and storing data that can be accessed within appropriate ways.

Why is Data Structure important?

Data structure is important because it is used in almost every program or software system.

It helps to write efficient code, structures the code and solve problems.
Data can be maintained more easily by encouraging a better design or implementation.

Data structure is just a container for the data that is used to store, manipulate and arrange. It can be processed by algorithms.

For example, while using a shopping website like Flipkart or Amazon, the

users know their last orders and can track them. The orders are stored in a database as records.

However, when the program needs them so that it can pass the data somewhere else (such as to a warehouse) or display it to the user, it loads the data in some form of data structure.

Types of Data Structure

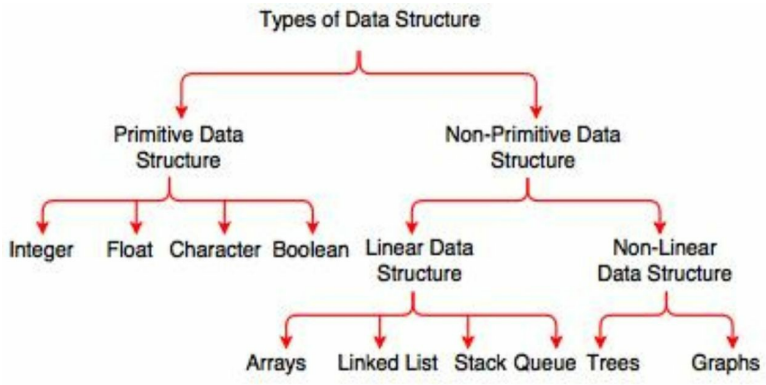


Fig. Types of Data Structure

A. Primitive Data Type

- Primitive data types are the data types available in most of the programming languages.
- These data types are used to represent single value.
- It is a basic data type available in most of the programming language.

Data type	Description
Integer	Used to represent a number without decimal point.
Float	Used to represent a number with decimal

	point.
Character	Used to represent single character.
Boolean	Used to represent logical values either true or false.

B. Non-Primitive Data Type

- Data type derived from primary data types are known as Non-Primitive data types.
- Non-Primitive data types are used to store group of values.

It can be divided into two types:

1. Linear Data Structure

- Linear data structure traverses the data elements sequentially.
- In linear data structure, only one data element can directly be reached.
- It includes array, linked list, stack and queues.

Types	Description
Arrays	Array is a collection of elements. It is used in mathematical proble matrix, algebra etc. each element of an array is referenced by a subs variable or value, called subscript or index enclosed in parenthesis.
Linked list	Linked list is a collection of data elements. It consists of two parts: I Link. Info gives information and Link is an address of next node. Lin can be implemented by using pointers.
Stack	Stack is a list of elements. In stack, an element may be inserted or de one end which is known as Top of the stack. It performs two operation and Pop. Push means adding an element in stack and Pop means remo element in stack. It is also called Last-in-First-out (LIFO).
Queue	Queue is a linear list of element. In queue, elements are added at c called rear and the existing elements are deleted from other end called is also called as First-in-First-out (FIFO).

2. Non-Linear Data Structure

- Non-Linear data structure is opposite to linear data structure.
- In non-linear data structure, the data values are not arranged in order and a data item is connected to several other data items.
- It uses memory efficiently. Free contiguous memory is not required for allocating data items.
- It includes trees and graphs.

Type	Description
Tree	Tree is a flexible, versatile and powerful non-linear structure. It is used to represent data it processing hierarchical relationship between grandfather and his children & grandchildren. It i ideal data structure for representing hierarchical data.
Graph	Graph is a non-linear data structure which consists finite set of ordered pairs called edges. Graph is a se elements connected by edges. Each element is call vertex and node.

Abstract Data Types

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called abstract because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.

The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented. So a user only needs to know what a data type can do but not how it will do it. We can think of ADT as a black box which hides the inner structure and design of the data type.

- ADT stands for **Abstract Data Type.**
- It is an abstraction of a data structure.
- Abstract data type is a mathematical model of a data structure.
- It describes a container which holds a finite number of objects where

  the objects may be associated through a given binary relationship.

- It is a logical description of how we view the data and the operations allowed without regard to how they will be implemented.
- ADT concerns only with what the data is representing and not with how it will eventually be constructed.
- It is a set of objects and operations. For example, List, Insert, Delete, Search, Sort.

It consists of following three parts:

1. Data describes the structure of the data used in the ADT.

2. Operation describes valid operations for the ADT. It describes its interface.

3. Error describes how to deal with the errors that can occur.

Advantages of ADT

- ADT is reusable and ensures robust data structure.
- It reduces coding efforts.
- Encapsulation ensures that data cannot be corrupted.
- ADT is based on principles of Object Oriented Programming (OOP) and Software Engineering (SE).
- It specifies error conditions associated with operations.

Now Ill define three ADTs namely List ADT, Stack ADT, Queue ADT.

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list.

get() Return an element from the list at any given position.

insert() Insert an element at any position of the list.

remove() Remove the first occurrence of any element from a non-empty list.

removeAt() Remove the element at a specified location from a non-empty list.

replace() Replace an element at any position by another element.

size() Return the number of elements in the list.

isEmpty() Return true if the list is empty, otherwise return false.

isFull() Return true if the list is full, otherwise return false.

Stack ADT

A Stack contains elements of same type arranged in sequential order. All operations takes place at a single end that is top of the stack and following operations can be performed:

push() Insert an element at one end of the stack called top.

pop() Remove and return the element at the top of the stack, if it is not empty.

peek() Return the element at the top of the stack without removing it, if the stack is not empty.

size() Return the number of elements in the stack.

isEmpty() Return true if the stack is empty, otherwise return false.

isFull() Return true if the stack is full, otherwise return false.

Queue ADT

A Queue contains elements of same type arranged in sequential order. Operations takes place at both ends, insertion is done at end and deletion is

done at front. Following operations can be performed:
enqueue() Insert an element at the end of the queue.
dequeue() Remove and return the first element of queue, if the queue is not empty.
peek() Return the element of the queue without removing it, if the queue is not empty.
size() Return the number of elements in the queue.
isEmpty() Return true if the queue is empty, otherwise return false.
isFull() Return true if the queue is full, otherwise return false.

Linked Lists

Linked lists can be thought of from a high level perspective as being a series of nodes. Each node has at least a single pointer to the next node, and in the last node's case a null pointer representing that there are no more nodes in the linked list.

In DSA our implementations of linked lists always maintain head and tail pointers so that insertion at either the head or tail of the list is a constant time operation. Random insertion is excluded from this and will be a linear operation. As such, linked lists in DSA have the following characteristics:

Insertion is $O(1)$
Deletion is $O(n)$
Searching is $O(n)$

Out of the three operations the one that stands out is that of insertion. In DSA we chose to always maintain pointers (or more aptly references) to the node(s) at the head and tail of the linked list and so performing a traditional insertion to either the front or back of the linked list is an $O(1)$ operation. An exception to this rule is performing an insertion before a node that is neither the head nor tail in a singly linked list. When the node we are inserting before is somewhere in the middle of the linked list (known as random insertion) the complexity is $O(n)$. In order to add before the designated node we need to traverse the linked list to find that node's current predecessor. This traversal yields an $O(n)$ run time.

This data structure is trivial, but linked lists have a few key points which at times make them very attractive:

1. The list is dynamically resized, thus it incurs no copy penalty like an array or vector would eventually incur; and

2. Insertion is $O(1)$.

2.1 Singly Linked List

Singly linked lists are one of the most primitive data structures you will find in this book. Each node that makes up a singly linked list consists of a value, and a reference to the next node (if any) in the list.

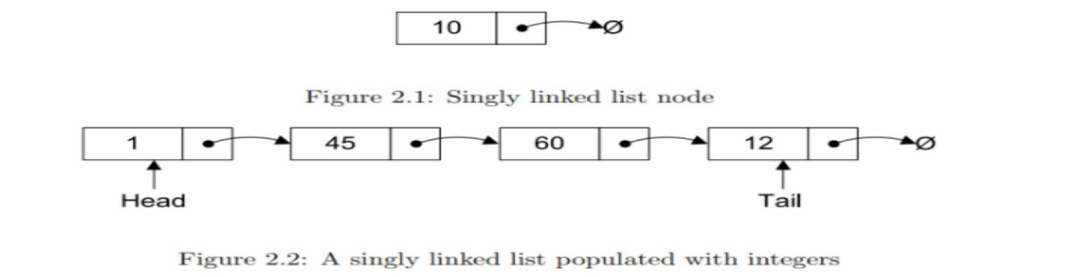


Figure 2.1: Singly linked list node

Figure 2.2: A singly linked list populated with integers

2.1.1 Insertion

In general when people talk about insertion with respect to linked lists of any form they implicitly refer to the adding of a node to the tail of the list. When you use an API like that of DSA and you see a general purpose method that adds a node to the list, you can assume that you are adding the node to the tail of the list not the head.

Adding a node to a singly linked list has only two cases:

1. *head* =null in which case the node we are adding is now both the *head* and *tail* of the list; or

2. we simply need to append our node onto the end of the list updating the *tail* reference appropriately.

algorithm Add(*value*)

Pre: *value* is the value to add to the list

Post: *value* has been placed at the tail of the list

n <- node(*value*)

if *head* =null*;*

head <- *n*

```
        tail <- n

      else
        tail.Next <- n

        tail <- n
      end if
end Add
```

As an example of the previous algorithm consider adding the following sequence of integers to the list: 1, 45, 60, and 12, the resulting list is that of Figure 2.2.

2.1.2 Searching

Searching a linked list is straightforward: we simply traverse the list checking the value we are looking for with the value of each node in the linked list. The algorithm listed in this section is very similar to that used for traversal in *x*2.1.4.

```
algorithm Contains(head, value)

    Pre: head is the head node in the list

       value is the value to search for
    Post: the item is either in the linked list, true; otherwise false

    n <- head

    while n !=null and n.Value != value

      n <- n.Next

    end while
```

```
        if n = null
          return false
        end if
        return true
    end Contains
```

2.1.3 Deletion

Deleting a node from a linked list is straightforward but there are a few cases we need to account for:

1. the list is empty; or
2. the node to remove is the only node in the linked list; or
3. we are removing the head node; or
4. we are removing the tail node; or
5. the node to remove is somewhere in between the head and tail; or
6. the item to remove doesn't exist in the linked list

The algorithm whose cases we have described will remove a node from anywhere within a list irrespective of whether the node is the *head* etc. If you know that items will only ever be removed from the *head* or *tail* of the list then you can create much more concise algorithms. In the case of always removing from the front of the linked list deletion becomes an $O(1)$ operation.

```
algorithm Remove(head, value)
    Pre: head is the head node in the list
         value is the value to remove from the list
    Post: value is removed from the list, true; otherwise false
    if head = null
        // case 1
        return false
    end if
    n <- head
    if n.Value = value
     if head = tail
      // case 2
      head <- null
      tail <- null
     else
      // case 3
      head <- head.Next
     end if
     return true
    end if
while n.Next !=null and n.Next.Value != value
```

```
 n <- n.Next

  end while
  if n.Next !=null

 if n.Next = tail
// case 4

tail <- n

 end if
 // case 5

 n.Next <- n.Next.Next

 return true

  end if

  // case 6

  return false

   end Remove
```

2.1.4 Traversing the list

Traversing a singly linked list is the same as that of traversing a doubly linked list (defined in *x*2.2). You start at the head of the list and continue until you come across a node that is *;*. The two cases are as follows:

1. *node* = null we have exhausted all nodes in the linked list; or

2. we must update the *node* reference to be *node*.Next.

The algorithm described is a very simple one that makes use of a simple *while* loop to check the first case.

```
  algorithm Traverse(head)

 Pre: head is the head node in the list

 Post: the items in the list have been traversed

 n <- head

 while n != 0

yield n.Value
n <- n.Next

 end while
  end Traverse
```

2.1.5 Traversing the list in reverse order

Traversing a singly linked list in a forward manner (i.e. left to right) is simple as demonstrated in *x*2.1.4. However, what if we wanted to traverse the nodes in the linked list in reverse order for some reason? The algorithm to perform such a traversal is very simple, and just like demonstrated in *x*2.1.3 we will need to acquire a reference to the predecessor of a node, even though the fundamental characteristics of the nodes that make up a singly linked list make this an expensive operation. For each node, finding its predecessor is an $O(n)$ operation, so over the course of traversing the whole list backwards the cost becomes $O(n^2)$.

Figure 2.3 depicts the following algorithm being applied to a linked list with the integers 5, 10, 1, and 40.

```
algorithm ReverseTraversal(head, tail)
    Pre: head and tail belong to the same list
    Post: the items in the list have been traversed in reverse order
    if tail != null
     curr <- tail
     while curr != head
       prev <- head
       while prev.Next != curr
         prev <- prev.Next
       end while
       yield curr.Value
       curr <- prev
     end while
     yield curr.Value
    end if
end ReverseTraversal
```

This algorithm is only of real interest when we are using singly linked lists, as you will soon see that doubly linked lists (defined in *x*2.2) make reverse list traversal simple and e±cient, as shown in *x*2.2.3.

2.2 Doubly Linked List

Doubly linked lists are very similar to singly linked lists. The only difference is that each node has a reference to both the next and previous nodes in the list.

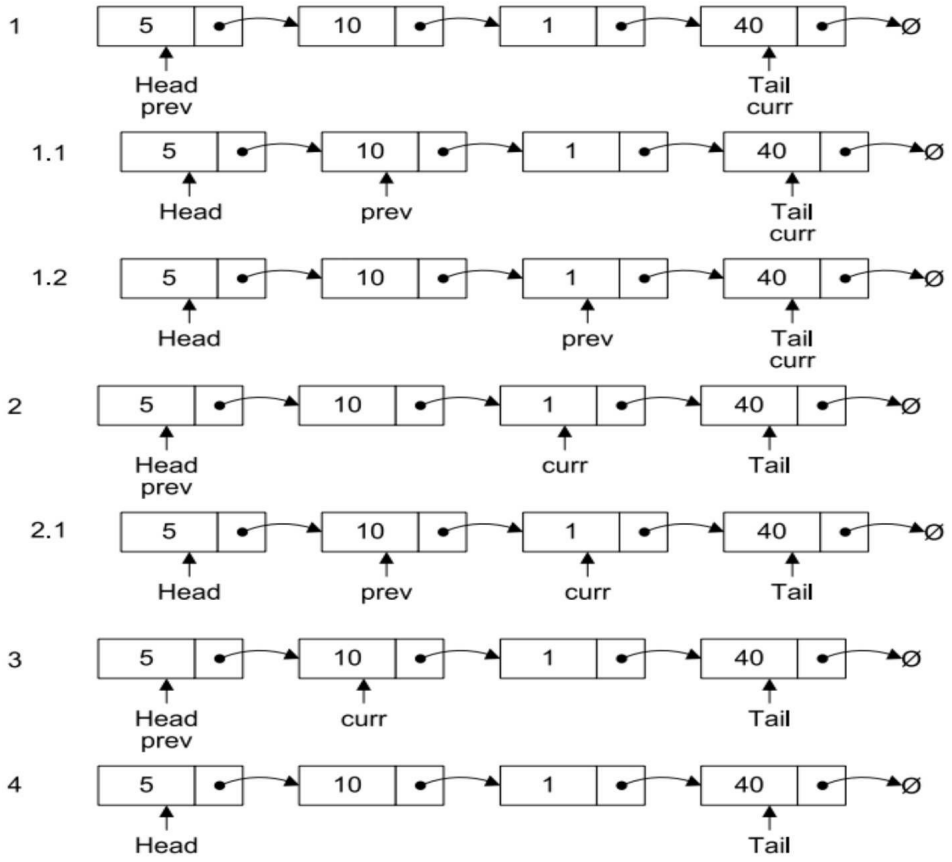


Figure 2.3: Reverse traveral of a singly linked list

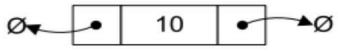


Figure 2.4: Doubly linked list node

The following algorithms for the doubly linked list are exactly the same as those listed
previously for the singly linked list:

1. Searching

2. Traversal

2.2.1 Insertion

The only major difference between the algorithm in *x*2.1.1 is that we need to remember to bind the previous pointer of *n* to the previous tail node if *n* was not the first node to be inserted into the list.

algorithm Add(*value*)

Pre: *value* is the value to add to the list

Post: *value* has been placed at the tail of the list

n <- node(*value*)

if *head* = null

head <- *n*

tail <- *n*

else
n.Previous <- *tail*

tail.Next <- *n*

tail <- *n*
end if
end Add

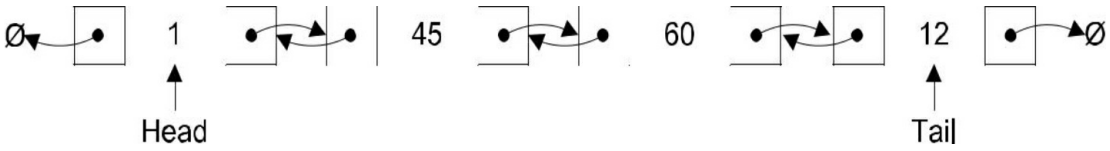


Figure 2.5 shows the doubly linked list after adding the sequence of 1.

Figure 2.5: Doubly linked list populated with integers

2.2.2 Deletion

As you may of guessed the cases that we use for deletion in a doubly linked list are exactly the same as those defined in *x*2.1.3. Like insertion we have the added task of binding an additional reference (*P revious*) to the correct value.

algorithm Remove(*head*, *value*)

Pre: *head* is the head node in the list

value is the value to remove from the list

Post: *value* is removed from the list, true; otherwise false

if *head* = null

```
 return false
  end if
  if value = head.Value
 if head = tail
head <- null
tail <- null
 else
head <- head.Next
head.Previous <- null;
 end if
 return true
  end if
  n <- head.Next
  while n !=null and value != n.Value
 n <- n.Next
  end while
  if n = tail
 tail <- tail.Previous
 tail.Next <- null
 return true
```

```
 else if n != null;
n.Previous.Next <- n.Next
n.Next.Previous <- n.Previous
return true
 end if
 return false
  end Remove
```

2.2.3 Reverse Traversal

Singly linked lists have a forward only design, which is why the reverse traversal algorithm defined in x2.1.5 required some creative invention. Doubly linked lists make reverse traversal as simple as forward traversal (defined in x2.1.4) except that we start at the tail node and update the pointers in the opposite direction. Figure 2.6 shows the reverse traversal algorithm in action.

```
 algorithm ReverseTraversal(tail)
     Pre: tail is the tail node of the list to traverse
     Post: the list has been traversed in reverse order
     n <- tail
     while n !=null;
       yield n.Value
       n <- n.Previous
```

```
        end while
    end ReverseTraversal
```

2.3 Summary

Linked lists are good to use when you have an unknown number of items to store. Using a data structure like an array would require you to specify the size up front; exceeding that size involves invoking a resizing algorithm which has a linear run time. You should also use linked lists when you will only remove nodes at either the head or tail of the list to maintain a constant run time. This requires maintaining pointers to the nodes at the head and tail of the list but the memory overhead will pay for itself if this is an operation you will be performing many times.

What linked lists are not very good for is random insertion, accessing nodes by index, and searching. At the expense of a little memory (in most cases 4 bytes would su±ce), and a few more read/writes you could maintain a *count* variable that tracks how many items are contained in the list so that accessing such a primitive property is a constant operation you just need to update *count* during the insertion and deletion algorithms.

Singly linked lists should be used when you are only performing basic insertions. In general doubly linked lists are more accommodating for non-trivial operations on a linked list.

We recommend the use of a doubly linked list when you require forwards and backwards traversal. For the most cases this requirement is present. For example, consider a token stream that you want to parse in a recursive descent fashion. Sometimes you will have to backtrack in order to create the correct parse tree.

Chapter 3

Binary Search Tree

Binary search trees (BSTs) are very simple to understand. We start with a root node with value *x*, where the left subtree of *x* contains nodes with values $< x$ and the right subtree contains nodes whose values are ¸ *x*. Each node follows the same rules with respect to nodes in their left and right subtrees.

BSTs are of interest because they have operations which are favourably fast: insertion, look up, and deletion can all be done in $O(log\ n)$ time. It is important to note that the $O(log\ n)$ times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self balancing properties see AVL tree defined in *x*7).

In the following examples you can assume, unless used as a parameter alias that *root* is a reference to the root node of the tree.

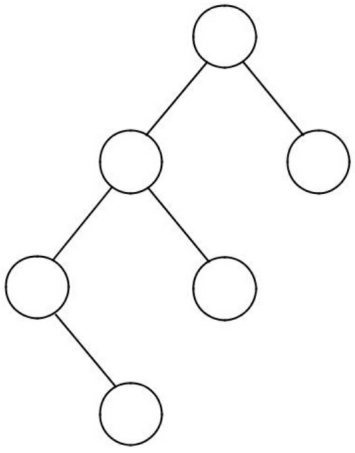


Figure 3.1: Simple unbalanced binary search tree

3.1 Insertion

As mentioned previously insertion is an *O*(*log n*) operation provided that the tree is moderately balanced.

```
algorithm Insert(value)
    Pre: value has passed custom type checks for type T
    Post: value has been placed in the correct location in the tree
    if root = null
     root <- node(value)
    else
     InsertNode(root, value)
    end if
end Insert
```

```
algorithm InsertNode(current, value)
Pre: current is the node to start from
Post: value has been placed in the correct location in the tree
    if value < current.Value
     if current.Left = null
      current.Left <- node(value)
     else
      InsertNode(current.Left, value)
     end if
    else
     if current.Right = null
      current.Right <- node(value)
     else
      nsertNode(current.Right, value)
     end if
    end if
 end InsertNode
```

The insertion algorithm is split for a good reason. The first algorithm (non-

recursive) checks a very core base case - whether or not the tree is empty. If the tree is empty then we simply create our root node and finish. In all other cases we invoke the recursive *InsertN ode* algorithm which simply guides us to the first appropriate place in the tree to put *value*. Note that at each stage we perform a binary chop: we either choose to recurse into the left subtree or the right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees.

3.2 Searching

Searching a BST is even simpler than insertion. The pseudocode is self-explanatory but we will look brie°y at the premise of the algorithm nonetheless.

We have talked previously about insertion, we go either left or right with the right subtree containing values that are ¸ *x* where *x* is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

1. the *root* = *;* in which case *value* is not in the BST; or

2. *root.*Value = *value* in which case *value* is in the BST; or

3. *value* < *root.*Value, we must inspect the left subtree of *root* for *value*; or

4. *value* > *root.*Value, we must inspect the right subtree of *root* for *value*.

algorithm Contains(*root*, *value*)

```
    Pre: root is the root node of the tree, value is what we would like
      to locate

    Post: value is either located or not

    if root = null

     return false

    end if

    if root.Value = value

     return true

    else if value < root.Value

     return Contains(root.Left, value)

    else

     return Contains(root.Right, value)

    end if

end Contains
```

3.3 Deletion

Removing a node from a BST is fairly straightforward, with four cases to con-sider:

1. the value to remove is a leaf node; or

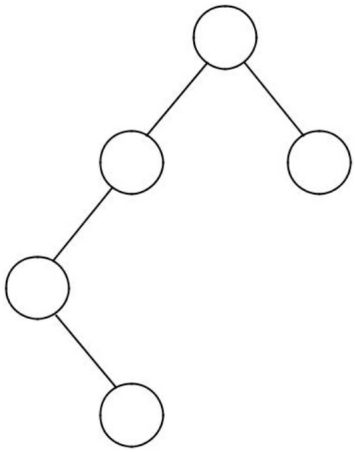
2. the value to remove has a right subtree, but no left subtree; or

3. the value to remove has a left subtree, but no right subtree; or

4. the value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.

There is also an implicit fifth case whereby the node to be removed is the only node in the tree. This case is already covered by the first, but should be noted as a possibility nonetheless.

Of course in a BST a value may occur more than once. In such a case the first occurrence of that value in the BST will be removed.



		23	#4: Right subtree
			and left subtree
	#3: Left subtree	14	31
	no right subtree		
#2: Right subtree	7		

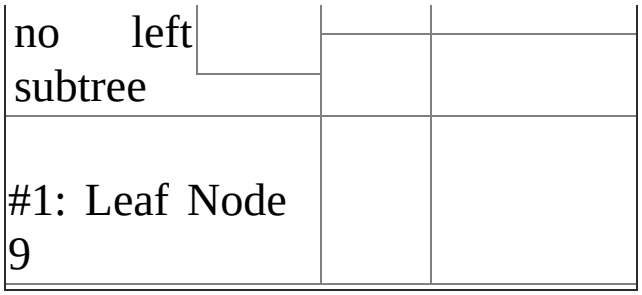


Figure 3.2: binary search tree deletion cases

The *Remove* algorithm given below relies on two further helper algorithms named *FindParent,* and *FindNode* which are described in *x*3.4 and *x*3.5 respectively.

```
algorithm Remove(value)
    Pre: value is the value of the node to remove, root is the root node of
the BST
    Count is the number of items in the BST
    Post: node with value is removed if found in which case yields true,
otherwise false
    nodeToRemove <- FindNode(value)
    if nodeToRemove = null
    return false // value not in BST
    end if
    parent <- FindParent(value)
    if Count = 1
    root <- null; // we are removing the only node in the BST
    else if nodeToRemove.Left = null and nodeToRemove.Right = null
    // case #1
    if nodeToRemove.Value < parent.Value
    parent.Left <- null;
    else
    parent.Right <- null;

    end if
    else if nodeToRemove.Left = null and nodeToRemove.Right !=null
    // case # 2
    if nodeToRemove.Value < parent.Value
```

```
        parent.Left <- nodeToRemove.Right
        else
        parent.Right <- nodeToRemove.Right
        end if
        else if nodeToRemove.Left !=null and nodeToRemove.Right =null
        // case #3
        if nodeToRemove.Value < parent.Value
        parent.Left <- nodeToRemove.Left
        else
        parent.Right <- nodeToRemove.Left
        end if
        else
        // case #4
        largestValue <- nodeToRemove.Left
        while largestValue.Right != null
        // find the largest value in the left subtree of nodeToRemove
        largestV alue <- largestV alue.Right
        end while
        // set the parents' Right pointer of largestValue to null
        FindParent(largestV alue.Value).Right <- null;
        nodeToRemove.Value <- largestV alue.Value
        end if
        Count <- Count ¡1
        return true
end Remove
```

3.4 Finding the parent of a given node

The purpose of this algorithm is simple - to return a reference (or pointer) to the parent node of the one with the given value. We have found that such an algorithm is very useful, especially when performing extensive tree transformations.

```
algorithm FindParent(value, root)
        Pre: value is the value of the node we want to find the parent of
        root is the root node of the BST and is ! = null
```

```
        Post: a reference to the parent node of value if found; otherwise null
        if value = root.Value
        return ;
        end if
        if value < root.Value
        if root.Left = null
        return ;
        else if root.Left.Value = value
        return root
        else
        return FindParent(value, root.Left)
        end if
        else
        if root.Right = null
        return ;
        else if root.Right.Value = value

        return root
        else
        return FindParent(value, root.Right)
        end if
        end if
end FindParent
```

A special case in the above algorithm is when the specified value does not exist in the BST, in which case we return *;*. Callers to this algorithm must take account of this possibility unless they are already certain that a node with the specified value exists.

3.5 Attaining a reference to a node

This algorithm is very similar to *x*3.4, but instead of returning a reference to the parent of the node with the specified value, it returns a reference to the node itself. Again null is returned if the value isn't found.

```
algorithm FindNode(root, value)
        Pre: value is the value of the node we want to find the parent of
```

```
        root is the root node of the BST
        Post: a reference to the node of value if found; otherwise null;
        if root = null
        return null
        end if
        if root.Value = value
        return root
        else if value < root.Value
        return FindNode(root.Left, value)
        else
        return FindNode(root.Right, value)
        end if
end FindNode
```

You will have noticed that the *FindNode* algorithm is exactly the same as the *Contains* algorithm (defined in *x*3.2) with the modification that we are returning a reference to a node not *true* or *f alse*. Given *FindNode*, the easiest way of implementing *Contains* is to call *FindNode* and compare the return value with *;*.

3.6 Finding the smallest and largest values in the binary search tree

To find the smallest value in a BST you simply traverse the nodes in the left subtree of the BST always going left upon each encounter with a node, terminating when you find a node with no left subtree. The opposite is the case when finding the largest value in the BST. Both algorithms are incredibly simple, and are listed simply for completeness.

The base case in both *FindMin*, and *FindMax* algorithms is when the Left (*FindMin*), or Right (*FindMax*) node references are null in which case we have reached the last node.

```
        algorithm FindMin(root)

            Pre: root is the root node of the BST
```

root != *null*

Post: the smallest value in the BST is located
if *root*.Left = null

return *root*.Value

end if

FindMin(*root*.Left)

end FindMin

algorithm FindMax(*root*)

Pre: *root* is the root node of the BST

root != *null*

Post: the largest value in the BST is located
if *root*.Right = null

return *root*.Value

end if

FindMax(*root*.Right)

end FindMax

3.7 Tree Traversals

There are various strategies which can be employed to traverse the items in a tree; the choice of strategy depends on which node visitation order you

require. In this section we will touch on the traversals that DSA provides on all data structures that derive from *BinarySearchT ree*.

3.7.1 Preorder

When using the preorder algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree. An example of preorder traversal is shown in Figure 3.3.

```
algorithm Preorder(root)

    Pre: root is the root node of the BST

    Post: the nodes in the BST have been visited in preorder

    if root != null

     yield root.Value
     Preorder(root.Left)

     Preorder(root.Right)

    end if

end Preorder
```

3.7.2 Postorder

This algorithm is very similar to that described in *x*3.7.1, however the value of the node is yielded after traversing both subtrees. An example of postorder traversal is shown in Figure 3.4.

```
algorithm Postorder(root)

    Pre: root is the root node of the BST
```

Post: the nodes in the BST have been visited in postorder

if *root* != *null*

Postorder(*root*.Left)
Postorder(*root*.Right)

yield *root*.Value

end if

end Postorder

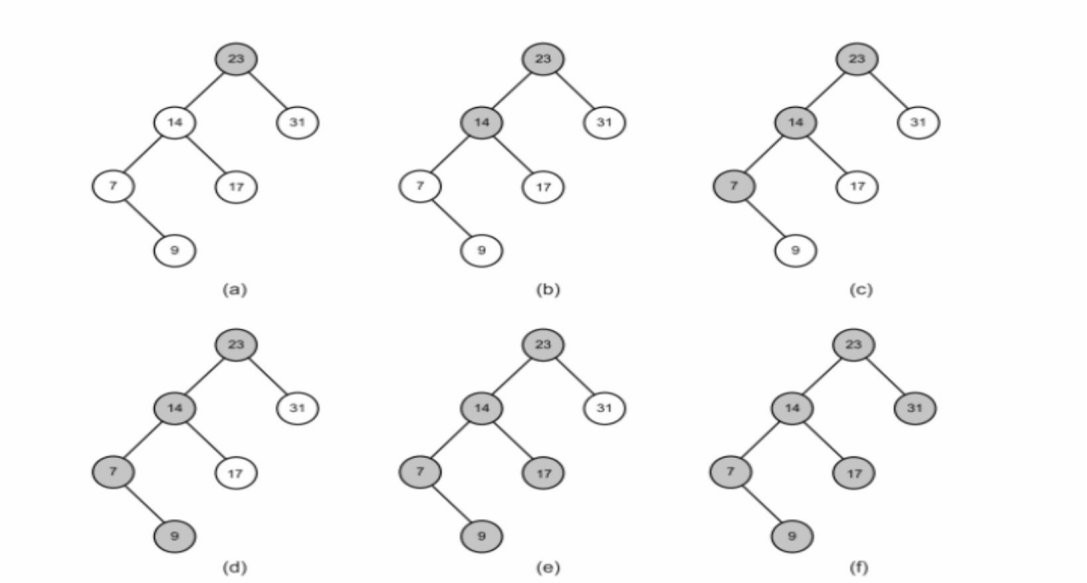


Figure 3.3: Preorder visit binary search tree example

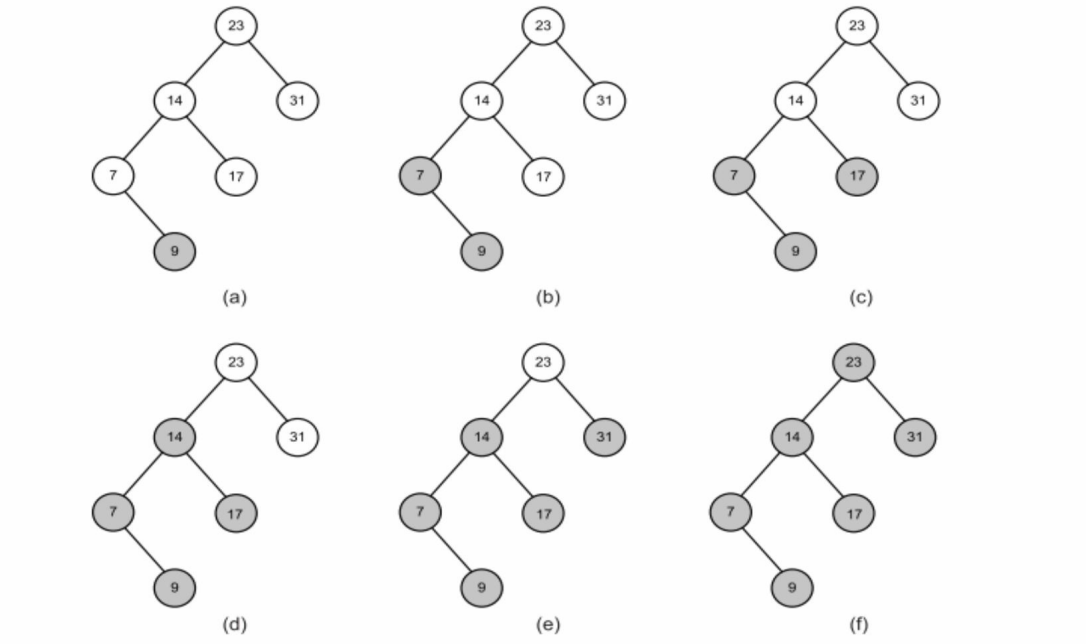


Figure 3.4: Postorder visit binary search tree example

3.7.3 Inorder

Another variation of the algorithms defined in *x*3.7.1 and *x*3.7.2 is that of inorder traversal where the value of the current node is yielded in between traversing the left subtree and the right subtree. An example of inorder traversal is shown in Figure 3.5.

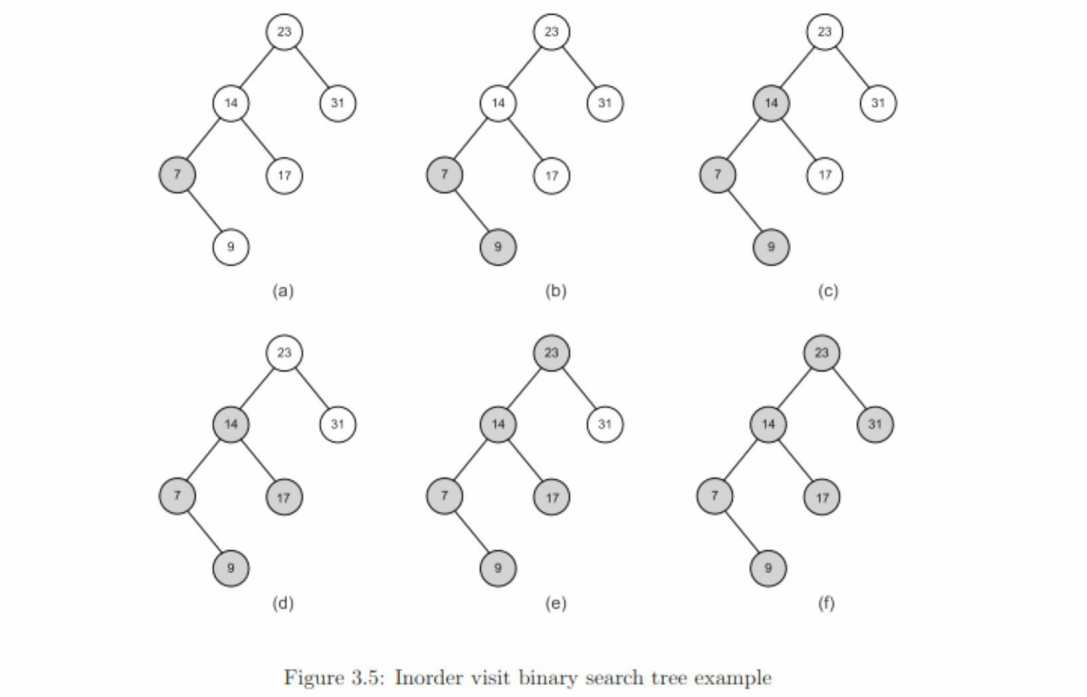


Figure 3.5: Inorder visit binary search tree example

algorithm Inorder(*root*)

Pre: *root* is the root node of the BST

Post: the nodes in the BST have been visited in inorder

if *root* != null

Inorder(*root*.Left)
yield *root*.Value

```
            Inorder(root.Right)
        end if
    end Inorder
```

3.7.4 Breadth First

Traversing a tree in breadth first order yields the values of all nodes of a particular depth in the tree before any deeper ones. In other words, given a depth *d* we would visit the values of all nodes at *d* in a left to right fashion, then we would proceed to *d* + 1 and so on until we hade no more nodes to visit. An example of breadth first traversal is shown in Figure 3.6.

Traditionally breadth first traversal is implemented using a list (vector, resizeable array, etc) to store the values of the nodes visited in breadth first order and then a queue to store those nodes that have yet to be visited.

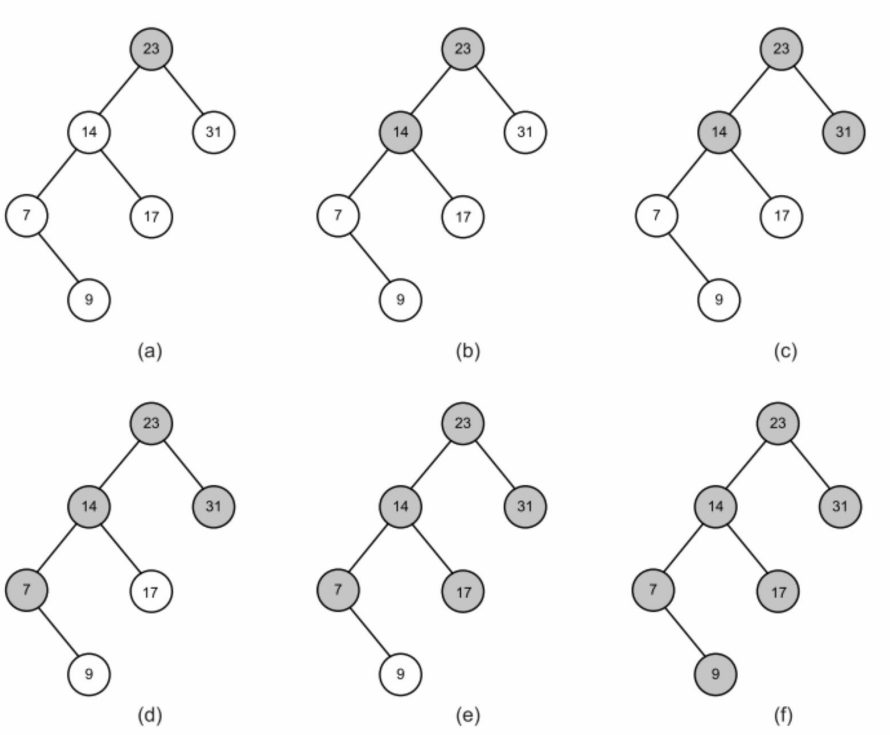


Figure 3.6: Breadth First visit binary search tree example

```
algorithm BreadthFirst(root)
        Pre: root is the root node of the BST
        Post: the nodes in the BST have been visited in breadth first order
        q <- queue
        while root != null
        yield root.Value
        if root.Left !=null
        q.Enqueue(root.Left)
        end if
        if root.Right != null
        q.Enqueue(root.Right)
        end if
        if !q.IsEmpty()
        root <- q.Dequeue()
        else
        root <- null;
        end if
        end while
```

```
end BreadthFirst
```

3.8 Summary

A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very efficient. Traversal remains linear, but there are many ways in which you can visit the nodes of a tree. Trees are recursive data structures, so typically you will find that many algorithms that operate on a tree are recursive.

The run times presented in this chapter are based on a pretty big assumption that the binary search tree's left and right subtrees are reasonably balanced. We can only attain logarithmic run times for the algorithms presented earlier when this is true. A binary search tree does not enforce such a property, and the run times for these operations on a pathologically unbalanced tree become linear: such a tree is effectively just a linked list. Later in *x7* we will examine an AVL tree that enforces self-balancing properties to help attain logarithmic run times.

Chapter 4
Heap

A heap can be thought of as a simple tree data structure, however a heap usually employs one of two strategies:

1. min heap; or

2. max heap

Each strategy determines the properties of the tree and its values. If you were to choose the min heap strategy then each parent node would have a value that is · than its children. For example, the node at the root of the tree will have the smallest value in the tree. The opposite is true for the max heap strategy. In this book you should assume that a heap employs the min heap strategy unless otherwise stated.

Unlike other tree data structures like the one defined in *x*3 a heap is generally implemented as an array rather than a series of nodes which each have references to other nodes. The nodes are conceptually the same, however, having at most two children. Figure 4.1 shows how the tree (not a heap data structure) (12 7(3 2) 6(9)) would be represented as an array. The array in Figure 4.1 is a result of simply adding values in a top-to-bottom, left-to-right fashion. Figure 4.2 shows arrows to the direct left and right child of each value in the array.

This chapter is very much centred around the notion of representing a tree as an array and because this property is key to understanding this chapter Figure 4.3 shows a step by step process to represent a tree data structure as an array. In Figure 4.3 you can assume that the default capacity of our array is eight.

Using just an array is often not su±cient as we have to be up front about the size of the array to use for the heap. Often the run time behaviour of a program can be unpredictable when it comes to the size of its internal data structures, so we need to choose a more dynamic data structure that contains the following properties:

1. We can specify an initial size of the array for scenarios where

we know the upper storage limit required; and

2. the data structure encapsulates resizing algorithms to grow the array as required

at run time.

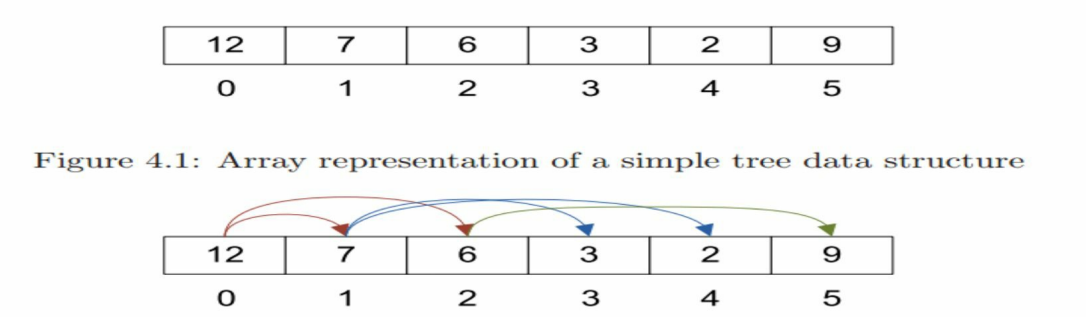


Figure 4.1: Array representation of a simple tree data structure

Figure 4.2: Direct children of the nodes in an array representation of a tree data structure

1. Vector

2. ArrayList

3. List

Figure 4.1 does not specify how we would handle adding null references to the heap. This varies from case to case; sometimes null values are prohibited entirely; in other cases we may treat them as being smaller than any non-null value, or indeed greater than any non-null value. You will have to resolve this ambiguity yourself having studied your requirements. For the sake of clarity we will avoid the issue by prohibiting null values.

Because we are using an array we need some way to calculate the index of a parent node, and the children of a node. The required expressions for this are defined as follows for a node at *index*:

1. (*index*- 1)/2 (parent index)
2. 2 * *index* + 1 (left child)
3. 2 * *index* + 2 (right child)

In Figure 4.4 a) represents the calculation of the right child of 12 (2 * 0 + 2); and b) calculates the index of the parent of 3 ((3-1)/2).

4.1 Insertion

Designing an algorithm for heap insertion is simple, but we must ensure that heap order is preserved after each insertion. Generally this is a post-insertion operation. Inserting a value into the next free slot in an array is simple: we just need to keep track of the next free index in the array as a counter, and increment it after each insertion. Inserting our value into the heap is the first part of the algorithm; the second is validating heap order. In the case of min-heap ordering this requires us to swap the values of a parent and its child if

the value of the child is < the value of its parent. We must do this for each subtree containing the value we just inserted.

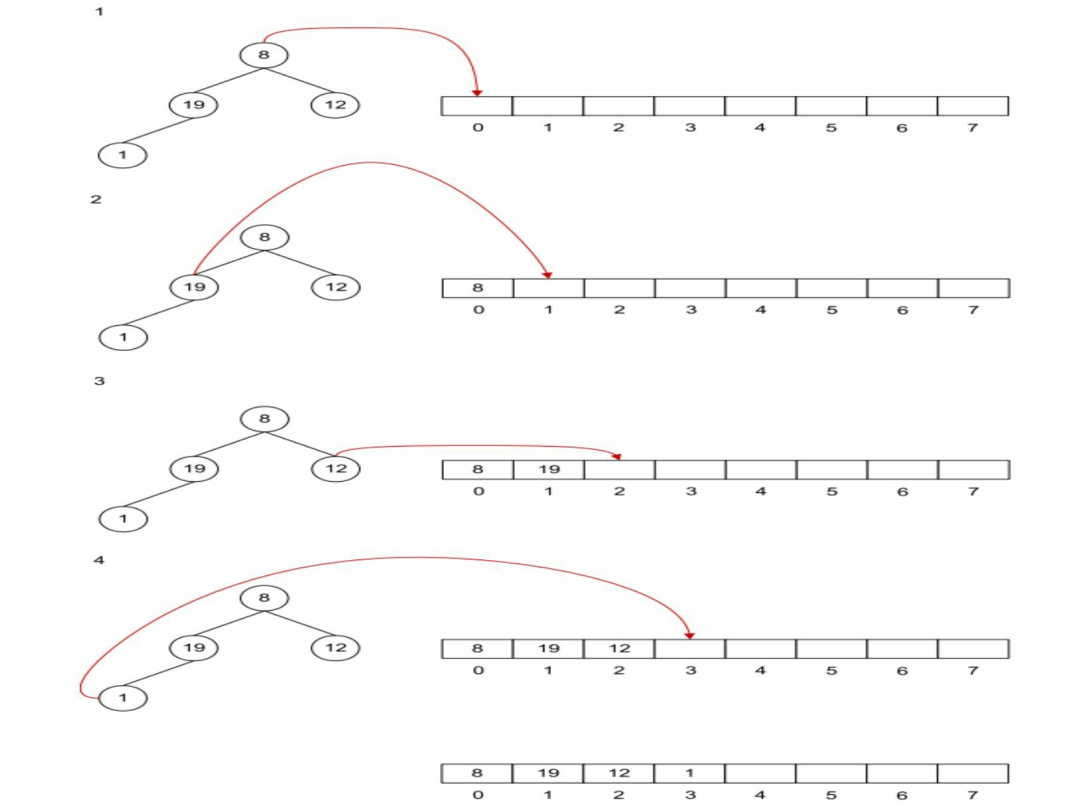


Figure 4.3: Converting a tree data structure to its array counterpart

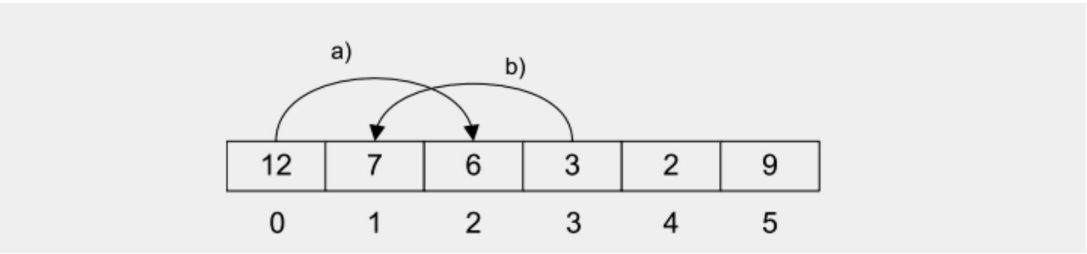
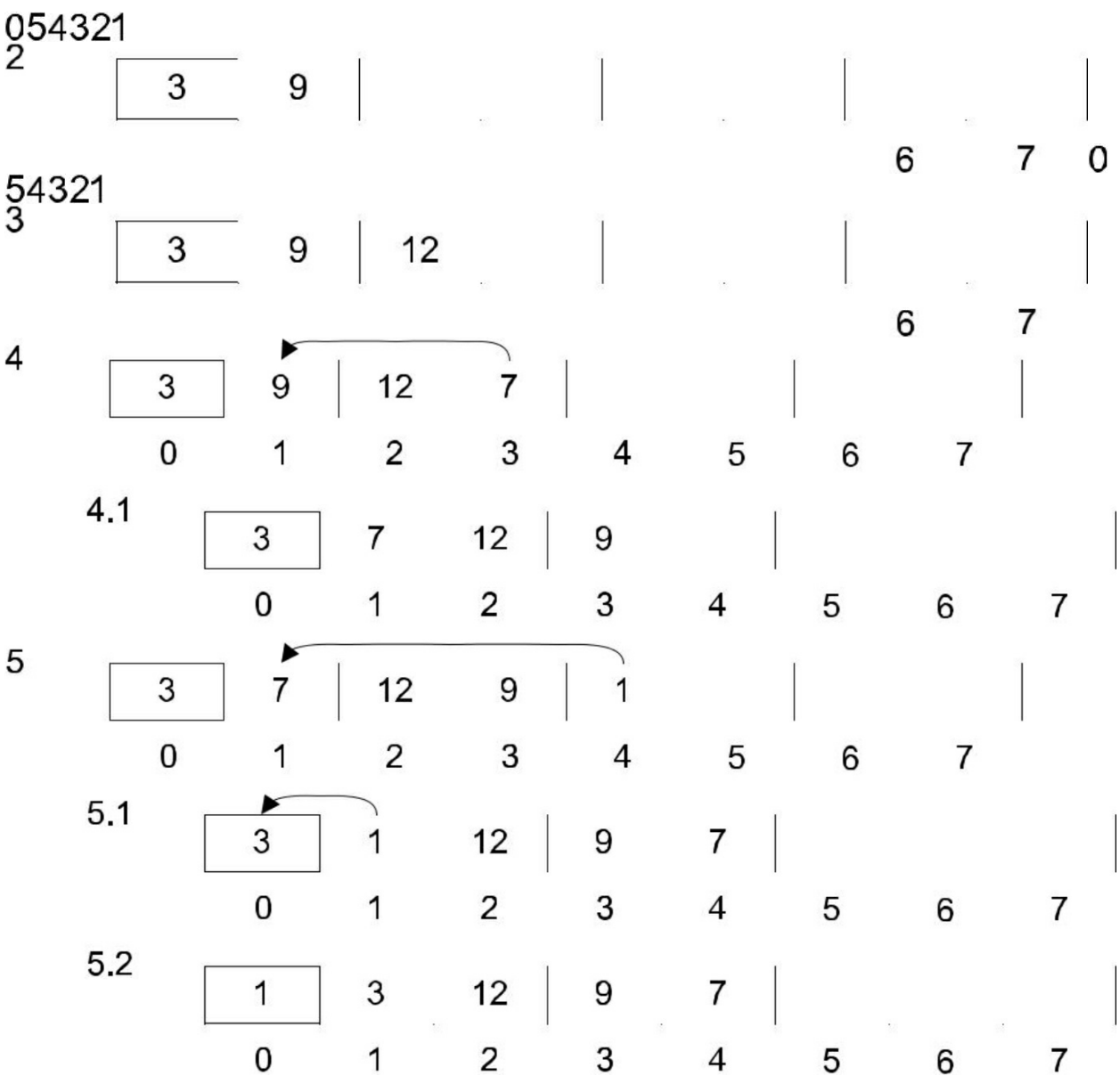


Figure 4.4: Calculating node properties

The run time e±ciency for heap insertion is $O(log\ n)$. The run time is a by product of verifying heap order as the first part of the algorithm (the actual

insertion into the array) is $O(1)$.

Figure 4.5 shows the steps of inserting the values 3, 9, 12, 7, and 1 into a min-heap.


054321
2
3
9
6
7
0
54321
3
3
9
12
6
7
4
3
9
12
7
0
1
2
3
4
5
6
7
4.1
3
7
12
9
0
1
2
3
4
5
6
7
5
3
7
12
9
1
0
1
2
3
4
5
6
7
5.1
3
1
12
9
7
0
1
2
3
4
5
6
7
5.2
1
3
12
9
7
0
1
2
3
4
5
6
7

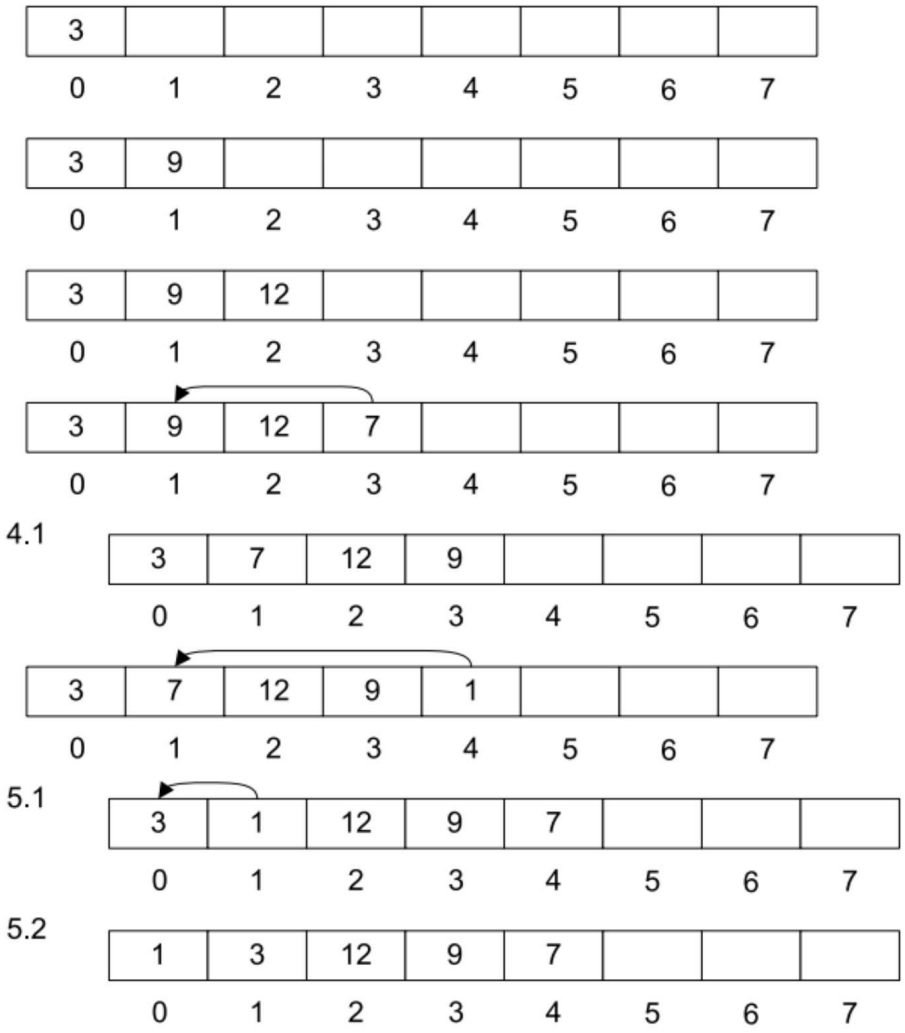


Figure 4.5: Inserting values into a min-heap

algorithm Add(*value*)

Pre: *value* is the value to add to the heap

```
        Count is the number of items in the heap
    Post: the value has been added to the heap
    heap[Count] <- value
    Count <- Count +1
    MinHeapify()
end Add


algorithm MinHeapify()
    Pre: Count is the number of items in the heap
       heap is the array used to store the heap items
    Post: the heap has preserved min heap ordering
    i <- Count - 1
    while i > 0 and heap[i] < heap[(i - 1)/2]
      Swap(heap[i], heap[(i - 1)/2]
      i <- (i - 1)/2
    end while
end MinHeapify
```

The design of the *MaxHeapify* algorithm is very similar to that of the *Min-*

Heapify algorithm, the only difference is that the < operator in the second condition of entering the while loop is changed to >.

4.2 Deletion

Just as for insertion, deleting an item involves ensuring that heap ordering is preserved. The algorithm for deletion has three steps:

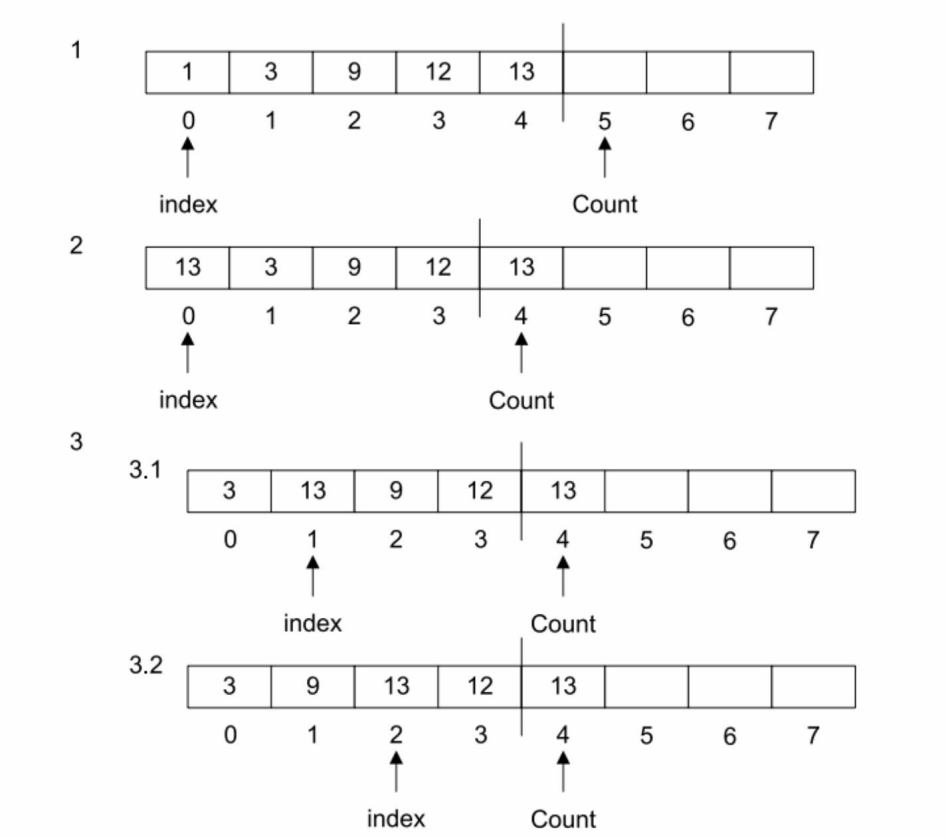
1. find the index of the value to delete
2. put the last value in the heap at the index location of the item to delete
3. verify heap ordering for each subtree which used to include the value

```
algorithm Remove(value)
    Pre: value is the value to remove from the heap
    left, and right are updated alias' for 2 * index + 1, and 2 * index + 2 respectively
    Count is the number of items in the heap
    heap is the array used to store the heap items
    Post: value is located in the heap and removed, true; otherwise false
    // step 1
    index <- FindIndex(heap, value)
    if index < 0
    return false
    end if
    Count <- Count - 1
    // step 2
    heap[index] <- heap[Count]
    // step 3
    while left < Count and heap[index] > heap[left] or heap[index] > heap[right]
    // promote smallest key from subtree
    if heap[leftt] < heap[right]
```

```
        Swap(heap, left, index)
        index <- le t
        else
        Swap(heap, right, index)
        index <- right
        end if
        end while
        return true
end Remove
```

Figure 4.6 shows the *Remove* algorithm visually, removing 1 from a heap containing the values 1, 3, 9, 12, and 13. In Figure 4.6 you can assume that we have specified that the backing array of the heap should have an initial capacity of eight.

Please note that in our deletion algorithm that we don't default the removed value in the *heap* array. If you are using a heap for reference types, i.e. objects that are allocated on a heap you will want to free that memory. This is important in both unmanaged, and managed languages. In the latter we will want to null that empty hole so that the garbage collector can reclaim that memory. If we were to not null that hole then the object could still be reached and thus won't be garbage collected.



4.3 Searching

Searching a heap is merely a matter of traversing the items in the heap array sequentially, so this operation has a run time complexity of $O(n)$. The search can be thought of as one that uses a breadth first traversal as defined in *x*3.7.4 to visit the nodes within the heap to check for the presence of a specified item.

algorithm Contains(*value*)

Pre: *value* is the value to search the heap for

Count is the number of items in the heap

heap is the array used to store the heap items

```
        Post: value is located in the heap, in which case true; otherwise
          false

        i <- 0

        while i < Count and heap[i] != value

          i <- i + 1

        end while
        if i < Count

          return true

        else

          return false

        end if

    end Contains
```

The problem with the previous algorithm is that we don't take advantage of the properties in which all values of a heap hold, that is the property of the heap strategy being used. For instance if we had a heap that didn't contain the value 4 we would have to exhaust the whole backing heap array before we could determine that it wasn't present in the heap. Factoring in what we know about the heap we can optimise the search algorithm by including logic which makes use of the properties presented by a certain heap strategy.

Optimising to deterministically state that a value is in the heap is not that straightforward, however the problem is a very interesting one. As an example consider a min-heap that doesn't contain the value 5. We can only

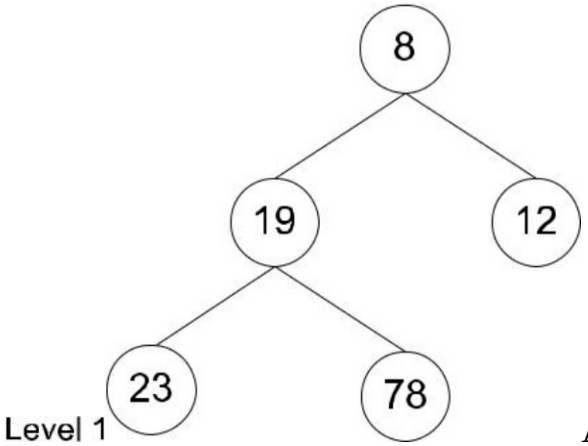
rule that the value is not in the heap if 5 > the parent of the current node being inspected and < the current node being inspected *8* nodes at the current level we are traversing. If this is the case then 5 cannot be in the heap and so we can provide an answer without traversing the rest of the heap. If this property is not satisfied for any level of nodes that we are inspecting then the algorithm will indeed fall back to inspecting all the nodes in the heap. The optimisation that we present can be very common and so we feel that the extra logic within the loop is justified to prevent the expensive worse case run time.

```
algorithm Contains(value)

    Pre: value is the value to search the heap for

      Count is the number of items in the heap

      heap is the array used to store the heap items

    Post: value is located in the heap, in which case true; otherwise
      false
    start <- 0

    nodes <- 1

    while start < Count
      start <- nodes - 1

      end <- nodes + start

      count <- 0

      while start < Count and start < end
        if value = heap[start]
```

```
            return true

          else if value > Parent(heap[start]) and value < heap[start]

           count <- count + 1

          end if
          start <- start + 1

        end while
        if count = nodes

          return false

        end if

        nodes <- nodes ¤ 2

      end while

      return false

  end Contains
```

The new *Contains* algorithm determines if the *value* is not in the heap by checking whether *count* = *nodes*. In such an event where this is true then we can confirm that *8* nodes *n* at level *i* : *value* > Parent(*n*), *value* < *n* thus there is no possible way that *value* is in the heap. As an example consider Figure 4.7. If we are searching for the value 10 within the min-heap displayed it is obvious that we don't need to search the whole heap to determine 9 is not present. We can verify this after traversing the nodes in the second level of the heap as the previous expression defined holds true.

4.4 Traversal



As mentioned in *x*4.3 traversal of a heap is usually done like that of any other array data structure which our heap implementation is based upon. As a result you traverse the array starting at the initial array index (0 in most languages) and then visit each value within the array until you have reached the upper bound of the heap. You will note that in the search algorithm that we use *Count* as this upper bound rather than the actual physical bound of the allocated array. *Count* is used to partition the conceptual heap from the actual array implementation of the heap: we only care about the items in the heap, not the whole array|the latter may contain various other bits of data as a result of heap mutation.

Figure 4.7: Determining 10 is not in the heap after inspecting the nodes of Level 2

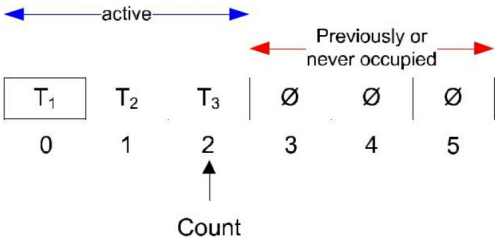


Figure 4.8: Living and dead space in the heap backing array

If you have followed the advice we gave in the deletion algorithm then a heap that has been mutated several times will contain some form of default value for items no longer in the heap. Potentially you will have at most *LengthOf*(*heapArray*) ¡ *Count* garbage values in the backing heap array data structure. The garbage values of course vary from platform to platform. To make things simple the garbage value of a reference type will be simple *;* and 0 for a value type.

Figure 4.8 shows a heap that you can assume has been mutated many times. For this example we can further assume that at some point the items in indexes 3 ¡ 5 actually contained references to live objects of type *T* . In Figure 4.8 subscript is used to disambiguate separate objects of *T* .

From what you have read thus far you will most likely have picked up that traversing the heap in any other order would be of little benefit. The heap

property only holds for the subtree of each node and so traversing a heap in any other fashion requires some creative intervention. Heaps are not usually traversed in any other way than the one prescribed previously.

4.5 Huffman Algorithm

Huffman algorithm or Huffman coding is an entropy encoding algorithm. It is used widely for data compression (like Winzip Compression-Winzip doesnt use it but!). Huffman coding is used in JPEG compression.

The key idea behind Huffman coding is to encode the most common characters using shorter strings of bits than those used for less common source characters. It works by creating a binary tree stored in an array. We also need to know the external path length (sum of all paths from root to external node) and internal path length (sum of all paths from root to internal node).

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

The Huffman Algorithm

> Step 1: Create a leaf node for each character. Add the character and its weight or frequency of occurrence to the

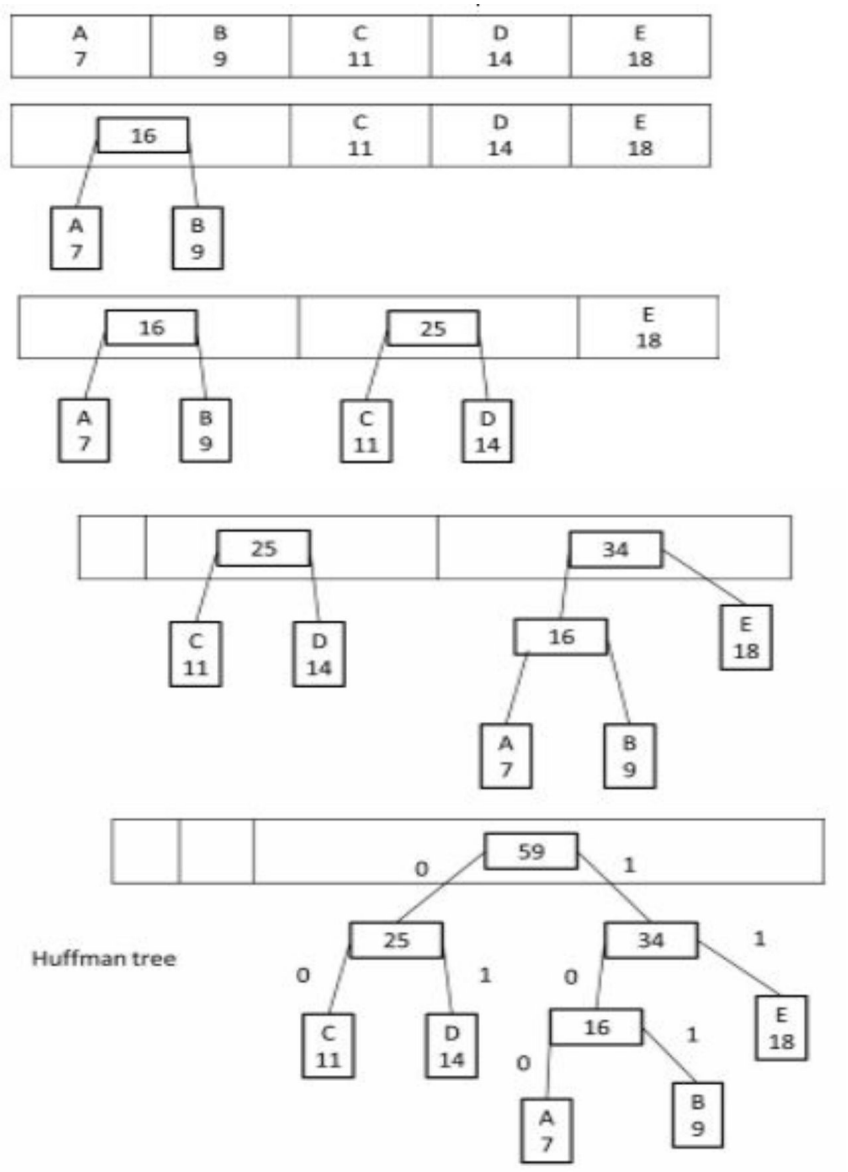
priority queue.

Step 2: Repeat Steps 3 to 5 while the total number of nodes in the queue is greater than 1

Step 3: Remove two nodes that have the lowest weight (or highest priority)

Step 4: Create a new internal node by merging these two nodes as children and with weight equal to the sum of the two nodes' weights.

Step 5: Add the newly created node to the queue.



So coding for C=00, D=01, A=100, B=101, C=11

4.6 Summary

Heaps are most commonly used to implement priority queues (see *x*6.2 for a sample implementation) and to facilitate heap sort. As discussed in both the insertion *x*4.1 and deletion *x*4.2 sections a heap maintains heap order according to the selected ordering strategy. These strategies are referred to as min-heap and max heap. The former strategy enforces that the value of a parent node is less than that of each of its children, the latter enforces that the value of the parent is greater than that of each of its children.

When you come across a heap and you are not told what strategy it enforces you should assume that it uses the min-heap strategy. If the heap can be configured otherwise, e.g. to use max-heap then this will often require you to state this explicitly. The heap abides progressively to a strategy during the invocation of the insertion, and deletion algorithms. The cost of such a policy is that upon each insertion and deletion we invoke algorithms that have logarithmic run time complexities. While the cost of maintaining the strategy might not seem overly expensive it does still come at a price. We will also have to factor in the cost of dynamic array expansion at some stage. This will occur if the number of items within the heap outgrows the space allocated in the heap's backing array. It may be in your best interest to research a good initial starting size for your heap array. This will assist in minimising the impact of dynamic array resizing.

Chapter 5

Sets

A set contains a number of values, in no particular order. The values within the set are distinct from one another.

Generally set implementations tend to check that a value is not in the set before adding it, avoiding the issue of repeated values from ever occurring.

This section does not cover set theory in depth; rather it demonstrates briefy the ways in which the values of sets can be defined, and common operations that may be performed upon them.

The notation $A = \{4; 7; 9; 12; 0\}$ defines a set A whose values are listed within the curly braces.

Given the set A defined previously we can say that 4 is a member of A denoted by $4 \in A$, and that 99 is not a member of A denoted by $99 \notin A$.

Often defining a set by manually stating its members is tiresome, and more importantly the set may contain a large number of values. A more concise way of defining a set and its members is by providing a series of properties that the values of the set must satisfy. For example, from the definition $A = \{x|x>0, x\%2=0\}$ the set A contains only positive integers that are even. x is an alias to the current value we are inspecting and to the right hand side of | are the properties that x must satisfy to be in the set A. In this example, x must be > 0, and the remainder of the arithmetic expression $x/2$ must be 0. You will be able to note from the previous definition of the set A that the set can

contain an infinite number of values, and that the values of the set *A* will be all even integers that are a member of the natural numbers set N, where N = *{1; 2; 3;.....}*.

Finally in this brief introduction to sets we will cover set intersection and union, both of which are very common operations (amongst many others) per-formed on sets. The union set can be defined as follows *A* U *B* = {*x* | x ϵ *A or x ϵ B*, and intersection *A Π B = {x| xϵ A and xϵ B*. Figure 5.1 demonstrates set intersection and union graphically.

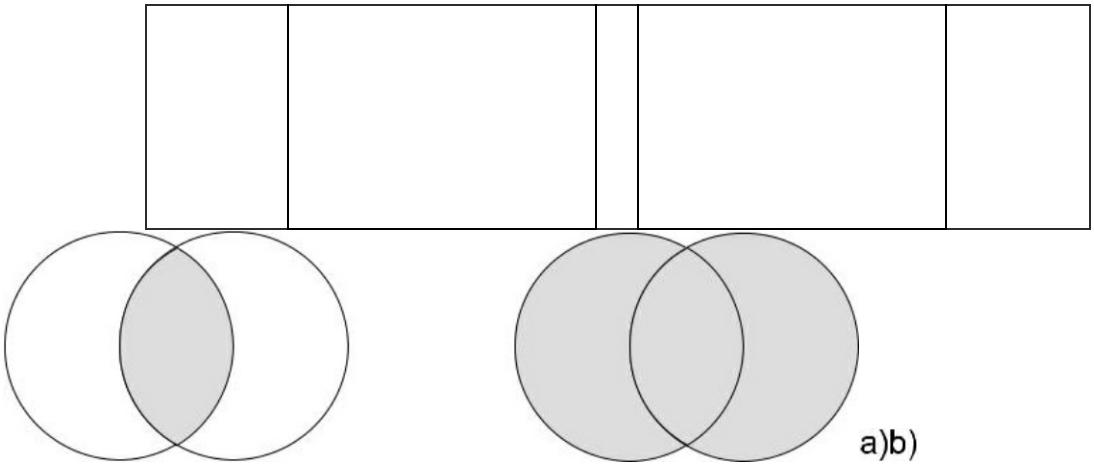


Figure 5.1: a) *A* \ *B*; b) *A [B*

Most of the algorithms defined in *System.Linq.Enumerable* deal mainly with sequences rather than sets exclusively.Set union can be implemented as a simple traversal of both sets adding each item of the two sets to a new union set.

algorithm Union(*set*1, *set*2)

Pre: *set*1, and *set*2 != *null*

```
        union is a set
    Post: A union of set1, and set2 has been created
    foreach item in set1
      union.Add(item)
    end foreach
    foreach item in set2
      union.Add(item)
    end foreach
    return union
end Union
```

The run time of our *Union* algorithm is $O(m + n)$ where m is the number of items in the first set and n is the number of items in the second set. This runtime applies only to sets that exhibit $O(1)$ insertions.

Set intersection is also trivial to implement. The only major thing worth pointing out about our algorithm is that we traverse the set containing the fewest items. We can do this because if we have exhausted all the items in the smaller of the two sets then there are no more items that are members of both sets, thus we have no more items to add to the intersection set.

```
algorithm Intersection(set1, set2)
    Pre: set1, and set2 != null
        intersection, and smallerSet are sets
```

```
        Post: An intersection of set1, and set2 has been created

        if set1.Count < set2.Count

         smallerSet <- set1

        else
         smallerSet <- set2

        end if

        foreach item in smallerSet

         if set1.Contains(item) and set2.Contains(item)

          intersection.Add(item)

         end if

        end foreach

        return intersection

    end Intersection
```

The run time of our *Intersection* algorithm is $O(n)$ where n is the number of items in the smaller of the two sets. Just like our *Union* algorithm a linear runtime can only be attained when operating on a set with $O(1)$ insertion.

5.1 Unordered

Sets in the general sense do not enforce the explicit ordering of their members. For example the members of $B = f6; 2; 9g$ conform to no ordering scheme because it is not required. Most libraries provide implementations of

unordered sets and so DSA does not; we simply mention it here to disambiguate between an unordered set and ordered set.

We will only look at insertion for an unordered set and cover brie°y why a hash table is an e±cient data structure to use for its implementation.

5.1.1 Insertion

An unordered set can be e±ciently implemented using a hash table as its backing data structure. As mentioned previously we only add an item to a set if that item is not already in the set, so the backing data structure we use must have a quick look up and insertion run time complexity.

A hash map generally provides the following:

1. $O(1)$ for insertion

2. approaching $O(1)$ for look up

The above depends on how good the hashing algorithm of the hash table is, but most hash tables employ incredibly e±cient general purpose hashing algorithms and so the run time complexities for the hash table in your library of choice should be very similar in terms of e±ciency.

5.2 Ordered

An ordered set is similar to an unordered set in the sense that its members are distinct, but an ordered set enforces some predefined comparison on each of its members to produce a set whose members are ordered appropriately.

In DSA 0.5 and earlier we used a binary search tree (defined in *x*3) as the

internal backing data structure for our ordered set. From versions 0.6 onwards we replaced the binary search tree with an AVL tree primarily because AVL is balanced.

The ordered set has its order realised by performing an inorder traversal upon its backing tree data structure which yields the correct ordered sequence of set members.

Because an ordered set in DSA is simply a wrapper for an AVL tree that additionally ensures that the tree contains unique items you should read *x*7 to learn more about the run time complexities associated with its operations.

5.3 Summary

Sets provide a way of having a collection of unique objects, either ordered or unordered.

When implementing a set (either ordered or unordered) it is key to select the correct backing data structure. As we discussed in *x*5.1.1 because we check first if the item is already contained within the set before adding it we need this check to be as quick as possible. For unordered sets we can rely on the use of a hash table and use the key of an item to determine whether or not it is already contained within the set. Using a hash table this check results in a near constant run time complexity. Ordered sets cost a little more for this check, however the logarithmic growth that we incur by using a binary search tree as its backing data structure is acceptable.

Another key property of sets implemented using the approach we describe is that both have favourably fast look-up times. Just like the check before insertion, for a hash table this run time complexity should be near constant. Ordered sets as described in 3 perform a binary chop at each stage when searching for the existence of an item yielding a logarithmic run time.

We can use sets to facilitate many algorithms that would otherwise be a little less clear in their implementation. For example in *x*11.4 we use an unordered set to assist in the construction of an algorithm that determines the number of repeated words within a string.

Chapter 6

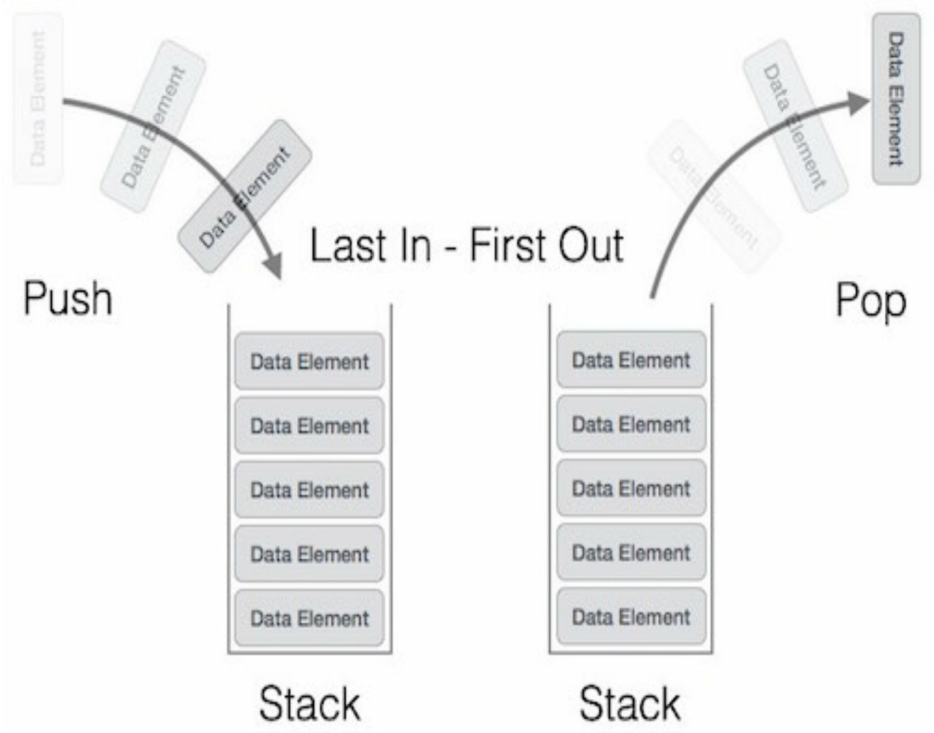
Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push ()** − Pushing (storing) an element on the stack.
- **Pop ()** − Removing (accessing) an element from the stack.

When data is pushed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **Peek ()** − get the top data element of the stack, without removing it.
- **Is Full()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last pushed data on the stack. As

this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
algorithm peek()
   return stack[top]
end peek
```

isfull()

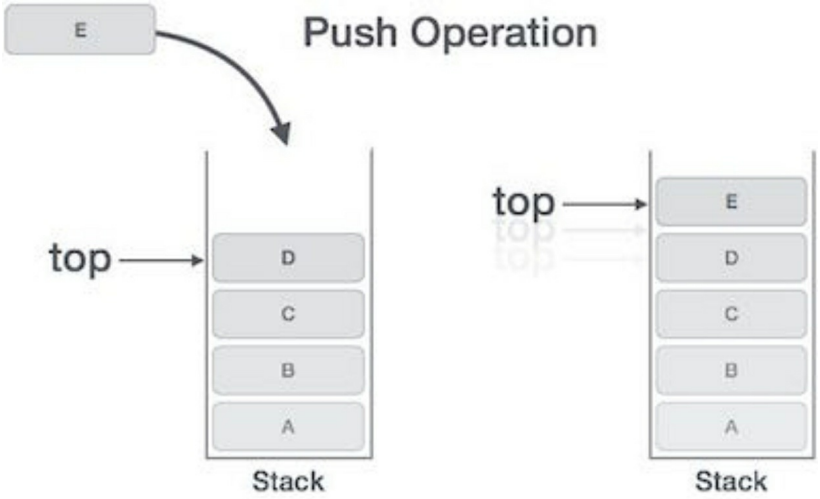
```
algorithm isfull()
   if top equals to MAXSIZE
      return true
   else
      return false
   endif
end isFull
```

```
algorithm isempty()
   if top less than 1
      return true
   else
      return false
   endif
end isEmpty
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

```
algorithm push( stack, data)
  if stack is full
    return null
  end if
  top ← top + 1
  stack[top] ← data
end push
```

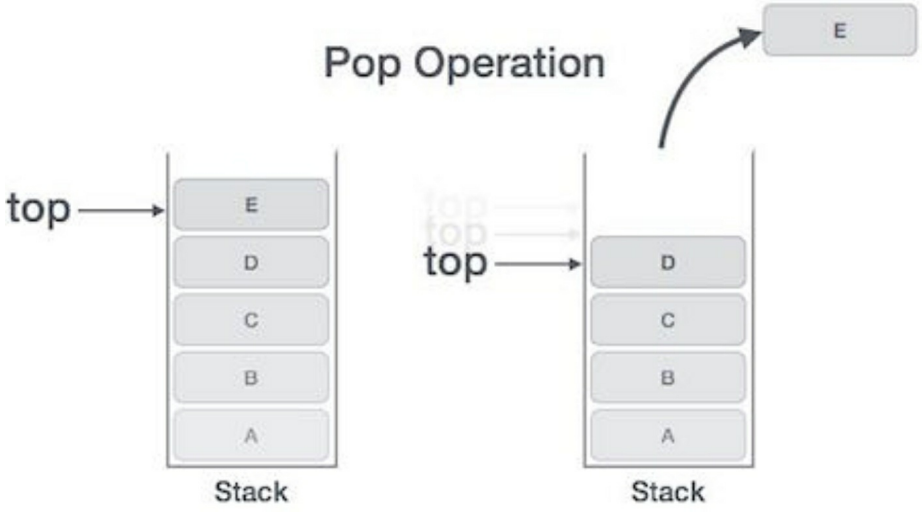
Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.

- **Step 5** − Returns success.




```
algorithm pop(stack)
   if stack is empty
      return null
   end if
   data ← stack[top]
   top ← top - 1
   return data
end pop
```

Expression Parsing using Stack

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used

in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗
6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a + b) − c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b*c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like **(a + b)*c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right

Step 2 − if it is an operand push it to stack

Step 3 − if it is an operator pull operand from stack and perform operation

Step 4 − store the output of step 3, back to stack

Step 5 − scan the expression until all operands are consumed

Step 6 − pop the stack and perform operation

Queues

Queues are an essential data structure that are found in vast amounts of software from user mode to kernel mode applications that are core to the system.

Fundamentally they honour a first in first out (FIFO) strategy, that is the item first put into the queue will be the first served, the second item added to the queue will be the second to be served and so on.

A traditional queue only allows you to access the item at the front of the queue; when you add an item to the queue that item is placed at the back of the queue.

Historically queues always have the following three core methods:

Enqueue: places an item at the back of the queue;

Dequeue: retrieves the item at the front of the queue, and removes it from the queue;

Peek: retrieves the item at the front of the queue without removing it from the queue

As an example to demonstrate the behaviour of a queue we will walk through a scenario whereby we invoke each of the previously mentioned methods observ-ing the mutations upon the queue data structure. The following list describes the operations performed upon the queue in Figure 6.1:

Enqueue(10)

Enqueue(12)

Enqueue(9)

Enqueue(8)

Enqueue(3)

Dequeue()

Peek()

Enqueue(33)

Peek()

Dequeue()

6.1 A standard queue

A queue is implicitly like that described prior to this section. In DSA we don't provide a standard queue because queues are so popular and such a core data structure that you will find pretty much every mainstream library provides a queue data structure that you can use with your language of choice. In this section we will discuss how you can, if required, implement an e±cient queue data structure.

The main property of a queue is that we have access to the item at the front of the queue. The queue data structure can be e±ciently implemented using a singly linked list (defined in *x*2.1). A singly linked list provides *O*(1) insertion and deletion run time complexities. The reason we have an *O*(1) run time complexity for deletion is because we only ever remove items from the front of queues (with the Dequeue operation). Since we always have a pointer to the item at the head of a singly linked list, removal is simply a case of returning the value of the old head node, and then modifying the head pointer to be the next node of the old head node. The run time complexity for searching a queue remains the same as that of a singly linked list: *O*(*n*).

6.2 Priority Queue

Unlike a standard queue where items are ordered in terms of who arrived first, a priority queue determines the order of its items by using a form of custom comparer to see which item has the highest priority. Other than the items in a priority queue being ordered by priority it remains the same as a normal queue: you can only access the item at the front of the queue.

A sensible implementation of a priority queue is to use a heap data structure (defined in *x*4). Using a heap we can look at the first item in the queue by simply returning the item at index 0 within the heap array. A heap provides us with the ability to construct a priority queue where the items with the highest priority are either those with the smallest value, or those with the largest.

6.3 Double Ended Queue

Unlike the queues we have talked about previously in this chapter a double ended queue allows you to access the items at both the front, and back of the queue. A double ended queue is commonly known as a deque which is the name we will here on in refer to it as.

A deque applies no prioritization strategy to its items like a priority queue does, items are added in order to either the front of back of the deque. The former properties of the deque are denoted by the programmer utilising the data structures exposed interface.

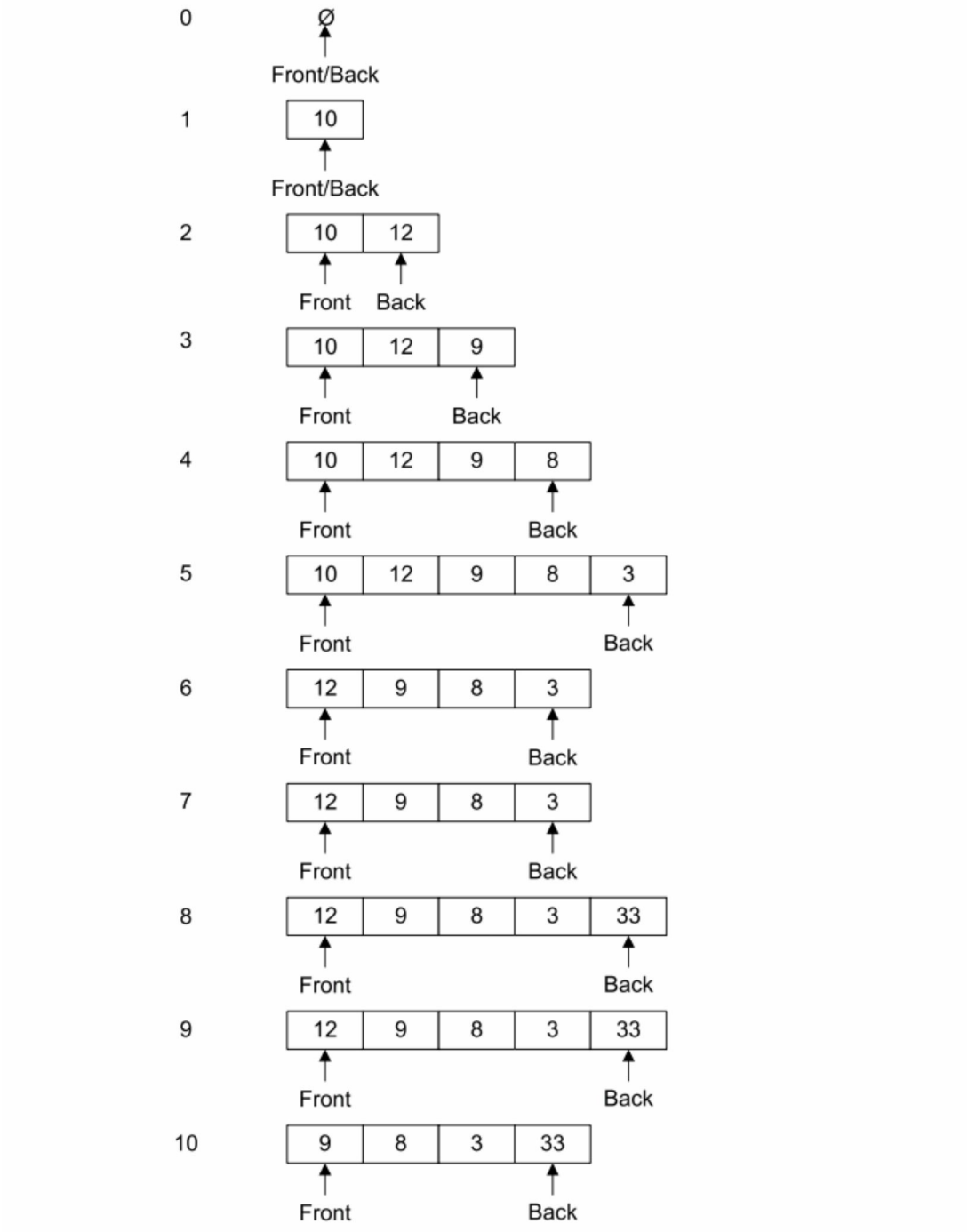


Figure 6.1: Queue mutations

Deque's provide front and back specific versions of common queue operations, e.g. you may want to enqueue an item to the front of the queue rather than the back in which case you would use a method with a name

along the lines of *EnqueueFront*. The following list identifies operations that are commonly supported by deque's:

EnqueueFront

EnqueueBack

DequeueFront

DequeueBack

PeekFront

PeekBack

Figure 6.2 shows a deque after the invocation of the following methods (in-order):

EnqueueBack(12)

EnqueueFront(1)

EnqueueBack(23)

EnqueueFront(908)

DequeueFront()

DequeueBack()

The operations have a one-to-one translation in terms of behaviour with those of a normal queue, or priority queue. In some cases the set of algorithms that add an item to the back of the deque may be named as they are with normal queues, e.g. *EnqueueBack* may simply be called *Enqueue* an so on. Some frameworks also specify explicit behaviour's that data structures must adhere

to. This is certainly the case in .NET where most collections implement an interface which requires the data structure to expose a standard *Add* method. In such a scenario you can safely assume that the *Add* method will simply enqueue an item to the back of the deque.

With respect to algorithmic run time complexities a deque is the same as a normal queue. That is enqueueing an item to the back of a the queue is $O(1)$, additionally enqueuing an item to the front of the queue is also an $O(1)$ operation.

A deque is a wrapper data structure that uses either an array, or a doubly linked list. Using an array as the backing data structure would require the pro-grammer to be explicit about the size of the array up front, this would provide an obvious advantage if the programmer could deterministically state the maxi-mum number of items the deque would contain at any one time. Unfortunately in most cases this doesn't hold, as a result the backing array will inherently incur the expense of invoking a resizing algorithm which would most likely be an $O(n)$ operation. Such an approach would also leave the library developer

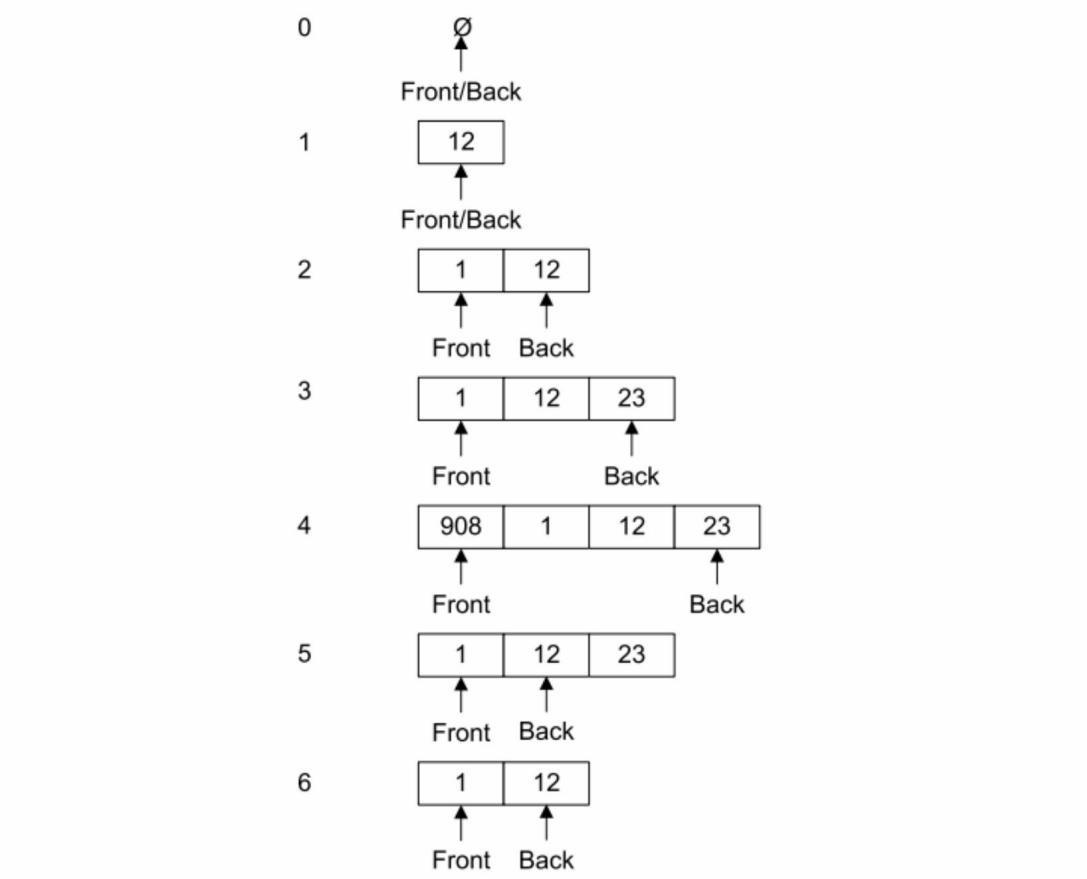


Figure 6.2: Deque data structure after several mutations

to look at array minimization techniques as well, it could be that after several invocations of the resizing algorithm and various mutations on the deque later that we have an array taking up a considerable amount of memory yet we are only using a few small percentage of that memory. An algorithm described would also be $O(n)$ yet its invocation would be harder to gauge strategically.

To bypass all the aforementioned issues a deque typically uses a doubly linked list as its baking data structure. While a node that has two pointers consumes more memory than its array item counterpart it makes redundant the need for expensive resizing algorithms as the data structure increases in size dynamically. With a language that targets a garbage collected virtual machine memory reclamation is an opaque process as the nodes that are no

longer ref-erenced become unreachable and are thus marked for collection upon the next invocation of the garbage collection algorithm. With C++ or any other lan-guage that uses explicit memory allocation and deallocation it will be up to the programmer to decide when the memory that stores the object can be freed.

6.4 Summary

With normal queues we have seen that those who arrive first are dealt with first; that is they are dealt with in a first-in-first-out (FIFO) order. Queues can be ever so useful; for example the Windows CPU scheduler uses a different queue for each priority of process to determine which should be the next process to utilise the CPU for a specified time quantum. Normal queues have constant insertion and deletion run times. Searching a queue is fairly unusual|typically you are only interested in the item at the front of the queue. Despite that, searching is usually exposed on queues and typically the run time is linear.

In this chapter we have also seen priority queues where those at the front of the queue have the highest priority and those near the back have the lowest.

One implementation of a priority queue is to use a heap data structure as its backing store, so the run times for insertion, deletion, and searching are the same as those for a heap (defined in *x*4).

Queues are a very natural data structure, and while they are fairly primitive they can make many problems a lot simpler. For example the breadth first search defined in *x*3.7.4 makes extensive use of queues.

Chapter 7

AVL Tree

In the early 60's G.M. Adelson-Velsky and E.M. Landis invented the first self-balancing binary search tree data structure, calling it AVL Tree.

An AVL tree is a binary search tree (BST, defined in *x*3) with a self-balancing condition stating that the difference between the height of the left and right subtrees cannot be no more than one, see Figure 7.1. This condition, restored after each tree modification, forces the general shape of an AVL tree. Before continuing, let us focus on why balance is so important. Consider a binary search tree obtained by starting with an empty tree and inserting some values in the following order 1,2,3,4,5.

The BST in Figure 7.2 represents the worst case scenario in which the running time of all common operations such as search, insertion and deletion are $O(n)$. By applying a balance condition we ensure that the worst case running time of each common operation is $O(\log n)$. The height of an AVL tree with n nodes is $O(\log n)$ regardless of the order in which values are inserted.

The AVL balance condition, known also as the node balance factor represents an additional piece of information stored for each node. This is combined with a technique that e±ciently restores the balance condition for the tree. In an AVL tree the inventors make use of a well-known technique called tree rotation.

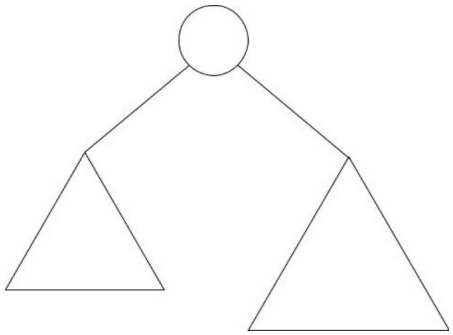


Figure 7.1: The left and right subtrees of an AVL tree differ in height by at most 1

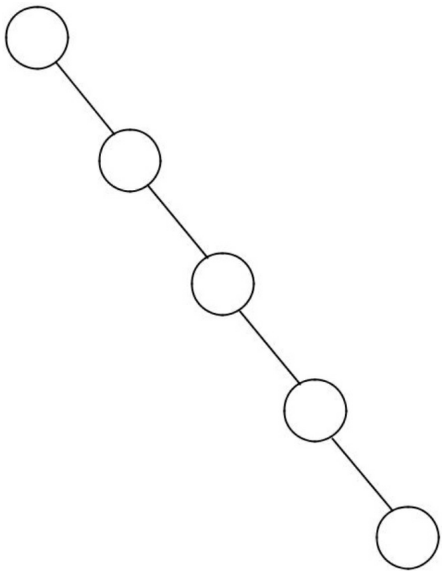


Figure 7.2: Unbalanced binary search tree

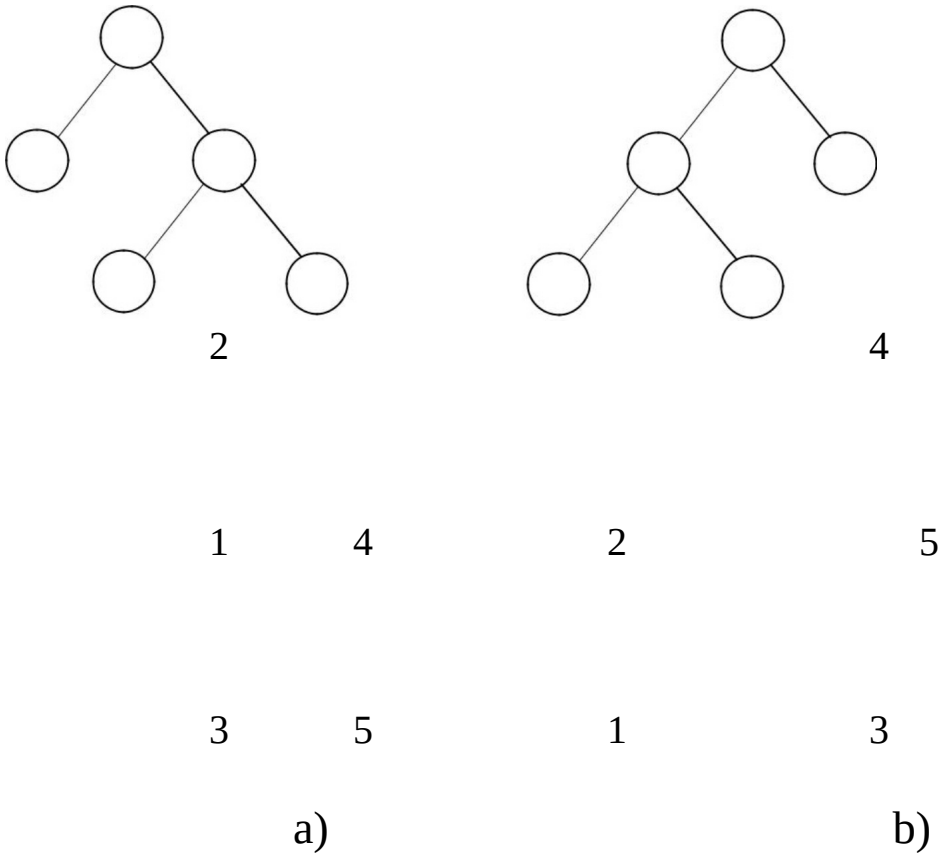


Figure 7.3: Avl trees, insertion order: -a)1,2,3,4,5 -b)1,5,4,3,2

7.1 Tree Rotations

A tree rotation is a constant time operation on a binary search tree that changes the shape of a tree while preserving standard BST properties. There are left and right rotations both of them decrease the height of a BST by moving smaller subtrees down and larger subtrees up.

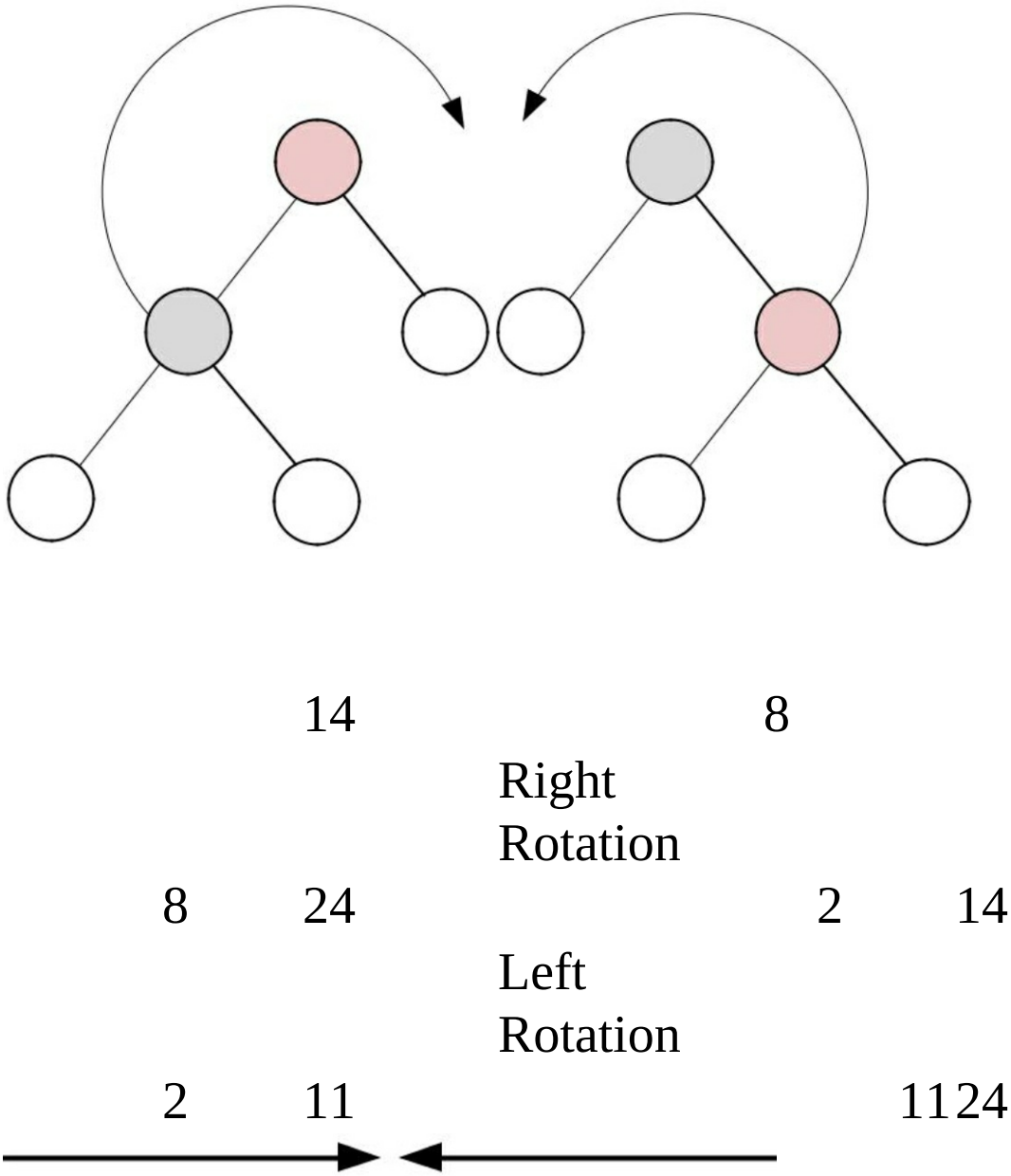


Figure 7.4: Tree left and right rotations

algorithm LeftRotation(*node*)

```
    Pre: node.Right ! = null

    Post: node.Right is the new root of the subtree,
      node has become node.Right's left child and,

      BST properties are preserved

    RightNode <- node.Right

    node.Right <- RightNode.Left
    RightNode.Left <- node

end LeftRotation


algorithm RightRotation(node)

    Pre: node.Left ! =null

    Post: node.Left is the new root of the subtree,
      node has become node.Left's right child and,

      BST properties are preserved

    LeftNode <- node.Left

    node.Left <- LeftNode.Right

    LeftNode.Right <- node

end RightRotation
```

The right and left rotation algorithms are symmetric. Only pointers are changed by a rotation resulting in an $O(1)$ runtime complexity; the other

fields present in the nodes are not changed.

7.2 Tree Rebalancing

The algorithm that we present in this section verifies that the left and right subtrees differ at most in height by 1. If this property is not present then we perform the correct rotation.

Notice that we use two new algorithms that represent double rotations. These algorithms are named *LeftAndRightRotation*, and *RightAndLeftRotation*. The algorithms are self documenting in their names, e.g. *LeftAndRightRotation* first performs a left rotation and then subsequently a right rotation.

algorithm CheckBalance(*current*)

Pre: *current* is the node to start from balancing

Post: *current* height has been updated while tree balance is if needed

restored through rotations

if *current*.Left = null and *current*.Right = null

current.Height = -1;

else

current.Height = Max(*Height*(*current*.Left),*Height*(*current:Right*)) + 1

```
        end if
        if Height(current.Left) - Height(current.Right) > 1
         if Height(current.Left.Left) - Height(current.Left.Right) > 0
          RightRotation(current)
         else
          LeftAndRightRotation(current)
         end if
        else if Height(current.Left) - Height(current.Right) < ¡1
           if Height(current.Right.Left) - Height(current.Right.Right) < 0
            LeftRotation(current)
           else
            RightAndLeftRotation(current)
           end if
        end if
    end CheckBalance
```

7.3 Insertion

AVL insertion operates first by inserting the given value the same way as BST insertion and then by applying rebalancing techniques if necessary. The latter is only performed if the AVL property no longer holds, that is the left

and right subtrees height differ by more than 1. Each time we insert a node into an AVL tree:

1. We go down the tree to find the correct point at which to insert the node, in the same manner as for BST insertion; then

2. We travel up the tree from the inserted node and check that the node balancing property has not been violated; if the property hasn't been violated then we need not rebalance the tree, the opposite is true if the balancing property has been violated.

algorithm Insert(*value*)

Pre: *value* has passed custom type checks for type *T*

Post: *value* has been placed in the correct location in the tree

if *root* = null

root <- node(*value*)

else
InsertNode(*root*, *value*)

end if

end Insert

algorithm InsertNode(*current*, *value*)

Pre: *current* is the node to start from

Post: *value* has been placed in the correct location in the tree while

preserving tree balance

```
        if value < current.Value
         if current.Left = null
          current.Left <- node(value)
         else
          InsertNode(current.Left, value)
         end if
        else
         if current.Right = null
          current.Right <- node(value)
         else
          InsertNode(current.Right, value)
         end if
        end if
        CheckBalance(current)
    end InsertNode
```

7.4 Deletion

Our balancing algorithm is like the one presented for our BST (defined in *x*3.3). The major difference is that we have to ensure that the tree still adheres to the AVL balance property after the removal of the node. If the tree doesn't need to be rebalanced and the value we are removing is contained within the tree then no further step are required. However, when the value is in the tree

and its removal upsets the AVL balance property then we must perform the correct rotation(s).

algorithm Remove(*value*)

Pre: *value* is the value of the node to remove, *root* is the root node

of the Avl

Post: node with *value* is removed and tree rebalanced if found in which

case yields true, otherwise false

nodeToRemove <- *root*

parent <- *null;*

Stackpath <- root

while *nodeToRemove* != *null* and *nodeToRemove:Value* = *Value*

parent = *nodeToRemove*
if *value* < *nodeToRemove*.Value

nodeToRemove <- nodeToRemove.Left

else
nodeToRemove <- nodeToRemove.Right

end if
path.Push(nodeToRemove)

end while

if *nodeToRemove* = null

```
 return false // value not in Avl

end if

parent <- FindParent(value)

if count = 1 // count keeps track of the # of nodes in the Avl
 root <- null; // we are removing the only node in the Avl

else if nodeT oRemove.Left = ; and nodeT oRemove.Right = null

 // case #1
 if nodeToRemove.Value < parent.Value
  parent.Left <- null

 else
  parent.Right <- null

 end if
else if nodeToRemove.Left = null and nodeToRemove.Right !=
   null

 // case # 2
 if nodeToRemove.Value < parent.Value

  parent.Left <- nodeToRemove.Right

 else
  parent.Right <- nodeToRemove.Right

 end if
else if nodeToRemove.Left != null and nodeToRemove.Right =
   null

 // case #3
 if nodeToRemove.Value < parent.Value
```

```
        parent.Left <- nodeT oRemove.Left

       else
        parent.Right <- nodeToRemove.Left

       end if

      else

       // case #4

       largestV alue <- nodeToRemove.Left

       while largestV alue.Right != null

        // find the largest value in the left subtree of nodeToRemove
        largestValue <- largestValue.Right
       end while

       // set the parents' Right pointer of largestV alue to ;

       FindParent(largestV alue.Value).Right <- null

       nodeToRemove.Value <- largestValue.Value

      end if
      while path:Count > 0

       CheckBalance(path.Pop()) // we trackback to the root node check
           balance

      end while

      count <- count - 1

      return true
  end Remove
```

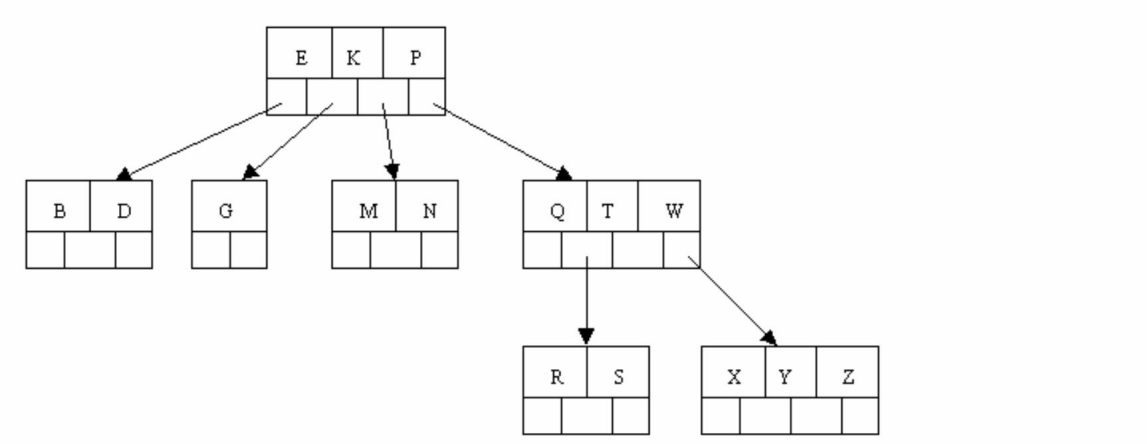
B-Trees

Introduction

A B-tree is a specialized multiway tree designed especially for use on disk. In a B-tree each node may contain a large number of keys. The number of subtrees of each node, then, may also be large. A B-tree is designed to branch out in this large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small. This means that only a small number of nodes must be read from disk to retrieve an item. The goal is to get fast access to the data, and with disk drives this means reading a very small number of records. Note that a large node size (with lots of keys in the node) also fits with the fact that with a disk drive one can usually read a fair amount of data at once.

Definitions

A *multiway tree of order m* is an ordered tree where each node has at most m children. For each node, if k is the actual number of children in the node, then k - 1 is the number of keys in the node. If the keys and subtrees are arranged in the fashion of a search tree, then this is called a *multiway search tree of order m*. For example, the following is a multiway search tree of order 4. Note that the first row in each node shows the keys, while the second row shows the pointers to the child nodes. Of course, in any useful application there would be a record of data associated with each key, so that the first row in each node might be an array of records where each record contains a key and its associated data. Another approach would be to have the first row of each node contain an array of records where each record contains a key and a record number for the associated data record, which is found in another file. This last method is often used when the data records are large. The example software will use the first method.



Then a multiway search tree of order 4 has to fulfill the following conditions related to the ordering of the keys:

A B-tree of order m is a multiway search tree of order m such that:

- All leaves are on the bottom level.
- All internal nodes (except the root node) have at least ceil(m / 2)

(nonempty) children.

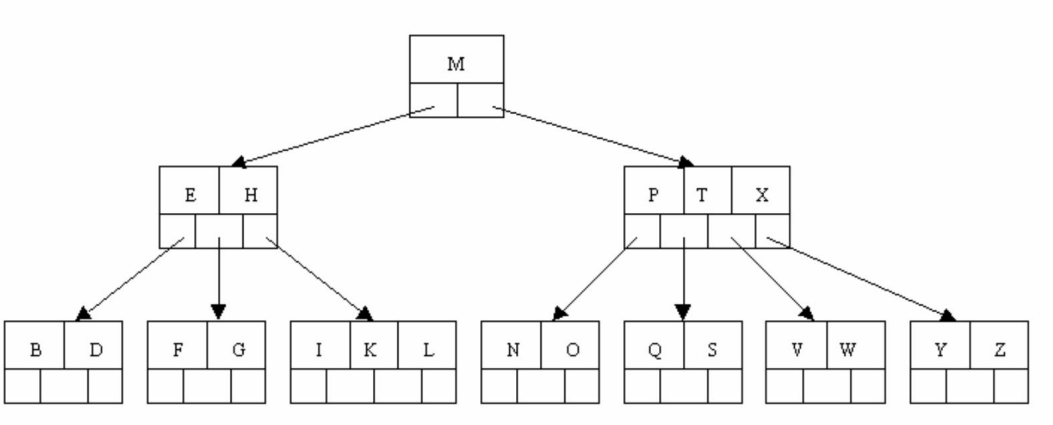
- The root node can have as few as 2 children if it is an internal node, and can obviously have no children if the root node is a leaf (that is, the whole tree consists only of the root node).
- Each leaf node must contain at least ceil(m / 2) - 1 keys.

Note that ceil(x) is the so-called ceiling function. It's value is the smallest integer that is greater than or equal to x. Thus ceil(3) = 3, ceil(3.35) = 4, ceil(1.98) = 2, ceil(5.01) = 6, ceil(7) = 7, etc.

A B-tree is a fairly well-balanced tree by virtue of the fact that all leaf nodes must be at the bottom. Condition (2) tries to keep the tree fairly bushy by insisting that each node have at least half the maximum number of children. This causes the tree to "fan out" so that the path from root to leaf is very short even in a tree that contains a lot of data.

Example B-Tree

The following is an example of a B-tree of order 5. This means that (other that the root node) all internal nodes have at least ceil(5 / 2) = ceil(2.5) = 3 children (and hence at least 2 keys). Of course, the maximum number of children that a node can have is 5 (so that 4 is the maximum number of keys). According to condition 4, each leaf node must contain at least 2 keys. In practice B-trees usually have orders a lot bigger than 5.



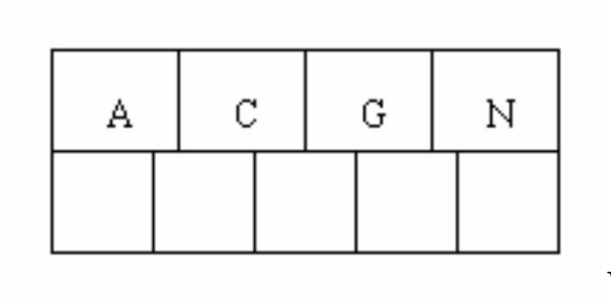
Operations on a B-Tree

Question: How would you search in the above tree to look up S? How about J? How would you do a sort-of "in-order" traversal, that is, a traversal that would produce the letters in ascending order? (One would only do such a traversal on rare occasion as it would require a large amount of disk activity and thus be very slow!)

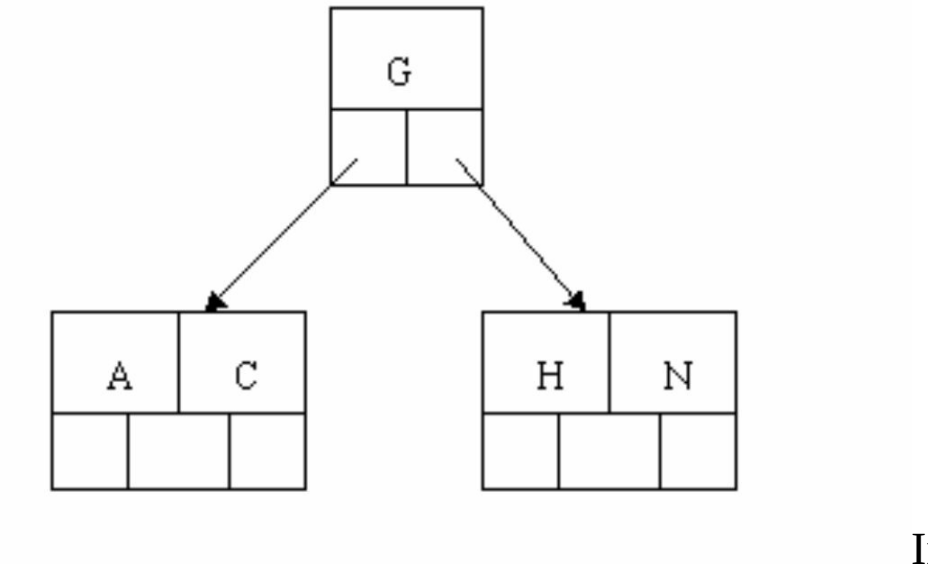
Inserting a New Item

According to Kruse (see reference at the end of this file) the insertion algorithm proceeds as follows: When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf. If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.) Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.

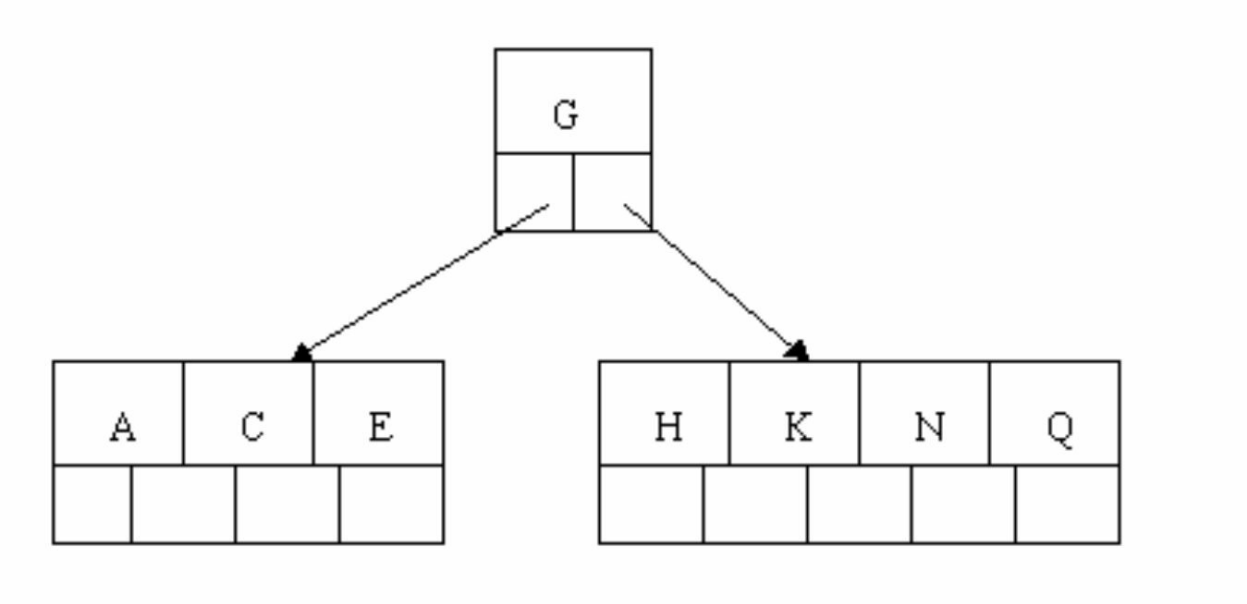
Let's work our way through an example similar to that given by Kruse. Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:



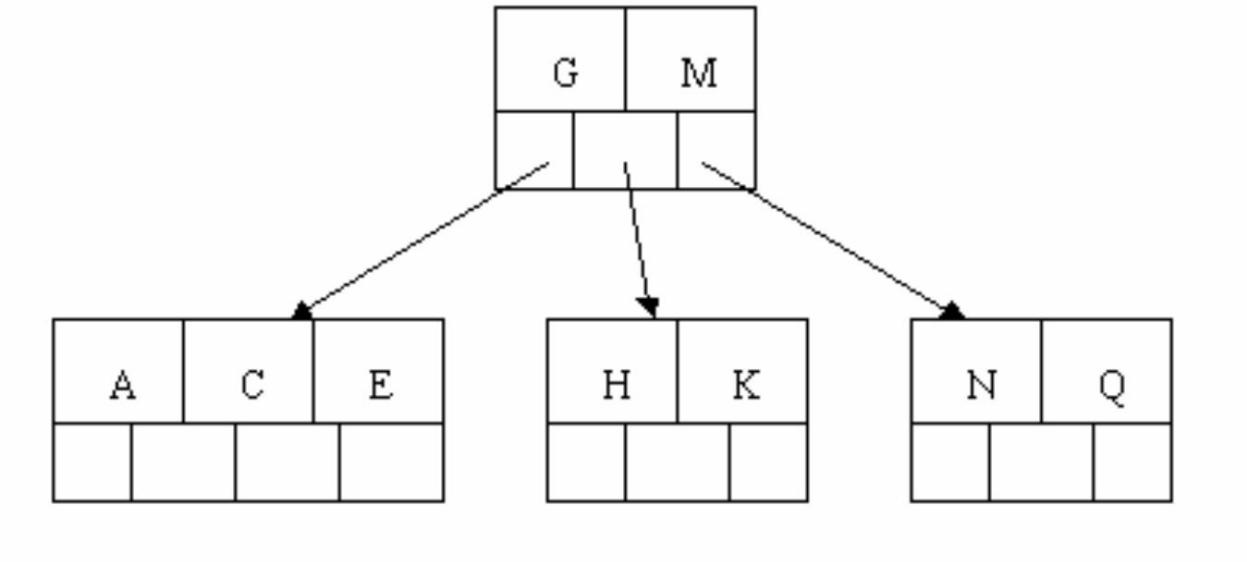
When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.



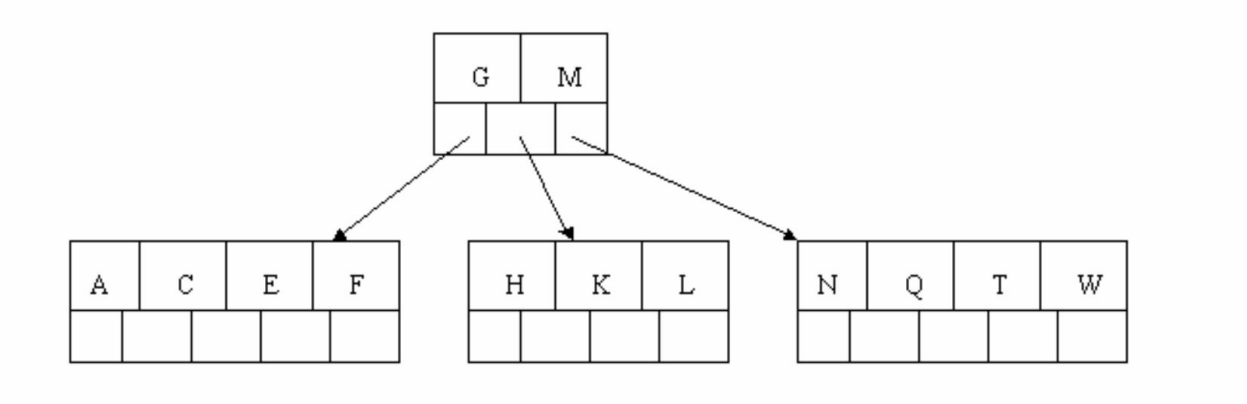
Inserting E, K, and Q proceeds without requiring any splits:



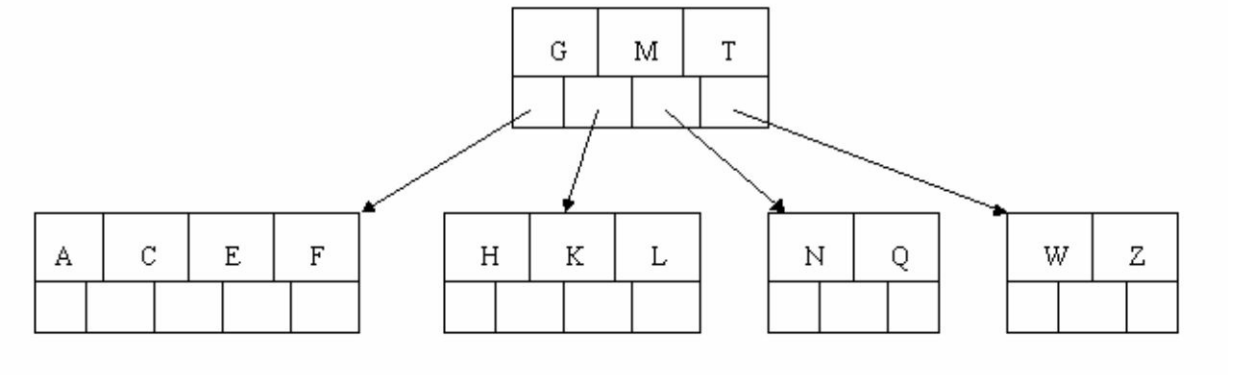
Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.



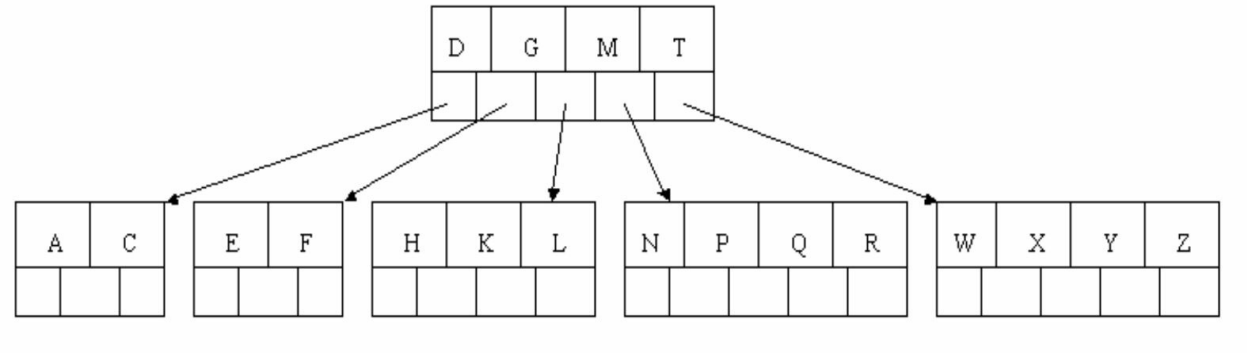
The letters F, W, L, and T are then added without needing any split.



When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.

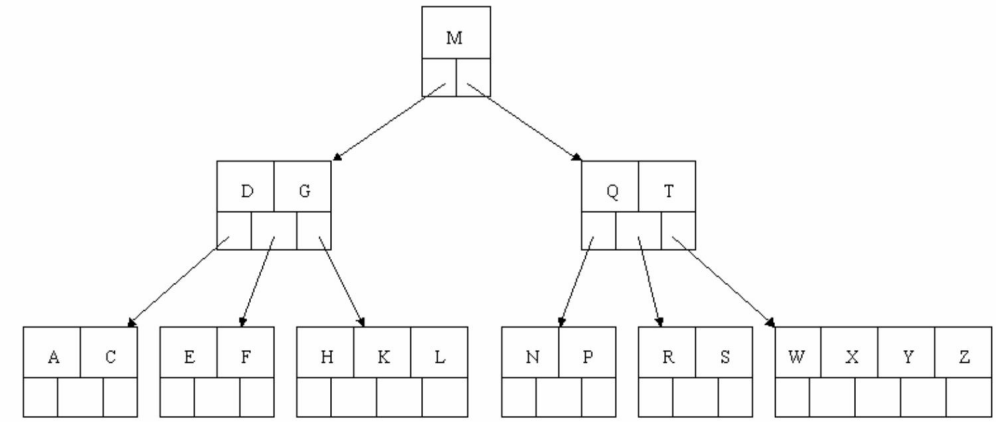


The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:



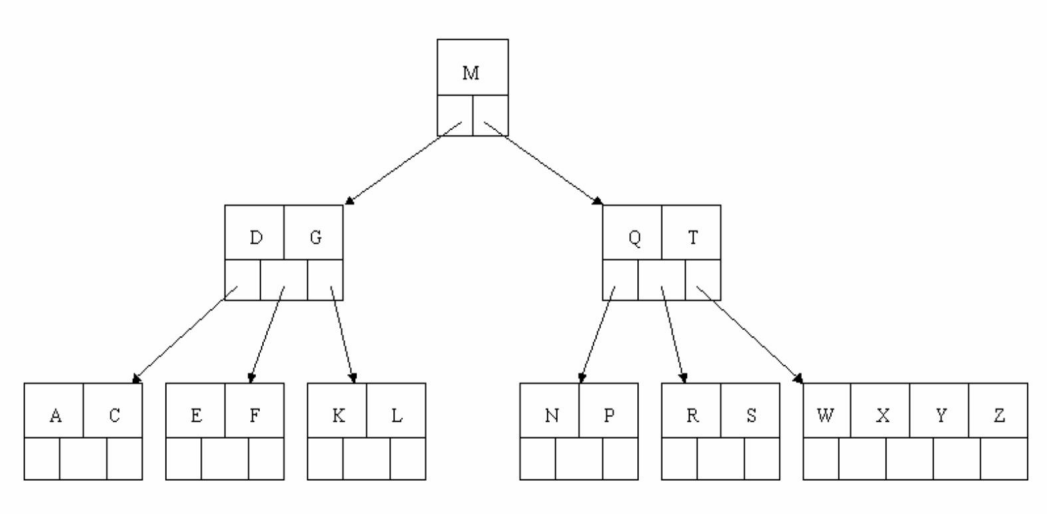
Finally, when S is added, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits,

sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.



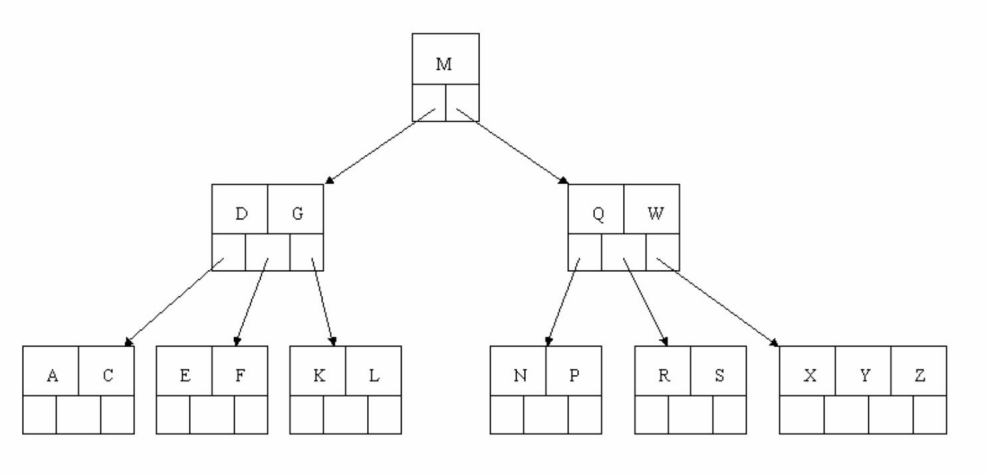
Deleting an Item

In the B-tree as we left it at the end of the last section, delete H. Of course, we first do a lookup to find H. Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been. This gives:

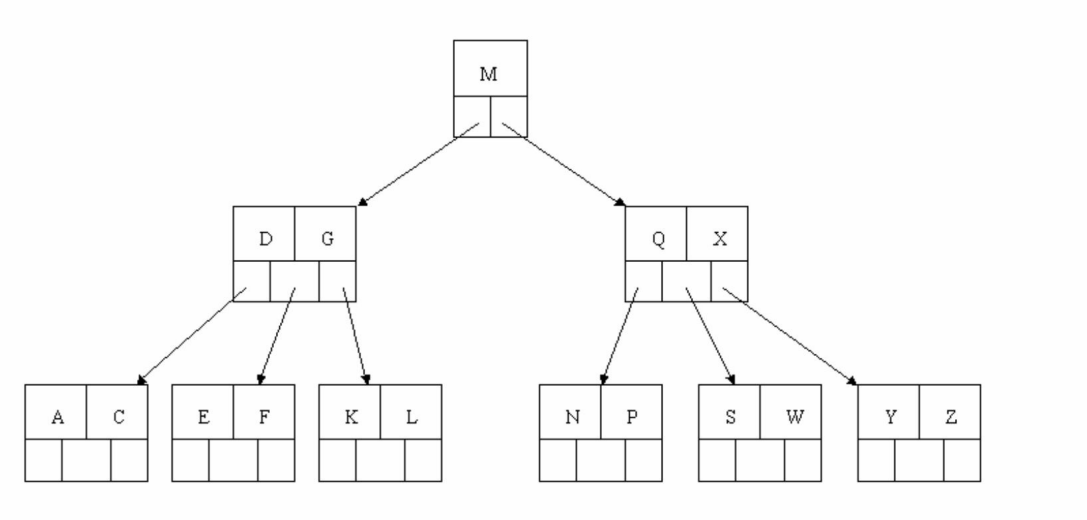


Next, delete the T. Since T is not in a leaf, we find its successor (the next item in ascending order), which happens to be W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL

cases we reduce deletion to a deletion in a leaf, by using this method.

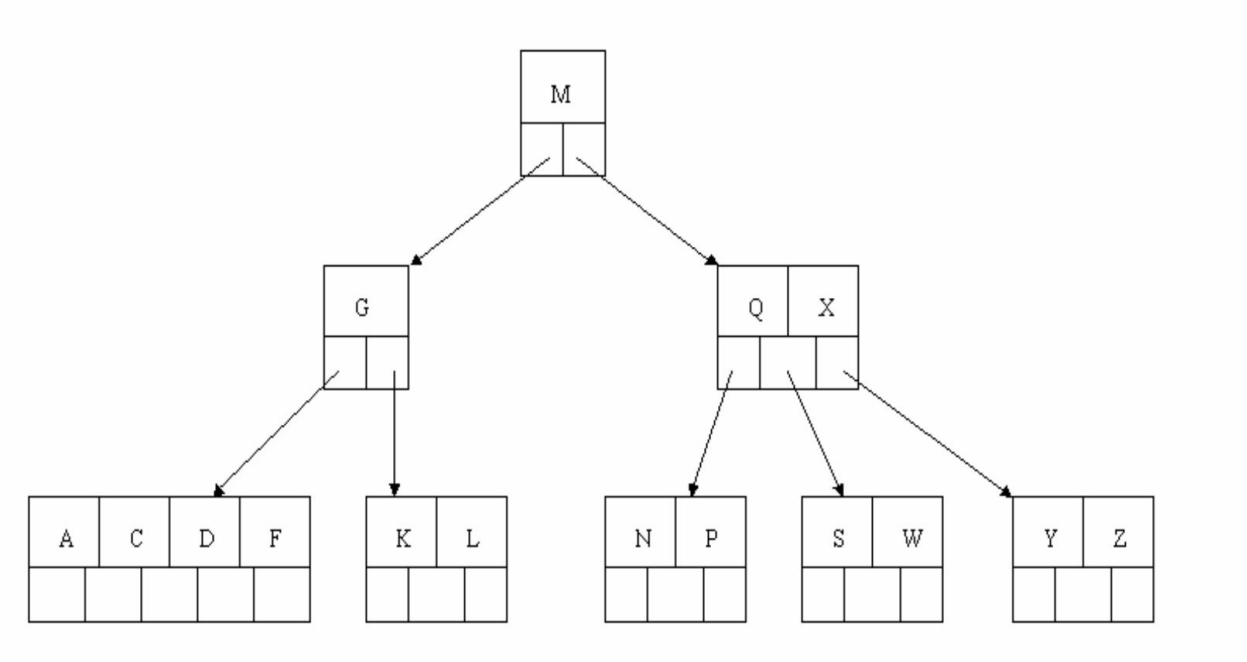


Next, delete R. Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (Of course, the S is moved over so that the W can be inserted in its proper place.)

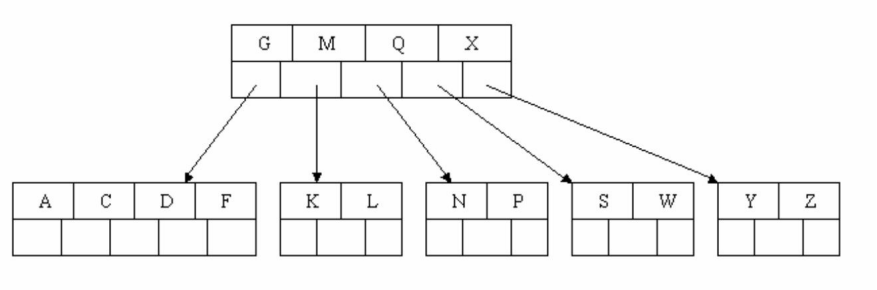


Finally, let's delete E. This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or

left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.

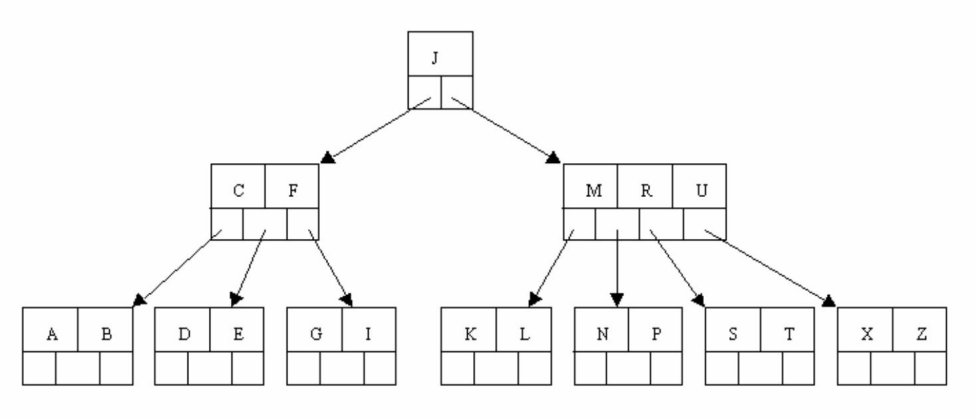


Of course, you immediately see that the parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Suppose for the moment that the right sibling (the node with Q X) had one more key in it somewhere to the right of Q. We would then move M down to the node with too few keys and move the Q up where the M had been. However, the old left subtree of Q would then have to become the right subtree of M. In other words, the N P node would be attached via the pointer field to the right of M's new location. Since in our example we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.

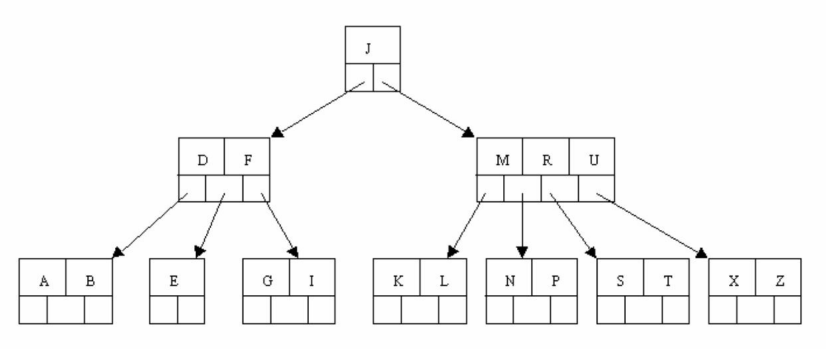

G
M
Q
X
A
C
D
F
K
L
N
P
S
W
Y
Z

Another Example

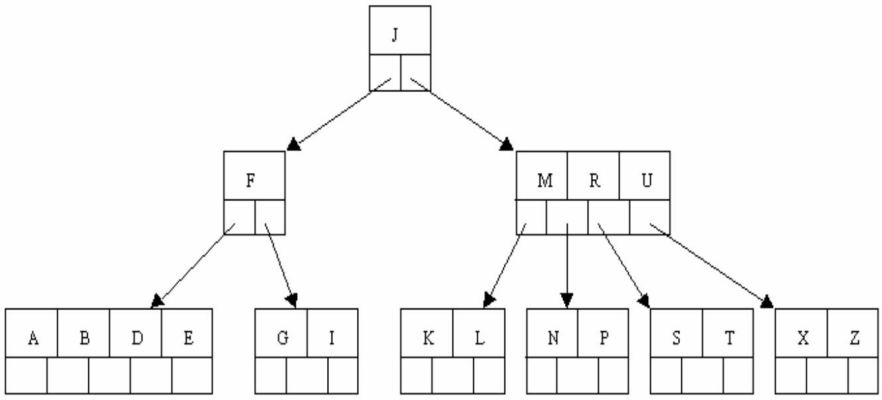
Here is a different B-tree of order 5. Let's try to delete C from it.



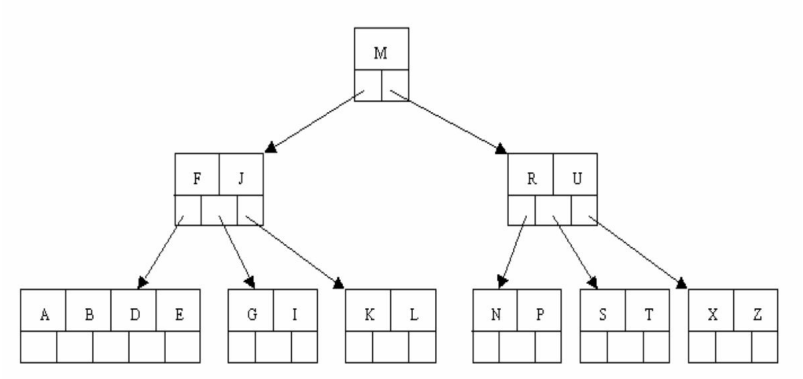
We begin by finding the immediate successor, which would be D, and move the D up to replace the C. However, this leaves us with a node with too few keys.



Since neither the sibling to the left or right of the node containing E has an extra key, we must combine the node with one of these two siblings. Let's consolidate with the A B node.



But now the node containing F does not have enough keys. However, its sibling has an extra key. Thus we borrow the M from the sibling, move it up to the parent, and bring the J down to join the F. Note that the K L node gets reattached to the right of the J.



7.5 Summary

The AVL tree is a sophisticated self balancing tree. It can be thought of as the smarter, younger brother of the binary search tree. Unlike its older brother the AVL tree avoids worst case linear complexity runtimes for its operations. The AVL tree guarantees via the enforcement of balancing algorithms that the left and right subtrees differ in height by at most 1 which yields at most a logarithmic runtime complexity.

Part II

Algorithms

Chapter 8

Sorting

All the sorting algorithms in this chapter use data structures of a specific type to demonstrate sorting, e.g. a 32 bit integer is often used as its associated operations (e.g. <, >, etc) are clear in their behaviour.

The algorithms discussed can easily be translated into generic sorting algorithms within your respective language of choice.

8.1 Bubble Sort

One of the most simple forms of sorting is that of comparing each item with every other item in some list, however as the description may imply this form of sorting is not particularly effecient $O(n^2)$. In it's most simple form bubble sort can be implemented as two loops.

lgorithm BubbleSort(*list*)

Pre: *list* != *null*

Post: *list* has been sorted into values of ascending order
for *i* <- 0 to *list:Count* - 1

```
    for j <- 0 to list:Count - 1

      if list[i] < list[j]
       Swap(list[i]; list[j])

      end if

    end for

  end for

  return list

nd BubbleSort
```

8.2 Merge Sort

Merge sort is an algorithm that has a fairly e±cient space time complexity - $O(n \log n)$ and is fairly trivial to implement. The algorithm is based on splitting a list, into two similar sized lists (*left*, and *right*) and sorting each list and then merging the sorted lists back together.

Note: the function MergeOrdered simply takes two ordered lists and makes them one.

```
 algorithm Mergesort(list)

  Pre: list != null

  Post: list has been sorted into values of ascending order
```

```
 if list.Count = 1 // already sorted
                    return list
 end if
 m <- list.Count / 2
 left <- list(m)
 right <- list(list.Count - m)
 for i <- 0 to left.Count - 1
                    left[i] <- list[i]
 end for
 for i <- 0 to right.Count - 1
                    right[i] <- list[i]
 end for
 left <- Mergesort(left)
 right <- Mergesort(right)
 return MergeOrdered(left, right)
end Mergesort
```

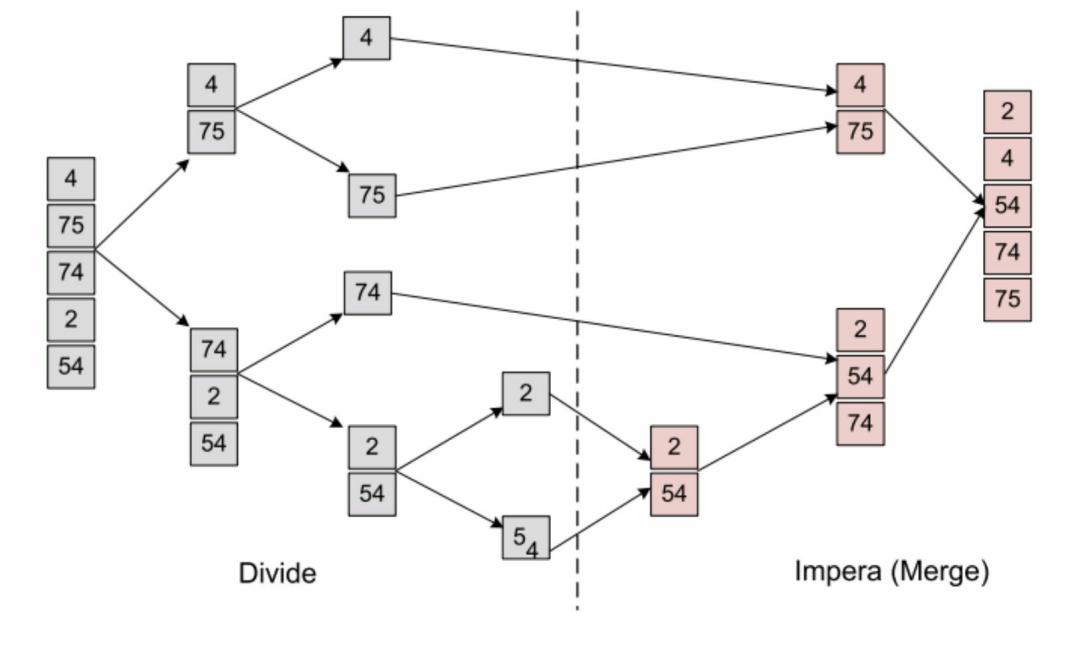


Figure 8.2: Merge Sort Divide et Impera Approach

8.3 Quick Sort

Quick sort is one of the most popular sorting algorithms based on divide et impera strategy, resulting in an $O(n \log n)$ complexity. The algorithm starts by picking an item, called pivot, and moving all smaller items before it, while all greater elements after it. This is the main quick sort operation, called partition, recursively repeated on lesser and greater sub lists until their size is one or zero - in which case the list is implicitly sorted.

Choosing an appropriate pivot, as for example the median element is fundamental for avoiding the drastically reduced performance of $O(n^2)$.

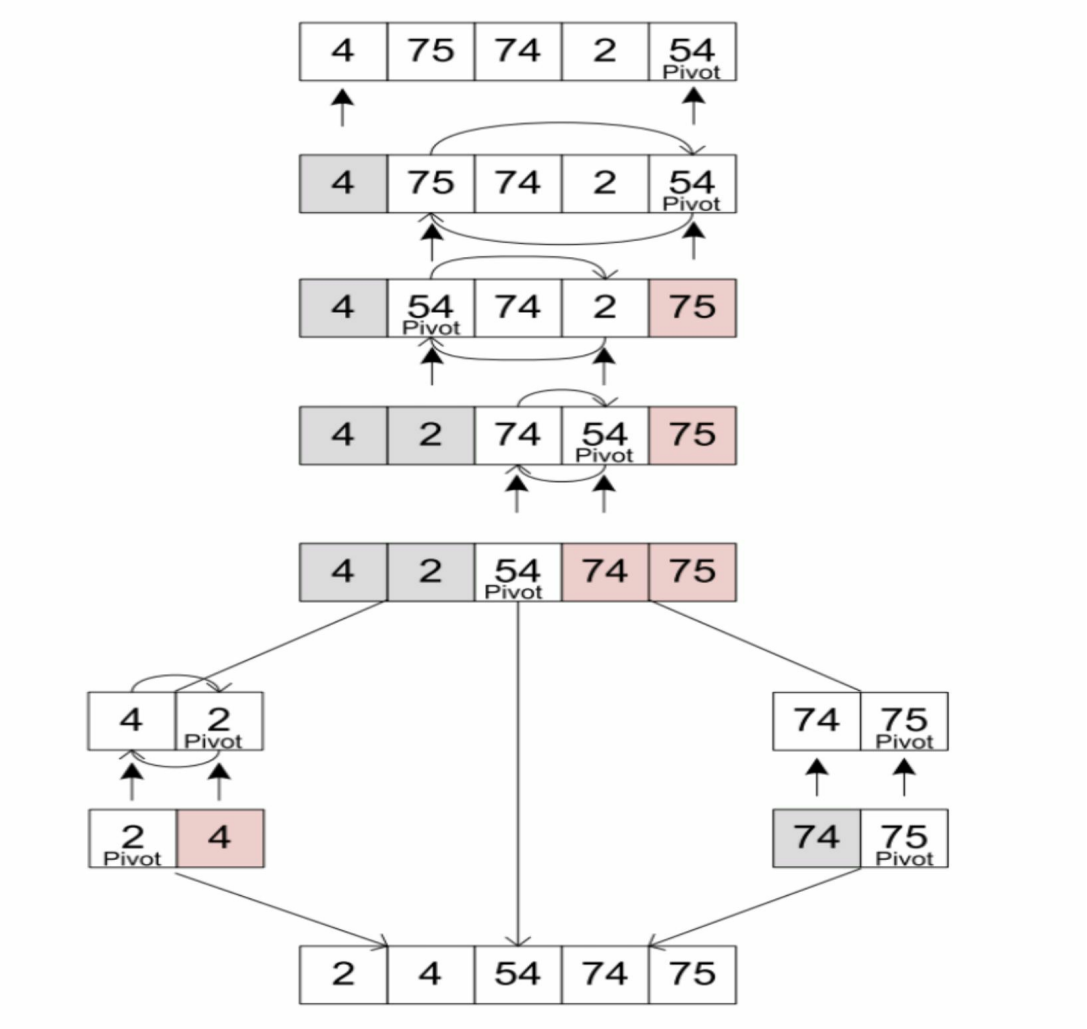


Figure 8.3: Quick Sort Example (pivot median strategy)

lgorithm QuickSort(*list*)

Pre: *list* != *null*

Post: *list* has been sorted into values of ascending order
if *list*.Count = 1 // already sorted

return *list*

end if

pivot <- MedianValue(*list*)

```
    for i <- 0 to list.Count - 1

                    if list[i] = pivot
                        equal.Insert(list[i])

                    end if

                    if list[i] < pivot

                        less.Insert(list[i])

                    end if

                    if list[i] > pivot

                        greater.Insert(list[i])

                    end if

                end for

                return Concatenate(QuickSort(less), equal,
        QuickSort(greater))
end Quicksort
```

8.4 Insertion Sort

Insertion sort is a somewhat interesting algorithm with an expensive runtime of $O(n^2)$. It can be best thought of as a sorting scheme similar to that of sorting a hand of playing cards, i.e. you take one card and then look at the rest with the intent of building up an ordered set of cards in your hand.

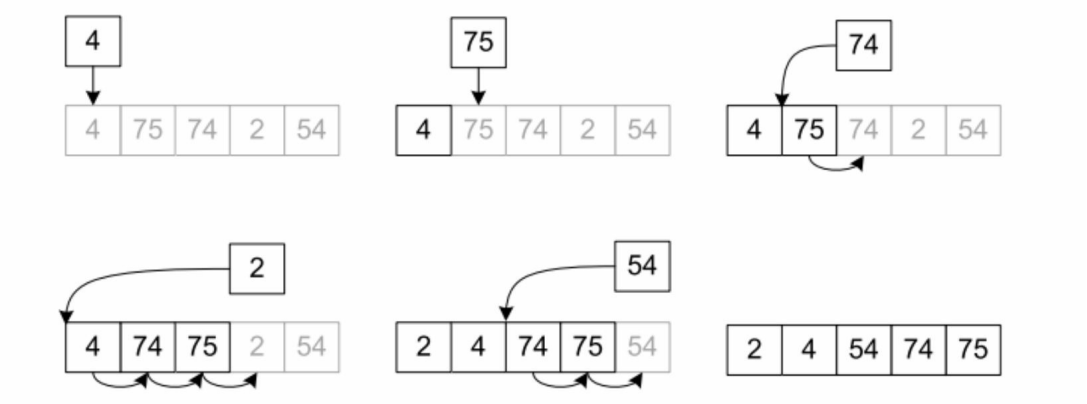


Figure 8.4: Insertion Sort Iterations

lgorithm Insertionsort(*list*)

Pre: *list* != *null*

Post: *list* has been sorted into values of ascending order
unsorted <- 1

while *unsorted* < *list*.Count
hold <- *list*[*unsorted*]

i <- *unsorted* - 1

while *i* >= 0 and *hold* < *list*[*i*]

list[*i* + 1] <- *list*[*i*]
i <- *i* - 1

end while
list[*i* + 1] <- *hold*

unsorted <- *unsorted* + 1

end while

return *list*

nd Insertionsort

8.5 Shell Sort

Put simply shell sort can be thought of as a more e±cient variation of insertion sort as described in *x*8.4, it achieves this mainly by comparing items of varying distances apart resulting in a run time complexity of $O(n\ log^2\ n)$.

Shell sort is fairly straight forward but may seem somewhat confusing at first as it differs from other sorting algorithms in the way it selects items to compare. Figure 8.5 shows shell sort being ran on an array of integers, the red coloured square is the current value we are holding.

```
algorithm ShellSort(list)

  Pre: list != null

  Post: list has been sorted into values of ascending order
  increment <- list.Count = 2

  while increment != 0

                    current <- increment

                    while current < list.Count
                        hold <- list[current]

                        i <- current - increment

                        while i >= 0 and hold < list[i]

                            list[i + increment] <- list[i]

                            i - = increment

                        end while
```

list[*i* + *increment*] <- *hold*

current <- *current* + 1

end while
increment / = 2

end while

return *list*

end ShellSort

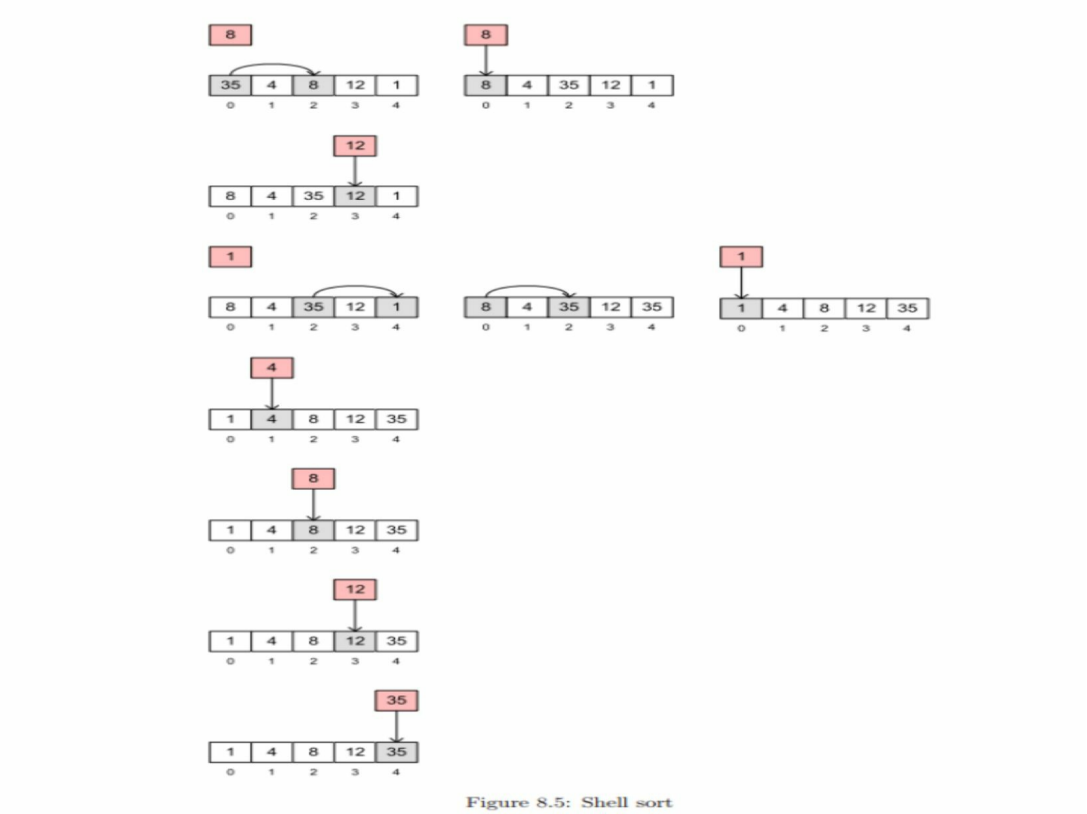


Figure 8.5: Shell sort

8.6 Bucket Sort

Bucket sort is mainly useful when input is uniformly distributed over a range. For example, consider the following problem.

Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

In the Bucket Sorting technique, the data items are distributed in a set of buckets. Each bucket can hold a similar type of data. After distributing, each bucket is sorted using another sorting algorithm. After that, all elements are gathered on the main list to get the sorted form. Following is the bucket algorithm;

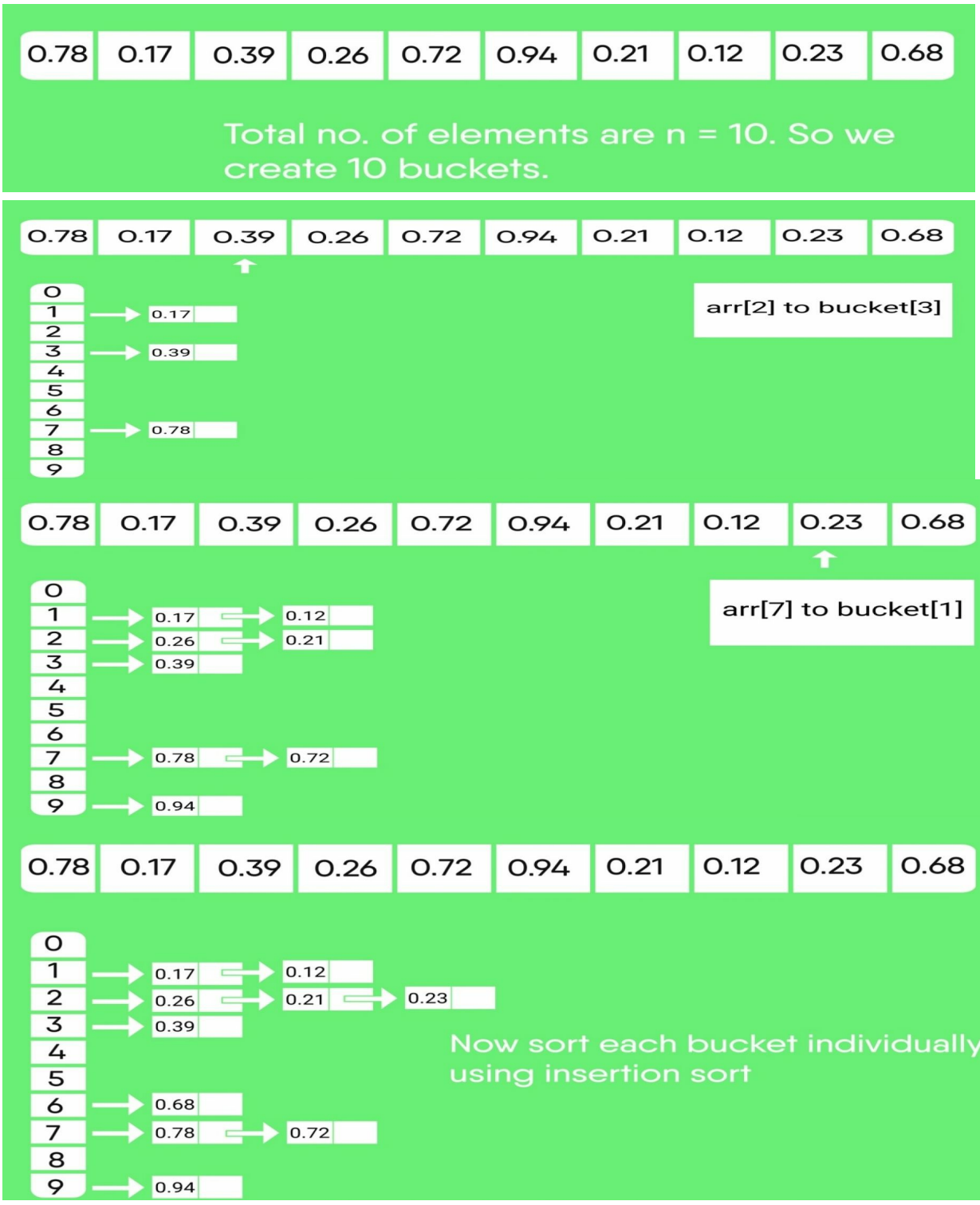
bucketSort(arr[], n)

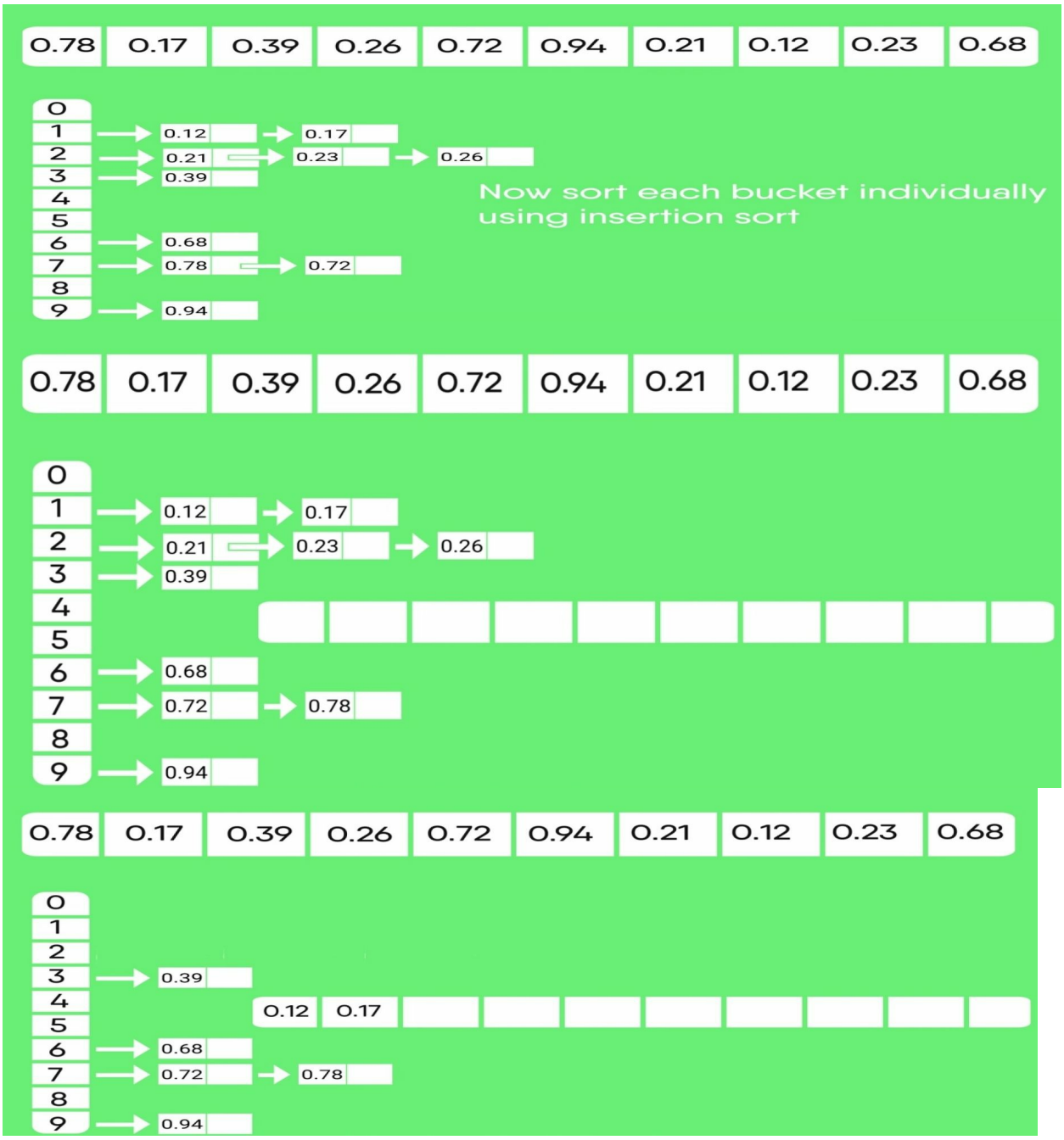
1) Create n empty buckets (Or lists).
2) Do following for every array element arr[i].
.....a) Insert arr[i] into bucket[n*array[i]]
3) Sort individual buckets using insertion sort.
4) Concatenate all sorted buckets.

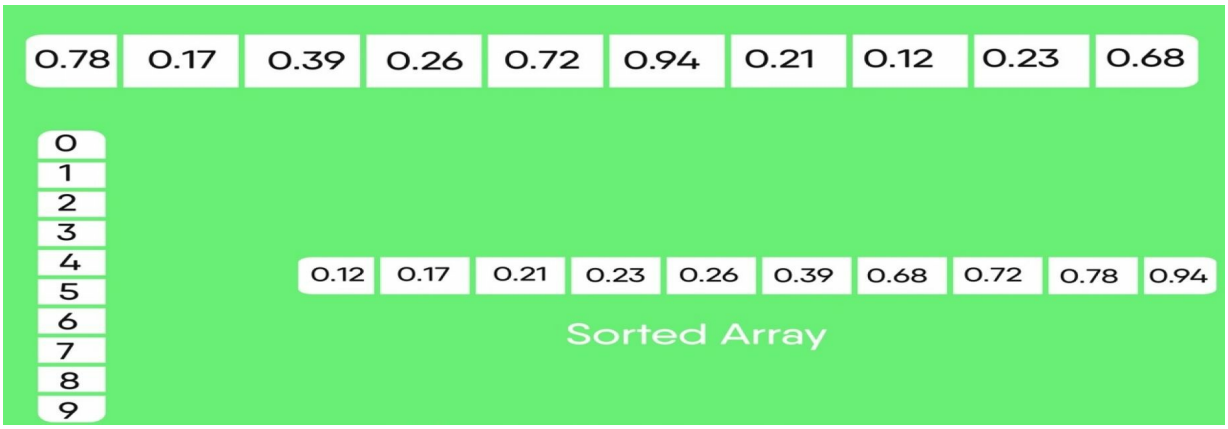
Time Complexity: If we assume that insertion in a bucket takes O(1) time then steps 1 and 2 of the above algorithm clearly take O(n) time. The O(1) is easily possible if we use a linked list to represent a bucket. Step 4 also takes O(n) time as there will be n items in all buckets.

The main step to analyze is step 3. This step also takes O(n) time on average if all numbers are uniformly distributed.

Snapshots:


0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
Total no. of elements are n = 10. So we create 10 buckets.
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
arr[2] to bucket[3]
0
1 0.17
2
3 0.39
4
5
6
7 0.78
8
9
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
arr[7] to bucket[1]
0
1 0.17 0.12
2 0.26 0.21
3 0.39
4
5
6
7 0.78 0.72
8
9 0.94
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
0
1 0.17 0.12
2 0.26 0.21 0.23
3 0.39
4
5
6 0.68
7 0.78 0.72
8
9 0.94
Now sort each bucket individually using insertion sort


0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
0
1 → 0.12 → 0.17
2 → 0.21 → 0.23 → 0.26
3 → 0.39
4
5
6 → 0.68
7 → 0.78 → 0.72
8
9 → 0.94
Now sort each bucket individually using insertion sort
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
0
1 → 0.12 → 0.17
2 → 0.21 → 0.23 → 0.26
3 → 0.39
4
5
6 → 0.68
7 → 0.72 → 0.78
8
9 → 0.94
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68
0
1
2
3 → 0.39
4 0.12 0.17
5
6 → 0.68
7 → 0.72 → 0.78
8
9 → 0.94



8.7 Radix Sort

Unlike the sorting algorithms described previously radix sort uses buckets to sort items, each bucket holds items with a particular property called a key. Normally a bucket is a queue, each time radix sort is performed these buckets are emptied starting the smallest key bucket to the largest. When looking at items within a list to sort we do so by isolating a specific key, e.g. in the example we are about to show we have a maximum of three keys for all items, that is the highest key we need to look at is hundreds. Because we are dealing with, in this example base 10 numbers we have at any one point 10 possible key values 0*::*9 each of which has their own bucket. Before we show you this first simple version of radix sort let us clarify what we mean by isolating keys. Given the number 102 if we look at the first key, the ones then we can see we have two of them, progressing to the next key - tens we can see that the number has zero of them, finally we can see that the number has a single hundred. The number used as an example has in total three keys:

Ones

Tens

Hundreds

For further clarification what if we wanted to determine how many thousands the number 102 has? Clearly there are none, but often looking at a number as final like we often do it is not so obvious so when asked the question how many thousands does 102 have you should simply pad the number with a zero in that location, e.g. 0102 here it is more obvious that the key value at the thousands location is zero.

The last thing to identify before we actually show you a simple implementation of radix sort that works on only positive integers, and requires you to specify the maximum key size in the list is that we need a way to isolate a specific key at any one time. The solution is actually very simple, but its not often you want to isolate a key in a number so we will spell it out clearly here. A key can be accessed from any integer with the following expression: *key* <- (*number = keyT oAccess*) % 10. As a simple example lets say that we want to access the tens key of the number 1290, the tens column is key 10 and so after substitution yields *key* <- (1290 = 10) % 10 = 9. The next key to look at for a number can be attained by multiplying the last key by ten working left to right in a sequential manner. The value of *key* is used in the following algorithm to work out the index of an array of queues to enqueue the item into.

algorithm Radix(*list*, *maxKeySize*)
  Pre: *list* !=*null*
    maxKeySize = 0 and represents the largest key size in the list
  Post: *list* has been sorted
  queues <- Queue[10]
  indexOf Key <- 1
  for *i* <- 0 to *maxKeySize* >= 1

```
    foreach item in list
      queues[GetQueueIndex(item, indexOf Key)].Enqueue(item)
    end foreach
    ist <- CollapseQueues(queues)
    ClearQueues(queues)
    indexOf Key <- indexOf Key ¤ 10
  end for
  return list
end Radix
```

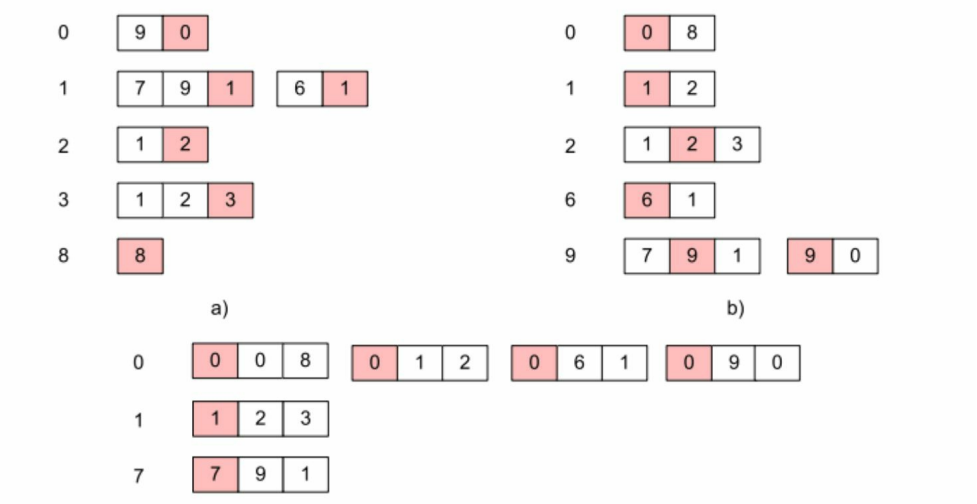


Figure 8.6: Radix sort base 10 algorithm

Figure 8.6 shows the members of *queues* from the algorithm described above operating on the list whose members are 90*;* 12*;* 8*;* 791*;* 123*;* and 61, the key we are interested in for each number is highlighted. Omitted queues in Figure 8.6 mean that they contain no items.

8.7 Summary

Throughout this chapter we have seen many different algorithms for sorting lists, some are very e±cient (e.g. quick sort defined in *x*8.3), some are not (e.g. bubble sort defined in *x*8.1).

Selecting the correct sorting algorithm is usually denoted purely by e±ciency, e.g. you would always choose merge sort over shell sort and so on. There are also other factors to look at though and these are based on the actual implementation. Some algorithms are very nicely expressed in a recursive fashion, however these algorithms ought to be pretty e±cient, e.g. implementing a linear, quadratic, or slower algorithm using recursion would be a very bad idea.

If you want to learn more about why you should be very, very careful when implementing recursive algorithms see Appendix C.

Chapter 9

Numeric

Unless stated otherwise the alias *n* denotes a standard 32 bit integer.

9.1 Primality Test

A simple algorithm that determines whether or not a given integer is a prime number, e.g. 2, 5, 7, and 13 are all prime numbers, however 6 is not as it can be the result of the product of two numbers that are < 6.

In an attempt to slow down the inner loop the p_n is used as the upper bound.

```
algorithm IsPrime(n)
  Post: n is determined to be a prime or not
  for i <- 2 to n do
                    for j <- 1 to sqrt(n) do
                        if i ¤ j = n
                        return false
                    end if
                end for
```

end for

end IsPrime

9.2 Base conversions

DSA contains a number of algorithms that convert a base 10 number to its equivalent binary, octal or hexadecimal form. For example 78_{10} has a binary representation of 1001110_2.

Table 9.1 shows the algorithm trace when the number to convert to binary is 742_{10}.

algorithm ToBinary(*n*)

Pre: *n* ¸ 0

Post: *n* has been converted into its base 2 representation
while *n* > 0

list:Add(*n* % 2)

n <- *n*=2

end while
return Reverse(*list*)

end ToBinary

n	*list*
742	*f* 0 *g*
371	*f* 0*;* 1 *g*
185	*f* 0*;* 1*;* 1 *g*
92	*f* 0*;* 1*;* 1*;* 0 *g*
46	*f* 0*;* 1*;* 1*;* 0*;* 1 *g*
23	*f* 0*;* 1*;* 1*;* 0*;* 1*;* 1 *g*
11	*f* 0*;* 1*;* 1*;* 0*;* 1*;* 1*;* 1

	g
5	*f* 0*;* 1*;* 1*;* 0*;* 1*;* 1*;* 1*;* 1 *g*
2	*f* 0*;* 1*;* 1*;* 0*;* 1*;* 1*;* 1*;* 1*;* 0 *g*
1	*f* 0*;* 1*;* 1*;* 0*;* 1*;* 1*;* 1*;* 1*;* 0*;* 1 *g*

Table 9.1: Algorithm trace of ToBinary

9.3 Attaining the greatest common denomina-tor of two numbers

A fairly routine problem in mathematics is that of finding the greatest common denominator of two integers, what we are essentially after is the greatest number which is a multiple of both, e.g. the greatest common denominator of 9, and 15 is 3. One of the most elegant solutions to this problem is based on Euclid's algorithm that has a run time complexity of $O(n^2)$.

algorithm GreatestCommonDenominator(*m*, *n*)

Pre: *m* and *n* are integers

Post: the greatest common denominator of the two integers is calculated

if *n* = 0

return *m*

end if

return GreatestCommonDenominator(*n*, *m* % *n*)

end GreatestCommonDenominator

9.4 omputing the maximum value for a num-ber of a specific base consisting of N digits

This algorithm computes the maximum value of a number for a given number of digits, e.g. using the base 10 system the maximum number we can have made up of 4 digits is the number 9999_{10}. Similarly the maximum number that consists of 4 digits for a base 2 number is 1111_2 which is 15_{10}.

The expression by which we can compute this maximum value for *N* digits is: B^N ¡ 1. In the previous expression *B* is the number base, and *N* is the number of digits. As an example if we wanted to determine the maximum value for a hexadecimal number (base 16) consisting of 6 digits the expression would be as follows: 16^6 ¡ 1. The maximum value of the previous example would be represented as $F\ F\ F\ F\ F\ F_{16}$ which yields 16777215_{10}.

In the following algorithm *numberBase* should be considered restricted to the values of 2, 8, 9, and 16. For this reason in our actual implementation *numberBase* has an enumeration type. The *Base* enumeration type is defined as:

Base = fBinary <- 2*; Octal* <- 8*; Decimal* <- 10*; Hexadecimal* <- 16*g*

The reason we provide the definition of *Base* is to give you an idea how this algorithm can be modelled in a more readable manner rather than using various checks to determine the correct base to use. For our implementation we cast the value of *numberBase* to an integer, as such we extract the value associated with the relevant option in the *Base* enumeration. As an example if we were to cast the option *Octal* to an integer we would get the value 8. In

the algorithm listed below the cast is implicit so we just use the actual argument *numberBase*.

algorithm MaxValue(*numberBase*, *n*)

Pre: *numberBase* is the number system to use, *n* is the number of digits

Post: the maximum value for *numberBase* consisting of *n* digits is computed

return Power(*numberBase; n*) ¡1

end MaxValue

9.5 Factorial of a number

Attaining the factorial of a number is a primitive mathematical operation. Many implementations of the factorial algorithm are recursive as the problem is re-cursive in nature, however here we present an iterative solution. The iterative solution is presented because it too is trivial to implement and doesn't suffer from the use of recursion (for more on recursion see *x*C).

The factorial of 0 and 1 is 0. The aforementioned acts as a base case that we will build upon. The factorial of 2 is 2¤ the factorial of 1, similarly the factorial of 3 is 3¤ the factorial of 2 and so on. We can indicate that we are after the factorial of a number using the form *N*! where *N* is the number we wish to attain the factorial of. Our algorithm doesn't use such notation but it is

handy to know.

```
algorithm Factorial(n)

  Pre: n ¸ 0, n is the number to compute the factorial of

  Post: the factorial of n is computed
  if n < 2

                    return 1

  end if

  f actorial <- 1

  for i <- 2 to n
                    f actorial <- f actorial ¤ i

  end for

  return f actorial

end Factorial
```

9.6 Summary

In this chapter we have presented several numeric algorithms, most of which are simply here because they were fun to design. Perhaps the message that the reader should gain from this chapter is that algorithms can be applied to several domains to make work in that respective domain attainable. Numeric algorithms in particular drive some of the most advanced systems on the planet computing such data as weather forecasts.

Chapter 10

Searching

10.1 Sequential Search

A simple algorithm that search for a specific item inside a list. It operates looping on each element $O(n)$ until a match occurs or the end is reached.

```
algorithm SequentialSearch(list, item)

  Pre:  list !=null
  Post: return index of item if found, otherwise -1

  index <- 0

  while index < list.Count and list[index] != item

                    index <- index + 1

  end while
  if index < list.Count and list[index] = item

                    return index

  end if

  return -1
```

end SequentialSearch

10.2 Probability Search

Probability search is a statistical sequential searching algorithm. In addition to searching for an item, it takes into account its frequency by swapping it with it's predecessor in the list. The algorithm complexity still remains at $O(n)$ but in a non-uniform items search the more frequent items are in the first positions, reducing list scanning time.

Figure 10.1 shows the resulting state of a list after searching for two items, notice how the searched items have had their search probability increased after each search operation respectively.

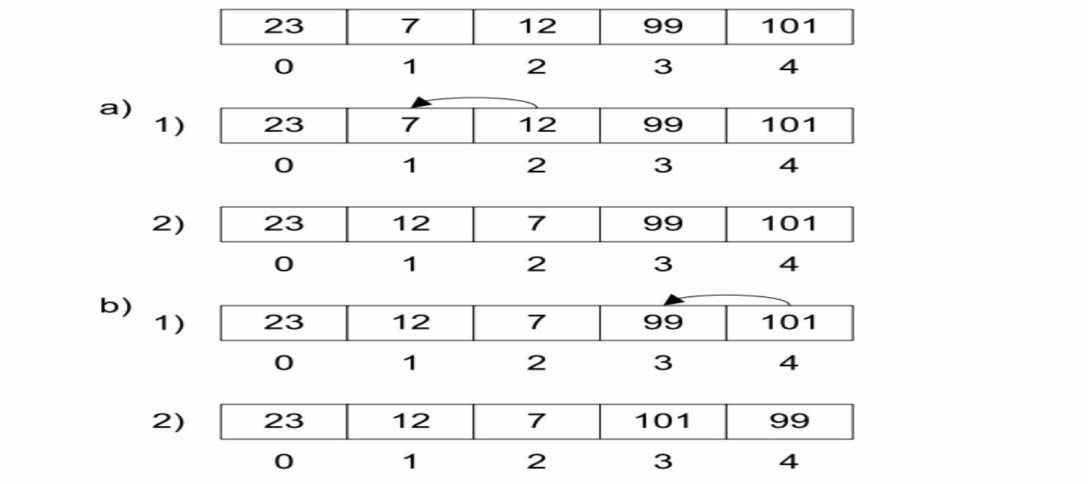


Figure 10.1: a) Search(12), b) Search(101)

algorithm ProbabilitySearch(*list, item*)

Pre: *list* !=null

Post: a boolean indicating where the item is found or not null

in the former case swap founded item with its predecessor

index <- 0

while *index* < *list*.Count and *list*[*index*] != *item*

index <- *index* + 1

end while
if *index* >= *list*.Count or *list*[*index*] != *item*

return false

end if

if *index* > 0

Swap(*list*[*index*]*; list*[*index* - 1])

end if
return true

end ProbabilitySearch

10.3 Binary search

is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the

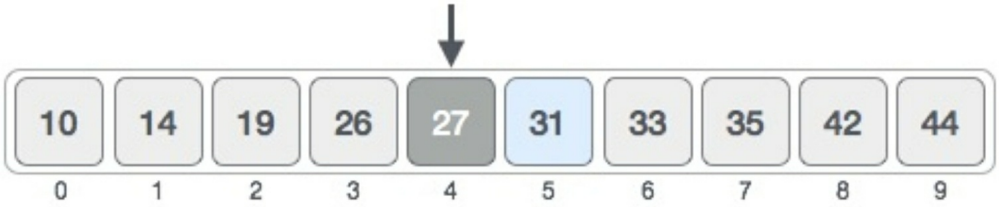
middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

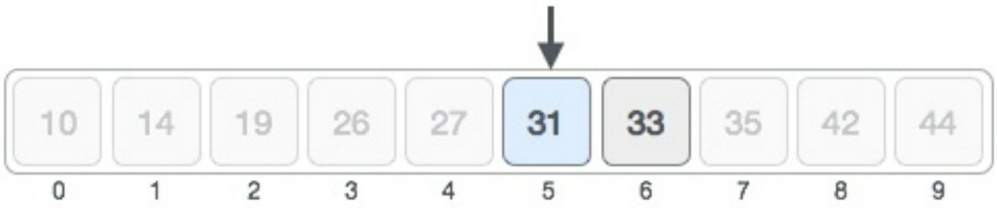
low = mid + 1

mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

```
algorithm binary_search(array,size)
  A ← sorted array
  n ← size of array
  x ← value to be searched
  Set lowerBound = 1
  Set upperBound = n
  while x not found
    if upperBound < lowerBound
      EXIT: x does not exists.
    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
    if A[midPoint] < x
```

```
      set lowerBound = midPoint + 1
    if A[midPoint] > x
      set upperBound = midPoint - 1
    if A[midPoint] = x
      EXIT: x found at location midPoint
  end while
end binary_search
```

10.4 Interpolation search

is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of O(n) whereas binary search has O(log n).

There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored.

Positioning in Binary Search

In binary search, if the desired data is not found then the rest of the list is divided in two parts, lower and higher. The search is carried out in either of them.

Even when the data is sorted, binary search does not take advantage to probe the position of the desired data.

Position Probing in Interpolation Search

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.

If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method −

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where −

A = list

Lo = Lowest index of the list

Hi = Highest index of the list

A[n] = Value stored at index n in the list

If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.

Runtime complexity of interpolation search algorithm is **O(log (log n))** as compared to **O(log n)** of BST in favorable situations.

Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing −

Step 1 − Start searching **data** from middle of the list.
Step 2 − If it is a match, return the index of the item, and exit.
Step 3 − If it is not a match, probe position.
Step 4 − Divide the list using probing formula and find the new midle.
Step 5 − If data is greater than middle, search in higher sub-list.
Step 6 − If data is smaller than middle, search in lower sub-list.
Step 7 − Repeat until match.

Pseudocode

```
A → Array list
N → Size of A
X → Target Value
algorithm Interpolation_Search()
  Set Lo  →  0
  Set Mid → -1
  Set Hi  →  N-1
  While X does not match
    if Lo equals to Hi OR A[Lo] equals to A[Hi]
      EXIT: Failure, Target not found
    end if
    Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])
    if A[Mid] = X
      EXIT: Success, Target found at Mid
    else
      if A[Mid] < X
        Set Lo to Mid+1
      else if A[Mid] > X
        Set Hi to Mid-1
      end if
    end if
  End While
End iterpolationSearch
```

10.5 Summary

In this chapter we have presented a few novel searching algorithms. We have presented more e±cient searching algorithms earlier on, like for instance the logarithmic searching algorithm that AVL and BST tree's use (defined in *x*3.2). We decided not to cover a searching algorithm known as binary chop (another name for binary search, binary chop usually refers to its array counterpart) as the reader has already seen such an algorithm in *x*3.

Searching algorithms and their e±ciency largely depends on the underlying data structure being used to store the data. For instance it is quicker to determine whether an item is in a hash table than it is an array, similarly it is quicker to search a BST than it is a linked list. If you are going to search for data fairly often then we strongly advise that you sit down and research the data structures available to you. In most cases using a list or any other primarily linear data structure is down to lack of knowledge. Model your data and then research the data structures that best fit your scenario.

Chapter 11
Strings

Strings have their own chapter in this text purely because string operations and transformations are incredibly frequent within programs. The algorithms presented are based on problems the authors have come across previously, or were formulated to satisfy curiosity.

11.1 Reversing the order of words in a sentence

Defining algorithms for primitive string operations is simple, e.g. extracting a sub-string of a string, however some algorithms that require more inventiveness can be a little more tricky.

The algorithm presented here does not simply reverse the characters in a string, rather it reverses the order of words within a string. This algorithm works on the principal that words are all delimited by white space, and using a few markers to define where words start and end we can easily reverse them.

```
algorithm ReverseWords(value)

  Pre:  value != null, sb is a string buffer

  Post: the words in value have been reversed
  last <- value.Length - 1

  start <- last

  while last >= 0
```

```
    // skip whitespace
    while start >= 0 and value[start] = whitespace

                        start <- start - 1

end while

last <- start

// march down to the index before the beginning of the word
while start >= 0 and start != whitespace

        start <- start - 1

end while
// append chars from start + 1 to length + 1 to string buffer sb

for i <- start + 1 to last

sb.Append(value[i])

end for

// if this isn't the last word in the string add some whitespace after the word in the buffer

if start > 0

        sb.Append(` ')

end if

last <- start - 1

start <- last

end while
```

```
// check if we have added one too many whitespace to sb

if sb[sb.Length - 1] = whitespace

// cut the whitespace
        sb.Length <- sb.Length -1

end if

return sb

end ReverseWords
```

11.2 Detecting a palindrome

Although not a frequent algorithm that will be applied in real-life scenarios detecting a palindrome is a fun, and as it turns out pretty trivial algorithm to design.

The algorithm that we present has a $O(n)$ run time complexity. Our algorithm uses two pointers at opposite ends of string we are checking is a palindrome or not. These pointers march in towards each other always checking that each character they point to is the same with respect to value. Figure 11.1 shows the *IsPalindrome* algorithm in operation on the string \Was it Eliot's toilet I saw?" If you remove all punctuation, and white space from the aforementioned string you will find that it is a valid palindrome.

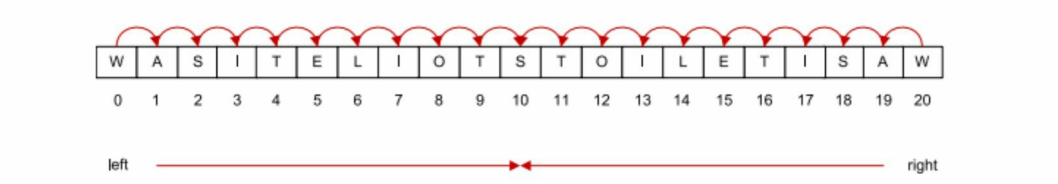


Figure 11.1: *left* and *right* pointers marching in towards one another

```
algorithm IsPalindrome(value)

  Pre:  value !=null

    Post: value is determined to be a palindrome or not
    word <- value.Strip().ToUpperCase()

    left <- 0

    right <- word.Length - 1

    while word[left] = word[right] and left < right
                          left <- left + 1
            right <- right - 1

end while
return word[left] = word[right]

end IsPalindrome
```

In the *IsPalindrome* algorithm we call a method by the name of *Strip*. This algorithm discards punctuation in the string, including white space. As a result *word* contains a heavily compacted representation of the original string, each character of which is in its uppercase representation.

Palindromes discard white space, punctuation, and case making these changes allows us to design a simple algorithm while making our algorithm fairly robust with respect to the palindromes it will detect.

11.3 Counting the number of words in a string

Counting the number of words in a string can seem pretty trivial at first, however there are a few cases that we need to be aware of:

1. tracking when we are in a string

2. updating the word count at the correct place

3. skipping white space that delimits the words

As an example consider the string \Ben ate hay" Clearly this string contains three words, each of which distinguished via white space. All of the previously listed points can be managed by using three variables:

1. *index*

2. *wordCount*

3. *inWord*

Figure 11.2: String with three words

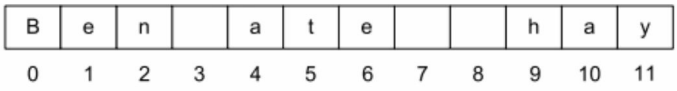


Figure 11.3: String with varying number of white space delimiting the words

Of the previously listed *index* keeps track of the current index we are at in the string, *wordCount* is an integer that keeps track of the number of words we have encountered, and finally *inW ord* is a Boolean °ag that denotes whether or not at the present time we are within a word. If we are not currently hitting white space we are in a word, the opposite is true if at the present index we are hitting white space.

What denotes a word? In our algorithm each word is separated by one or more occurrences of white space. We don't take into account any particular

splitting symbols you may use, e.g. in .NET *String.Split*[1] can take a char (or array of characters) that determines a delimiter to use to split the characters within the string into chunks of strings, resulting in an array of sub-strings.

In Figure 11.2 we present a string indexed as an array. Typically the pattern is the same for most words, delimited by a single occurrence of white space. Figure 11.3 shows the same string, with the same number of words but with varying white space splitting them.

```
algorithm WordCount(value)

  Pre:  value != null

  Post: the number of words contained within value is determined

  inWord <- true

  wordCount <- 0

  index <- 0

  // skip initial white space
  while value[index] = whitespace and index < value.Length - 1

                    index <- index + 1

  end while
  // was the string just whitespace?

  if index = value.Length and value[index] = whitespace

                    return 0

end if
```

```
while index < value.Length
                    value[index] = whitespace
                        // skip all whitespace
                         value[index] = whitespace and index < value.Length -
        1
                              index <- index + 1
    end while
    inWord <- f alse
    wordCount <- wordCount + 1
else
                         inWord <- true
end if
        index <- index + 1
end while
// last word may have not been followed by whitespace
if inWord
        wordCount <- wordCount + 1
end if
        return wordCount
end WordCount
```

11.4 Determining the number of repeated words within a string

With the help of an unordered set, and an algorithm that can split the words within a string using a specified delimiter this algorithm is straightforward to implement. If we split all the words using a single occurrence of white space as our delimiter we get all the words within the string back as elements of an array. Then if we iterate through these words adding them to a set which contains only unique strings we can attain the number of unique words from the string. All that is left to do is subtract the unique word count from the total number of stings contained in the array returned from the split operation. The split operation that we refer to is the same as that mentioned in *x*11.3.

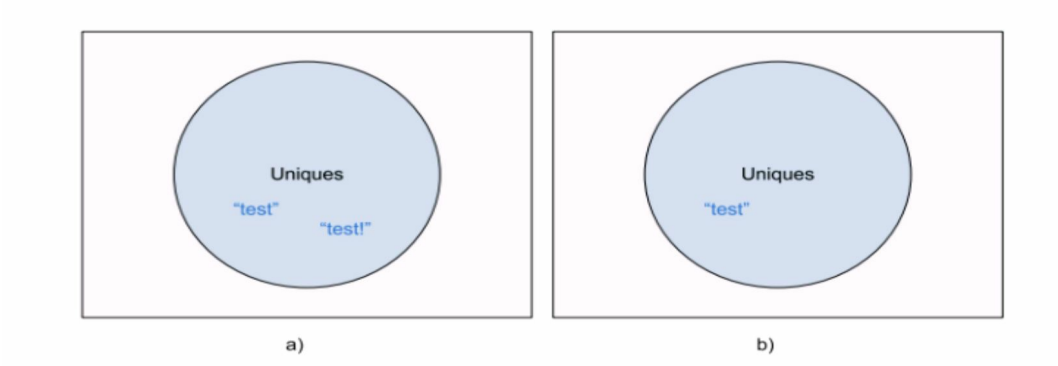


Figure 11.4: a) Undesired *uniques* set; b) desired *uniques* set

algorithm RepeatedWordCount(*value*)

Pre: *value* != *null*

Post: the number of repeated words in *value* is returned

```
words <- value.Split(' ')

uniques <- Set

foreach word in words
                uniques.Add(word.Strip())

end foreach

return words.Length ¡uniques.Count

end RepeatedWordCount
```

1.5 Determining the first matching character between two strings

The algorithm to determine whether any character of a string matches any of the characters in another string is pretty trivial. Put simply, we can parse the strings considered using a double loop and check, discarding punctuation, the equality between any characters thus returning a non-negative index that represents the location of the first character in the match (Figure 11.5); otherwise we return -1 if no match occurs. This approach exhibit a run time complexity of $O(n^2)$.

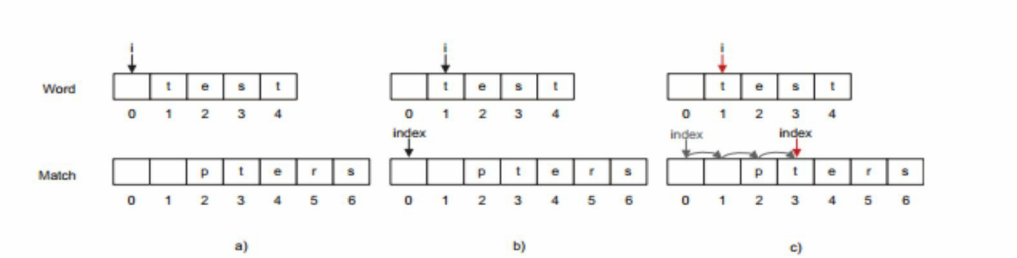


Figure 11.5: a) First Step; b) Second Step c) Match Occurred

algorithm Any(*word*,*match*)

Pre: *word; match* !=null

Post: *index* representing match location if occured, ¡1 otherwise

for *i* <- 0 to *word:Length* - 1

while *word*[*i*] = whitespace
i <- *i* + 1

end while
for *index* <- 0 to *match:Length* - 1

while *match*[*index*] = whitespace
index <- *index* + 1

end while
if *match*[*index*] = *word*[*i*]

return *index*

end if

end for

end for

return - 1

end Any

11.6 Summary

We hope that the reader has seen how fun algorithms on string data types are. Strings are probably the most common data type (and data structure - remember we are dealing with an array) that you will work with so its important that you learn to be creative with them. We for one find strings fascinating. A simple Google search on string nuances between languages and encodings will provide you with a great number of problems. Now that we have spurred you along a little with our introductory algorithms you can devise some of your own.

Chapter 12

Graph

12.1 Introduction

Since many situations and structures give rise to graphs, graph theory becomes an important mathematical tool in a wide variety of subjects, ranging from operations research, computing science and linguistics to chemistry and genetics.

Recently, there has been considerable interest in tree structures arising in the computer science and artificial intelligence. We often organize data in a computer memory store or the flow of information through a system in tree structure form. Indeed, many computer operating systems are designed to be tree structures.

12.2 What is a Graph

Let us consider Figures 4.1 and 4.2 which depict, respectively, part of an electrical network and part of a road map.

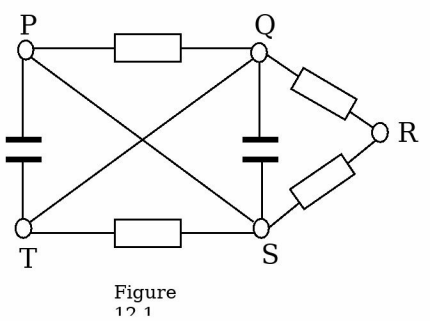


Figure 12.1

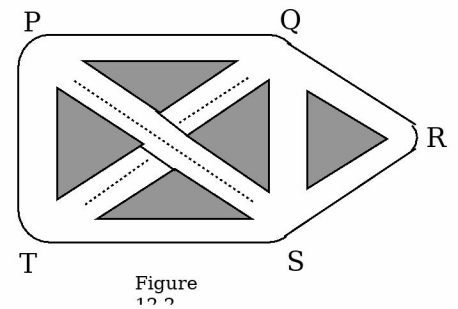
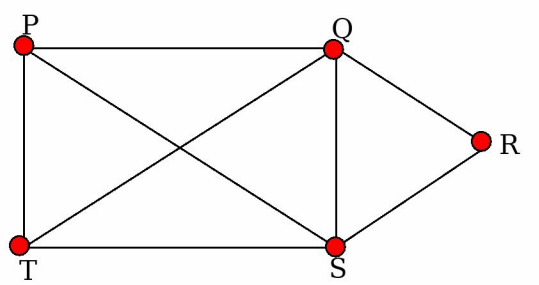


Figure 12.2

It is clear that either of them can be represented diagrammatically by means of points and lines in Figure 4.3. The points P, Q, R, S and T are called **vertices** and the lines are called **edges**; the whole diagram is called a **graph**.



Figure

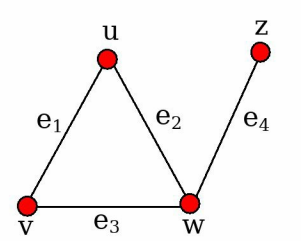
12.3 Definitions

12.3.1 The Definition of a Graph

An **undirected graph** G consists of a non-empty set of elements, called **vertices**, and a set of **edges**, where each edge is associated with a list of **unordered** pairs of either one or two vertices called its **endpoints**. The correspondence from edges to end-points is called the **edge-endpoint function**. The set of vertices of the graph G is called-set of **vertex-set** of G, denoted by **V(G)**, and the set of edges is called the **edge-set** of G, denoted by **E(G)**.

Example 12.1

Figure 12.4 represents the simple undirected graph G



Figure

whose

vertex-set V(G) = {u, v, w, z}

edge-set E(G) = {e_1, e_2, e_3, e_4}

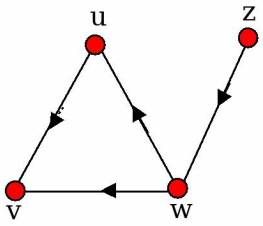
edge-endpoint function

and G = {V(G), E(G)}

= {{u, v, w, z}, {e_1, e_2, e_3, e_4}}

A **directed graph** G consists of vertices, and a set of edges, where each edge is associated with a list of **ordered** pair endpoints.

Example 12.2



Figure

Figure 12.5 represents the directed graph G

e_4 e_2 e_1

whose

vertex-set V(G) = {u, v, w, z}

e_3

edge-set $E(G) = \{e_1, e_2, e_3, e_4\}$

and edge-endpoint function;

Two or more edges joining the same pair of vertices are called **multiple edges** and an edge joining a vertex to itself is called a **loop**. A graph with no loops or multiple edges is called a **simple graph**.

The **degree** of a vertex is the number of edges meeting at a given vertex, and is denoted by **deg** v. Each loop contributes 2 to the degree of the corresponding vertex. The **total degree** of G is the sum of the degrees of all the vertices of G.

Example 12.3

Figure 4.6 illustrates these definitions.

(G) = {u, v, w, z}

$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$

deg u = 6, deg v = 5, deg w = 2, deg z = 1

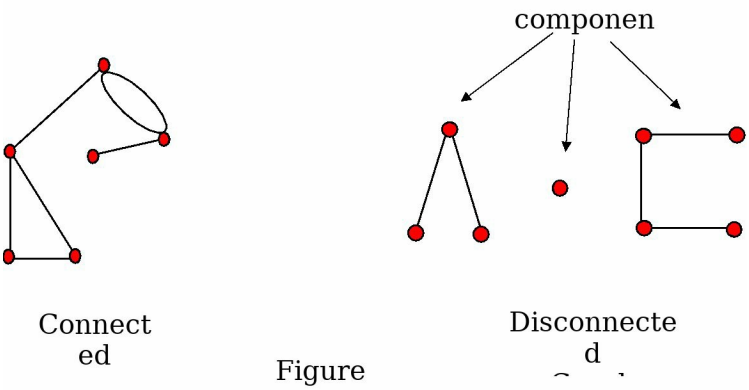
total degree = 6 + 5 + 2 + 1 = 14

edge-endpoint function:

Two vertices v and w of a graph G are said to be **adjacent** if there is an edge joining them; the vertices v and w are then said to be **incident** to such an edge. Similarly, two distinct edges of G are **adjacent** if they have at least one vertex in common.

12.3.2 Connected and Disconnected Graphs

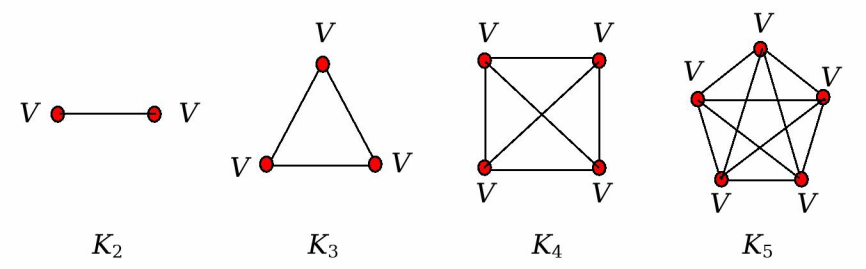
A graph G is **connected** if there is a path in G between any given pair of vertices, and **disconnected** otherwise. Every disconnected graph can be split up into a number of connected subgraphs, called **components**.



Figure

Example 12.4

12.3.3 Complete graph



A complete graph on n vertices , denoted K_n, is a simple graph with n vertices v_1, v_2, v_n whose set of edges contains exactly one edge for each pair of distinct vertices.

Example 12.5

Figure 12.8

Furthermore, the graph K_n is regular of degree n – 1, and has $\frac{1}{2}n(n-1)$ edges.

12.4 The Handshaking Lemma

In any graph, the sum of all the vertex-degrees is equal to twice the number of edges.

As the consequences of the Handshaking Lemma, the sum of all the vertex-degrees is an **even** number and the number of vertices of **odd** degree is **even**.

In Figure 12.6,

the number of edges = 7

the sum of all the vertex-degrees = 6 + 5 + 2 + 1 = 14 (even)

$= 2 \times 7$

= twice the number of edges

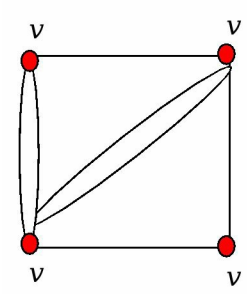
the number of odd degree = 2 (even)

12.5 Matrix Representations of Graphs

A given graph can be specified and stored in matrix format. The adjacency matrix and the incidence matrix are often used in practice.

12.5.1 Matrices and Undirected Graphs

The **Adjacency Matrix A(G)** is the $n \times n$ matrix in which the entry in row i and column j is the number of edges joining the vertices i and j.



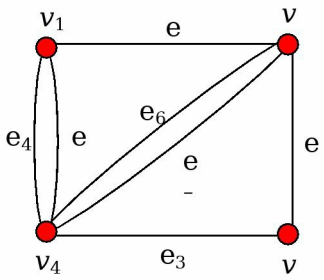
Figure

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 2 \\ 1 & 0 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 2 & 2 & 1 & 0 \end{bmatrix} \end{array}$$

Example 12.6

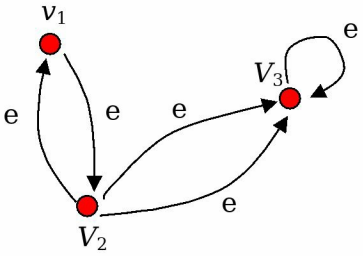
The **Incidence Matrix I(G)** is the n × m matrix in which the entry in row i and column j is 1 if vertex i is incident with edges j, and 0 otherwise.

Example 12.7



Figure

	e_1	e_2	e_3	e_4	e_5	e_6	e_7
v_1	1	0	0	1	1	0	0
v_2	1	1	0	0	0	1	1
v_3	0	1	1	0	0	0	0
v_4	0	0	1	1	1	1	1



Figure

	v_1	v_2	v_3
v_1	0	1	0
v_2	1	0	2
v_3	0	0	1

	e_1	e_2	e_3	e_4	e_5
v_1	1	1	0	0	0
v_2	1	1	1	1	0
v_3	0	0	1	1	2

12.5.2 Matrices and Directed Graphs

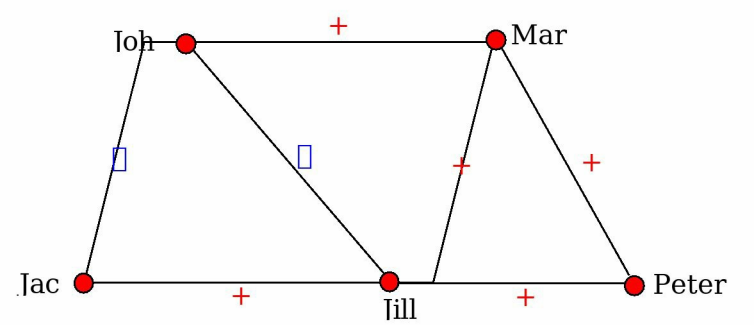
Example 12.8

12.6 Applications of Graphs

12.6.1 Social Sciences

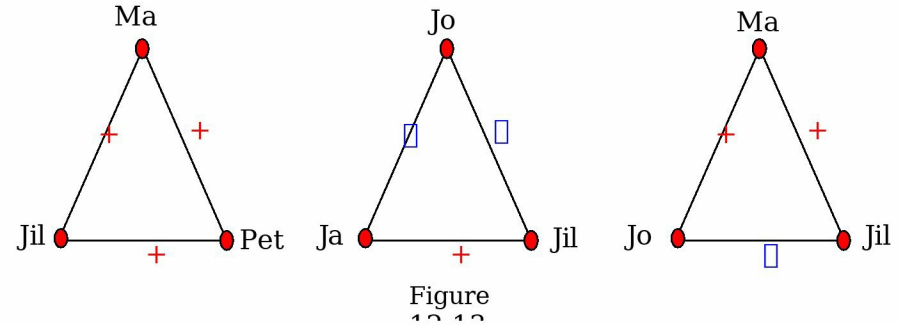
Graphs have been used extensively to represent interpersonal (international) relationships. The vertices correspond to individuals in a group (nations), and the edges join pairs of individuals who are related in some way; such as likes, hates, agrees with, avoids, communicates, etc.(allied, maintain diplomatic relations, agree on a particular strategy, etc)

Consider the following **signed** graph (Figure 4.12), which shows the working relationships with four employees. The graph with either + or − associated with each edge, indicating a positive relationship (likes) or a negative one (dislikes). There are no strong feelings about each other for no connection.



Figure

Now consider the following diagrams, which illustrate some of the situations that can occur when three people work together.

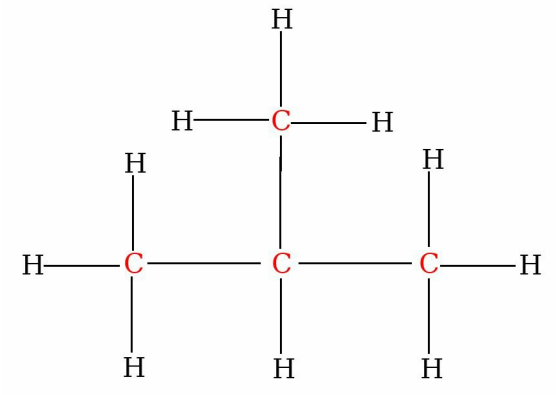


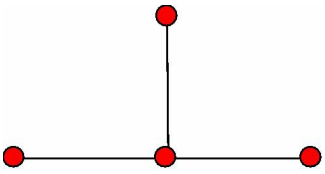
Figure

In the first case, all three get on well. In the second case, Jack and Jill get on well and both dislike John; the result is that John works on his own. In the third case, Mary would like to work with both John and Jill, but Jack and Jill do not wish to work together; in this case, no suitable working arrangement can be found and there is tension.

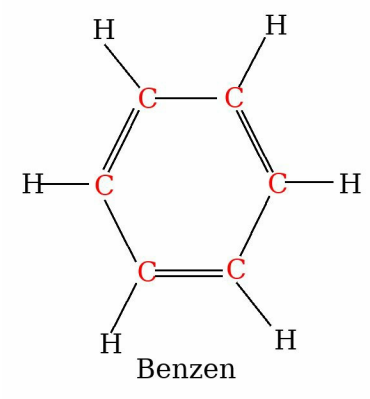
12.6.2 Chemistry

Chemical molecule can be represented as a graph whose vertices correspond to the atoms and whose edges correspond to the chemical bonds connecting them.





2- Carbon-graph



Benzen

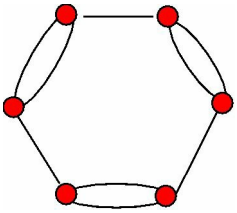


Figure 12.14

Carbon-graph

12.6.3 Circulation diagram

Graphs are used to analyze the movements of people in large buildings. In particular, they have been used in the designing of airports, and in planning the layout of supermarkets.

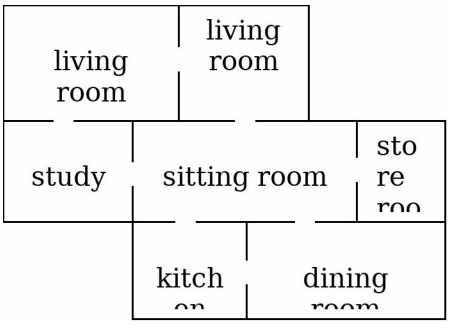


Figure 12.15

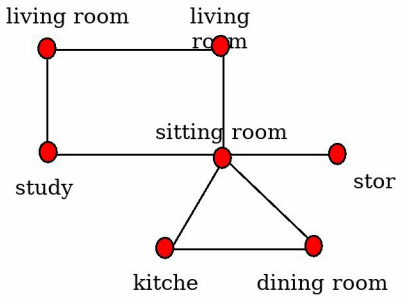


Figure 12.16

12.7 Paths and Circuits

12.7.1 Definitions

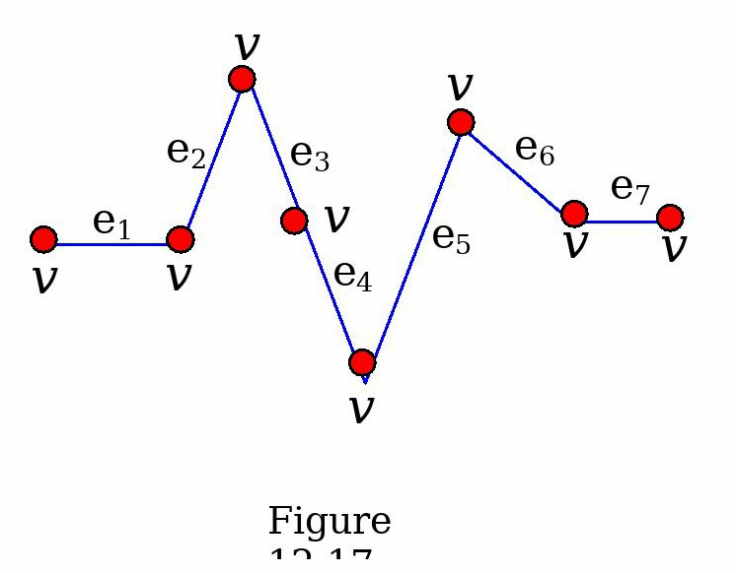


Figure 12.17

A **walk of length** k between r and y in a graph G is a succession of k edges of G of the form $v_1e_1v_2$, $v_2e_2v_3$, $v_3e_3v_4$, ,$v_7e_7v_8$.

We denote this walk by $v_1e_1v_2e_2v_3e_3v_4e_4v_5e_5v_6e_6v_7e_7v_8$.

A **closed walk** is a walk that starts and ends at the same vertex.

For example, in the following graph

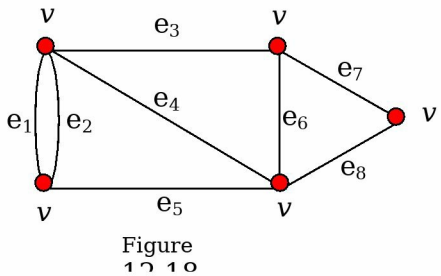
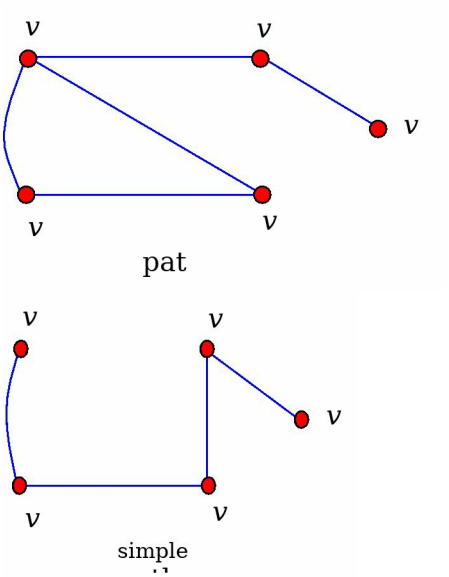


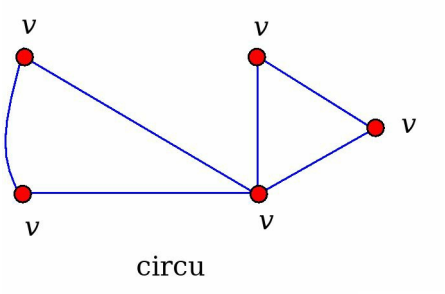
Figure 12.18

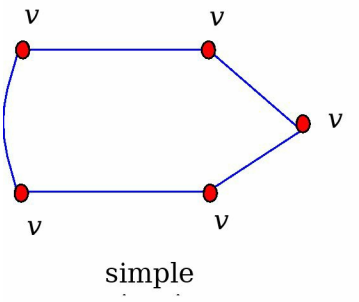
$v_1e_4v_4e_5v_2e_1v_1e_3v_3e_7v_5$ is a walk of length 5 between v_1 and v_5.

If all the edges (but not necessarily all the vertices) of a walk are different, then the walk is called a **path**. If , in addition, all the vertices are different, then the trail is called **simple path**. A closed path is called a **circuit**. A **simple circuit** is a circuit that does not have any other repeated vertex except the first and last.



Example 12.9 In Figure 12.19, $v_1\ v_2\ v_4\ v_1\ v_3\ v_5$ is a path, $v_1\ v_2\ v_4\ v_3\ v_5$ is a simple path, $v_1\ v_2\ v_4\ v_3\ v_5\ v_4\ v_1$ is a circuit and $v_1\ v_2\ v_4\ v_5\ v_3\ v_1$ is a simple circuit.





Figure

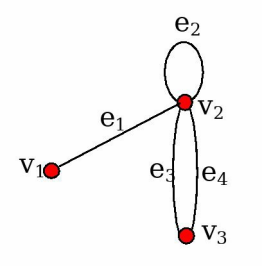
12.7.2 Counting Walks of Length N

Theorem 12.1

If G is a graph with vertices v_1, v_2, v_m and A(G) is the adjacency matrix of G, then for each positive integer n, the *ij*th entry of A^n = the number of walks of length n form v_i to v_j.

Example 12.10

Consider the following graph G.



Figure

The adjacency matrix A(G) is

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

Then

$$A^2 = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 6 & 2 \\ 2 & 2 & 4 \end{bmatrix}$$

Therefore,

one walk of length 2 connecting v_1 to v_1;$(v_1 e_1 v_2 e_1 v_1)$

one walk of length 2 connecting v_1 to v_2; $(v_1 e_1 v_2 e_2 v_2)$

two walks of length 2 connecting v_1 to v_3; $(v_1 e_1 v_2 e_3 v_3, v_1 e_1 v_2 e_4 v_3)$

six walks of length 2 connecting v_2 to v_2;

$(v_2e_1v_1e_1v_2, v_2e_2v_2e_2v_2, v_2e_3v_3e_3v_2, v_2e_3v_3e_4v_2, v_2e_4v_3e_4v_2, v_2e_4v_3e_3v_2)$

two walks of length 2 connecting v_2 to v_3;

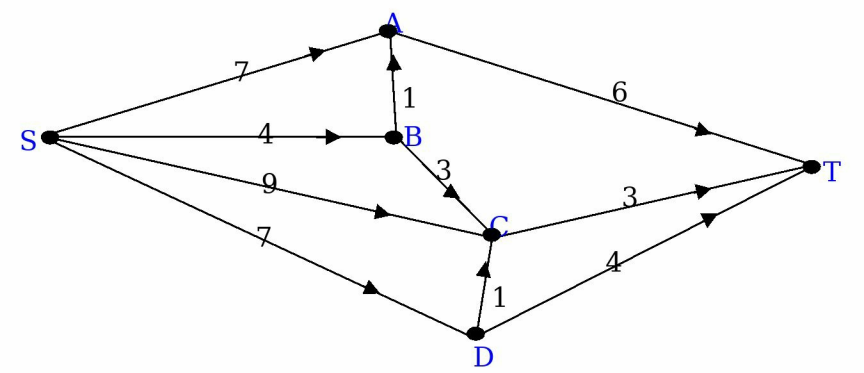
$(v_2e_2v_2e_3v_3, v_2e_2v_2e_4v_3)$

four walks of length 2 connecting v_3 to v_3.

$(v_3e_3v_2e_3v_3, v_3e_3v_2e_4v_3, v_3e_4v_2e_4v_3, v_3e_4v_2v_{3)}$

12.7.3 The Shortest Path Algorithm

The objective of this algorithm is to find the shortest path from vertex S to vertex T in a given network. We demonstrate the steps of the algorithm by finding the shortest distance from S to T in the following network:

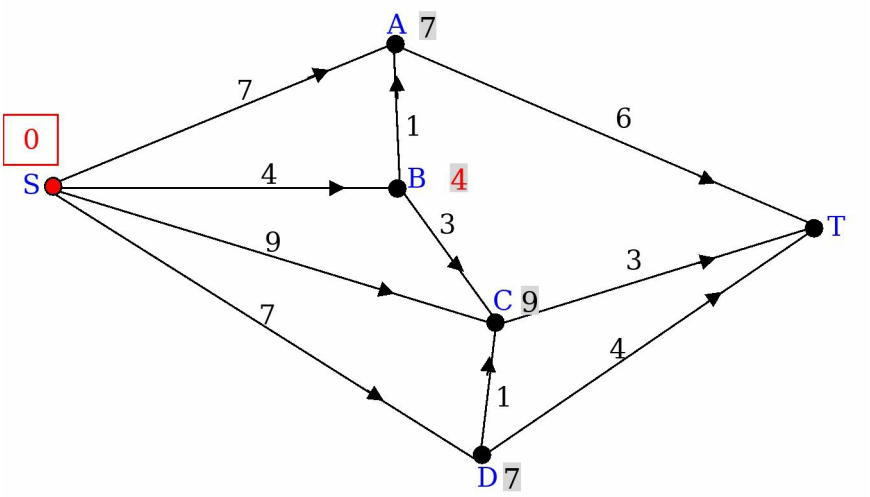


Example 12.11

Figure

Step 1 Initialization

- Assign to vertex S potential 0.
- Label each vertex V reached directly from S with distance from S to V.



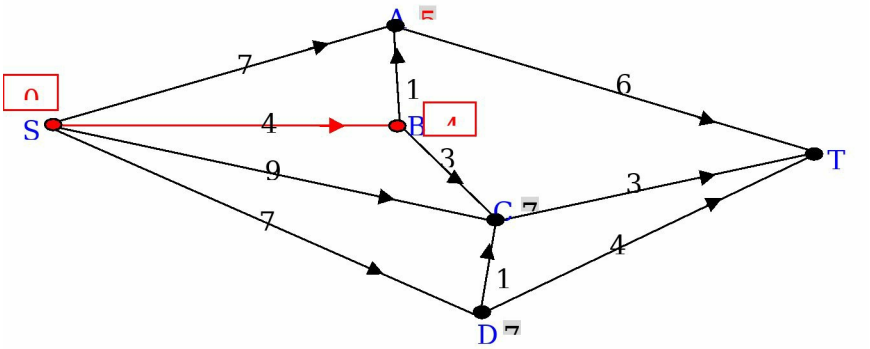
- Choose the smallest of these labels, and make it the potential of the corresponding vertex or vertices.

Figure

Step 2 General Step

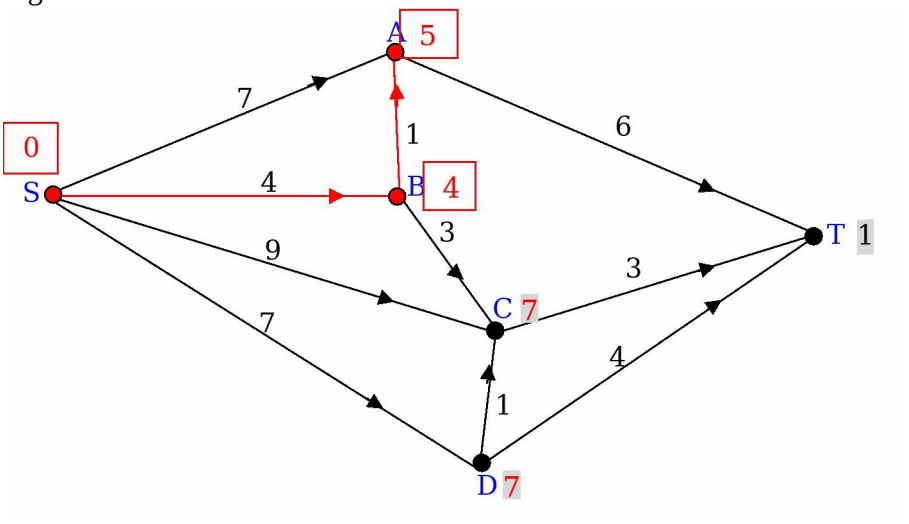
Consider the vertex or vertices just assigned a potential;
for each vertex V, look at each vertex W reached directly from V and assign W the label (potential of V) + (distance from V to W). unless W already has a smaller label.

Choose the smallest of these labels, and make it the potential of the corresponding vertex or vertices.



Repeat the general step with the new potential.

Figure 22.b



Figure

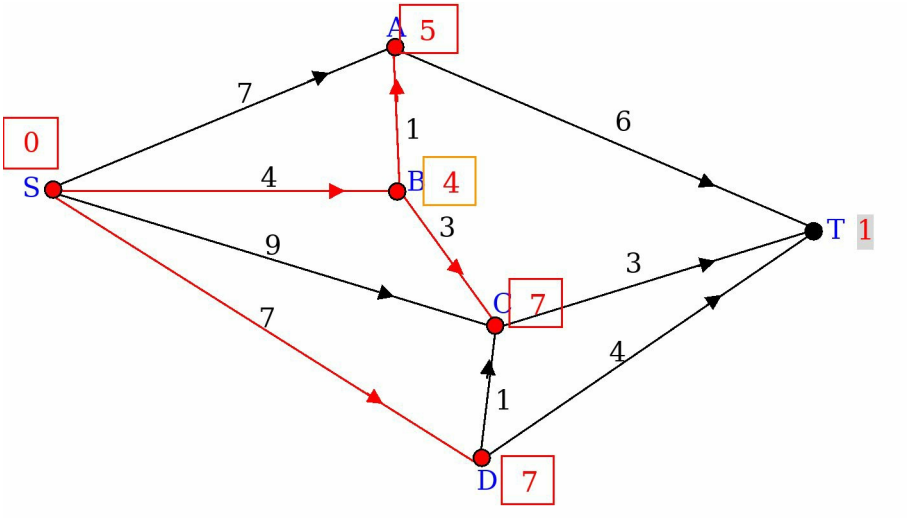
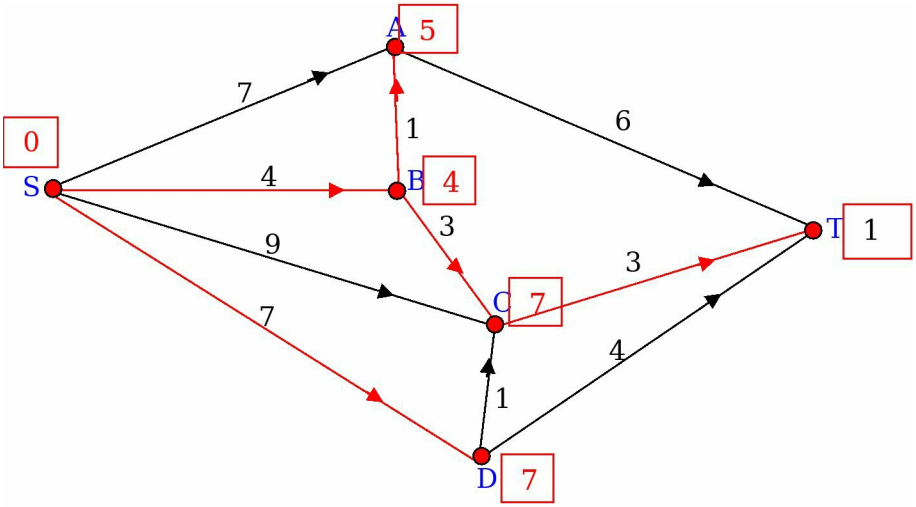


Figure12.2

STEP 3 Stop



When vertex T has been assigned a potential; this is the shortest distance from S to T.

Figure

STEP 4 Conclusion

Therefore, the shortest path from S to T is SBCT with path length 10.

12.7.4 Kruskals Minimum Spanning Tree (MST)

What is Minimum Spanning Tree?

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?

A minimum spanning tree has (V 1) edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?

Below are the steps for finding MST using Kruskals algorithm

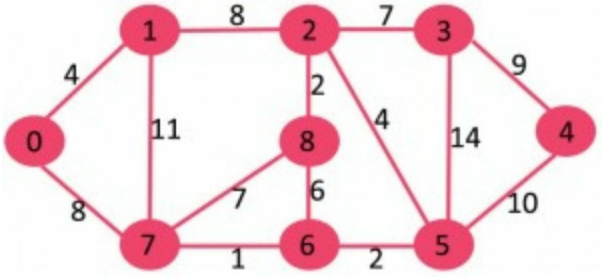
Algorithm

1. *Sort all the edges in non-decreasing order of their weight.*

2. *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*

3. *Repeat step#2 until there are (V-1) edges in the spanning tree.*

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree

formed will be having (9 1) = 8 edges.

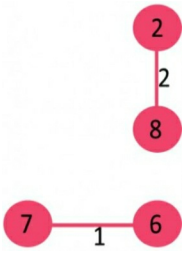
After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

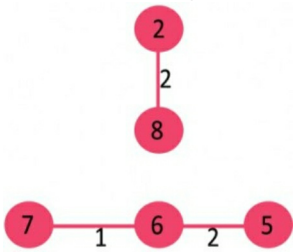
Now pick all edges one by one from sorted list of edges

1. *Pick edge 7-6:* No cycle is formed, include it.

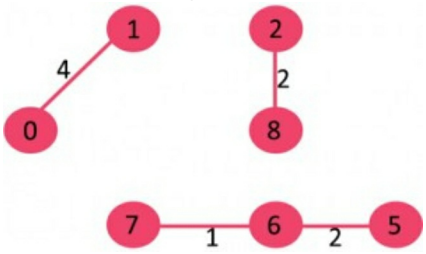
2. *Pick edge 8-2:* No cycle is formed, include it.



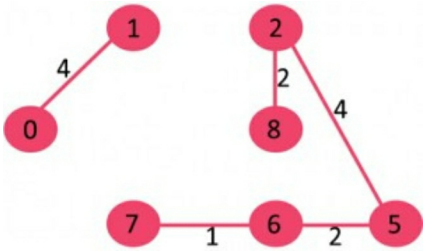
3. *Pick edge 6-5:* No cycle is formed, include it.



4. *Pick edge 0-1:* No cycle is formed, include it.

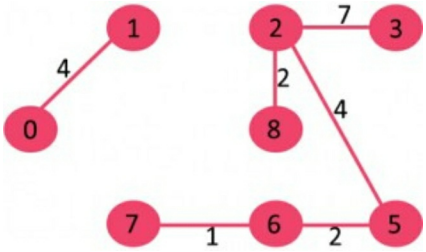


5. *Pick edge 2-5:* No cycle is formed, include it.



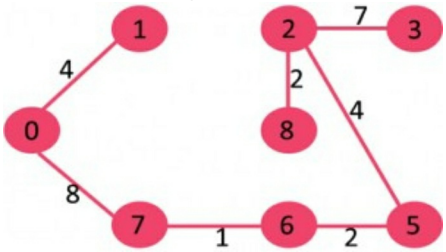
6. *Pick edge 8-6:* Since including this edge results in cycle, discard it.

7. *Pick edge 2-3:* No cycle is formed, include it.



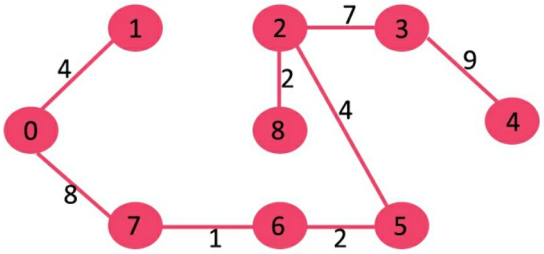
8. *Pick edge 7-8:* Since including this edge results in cycle, discard it.

9. *Pick edge 0-7:* No cycle is formed, include it.



10. *Pick edge 1-2:* Since including this edge results in cycle, discard it.

11. *Pick edge 3-4:* No cycle is formed, include it.



Since the number of edges included equals (V 1), the algorithm stops here.

12.7.5 Prims Minimum Spanning Tree (MST)

Like Kruskals algorithm, Prims algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph theory. *So, at every step of Prims algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).*

How does Prims Algorithm Work? The idea behind Prims algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning* Tree. And they must be connected with the minimum weight edge to make it a *Minimum* Spanning Tree.

Algorithm

1) Create a set *mstSet* that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

3) While mstSet doesnt include all vertices

.**a)** Pick a vertex *u* which is not there in *mstSet* and has minimum key value.

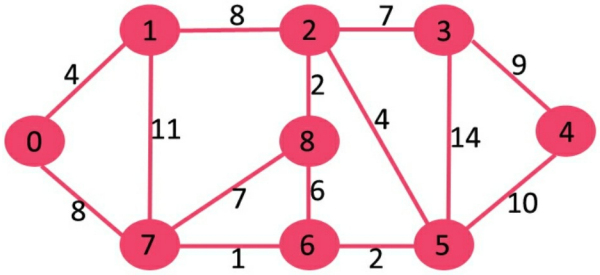
.**b)** Include *u* to mstSet.

.**c)** Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of

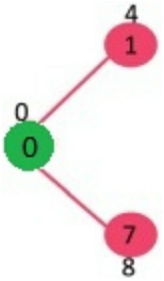
edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

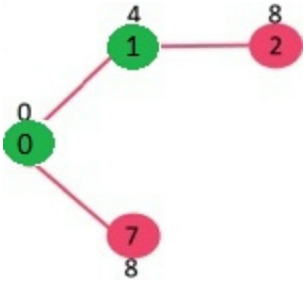
Let us understand with the following example:



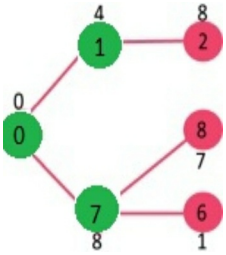
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



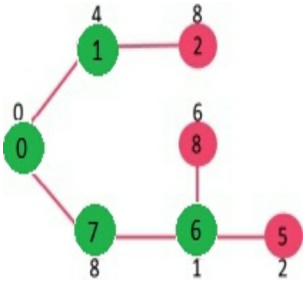
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



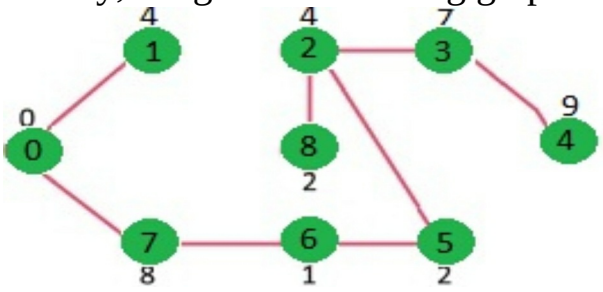
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prims algorithm can be reduced to O(E log V) with the help of binary heap.

12.8 Eulerian Graphs

12.8.1 Definition

A connected graph G is Eulerian if there is a closed path which includes every edge of G; such path is called an Eulerian path.

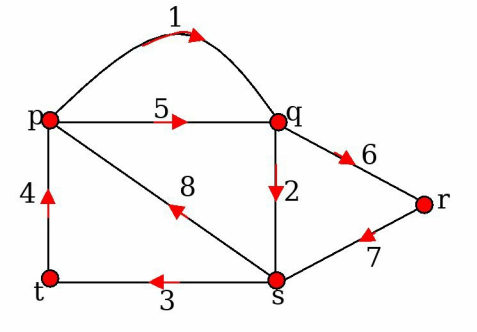


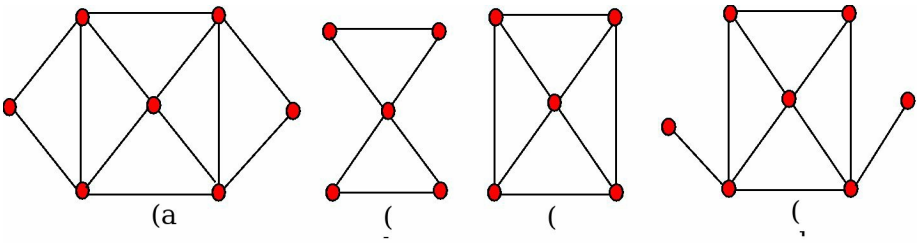
Figure 12.34 Eulerian

12.8.2 Theorem 12.2

Let G be a connected graph. Then G is Eulerian if and only if every vertex of G has even degree.

For example, since K_4 graph has odd degree vertices (deg 3), K_4 graph is not Eulerian. However all the vertices of K_5 are even (deg 4). Therefore by theorem 12.2, it is Eulerian.

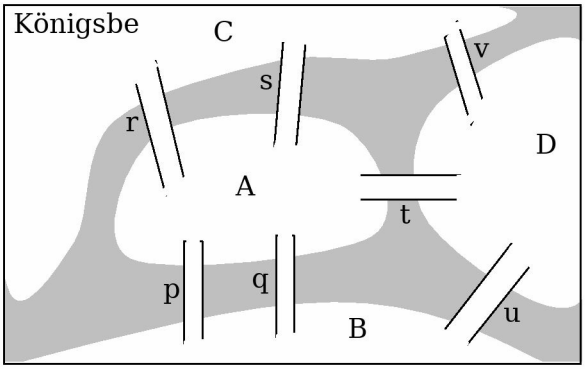
Example 12.12
Consider the following four graphs



Figure

Which graph(s) is/are Eulerian?
Answer: Graph (a) and (b) are Eulerian.

12.8.3 Königsberg bridges problem

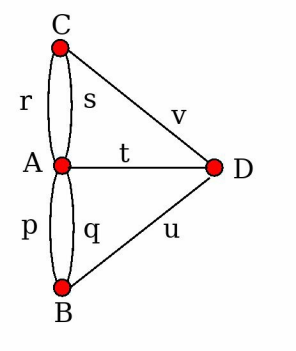


The four parts (A, B, C and D) of the city Königsberg were interconnected by seven bridges (p, q, r, s, t, u and v) as shown in the following diagram. Is it possible to find a route crossing each bridge exactly once ?

Figure 12.36
Königsberg map

We can express the Königsberg bridges problem in terms of a graph by taking the four land areas as vertices and the seven

bridges as edges joining the corresponding pairs of vertices.



Figure

By theorem 12.2, it is not a Eulerian graph. It follows that there is no route of the desired kind crossing the seven bridges of Königsberg.

12.8.4 Fleurys Algorithm

If G is an Eulerian graph, then the following steps can always be carried out, and produce an Eulerian path in G:

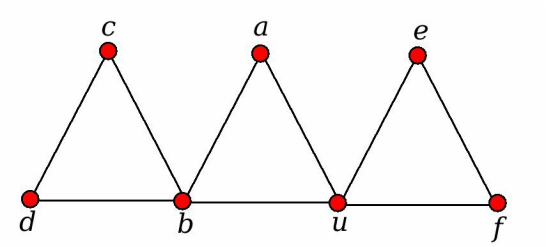
STEP 1 Choose a starting vertex u.

STEP 2 At each stage, traverse any available edge, choosing a bridge only if there is no alternative.

STEP 3 After traversing each edge, erase it (erasing any vertices of degree 0 which result), and then choose another available edge.

STEP 4 STOP when there are no more edges.

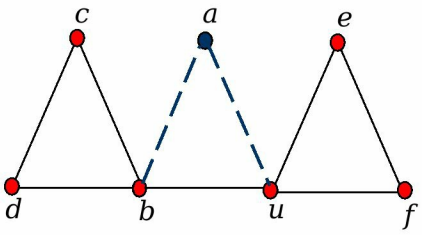
Example 12.13



Find the Eulerian path of the following graph.

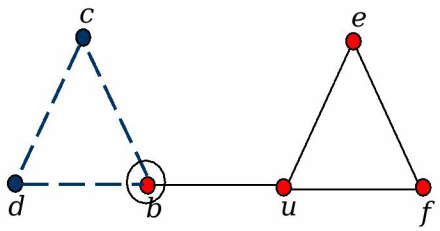
Figure

Starting at u, we may choose the edge *ua,* followed by *ab*. Erasing these edges and the vertex a give us graph (b).



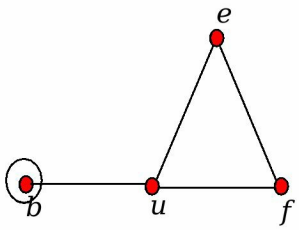
(a

We cannot use the edge *bu* since it is a bridge, so we choose the edge *bc*, followed by *cd* and *db*. Erasing these edges and the vertices *c* and *d*



(b

We have to traverse the bridge *bu*. Traversing the cycle *uefu* completes the Eulerian path. The path is therefore *uabcdbuefu*.



(c

Figure

12.9 Hamiltonian Graphs

12.9.1 Definition

A connected graph G is **Hamiltonian** if there is a cycle which includes every vertex of G ; such a cycle is called a **Hamiltonian** cycle.

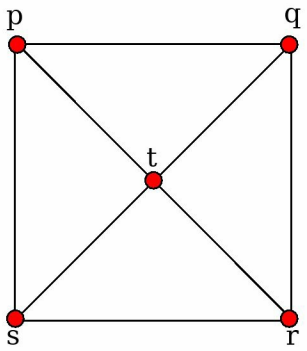


Figure 12.40
Hamiltonian

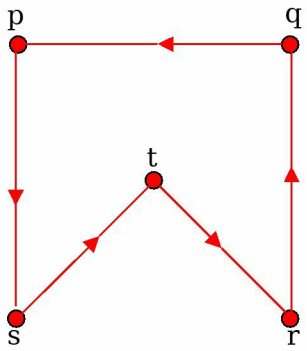
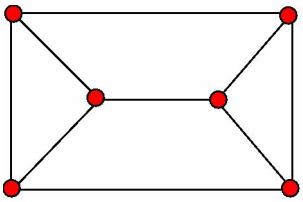


Figure 12.41
Hamiltonian

12.9.2 Theorem 12.3 (DIRACS THEOREM)



Let G be a simple graph with n vertices, where n ≥ 3. If deg $v \geq \frac{1}{2}$ n for each vertex v, then G is Hamiltonian.

Example 12.14

Figure

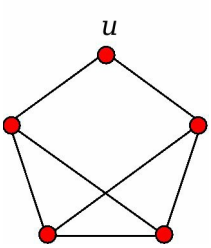
For the above graph, n = 6 and deg v = 3 for each vertex v, so this graph is Hamitonian by Diracs theorem.

12.9.3 Theorem 12.4 (ORES THEOREM)

Let G be a simple graph with n vertices, where n ≥ 3.

deg v + deg $w \geq$ n,

for each pair of non-adjacent vertices v and w, then G is Hamiltonian.



Example 12.15

Figure

For figure 4.43, n = 5 but deg u = 2, so Diracs theorem does not apply. However, deg v + deg $w \geq$ 5 for all pairs of non-adjacent vertices v and w, so this graph is Hamiltonian by Ores theorem.

Summary

A **directed graph** G consists of vertices, and a set of edges, where each edge is associated with a list of **ordered** pair endpoints.

A **undirected graph** G consists of vertices, and a set of edges, where each edge is associated with a list of **unordered** pair endpoints.

A graph with no loops or multiple edges is called a **simple graph**.

The **degree** of a vertex is the number of edges meeting at a given vertex,

For a complete graph K_n : it is regular of degree n − 1, and has $\frac{1}{2}n(n-1)$ edges.

Handshaking Lemma, the sum of all the vertex-degrees is an **even** number and the number of vertices of **odd** degree is **even**.

A **closed walk** is a walk that starts and ends at the same vertex.

If all the edges (but not necessarily all the vertices) of a walk are different, then the walk is called a **path**. If , in addition, all the vertices are different, then the trail is called **simple path**. A closed path is called a **circuit**. A **simple circuit** is a circuit that does not have any other repeated vertex except the first and last.

The Shortest Path Algorithm: initialization, general, stop

A connected graph G is Eulerian if there is a closed path which includes every edge of G

Then G is Eulerian if and only if every vertex of G has even degree.

A connected graph G is **Hamiltonian** if there is a cycle which includes every vertex of G ; such a cycle is called a **Hamiltonian** cycle.

DIRACS THEOREM)

Let G be a simple graph with n vertices, where $n \geq 3$. If $\deg v \geq \frac{1}{2} n$ for each vertex v, then G is Hamiltonian.

ORES THEOREM)

Let G be a simple graph with n vertices, where $n \geq 3$.

$\deg v + \deg w \geq n$,

for each pair of non-adjacent vertices v and w, then G is

Hamiltonian.

A tree is a connected graph which contains no cycles

Chapter 13

Application of Data Structures

Computer has to process lots and lots of data. To systematically process those data e ciently, those data are organized as a whole, appropri-ate for the application, called a data structure.

The important issue is that di erent kinds of data structures are suited to di erent kinds of applications, and some are highly specialized to specific tasks.

An example

When we talked about algorithms, we looked at the example of 555-1212, where the phone records are organized as tables, i.e., a list of phone records.

We also saw that if we use a table where the phone records are not alphabetically sorted, only the sequential search algorithm can be ap-plied. On the other hand, if those records are sorted, then the much faster binary search ap-proach can be followed.

One of the two

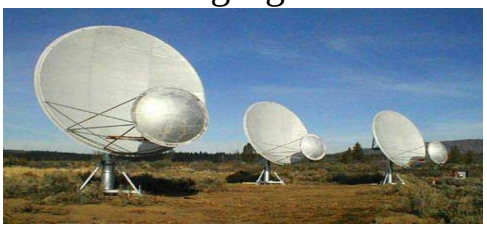
It is long recognized that there is a close re-lationship between algorithms and data struc-ture, indeed, as stated in Worths famous book,

Algorithms + Data Structures = Programs

Hence, data structures are used in every non-trivial program or software system. In many cases, an e cient algorithm is based on a clev-erly designed data structure. An appropriate data structure is also the key to manage huge amounts of data, such as large databases, the bloodline of todays information age.
We now look at several simple, but often used, data structures.

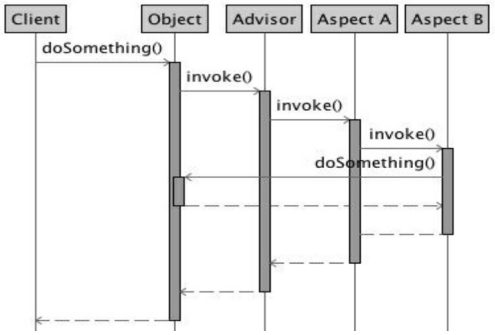
Array

As we already discussed, the backbone of the 555-1212 system is a table of phone records. Technically, we call such a data structure as an array of data. Array is perhaps one of the most used data structure.
This data structure is also used in our daily life.
The following figure shows an array of tele-scopes.



Stack

We once discussed in the topic of Analytical Engine about the concept of sub-routine, a group of steps that can be used as a whole.
For example, when we want to doSomething, we will use the Advisor, which uses Aspect A, then Aspect B, which again doSome-thing before it completes.
When Aspect B completes, we continue run Aspect A, then return to the Advisor, fi-nally come back to doSomething until it is over.



Question: **What is buried here?**

Answer: The issue here is that we take o from doSomething first, but will come back to this as the last step.
Similarly, we come to the Aspect B as the last step of the whole sequence, but finishes it o as the first one. When we use a data struc-ture to manage such a sequence of FirstOut-LastIn or LastInFirstOut, we will use the Stack data structure.
Stack is actually a restricted version of a list, but when we either add in something into a Stack, or remove something from it, we always do it in only one of the two ends.

A general list represents a sequence where we can either add something in, or take something out, anywhere.
Stack is also used in our daily life.
For example, when we want to take a dish out of a deck of plates, as shown in the following figure, which one do you take?

Although you can take out any one, I bet you must take the one at the top. On the other hand, when we want to put a dish back to such a deck, where do you put it back to? Again at the top.

Thus, such a deck of dish is really a stack.

Queue

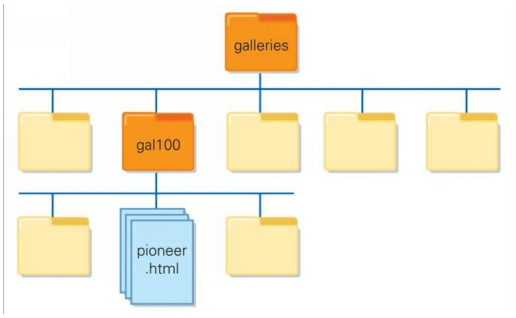
Queue is another very simple but useful data structure. In a computer lab when one printer is shared by all the computers in that lab, what happens if five people want to print out their papers? The answer is very simple: they take in turns. What happens is that those papers will be put into a queue according to the order they arrive, then wait there until it comes to the front.

Queue is also a restricted list where we add things in one end, and remove from the other. It is actually called FirstInFirstOut or LastIn-LastOut list.
Not just here again...
Queue is certainly also used quite a bit in our daily life.

For example, in a civilized society, we would wait in a line for our turns, but not cut in as some of the students did when buying tickets for the Drake show in last years Spring Fling weekend.

Tree

Array, stack and queue are examples of linear data structures, where everything is lined up. Tree is an example of a more complicated data structure.
Indeed, everything inside the computer is orga-nized as a tree. For example, the following fig-ure shows the composition of a web site: The folder galleries contains a few folders, one of which is called gal100, which contains a few folders and a few documents, one being pioneer.html.

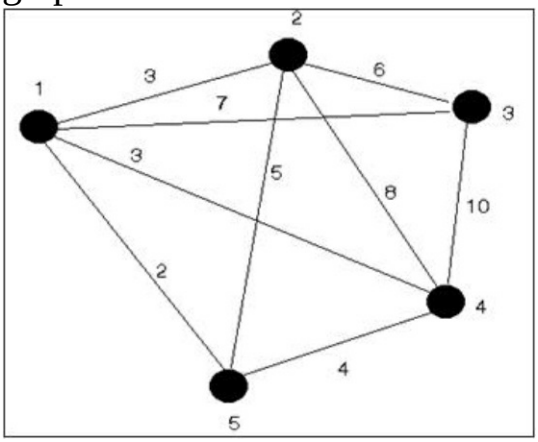


A key property of using the tree structure to organize such data is that we can provide a unique path for the computer to find the pioneer.html file.

Tree structure is also widely used in our life. For example, the following figure shows the relationship among persons in a family.

Graph

Graph is the most complicated data structure. The following shows a simple graph that con-nects five vertices with nice edges.



In the above graph, we represent the associa-tion between various vertices as well as some information attached to such association.
Internet is also organized as a graph.

Topological sort

The following is an algorithm looking for a topological sorting in G, where we label all the vertices to come up with an order in which these courses will

be taken:

1. Set n to 1

2. Wile not all vertices labeled yet

3. Find v, s.t. there is no edge coming into v;

4. Label v with n;

5. n<-n+1
   Retract v together with all the edges outgoing from v;

Lets apply this algorithm to the above graph.

Minimum spanning tree

Lets assume that we want to hook up a collection of sites, houses, buildings on a campus, with Internet lines, each connection comes at di erent cost. How could we do it with mini-mum cost.

Appendix A

Algorithm Walkthrough

Learning how to design good algorithms can be assisted greatly by using a structured approach to tracing its behaviour. In most cases tracing an algorithm only requires a single table. In most cases tracing is not enough,

you will also want to use a diagram of the data structure your algorithm operates on. This diagram will be used to visualise the problem more effectively. Seeing things visually can help you understand the problem quicker, and better.

The trace table will store information about the variables used in your algorithm. The values within this table are constantly updated when the algorithm mutates them. Such an approach allows you to attain a history of the various values each variable has held. You may also be able to infer patterns from the values each variable has contained so that you can make your algorithm more e±cient.

We have found this approach both simple, and powerful. By combining a visual representation of the problem as well as having a history of past values generated by the algorithm it can make understanding, and solving problems much easier.

In this chapter we will show you how to work through both iterative, and recursive algorithms using the technique outlined.

A.1 Iterative algorithms

We will trace the *IsPalindrome* algorithm (defined in *x*11.2) as our example iterative walkthrough. Before we even look at the variables the algorithm uses, first we will look at the actual data structure the algorithm operates on. It should be pretty obvious that we are operating on a string, but how is this represented? A string is essentially a block of contiguous memory that consists of some char data types, one after the other. Each character in the string can be accessed via an index much like you would do when accessing items within an array. The picture should be presenting itself - a string can be thought of as an array of characters.

For our example we will use *IsPalindrome* to operate on the string \Never odd or even" Now we know how the string data structure is represented, and the value of the string we will operate on let's go ahead and draw it as shown in Figure A.1.

Figure A.1: Visualising the data structure we are operating on

value	*word*	*left*	*right*

Table A.1: A column for each variable we wish to track

The *IsPalindrome* algorithm uses the following list of variables in some form throughout its execution:

value

word

left

right

Having identified the values of the variables we need to keep track of we simply create a column for each in a table as shown in Table A.1.

Now, using the *IsPalindrome* algorithm execute each statement updating the variable values in the table appropriately. Table A.2 shows the final table values for each variable used in *IsPalindrome* respectively.

While this approach may look a little bloated in print, on paper it is much more compact. Where we have the strings in the table you should annotate these strings with array indexes to aid the algorithm walkthrough.

There is one other point that we should clarify at this time - whether to include variables that change only a few times, or not at all in the trace table. In Table A.2 we have included both the *value*, and *word* variables because it was convenient to do so. You may find that you want to promote these values to a larger diagram (like that in Figure A.1) and only use the trace table for variables whose values change during the algorithm. We recommend that you promote the core data structure being operated on to a larger diagram outside of the table so that you can interrogate it more easily.

value	*word*	*left*	*right*
\Never odd or even"	\NEVERODDOREVEN"	0	13
		1	12
		2	11
		3	10
		4	9
		5	8
		6	7
		7	6

Table A.2: Algorithm trace for *IsPalindrome*

We cannot stress enough how important such traces are when designing your algorithm. You can use these trace tables to verify algorithm correctness. At the cost of a simple table, and quick sketch of the data structure you are operating on you can devise correct algorithms quicker. Visualising the problem domain and keeping track of changing data makes problems a lot easier to solve. Moreover you always have a point of reference which you can look back on.

A.2 Recursive Algorithms

For the most part working through recursive algorithms is as simple as

walking through an iterative algorithm. One of the things that we need to keep track of though is which method call returns to who. Most recursive algorithms are much simple to follow when you draw out the recursive calls rather than using a table based approach. In this section we will use a recursive implementation of an algorithm that computes a number from the Fiboncacci sequence.

algorithm Fibonacci(*n*)

Pre: *n* is the number in the fibonacci sequence to compute

Post: the fibonacci sequence number *n* has been computed

if *n* < 1

return 0

else if *n* < 2

return 1

end if

return Fibonacci(*n* ¡ 1) + Fibonacci(*n* ¡ 2)

end Fibonacci

Before we jump into showing you a diagrammtic representation of the algorithm calls for the *Fibonacci* algorithm we will brie°y talk about the cases of the algorithm. The algorithm has three cases in total:

n < 1

n < 2

n ¸ 2

The first two items in the preceeding list are the base cases of the algorithm. Until we hit one of our base cases in our recursive method call tree we won't return anything. The third item from the list is our recursive case.

With each call to the recursive case we etch ever closer to one of our base cases. Figure A.2 shows a diagrammtic representation of the recursive call chain. In Figure A.2 the order in which the methods are called are labelled. Figure A.3 shows the call chain annotated with the return values of each method call as well as the order in which methods return to their callers. In Figure A.3 the return values are represented as annotations to the red arrows.

It is important to note that each recursive call only ever returns to its caller upon hitting one of the two base cases. When you do eventually hit a base case that branch of recursive calls ceases. Upon hitting a base case you go back to

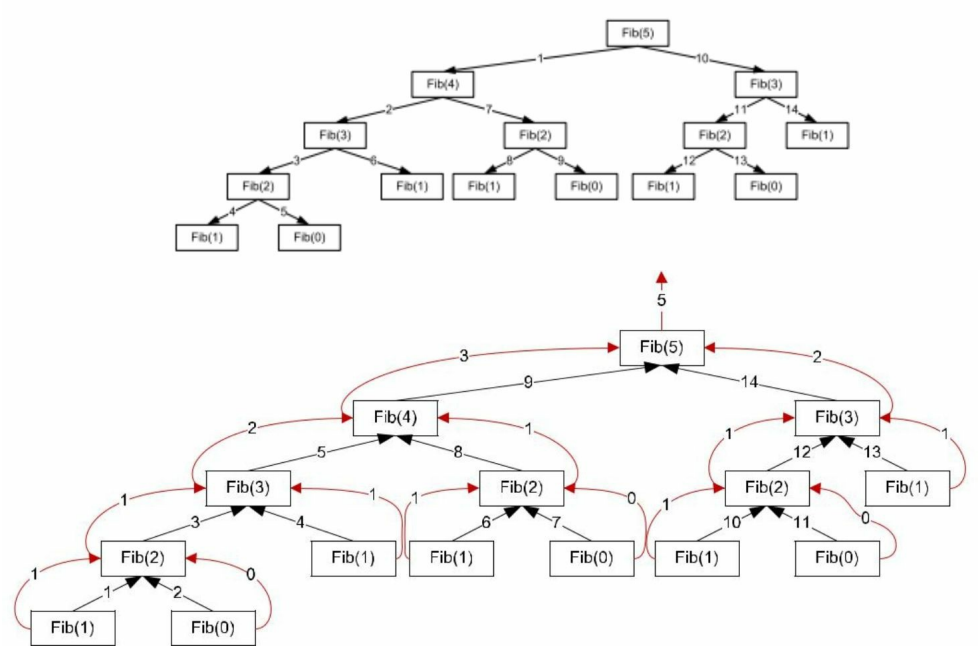


Figure A.2: Call chain for *Fibonacci* algorithm

Figure A.3: Return chain for *Fibonacci* algorithm

the caller and continue execution of that method. Execution in the caller is contiued at the next statement, or expression after the recursive call was made.

In the *Fibonacci* algorithms' recursive case we make two recursive calls. When the first recursive call (Fibonacci(n ¡ 1)) returns to the caller we then execute the the second recursive call (Fibonacci(n ¡ 2)). After both recursive calls have returned to their caller, the caller can then subesequently return to its caller and so on.

Recursive algorithms are much easier to demonstrate diagrammatically as Figure A.2 demonstrates. When you come across a recursive algorithm draw method call diagrams to understand how the algorithm works at a high level.

A.3 Summary

Understanding algorithms can be hard at times, particularly from an implemen-tation perspective. In order to understand an algorithm try and work through it using trace tables. In cases where the algorithm is also

recursive sketch the recursive calls out so you can visualise the call/return chain.

In the vast majority of cases implementing an algorithm is simple provided that you know how the algorithm works. Mastering how an algorithm works from a high level is key for devising a well designed solution to the problem in hand.

Appendix B

Translation Walkthrough

The conversion from pseudo to an actual imperative language is usually very straight forward, to clarify an example is provided. In this example we will convert the algorithm in *x*9.1 to the C language.

```
public static bool IsPrime(int number)
{
    if (number < 2)
    {
      return false;
    }
    int innerLoopBound = (int)Math.Floor(Math.Sqrt(number));
    for (int i = 1; i < number; i++)
    {
      for(int j = 1; j <= innerLoopBound; j++)
      {
        if (i % j == number)
```

```
            {
              return false;
            }
          }
        }
        return true;
    }
```

For the most part the conversion is a straight forward process, however you may have to inject various calls to other utility algorithms to ascertain the correct result.

A consideration to take note of is that many algorithms have fairly strict preconditions, of which there may be several - in these scenarios you will need to inject the correct code to handle such situations to preserve the correctness of the algorithm. Most of the preconditions can be suitably handled by throwing the correct exception.

B.1 Summary

As you can see from the example used in this chapter we have tried to make the translation of our pseudo code algorithms to mainstream imperative languages as simple as possible.

Whenever you encounter a keyword within our pseudo code examples that you are unfamiliar with just browse to Appendix E which descirbes each key-

word.

Appendix C

Recursive Vs. Iterative Solutions

One of the most succinct properties of modern programming languages like C++, C#, and Java (as well as many others) is that these languages allow you to define methods that reference themselves, such methods are said to be recursive. One of the biggest advantages recursive methods bring to the table is that they usually result in more readable, and compact solutions to problems.

A recursive method then is one that is defined in terms of itself. Generally a recursive algorithms has two main properties:

One or more base cases; and

A recursive case

For now we will brie°y cover these two aspects of recursive algorithms. With each recursive call we should be making progress to our base case otherwise we are going to run into trouble. The trouble we speak of manifests itself typically as a stack over°ow, we will describe why later.

Now that we have brie°y described what a recursive algorithm is and why you might want to use such an approach for your algorithms we will now talk about iterative solutions. An iterative solution uses no recursion whatsoever. An iterative solution relies only on the use of loops (e.g. for, while, do-while, etc). The down side to iterative algorithms is that they tend not to be as clear as to their recursive counterparts with respect to their operation. The major

advantage of iterative solutions is speed. Most production software you will find uses little or no recursive algorithms whatsoever. The latter property can sometimes be a companies prerequisite to checking in code, e.g. upon checking in a static analysis tool may verify that the code the developer is checking in contains no recursive algorithms. Normally it is systems level code that has this zero tolerance policy for recursive algorithms.

Using recursion should always be reserved for fast algorithms, you should avoid it for the following algorithm run time deficiencies:

$O(n^2)$
$O(n^3)$
$O(2^n)$

If you use recursion for algorithms with any of the above run time e±ciency's you are inviting trouble. The growth rate of these algorithms is high and in most cases such algorithms will lean very heavily on techniques like divide and conquer. While constantly splitting problems into smaller problems is good practice, in these cases you are going to be spawning a lot of method calls. All this overhead (method calls don't come *that* cheap) will soon pile up and either cause your algorithm to run a lot slower than expected, or worse, you will run out of stack space. When you exceed the allotted stack space for a thread the process will be shutdown by the operating system. This is the case irrespective of the platform you use, e.g. .NET, or native C++ etc. You can ask for a bigger stack size, but you typically only want to do this if you have a very good reason to do so.

C.1 Activation Records

An activation record is created every time you invoke a method. Put simply an activation record is something that is put on the stack to support method invocation. Activation records take a small amount of time to create, and are pretty lightweight.

Normally an activation record for a method call is as follows (this is very general):

The actual parameters of the method are pushed onto the stack

The return address is pushed onto the stack

The top-of-stack index is incremented by the total amount of memory required by the local variables within the method

A jump is made to the method

In many recursive algorithms operating on large data structures, or algorithms that are ine±cient you will run out of stack space quickly. Consider an algorithm that when invoked given a specific value it creates many recursive calls. In such a case a big chunk of the stack will be consumed. We will have to wait until the activation records start to be unwound after the nested methods in the call chain exit and return to their respective caller. When a method exits it's activation record is unwound. Unwinding an activation record results in several steps:

The top-of-stack index is decremented by the total amount of memory consumed by the method

The return address is popped off the stack

The top-of-stack index is decremented by the total amount of memory consumed by the actual parameters

While activation records are an e±cient way to support method calls they can build up very quickly. Recursive algorithms can exhaust the stack size allocated to the thread fairly fast given the chance.

Just about now we should be dusting the cobwebs off the age old example of an iterative vs. recursive solution in the form of the Fibonacci algorithm. This is a famous example as it highlights both the beauty and pitfalls of a recursive algorithm. The iterative solution is not as pretty, nor self documenting but it does the job a lot quicker. If we were to give the Fibonacci algorithm an input of say 60 then we would have to wait a while to get the value back because it has an $O(g^n)$ run time. The iterative version on the other hand has a $O(n)$ run time. Don't let this put you off recursion. This example is mainly used to shock programmers into thinking about the ramifications of recursion rather than warning them off.

C.2 Some problems are recursive in nature

Something that you may come across is that some data structures and algo-rithms are actually recursive in nature. A perfect example of this is a tree data structure. A common tree node usually contains a value, along with two point-ers to two other nodes of the same node type. As you can see tree is recursive in its makeup wit each node possibly pointing to two other nodes.

When using recursive algorithms on tree's it makes sense as you are simply adhering to the inherent design of the data structure you are operating on. Of course it is not all good news, after all we are still bound by the limitations we have mentioned previously in this chapter.

We can also look at sorting algorithms like merge sort, and quick sort. Both of these algorithms are recursive in their design and so it makes sense to model them recursively.

C.3 Summary

Recursion is a powerful tool, and one that all programmers should know of. Often software projects will take a trade between readability, and e±ciency in which case recursion is great provided you don't go and use it to implement

an algorithm with a quadratic run time or higher. Of course this is not a rule of thumb, this is just us throwing caution to the wind. Defensive coding will always prevail.

Many times recursion has a natural home in recursive data structures and algorithms which are recursive in nature. Using recursion in such scenarios is perfectly acceptable. Using recursion for something like linked list traversal is a little overkill. Its iterative counterpart is probably less lines of code than its recursive counterpart.

Because we can only talk about the implications of using recursion from an abstract point of view you should consult your compiler and run time environ-ment for more details. It may be the case that your compiler recognises things like tail recursion and can optimise them. This isn't unheard of, in fact most commercial compilers will do this. The amount of optimisation compilers can do though is somewhat limited by the fact that you are still using recursion.

You, as the developer have to accept certain accountability's for performance.

Appendix D
Symbol Definitions

Throughout the pseudocode listings you will find several symbols used, describes the meaning of each of those symbols.

Symbol	Description
<- *or* ->	Assignment.
=	Equality.
<=	Less than or equal to.
<	Less than.
>=	Greater than or equal to.
>	Greater than.
!=	Inequality.
∅	Null.
And	Logical and.
Or	Logical or.
Whitespace	Single occurrence of whitespace.
Yield	Like return but builds a sequence.

Table D.1: Pseudo symbol definitions

This symbol has a direct translation with the vast majority of imperative counterparts.