Problem Solving with Algorithms and Data Structures

Problem Solving with Algorithms and Data Structures

- Introduction
  - Review of Basic Python
    - Data
    - Input and Output
    - Control Structures
    - Exception Handing
    - Defining Functions
    - Object-Oriented Programming
- ALGORITHM ANALYSIS
  - What is Algothrm Analysis
    - Big-O Notation
    - Anagram Detection
  - Performance of Python Data Structures
    - List
    - Dict
- BASIC DATA STRUCTURES
  - What Are Linear Structures
  - Stacks
    - The Stack Abstract Data Type
    - Implementing A Stack in Python
    - Simple Balance Parentheses
    - Balanced Symbols (A General Case)
    - Converting Decimal Numbers to Binary Numbers
    - Infix, Prefix, and Postfix Expressions
  - Queues
    - The Queue Abstract Data Type
    - Implementing A Queue in Python
    - Simulation: Hot Potato
    - Simulation: Printing Tasks
  - Deques
    - The Deque Abstract Data Type
    - Implementing a Deque in Python
    - Palindrome Checker
  - Lists
    - The Unordered List Abstract Data Type
    - Implementing an Unordered List: Linked Lists
    - The Ordered List Abstract Data Type
    - Implementing an Ordered List
    - Analysis of Linked Lists
- RECURSION
  - What is Recursion
    - Calculating the Sum of a List of Numbers
    - The Three Laws of Recursion
    - Converting an Integer to a String in Any Base

- Stack Frames: Implementing Recursion
- Visualising Recursion
- Complex Recursive Problems
  - The Towers Of Hanoi
- Exploring a Maze

SORTING AND SEARCHING

- Searching
  - The Sequential Search
  - The Binary Search
  - Hashing
    - Hash Functions
    - Collision Resolution
    - Implementing the Map Abstract Data Type
- Sorting
  - Bubble Sort
  - Selection Sort
  - The Insertion Sort
  - Shell Sort
  - The Merge Sort
  - The Quick Sort
- Summary

TREES AND TREE ALGORITHMS

- Examples of Trees
- Vocabulary and Definition
  - Vocabulary
  - Definition
- Implementation
  - List of Lists Representation
  - Nodes and References
- Binary Tree Applications
  - Parse Tree
  - Tree Traversals
- Priority Queues with Binary Heaps
  - Binary Heap Operations
  - Binary Heap Implementation
    - The Structure Property
    - The Heap Order Property
    - Heap Operations
- Binary Search Trees
  - Search Tree Operations
  - Search Tree Implementation
  - Search Tree Analysis
- Balanced Binary Search Trees
  - AVL Tree Performance
  - AVL Tree Implementation
- Summary of Map ADT Implementations

Graphs and Graph Algorithms

- Vocabulary and Definitions
- The Graph Abstract Data Type
  - An Adjacency Matrix
  - An Adjacency List
- Breadth First Search

        The Word Ladder Problem
        Building the Word Ladder Graph
        Implementing Breadth First Search
        Breadth First Search Analysis
        The Knight's Tour Problem
    Depth First Search
        Building the Knight's Tour Graph
        Implementing Knight's Tour
        Knight's Tour Analysis
        General Depth First Search
        Depth First Search Analysis
    Topological Sorting
    Strongly Connected Components
    Shortest Path Problems
        Dijkstra's Algorithm
        Analysis of Dijkstra's Algorithm
        Prim's Spanning Tree Algorithm

Introduction

Review of Basic Python

Data

int float \ bool \
list string tuple \ set dict

Input and Output

input \ output

Control Structures

while for

Exception Handing

try except

Defining Functions

def

Object-Oriented Programming

class

ALGORITHM ANALYSIS

What is Algothrm Analysis

Big-O Notation

Our goal is to show how the algorithm's execution time changes
with respect to the size of the problem.
order of magnitude, it provides a useful approximation to the actual number of steps in the computation.

Anagram Detection

Checking Off: $ *O*(n^2) $
Sort and Compare: $ *O*(n^2) $ or $ *O*(nlogn) $
Brute Force: $ O(n!) $
Count and Compare: $ O(n) $

Performance of Python Data Structures

the Big-O performance for the operations on Python lists and dictionaries

List

common operations: indexing and assign to an index position $ O(1) $ \ grow a list

generate a list of $ n $ numbers:

- concatenation: $ O(k) $
- append: $ O(1) $
- comprehension:
- range function:

Big-O Efficiency of Python List Operations

Operation	Big-O Efficiency
indexx[]	$O(1)$
index assignment	$O(1)$
append	$O(1)$
pop()	$O(1)$
pop(i)	$O(n)$
insert(i,item)	$O(n)$
del operator	$O(n)$
iteration	$O(n)$
contains (in)	$O(n)$
get slice [x:y]	$O(k)$
del slice	$O(n)$
set slice	$O(n+k)$
reverse	$O(n)$
concatenate	$O(k)$
sort	$O(n \log n)$
multiply	$O(nk)$

Table 2.2: Big-O Efficiency of Python List Operators

Dict

Operation	Big-O Efficiency
copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

Table 2.3: Big-O Efficiency of Python Dictionary Operations

BASIC DATA STRUCTURES

What Are Linear Structures

linear structure: stacks, queues, deques, and lists (whose items are ordered depending on how they are added or removed)

Stacks

an ordered collection of items
top base
LIFO(last-in-first-out)

The Stack Abstract Data Type

stack operations:

- Stack()
- push(item)
- pop()
- peek()
- is_empty()
- size()

Stack Operation	Stack Contents	Return Value
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.is_empty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Table 3.1: Sample Stack Operations

Implementing A Stack in Python

class Stack()

Simple Balance Parentheses

whether the string of parentheses is balanced

Balanced Symbols (A General Case)

When a closing symbol does appear, the only difference is that we must check to be sure that it correctly matches the type of the opening symbol on top of the stack.

Converting Decimal Numbers to Binary Numbers

Divide by 2

Infix, Prefix, and Postfix Expressions

- Conversion of Infix Expressions to Prefix and Postfix

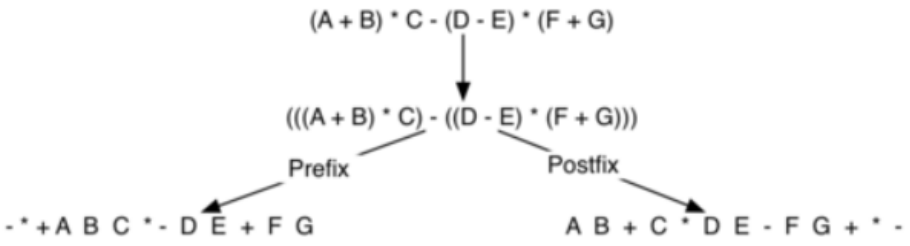


Figure 3.8: Converting a Complex Expression to Prefix and Postfix Notations)

- General Infix-to-Postfix Conversion

When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

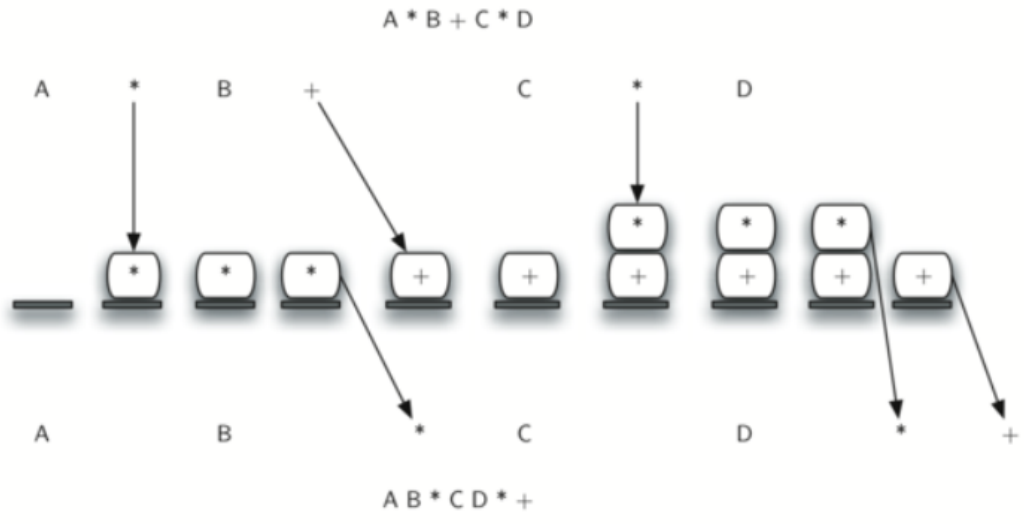


Figure 3.9: Converting $A * B + C * D$ to Postfix Notation)

Steps:

1. Create an empty stack called op_stack for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method split.
3. Scan the token list from left to right.
   • If the token is an operand, append it to the end of the output list.
   • If the token is a left parenthesis, push it on the op_stack.
   • If the token is a right parenthesis, pop the op_stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
   • If the token is an operator, *, /, +, or −, push it on the op_stack. However, first remove any operators already on the op_stack that have higher or equal precedence and append them to the output list.
4. When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

- Postfix Evaluation

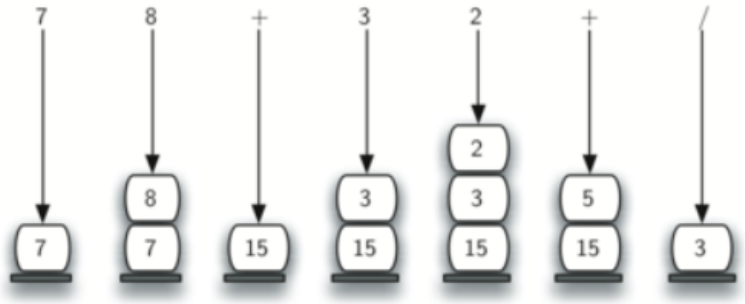


Figure 3.11: A More Complex Example of Evaluation)

1. Create an empty stack called operand_stack.
2. Convert the string to a list by using the string method split.
3. Scan the token list from left to right.
   • If the token is an operand, convert it from a string to an integer and push the value onto the operand_stack.
   • If the token is an operator, *, /, +, or −, it will need two operands. Pop the operand_stack twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the operand_stack.
4. When the input expression has been completely processed, the result is on the stack. Pop the operand_stack and return the value.

Queues

an ordered collection of items
front(remove) rear(add)
FIFO, first-in first-out

The Queue Abstract Data Type

- Queue()
- enqueue(item): $ O(n) $
- dequeue(): $ O(1) $
- is_empty()
- size()

Queue Operation	Queue Contents	Return Value
`q.is_empty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.is_empty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

Table 3.5: Example Queue Operations

Implementing A Queue in Python

the rear is at position 0 in the list.
class Queue()

Simulation: Hot Potato

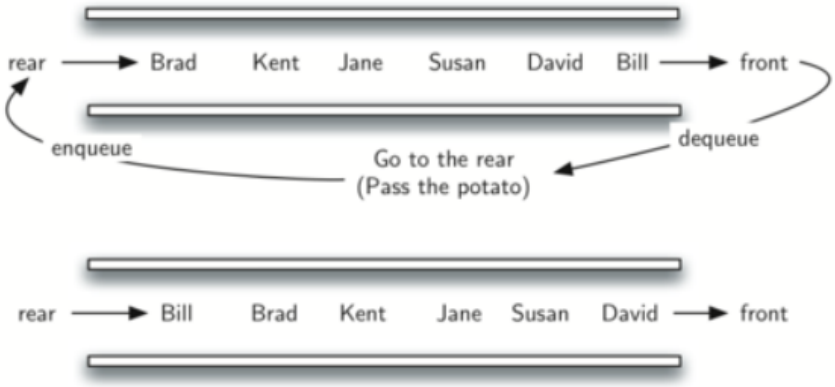


Figure 3.14: A Queue Implementation of Hot Potato)

Simulation: Printing Tasks

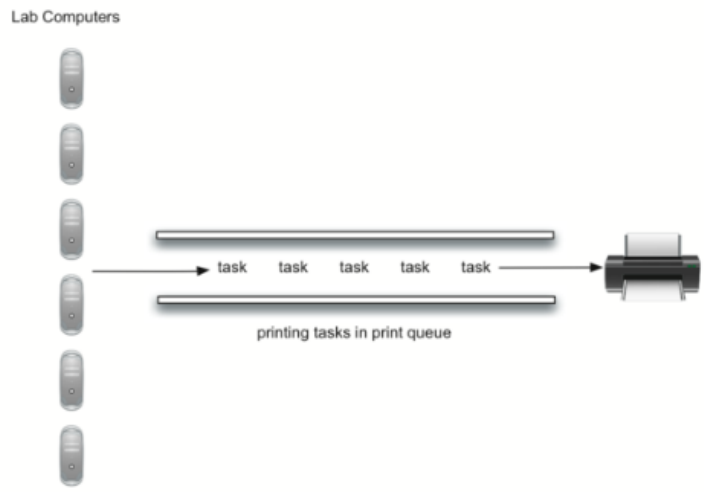


Figure 3.15: Computer Science Laboratory Printing Queue)

Main Simulation Steps:

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.
2. For each second (current_second):
   • Does a new print task get created? If so, add it to the queue with the current_second as the timestamp.
   • If the printer is not busy and if a task is waiting,
   – Remove the next task from the print queue and assign it to the printer.
   – Subtract the timestamp from the current_second to compute the waiting time for that task.
   – Append the waiting time for that task to a list for later processing.
   – Based on the number of pages in the print task, figure out how much time will be required.
   • The printer now does one second of printing if necessary. It also subtracts one second from the time required for that task.
   • If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
3. After the simulation is complete, compute the average waiting time from the list of waiting times generated.

Deques

an ordered collection of items
It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rea

The Deque Abstract Data Type

- Deque()
- add_front()
- add_rear()
- remove_front

- remove_rear()
- is_empty()
- size()

Implementing a Deque in Python

class Deque()

Palindrome Checker

A palindrome is a string that reads the same forward and backward, for example, radar, toot, and madam.

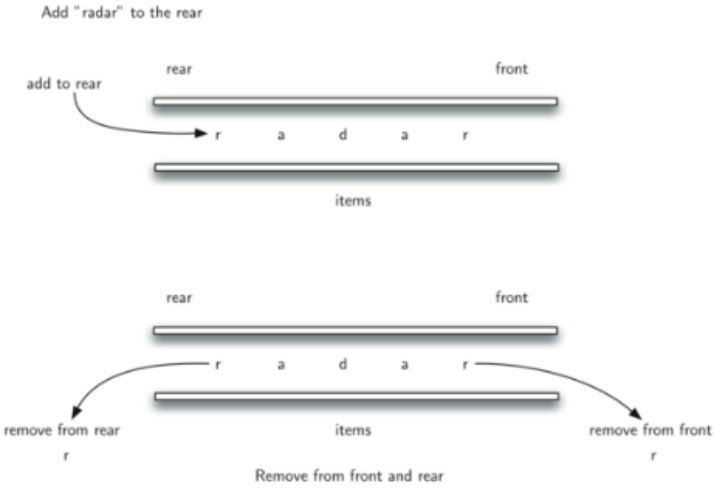


Figure 3.17: A Deque)

Lists

The Unordered List Abstract Data Type

The structure of an unordered list, as described above, is a collection of items where each item holds a relative position with respect to the others.

Some possible unordered list operations:

- List()
- add(item)
- remove(item)
- search(item): return a boolean value
- is_empty()
- size()
- append(item)
- index(item)
- insert(pos, item)
- pop()
- pop(pos)

Implementing an Unordered List: Linked Lists

- The Node Class

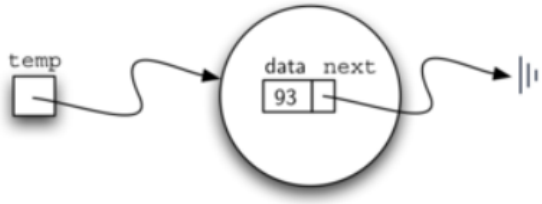


Figure 3.20: A Node Object Contains the Item and a Reference to the Next Node

- The Unordered List Class
  the linked list structure provides us with only one entry point, the head of the list. we will make the new item the first item of the list and the existing
  items will need to be linked to this new first item so that they follow.

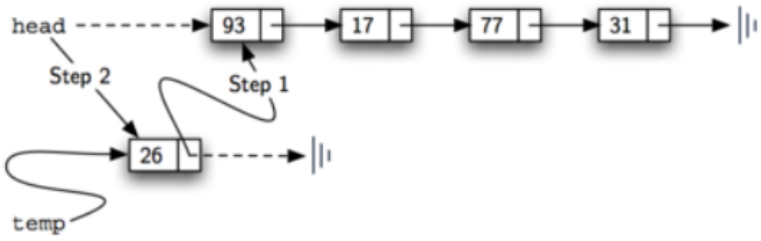


Figure 3.24: Adding a New Node is a Two-Step Process

The Ordered List Abstract Data Type

The structure of an ordered list is a collection of items where each item holds a relative position that is based upon some underlying characteristic of the item. The ordering is typically either ascending or descending and we assume that list items have a meaningful comparison operation that is already defined.

- OrderedList()
- add(item)
- remove(item)
- search(item)
- is_empty()
- size()
- index(item)
- pop()
- pop(pos)

Implementing an Ordered List

overwrite search and add method

Analysis of Linked Lists

is_empty: $ O(1) $
size: $ O(n) $
add for unordered list: $ O(1) $
add search remove for ordered list: $ O(n) $

> linked lists are not the way Python lists are implemented. The actual implementation of a Python list is based on the notion of an array.

RECURSION

What is Recursion

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially.

Calculating the Sum of a List of Numbers

list_sum(num_list) = first(num_list) + list_sum(rest(num_list))

The Three Laws of Recursion

1. A recursive algorithm must have a base case.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

Converting an Integer to a String in Any Base

Knowing what our base is suggests that the overall algorithm will involve three components:

1. Reduce the original number to a series of single-digit numbers.
2. Convert the single digit-number to a string using a lookup.
3. Concatenate the single-digit strings together to form the final result

Stack Frames: Implementing Recursion

When a function is called in Python, a stack frame is allocated to handle the local variables of the function. When the function returns, the return value is left on top of the stack for the calling function to access

Visualising Recursion

- spiral
- tree

- Sierpinski Triangle

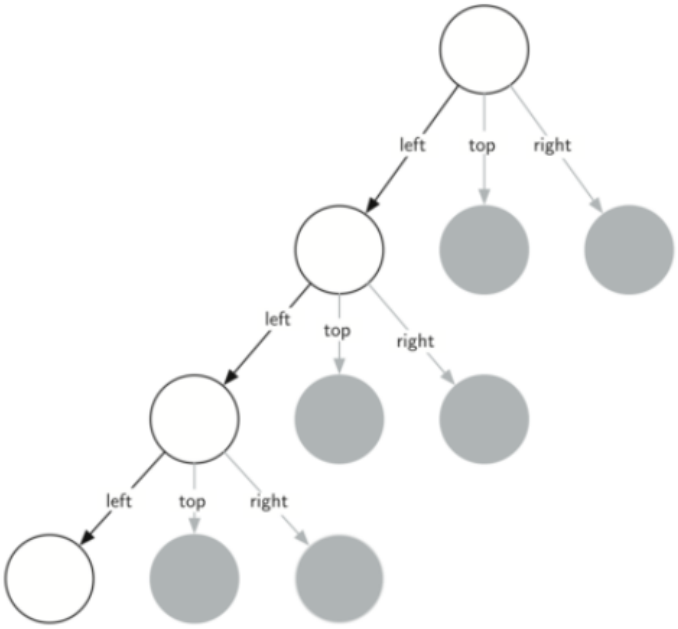


Figure 4.10: Building a Sierpinski Triangle

Complex Recursive Problems

The Towers Of Hanoi

Here is a high-level outline of how to move a tower from the starting pole, to the goal pole, using an intermediate pole:

1. Move a tower of height-1 to an intermediate pole, using the final pole.
2. Move the remaining disk to the final pole.
3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.

Exploring a Maze

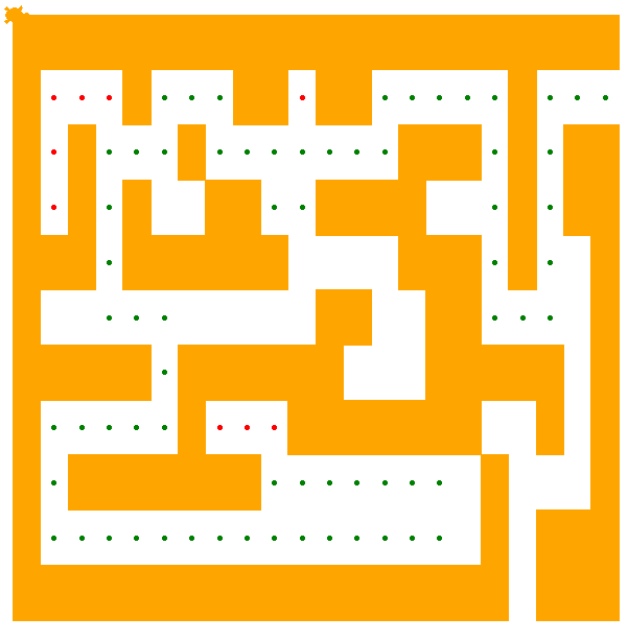
a systematic procedure:

- From our starting position we will first try going North one square and then recursively try our procedure from there.
- If we are not successful by trying a Northern path as the first step then we will take a step to the South and recursively repeat our procedure.
- If South does not work then we will try a step to the West as our first step and recursively apply our procedure.
- If North, South, and West have not been successful then apply the procedure recursively from a position one step to our East.
- If none of these directions works then there is no way to get out of the maze and we fail.

> in order to avoid infinite loop, we must have a strategy to remember where we have been. In this case we will assume that we have a bag of bread crumbs we can drop along our way. If we take a step in a certain direction and find that there is a bread crumb already on that square, we know that we should immediately back up and try the next direction in our procedure.

In this algorithm, there are four base cases to consider:

1. The turtle has run into a wall. Since the square is occupied by a wall no further exploration can take place.
2. The turtle has found a square that has already been explored. We do not want to continue exploring from this position or we will get into a loop.
3. We have found an outside edge, not occupied by a wall. In other words we have found an exit from the maze.
4. We have explored a square unsuccessfully in all four directions.



SORTING AND SEARCHING

Searching

The Sequential Search

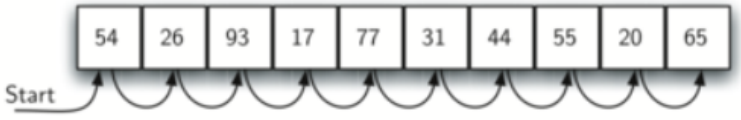


Figure 5.1: The Sequential Search of a List of Integers

Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items.

- unordered list

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

Table 5.1: Comparisons Used in a Sequential Search of an Unordered List

the complexity of the sequential search, is $O(n)$

- ordered list

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	1	n	$\frac{n}{2}$

Table 5.2: Comparisons Used in Sequential Search of an Ordered List

the complexity of the sequential search, is $O(n)$

The Binary Search

the binary search is designed for an ordered list
a binary search will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

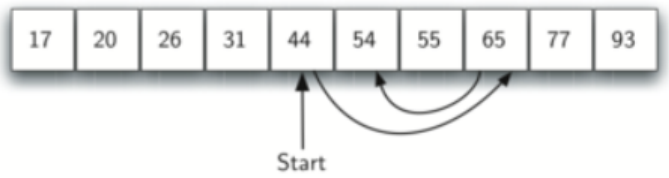


Figure 5.3: Binary Search of an Ordered List of Integers

the complexity of the binary search, is $O(\log n)$

Hashing

build a data structure that can be searched in $O(1)$ time. This concept is referred to as hashing.
A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0.
The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and $m-1$.

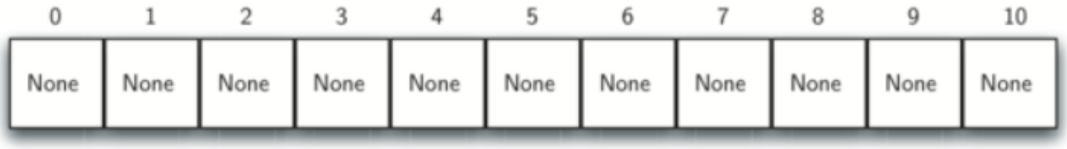


Figure 5.4: Hash Table with 11 Empty Slots

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Table 5.4: Simple Hash Function Using Remainders

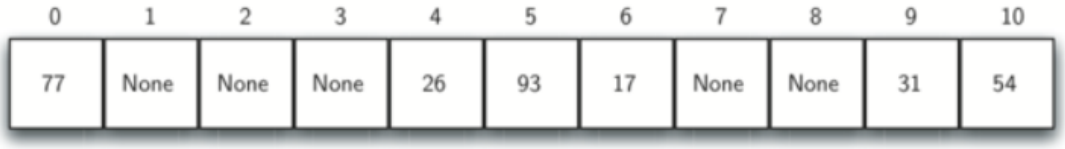


Figure 5.5: Hash Table with Six Items

Hash Functions

Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table.

- folding method
  The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value.436-555-
  436-555-4601, (43, 65, 55, 46, 01), sum() = 210, 210%11 = 1
  some folding methods reverse every other piece before addition
- mid-square method
  we first square the item, and then extract some portion of the resulting digits
  44, 44^2 = 1936, extract 93, 93%11 = 5
- hash functions for character-based items such as strings

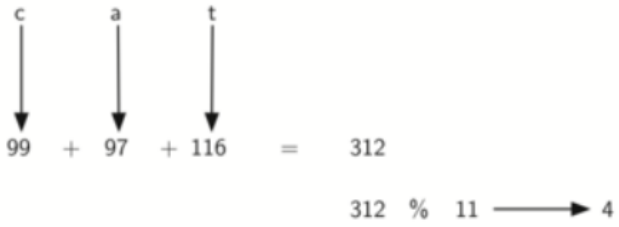


Figure 5.6: Hashing a String Using Ordinal Values

We can then take these three ordinal values, add them up, and use the remainder method to get a hash value.
cat, 99+97+116 = 312, 312%11 = 4

Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table.

- linear probing(open addressing)
  start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty.

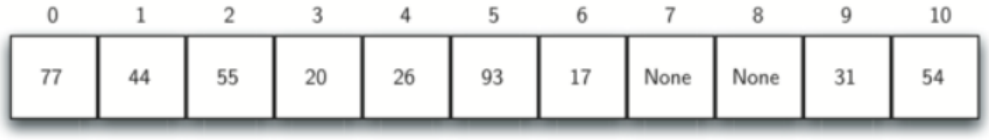


Figure 5.8: Collision Resolution with Linear Probing

A disadvantage to linear probing is the tendency for clustering; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution.
One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions.

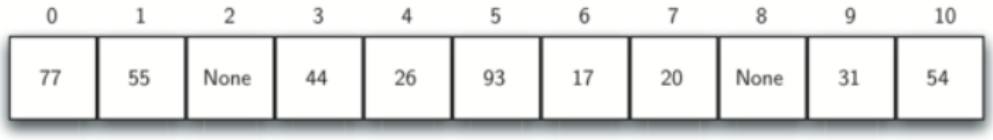


Figure 5.10: Collision Resolution Using "Plus 3"

new_hash_value = rehash(old_hash_value)
rehash(pos) = (pos + 1)%size_of_table
rehash(pos) = (pos + 3)%size_of_table
in general, rehash(pos) = (pos + skip)%size_of_table

- quadratic probing
  Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1,3,5,7,9, and so on.
  This means that if the first hash value is h, the successive values are h+1,h+4,h+9,h+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

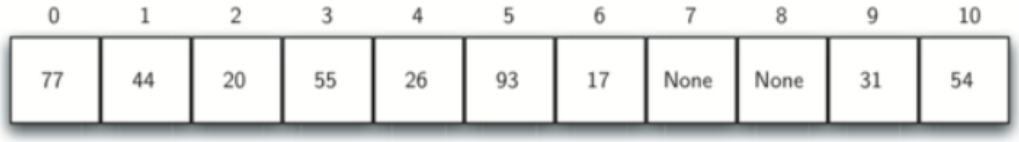


Figure 5.11: Collision Resolution with Quadratic Probing

- Chaining
  Chaining allows many items to exist at the same location in the hash table.

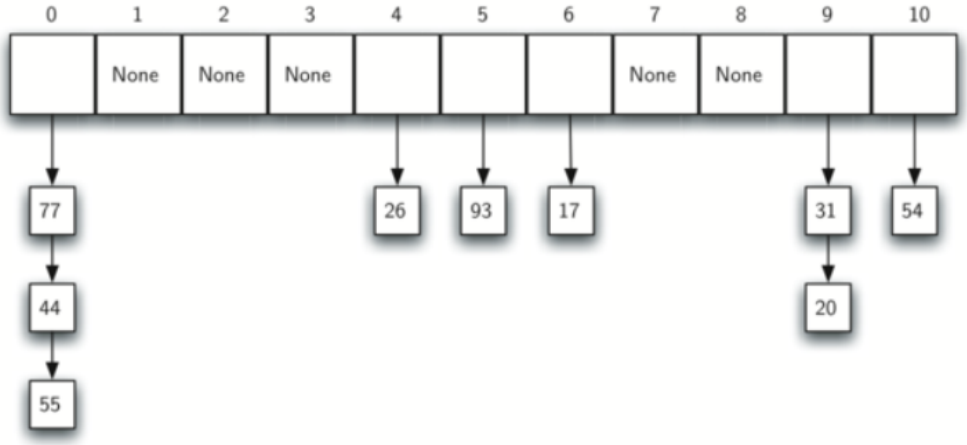


Figure 5.12: Collision Resolution with Quadratic Probing

Implementing the Map Abstract Data Type

The map abstract data type is defined as follows. The structure is an unordered collection of associations between a key and a data value(dictionary). The keys in a map are all unique so that there is a one-to-one relationship between a key and a value.

operations:

- Map()
- put(key, value)
- get(key)
- del map(key)
- len()
- in

we will have a result for both a successful and an unsuccessful search.
For a successful search using open addressing with linear probing, the average number of comparisons is approximately

$$\frac{1}{2}(1+\frac{1}{1-\lambda})$$

and an unsuccessful search gives

$$\frac{1}{2}(1+(\frac{1}{1-\lambda})^{2})$$

If we are using chaining, the average number of comparisons is

$$1+\frac{1}{\lambda}$$

for the successful case, and simply λ comparisons if the search is unsuccessful

$$\lambda$$

Sorting

Bubble Sort

the complexity of the bubble sort, is $O(n^2)$
in the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.
short bubble sort: if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted.

Selection Sort

the complexity of the selection sort, is $O(n^2)$
The selection sort improves on the bubble sort by making only one exchange for every pass through the list and it executes faster in benchmark studies.

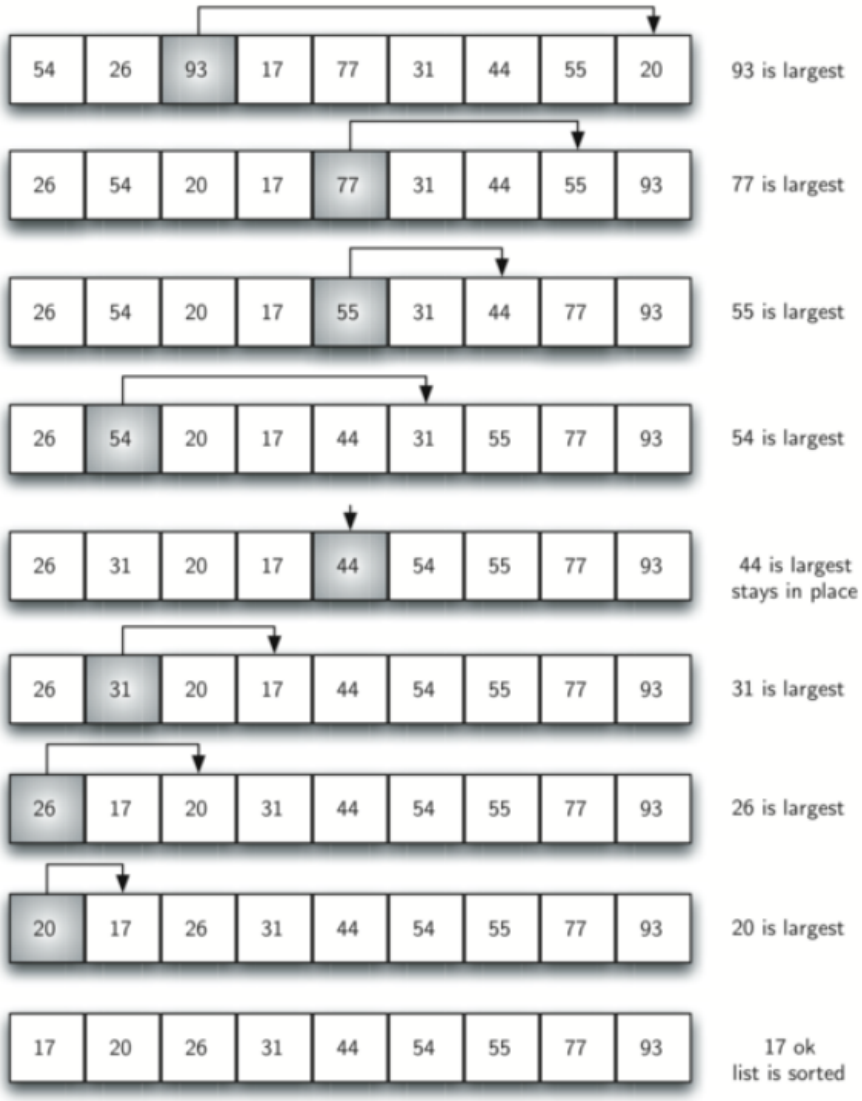


Figure 5.15: Selection Sort

The Insertion Sort

the complexity of the insertion sort, is $O(n^2)$
It always maintains a sorted sublist in the lower positions of the list. Each new item is then "inserted" back into the previous sublist such that the sorted sublist is one item larger. However, in the best case, only one comparison needs to be done on each pass. This would be the case for an already sorted list.

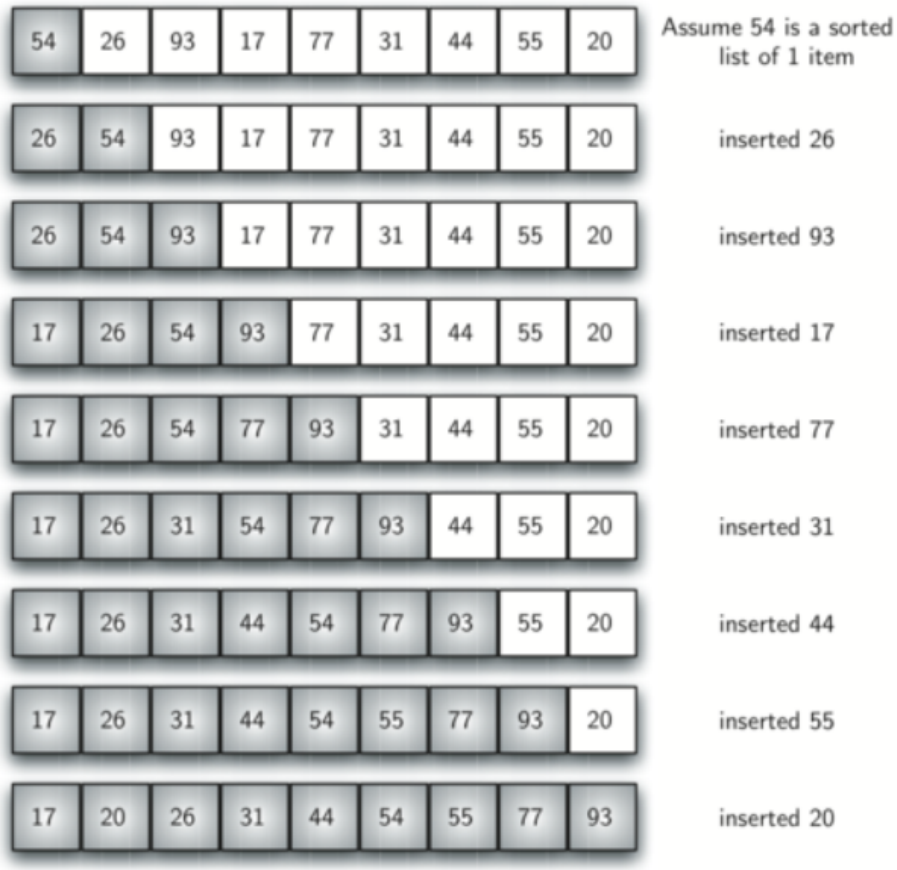


Figure 5.16: Insertion Sort

Shell Sort

the complexity of shell sort tends to fall somewhere between $O(n)$ and $O(n2)$
By changing the increment, for example using $2^k - 1$ (1,3,7,15,31, and so on), a shell sort can perform at $O(n^3/2)$
The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. the shell sort uses an increment *i*, sometimes called the

gap, to create a sublist by choosing all items that are *i* items apart.

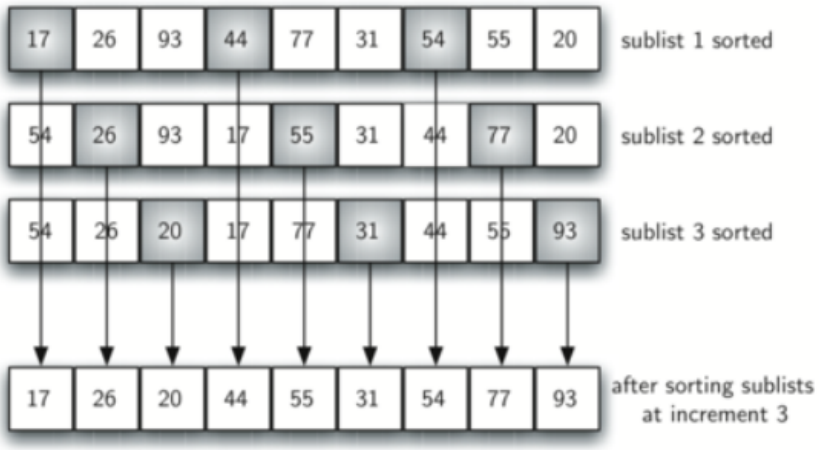


Figure 5.19: A Shell Sort after Sorting Each Sublist

The Merge Sort

the complexity of merge sort is $O(n\log n)$, split and merge
Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

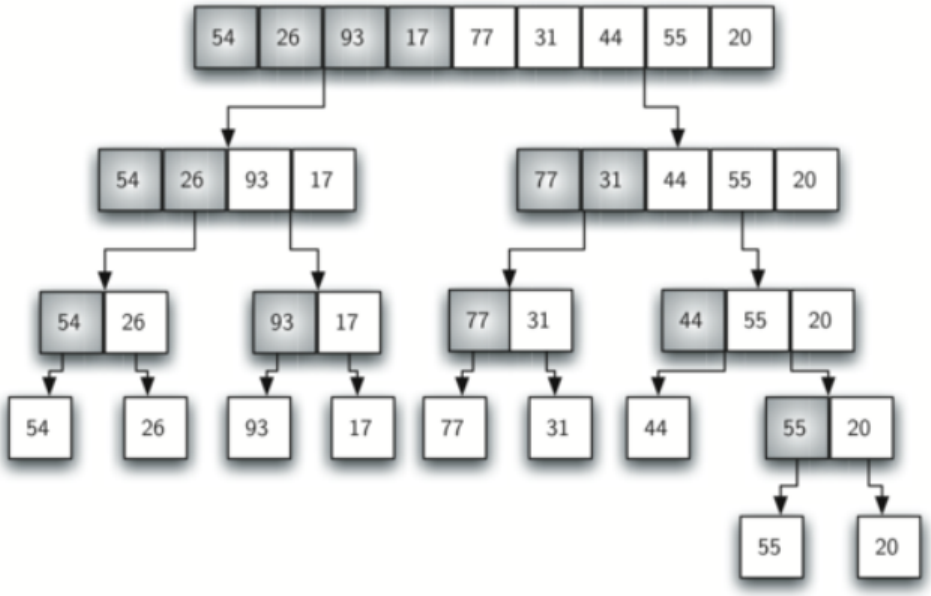


Figure 5.22: Splitting the List in a Merge Sort

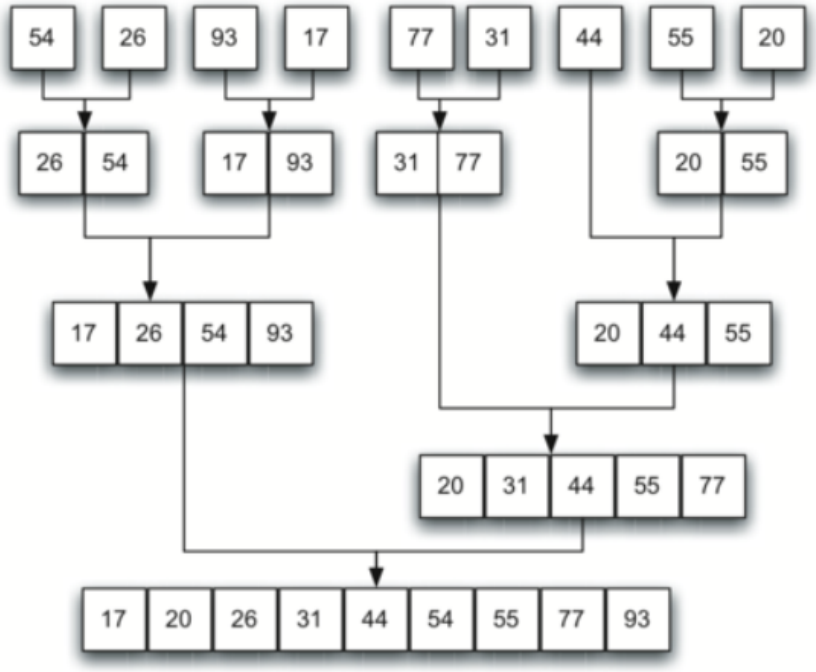


Figure 5.23: Lists as They Are Merged Together

the merge_sort function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical
factor if the list is large and can make this sort problematic when working on large data sets.

The Quick Sort

the complexity of quick sort is $O(n\log n)$. for a list of length n, if the partition always occurs in the middle of the list, there will again be log n divisions. In order to find the split point, each of the n items needs to be checked against the pivot value. in the worst case, the split points may not be in the middle and can be very skewed to the left or the right, leaving a

very uneven division. In this case, sorting a list of n items divides into sorting a list of 0 items and a list of n − 1 items. Then sorting a list of n − 1 divides into a list of size 0 and a list of size n − 2, and so on. The result is an $O(n$^2) sort with all of the overhead that recursion requires.

The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage.

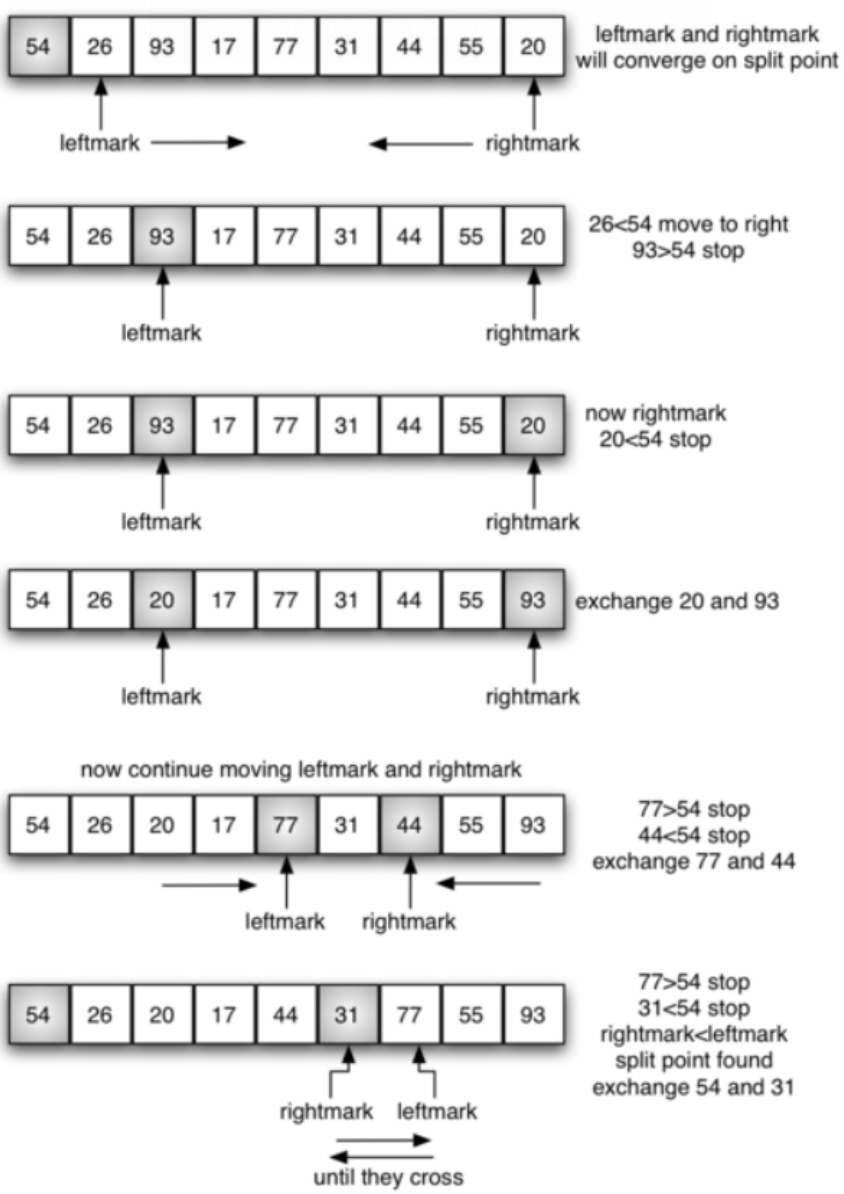


Figure 5.25: Finding the Split Point for 54

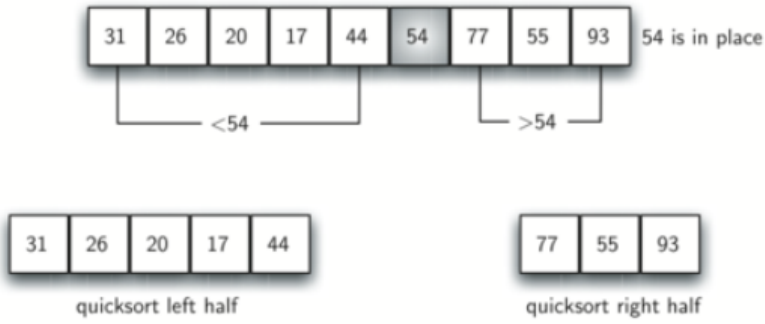


Figure 5.26: Completing the Partition Process to Find the Split Point for 54

Summary

• A sequential search is $O(n)$ for ordered and unordered lists.
• A binary search of an ordered list is $O(\log n)$ in the worst case.
• Hash tables can provide constant time searching.
• A bubble sort, a selection sort, and an insertion sort are $O(n$^2) algorithms.
• A shell sort improves on the insertion sort by sorting incremental sublists. It falls between $O(n)$ and $O(n$^2).
• A merge sort is $O(n\log n)$, but requires additional space for the merging process.
• A quick sort is $O(n\log n)$, but may degrade to $O(n$^2) if the split points are not near the middle of the list. It does not require additional space

TREES AND TREE ALGORITHMS

Examples of Trees

several properties of trees:

- trees are hierarchical
- all of the children of one node are independent of the children of another node
- each leaf node is unique

Vocabulary and Definition

Vocabulary

- Node
- Edge
- Root
- Path
- Children
- Parent
- Sibling
- Subtree
- Leaf Node
- Level: the number of edges on the path from the root node to the node
- Height: the maximum level of any node in the tree

Definition

Definition One: A tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:

- One node of the tree is designated as the root node.
- Every node *n*, except the root node, is connected by an edge from exactly one other node *p*, where pp is the parent of *n*.
- A unique path traverses from the root to each node.
- If each node in the tree has a maximum of two children, we say that the tree is a **binary tree**.

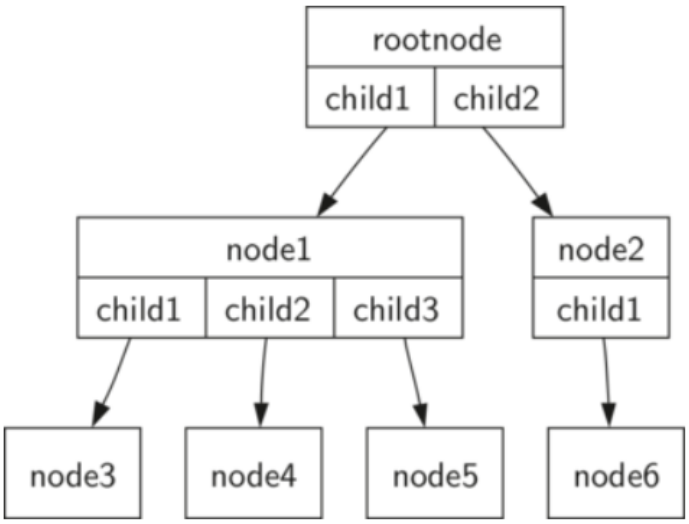


Figure 3: A Tree Consisting of a Set of Nodes and Edges

Definition Two: A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.

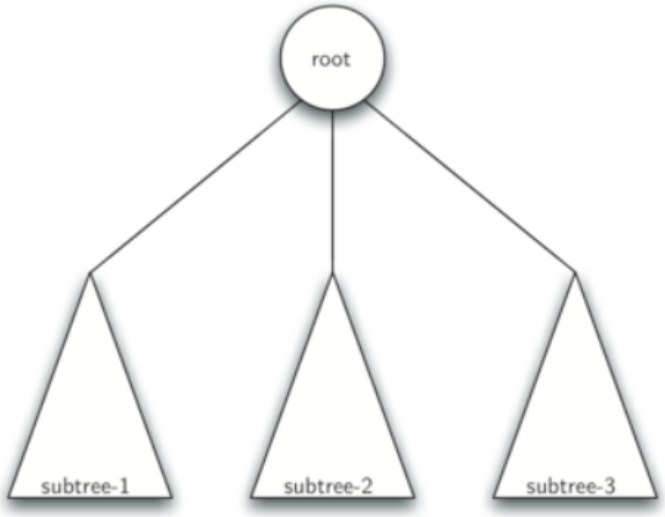


Figure 4: A recursive Definition of a tree

Implementation

The key decision in implementing a tree is choosing a good internal storage technique. Python allows us two very interesting possibilities, so we will examine both before choosing one. The first technique we will call “list of lists,” the second technique we will call “nodes and references.”

List of Lists Representation

In a list of lists tree, we will store the value of the root node as the first element of the list. The second element of the list will itself be a list that represents the left subtree. The third element of the list will be another list that represents the right subtree.

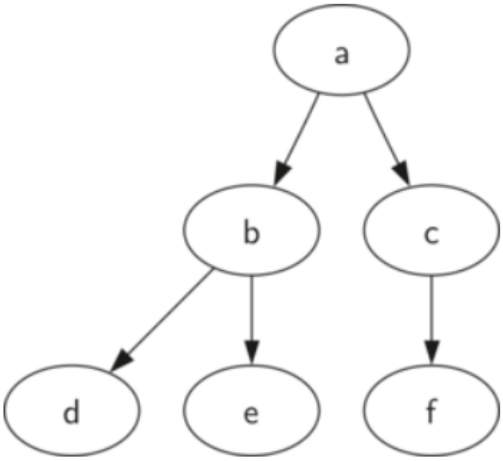


Figure 1: A Small Tree

```
myTree = ['a',   #root
      ['b',  #left subtree
       ['d', [], []],
       ['e', [], []] ],
      ['c',  #right subtree
       ['f', [], []],
       [] ]
     ]
```

One very nice property of this list of lists approach is that the structure of a list representing a subtree adheres to the structure defined for a tree; the structure itself is recursive! A subtree that has a root value and two empty lists is a leaf node. Another nice feature of the list of lists approach is that it generalizes to a tree that has many subtrees. In the case where the tree is more than a binary tree, another subtree is just another list.

Nodes and References

In this case we will define a class that has attributes for the root value, as well as the left and right subtrees. this representation more closely follows the object-oriented programming paradigm.

Binary Tree Applications

Parse Tree

((7+3)∗(5−2))

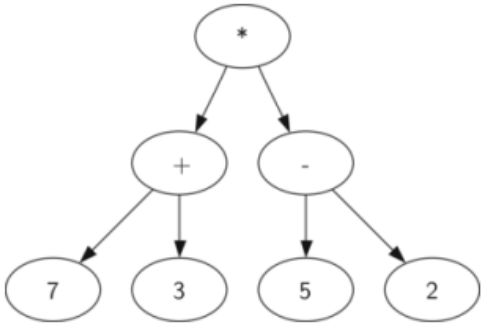


Figure 6.14: Parse Tree for $((7+3)*(5-2))$

1. If the current token is a `'('`, add a new node as the left child of the current node, and descend to the left child.
2. If the current token is in the list `['+','-','/','*']`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a `')'`, go to the parent of the current node.

Tree Traversals

There are three commonly used patterns to visit all the nodes in a tree. The difference between these patterns is the order in which each node is visited. We call this visitation of the nodes a "traversal." The three traversals we will look at are called **preorder**, **inorder**, and **postorder**.

- **preorder**
  In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.
- **inorder**
  In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.
- postorder
  In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

Priority Queues with Binary Heaps

A priority queue acts like a queue in that you dequeue an item by removing it from the front. However, in a priority queue the logical order of items inside a queue is determined by their priority. The highest priority items are at the front of the queue and the lowest priority items are at the back.

The classic way to implement a priority queue is using a data structure called a **binary heap**. A binary heap will allow us both enqueue and dequeue items in *O(log n)*.

The binary heap is interesting to study because when we diagram the heap it looks a lot like a tree, but when we implement it we use only a single list as an internal representation. The binary heap has two common variations: the **min heap**, in which the smallest key is always at the front, and the **max heap**, in which the largest key value is always at the front.

Binary Heap Operations

- BinaryHeap()
- insert(k)
- findMin()
- delMin()
- isEmpty()

- size()
- buildHeap(list)

Binary Heap Implementation

The Structure Property

In our heap implementation we keep the tree balanced by creating a **complete binary tree**. each level has all of its nodes except the bottom level.

Another interesting property of a complete tree is that we can represent it using a single list. the left child of a parent (at position p) is the node that is found in position 2p in the list. Similarly, the right child of the parent is at position 2p+1 in the list.

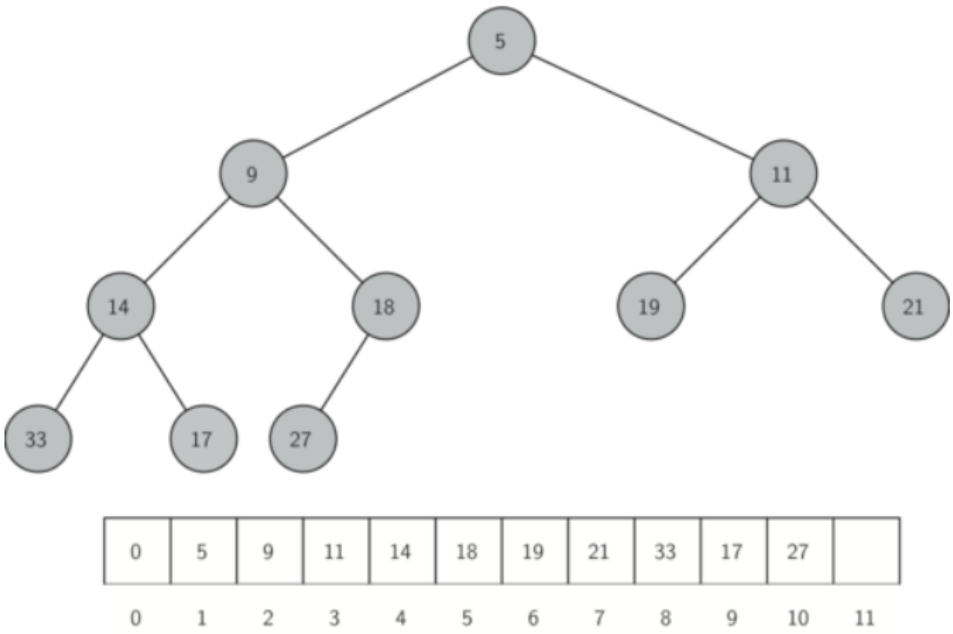


Figure 2: A Complete Binary Tree, along with its List Representation

The Heap Order Property

In a heap, for every node *x* with parent *p*, the key in pp is smaller than or equal to the key in *x*.

Heap Operations

The assertion that we can build the heap in O(n)

Binary Search Trees

Search Tree Operations

- Map()
- put(key, val)
- get(key)
- del
- len()
- in

Search Tree Implementation

A binary search tree relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree. We will call this the **bst property**.

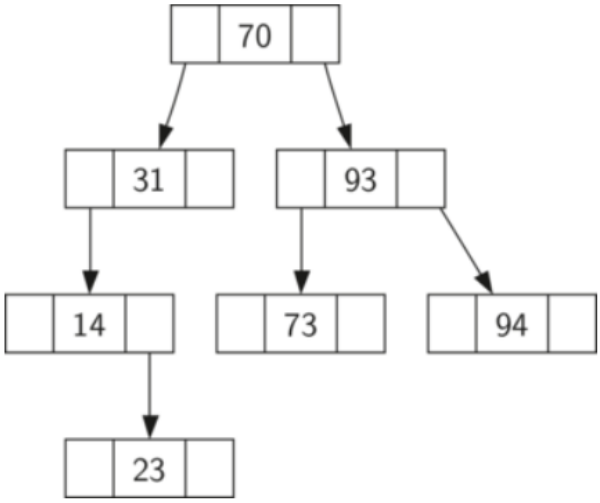


Figure 1: A Simple Binary Search Tree

inserting a new node in the tree:

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.
- When there is no left (or right) child to search, we have found the position in the tree where the new node should be installed.
- To add a node to the tree, create a new `TreeNode` object and insert the object at the point discovered in the previous step.

Search Tree Analysis

If the keys are added in a random order, the height of the tree is going to be around log2n where n is the number of nodes in the tree.

The total number of nodes in a perfectly balanced binary tree is 2^(h+1)−1, where h represents the height of the tree.

In a balanced binary tree, the worst-case performance of `put` is *O(log2 n)*, where n is the number of nodes in the tree. it is possible to construct a search tree that has height n simply by inserting the keys in sorted order. In this case the performance of the `put` method is *O(n)*.

Balanced Binary Search Trees

AVL tree: automatically make sure that the tree remains balanced at all times.

we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$balanceFactor = height(leftSubTree) - height(rightSubTree)$$

The balance factor is greater than zero then the subtree is left-heavy, and the subtree is right heavy if the balance factor is less than zero. If the balance factor is zero then the tree is perfectly in balance.

we define a tree to be in balance if the balance factor is -1, 0, or 1.

AVL Tree Performance

$$h = 1.44logN_h$$

At any time the height of our AVL tree is equal to a constant(1.44) times the log of the number of nodes in the tree. This is great news for searching our AVL tree because it limits the search to O(logN).

AVL Tree Implementation

Since all new keys are inserted into the tree as leaf nodes and we know that the balance factor for a new leaf is zero, there are no new requirements for the node that was just inserted. But once the new leaf is added we must update the balance factor of its parent. How this new leaf affects the parent's balance factor depends on whether the leaf node is a left child or a right child. If the new node is a right child the balance factor of the parent will be reduced by one. If the new node is a left child then the balance factor of the parent will be increased by one. This relation can be applied recursively to the grandparent of the new node, and possibly to every ancestor all the way up to the root of the tree. Since this is a recursive procedure let us examine the two base cases for updating balance factors:

- The recursive call has reached the root of the tree.
- The balance factor of the parent has been adjusted to zero. You should convince yourself that once a subtree has a balance factor of zero, then the balance of its ancestor nodes does not change.

Summary of Map ADT Implementations

Table 1: Comparing the Performance of Different Map Implementations

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
put	$O(n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
get	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
in	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
del	$O(n)$	$O(1)$	$O(n)$	$O(\log_2 n)$

Graphs and Graph Algorithms

Vocabulary and Definitions

- vertex(node): key, payload
- edge(arc): edges may be one-way or two-way, directed gragh or digraph that all edges are all one-way
- weight: cost to go from one vertex to another
- path: a sequence of vertices that are connected by edges
- cycle: a path that starts and ends at the same vertex. acyclic graph with no cycles, directed acyclic graph

The Graph Abstract Data Type

- `Graph()` creates a new, empty graph.
- `addVertex(vert)` adds an instance of `Vertex` to the graph.
- `addEdge(fromVert, toVert)` Adds a new, directed edge to the graph that connects two vertices.
- `addEdge(fromVert, toVert, weight)` Adds a new, weighted, directed edge to the graph that connects two vertices.
- `getVertex(vertKey)` finds the vertex in the graph named `vertKey`.
- `getVertices()` returns the list of all vertices in the graph.
- `in` returns `True` for a statement of the form `vertex in graph`, if the given vertex is in the graph, `False` otherwise.

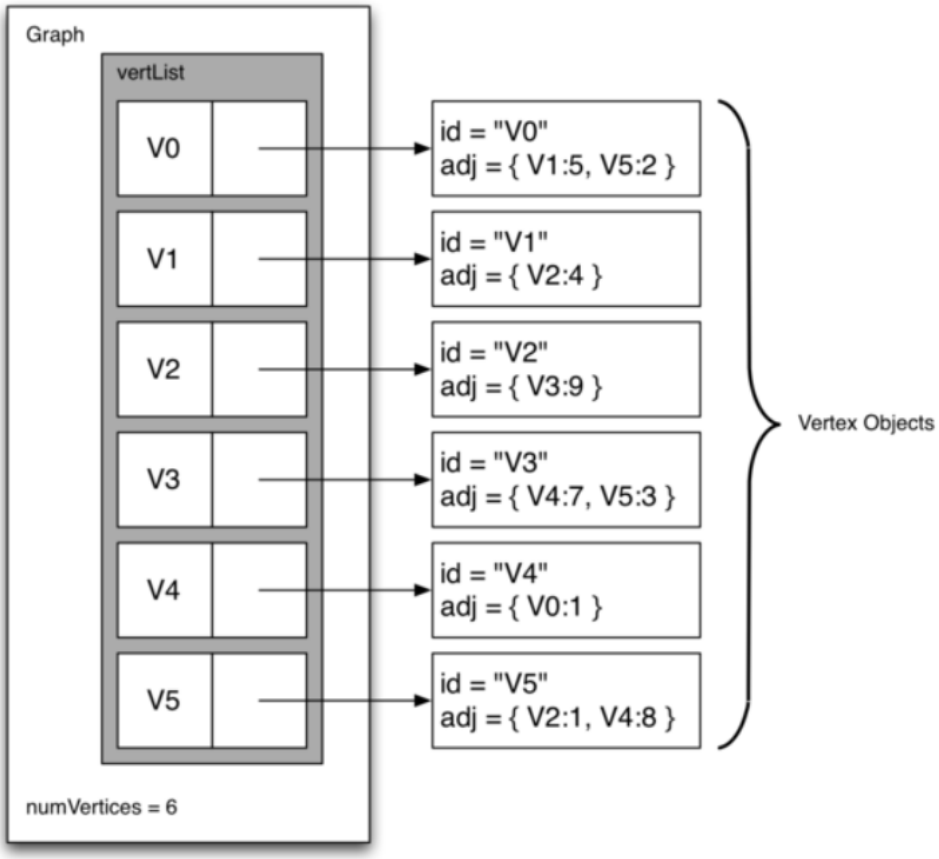
There are two well-known implementations of a graph, the **adjacency matrix** and the **adjacency list**.

An Adjacency Matrix

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

The advantage of the adjacency matrix is that it is simple, and for small graphs it is easy to see which nodes are connected to other nodes. However, notice that most of the cells in the matrix are empty. Because most of the cells are empty we say that this matrix is “sparse.”

An Adjacency List



In an adjacency list implementation we keep a master list of all the vertices in the Graph object and then each vertex object in the graph maintains a list of the other vertices that it is connected to. In our implementation of the `Vertex` class we will use a dictionary rather than a list where the dictionary keys are the vertices, and the values are the weights.

The advantage of the adjacency list implementation is that it allows us to compactly represent a sparse graph. The adjacency list also allows us to easily find all the links that are directly connected to a particular vertex.

Breadth First Search

The Word Ladder Problem

In a word ladder puzzle you must make the change occur gradually by changing one letter at a time. At each step you must transform one word into another word, you are not allowed to transform a word into a non-word.

```
fail
fall
pall
pole
poll
pope
sale
pale
page
```

```
sage
pool
cool
fool
foul
foil
```

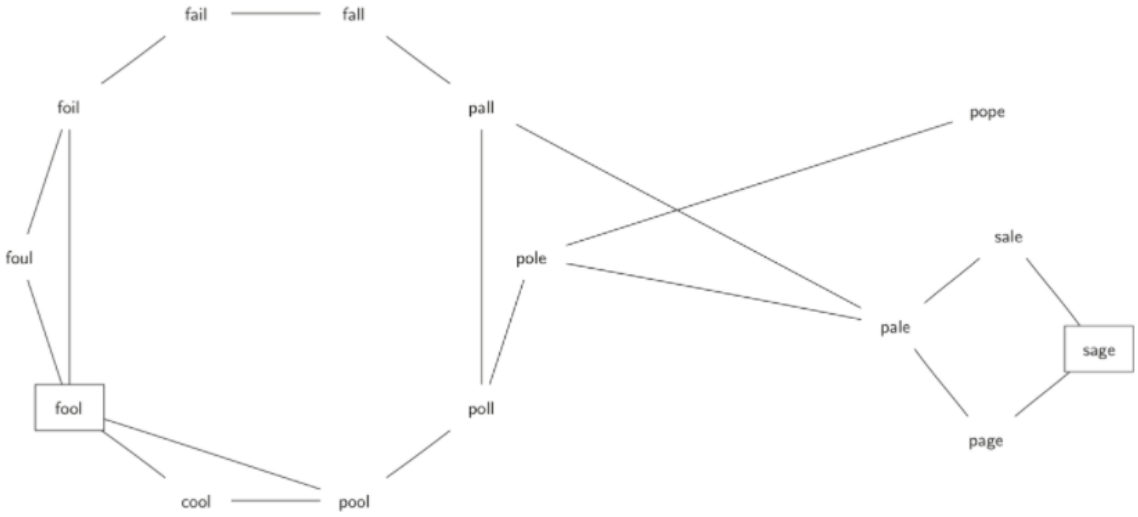


Figure 1: A Small Word Ladder Graph

Building the Word Ladder Graph

To create the graph, we should connect the words first. Suppose that we have a huge number of buckets, each of them with a four-letter word on the outside, except that one of the letters in the label has been replaced by an underscore. For example, we might have a bucket labeled "pop." *As we process each word in our list we compare the word with each bucket, using the* " as a wildcard, so both "pope" and "pops" would match "pop_." Every time we find a matching bucket, we put our word in that bucket. Once we have all the words in the appropriate buckets we know that all the words in the bucket must be connected.

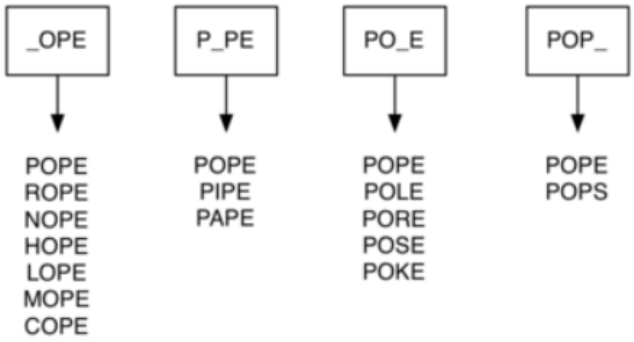


Figure 2: Word Buckets for Words That are Different by One Letter

Implementing Breadth First Search

BFS begins at the starting vertex `s` and colors `start` gray to show that it is currently being explored. Two other values, the distance and the predecessor, are initialized to 0 and `None` respectively for the starting vertex. Finally, `start` is placed on a `Queue`. The next step is to begin to systematically explore vertices at the front of the queue. We explore each new node at the front of the queue by iterating over its adjacency list. As each node on the

adjacency list is examined its color is checked. If it is white, the vertex is unexplored, and four things happen:

1. The new, unexplored vertex `nbr`, is colored gray.
2. The predecessor of `nbr` is set to the current node `currentVert`
3. The distance to `nbr` is set to the distance to `currentVert + 1`
4. `nbr` is added to the end of a queue. Adding `nbr` to the end of the queue effectively schedules this node for further exploration, but not until all the other vertices on the adjacency list of `currentVert` have been explored.

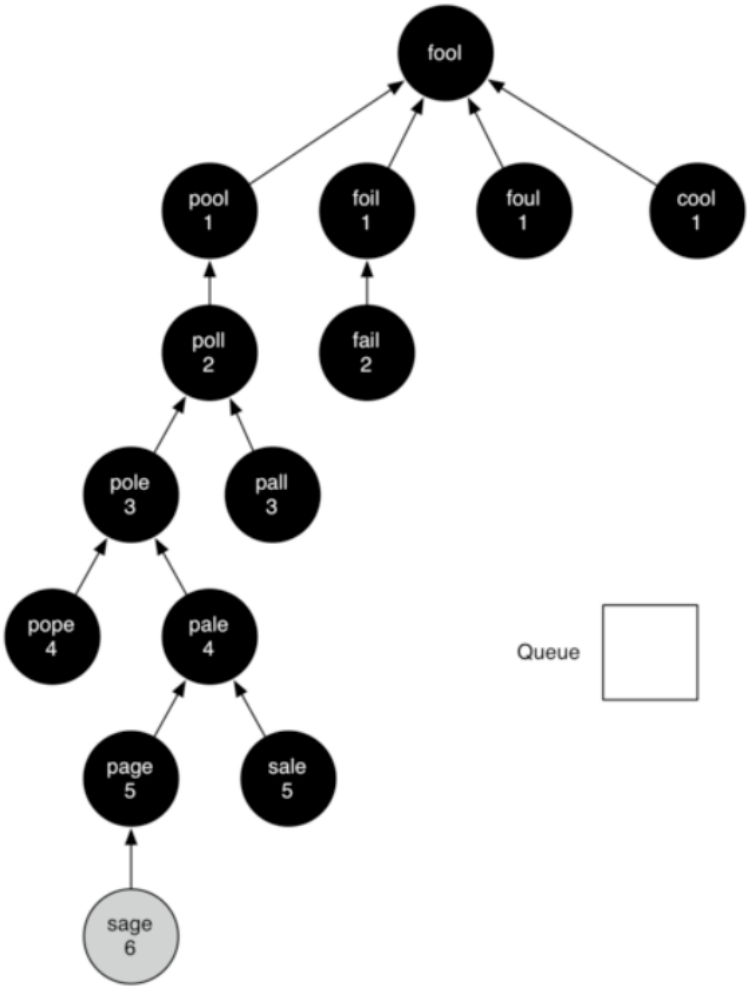


FIgure 6: Final Breadth First Search Tree

Breadth First Search Analysis

The first thing to observe is that the while loop is executed, at most, one time for each vertex in the graph |V|. You can see that this is true because a vertex must be white before it can be examined and added to the queue. This gives us O(V) for the while loop. The for loop, which is nested inside the while is executed at most once for each edge in the graph, |E|. The reason is that every vertex is dequeued at most once and we examine an edge from node u to node v only when node u is dequeued. This gives us O(E) for the for loop. combining the two loops gives us O(V+E).

Of course doing the breadth first search is only part of the task. Following the links from the starting node to the goal node is the other part of the task. The worst case for this would be if the graph was a single long chain. In this case traversing through all of the vertices would be O(V). The normal case is going to be some fraction of |V| but we would still write O(V).

The Knight's Tour Problem

The knight's tour puzzle is played on a chess board with a single chess piece, the knight. The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once.

A graph search is one of the easiest to understand and program. Once again we will solve the problem using two main steps:

- Represent the legal moves of a knight on a chessboard as a graph.
- Use a graph algorithm to find a path of length rows×columns−1rows×columns−1 where every vertex on the graph is visited exactly once.

Depth First Search

Building the Knight's Tour Graph

To represent the knight's tour problem as a graph we will use the following two ideas: Each square on the chessboard can be represented as a node in the graph. Each legal move by the knight can be represented as an edge in the graph.

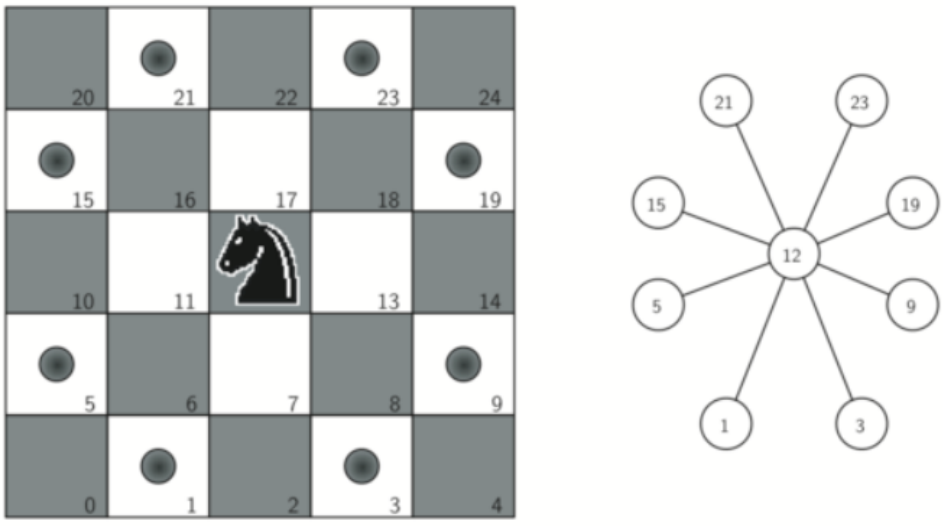


Figure 1: Legal Moves for a Knight on Square 12, and the Corresponding Graph

There are exactly 336 edges in the graph. Notice that the vertices corresponding to the edges of the board have fewer connections (legal moves) than the vertices in the middle of the board.

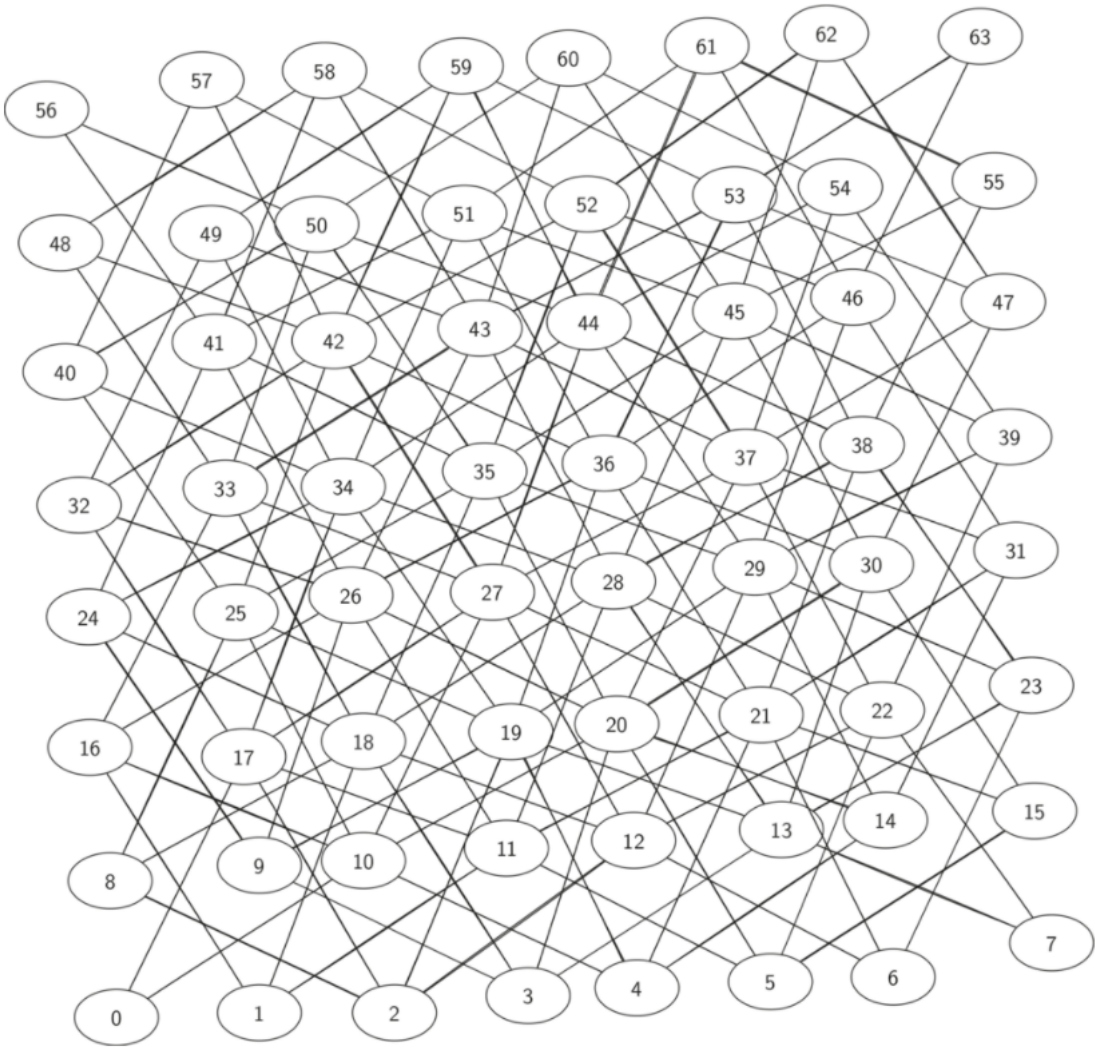


Figure 2: All Legal Moves for a Knight on an 8×8 Chessboard

Implementing Knight's Tour

Whereas the breadth first search algorithm discussed in the previous section builds a search tree one level at a time, a depth first search creates a search tree by exploring one branch of the tree as deeply as possible.

In this section we will look at two algorithms that implement a depth first search. The first algorithm we will look at directly solves the knight's tour problem by explicitly forbidding a node to be visited more than once. The second implementation is more general, but allows nodes to be visited more than once as the tree is constructed.

Knight's Tour Analysis

the knight's tour problem as we have implemented it so far is an exponential algorithm of size $ O(k^N) $, where N is the number of squares on the chess board, and k is a small constant.

The problem with using the vertex with the most available moves as your next vertex on the path is that it tends to have the knight visit the middle squares early on in the tour. When this happens it is easy for the knight to get stranded on one side of the board where it cannot reach unvisited squares on the other side of the board. On the other hand, visiting the squares with the fewest available moves first pushes the knight to visit the squares around the edges of the board first. This ensures that the knight will visit the hard-

to-reach corners early and can use the middle squares to hop across the board only when necessary. Utilizing this kind of knowledge to speed up an algorithm is called a heuristic. Humans use heuristics every day to help make decisions, heuristic searches are often used in the field of artificial intelligence.

General Depth First Search

The knight's tour is a special case of a depth first search where the goal is to create the deepest depth first tree, without any branches. The more general depth first search is actually easier. Its goal is to search as deeply as possible, connecting as many nodes in the graph as possible and branching where necessary.

It is even possible that a depth first search will create more than one tree. When the depth first search algorithm creates a group of trees we call this a **depth first forest**. As with the breadth first search our depth first search makes use of predecessor links to construct the tree. In addition, the depth first search will make use of two additional instance variables in the `vertex` class. The new instance variables are the discovery and finish times. The discovery time tracks the number of steps in the algorithm before a vertex is first encountered. The finish time is the number of steps in the algorithm before a vertex is colored black. As we will see after looking at the algorithm, the discovery and finish times of the nodes provide some interesting properties we can use in later algorithms.

Depth First Search Analysis

The general running time for depth first search is as follows. The loops in `dfs` both run in O(V), not counting what happens in `dfsvisit`, since they are executed once for each vertex in the graph. In `dfsvisit` the loop is executed once for each edge in the adjacency list of the current vertex. Since `dfsvisit` is only called recursively if the vertex is white, the loop will execute a maximum of once for every edge in the graph or O(E). So, the total time for depth first search is O(V+E).

Topological Sorting

A topological sort takes a directed acyclic graph and produces a linear ordering of all its vertices such that if the graph G contains an edge (v,w) then the vertex v comes before the vertex w in the ordering. Directed acyclic graphs are used in many applications to indicate the precedence of events. Making pancakes is just one example; other examples include software project schedules, precedence charts for optimizing database queries, and multiplying matrices.

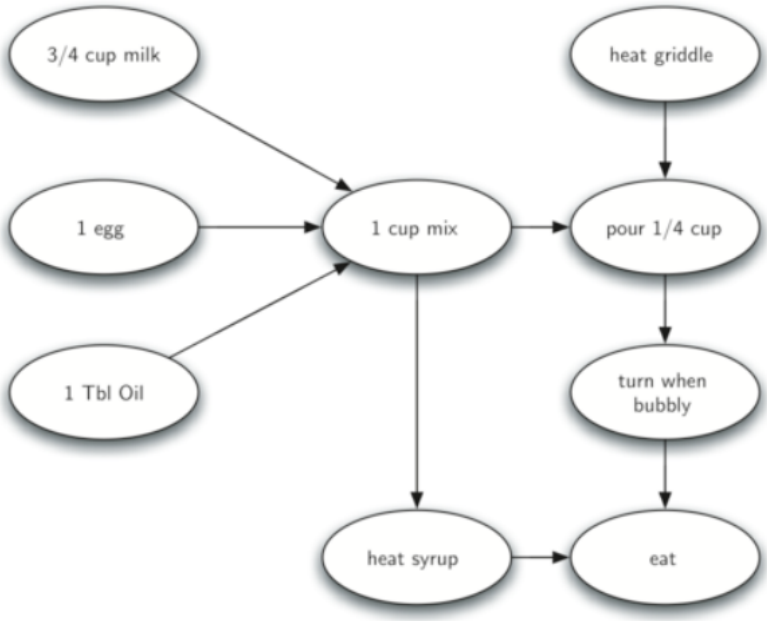


Figure 27: The Steps for Making Pancakes

The topological sort is a simple but useful adaptation of a depth first search. The algorithm for the topological sort is as follows:

1. Call `dfs(g)` for some graph `g`. The main reason we want to call depth first search is to compute the finish times for each of the vertices.
2. Store the vertices in a list in decreasing order of finish time.
3. Return the ordered list as the result of the topological sort.

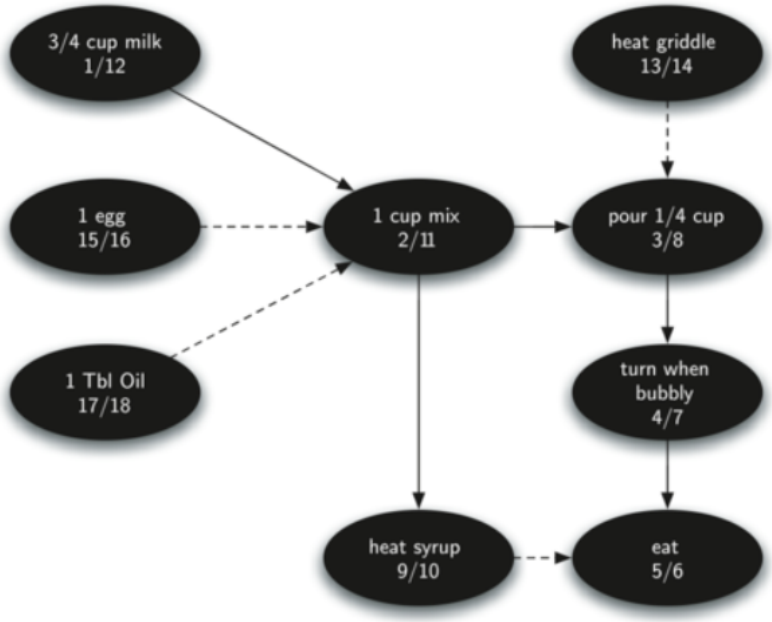


Figure 28: Result of Depth First Search on the Pancake Graph

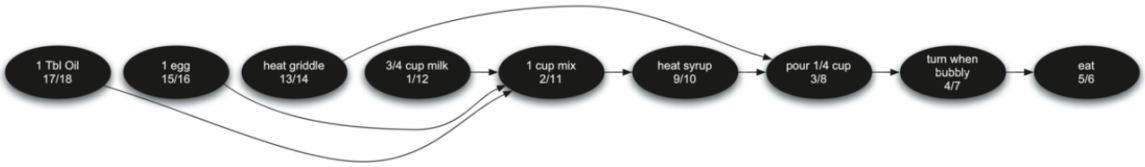


Figure 29: Result of Topological Sort on Directed Acyclic Graph

Strongly Connected Components

One graph algorithm that can help find clusters of highly interconnected vertices in a graph is called the strongly connected components algorithm (**SCC**). We formally define a **strongly connected component**, C, of a graph G, as the largest subset of vertices $C \subset V$ such that for every pair of vertices $v,w \in C$ we have a path from v to w and a path from w to v.

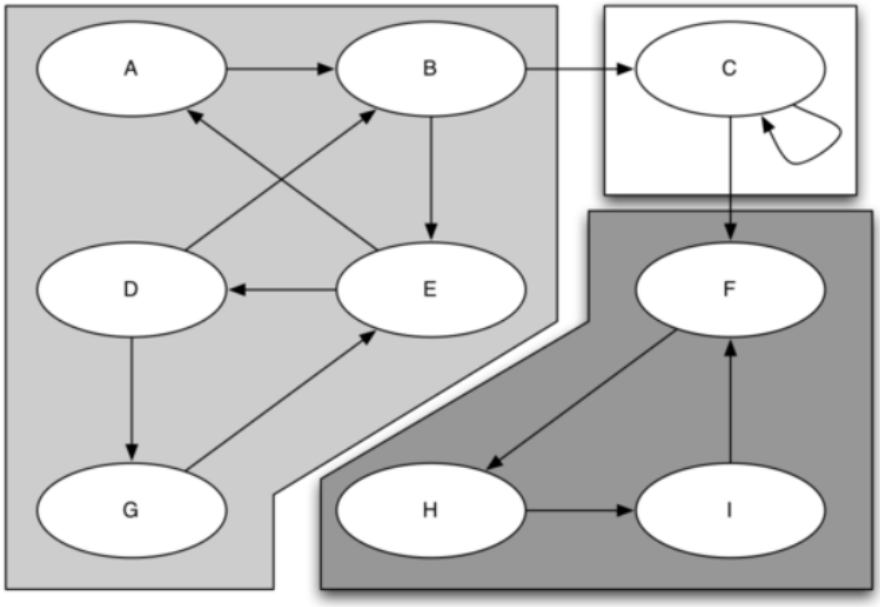


Figure 31: A Directed Graph with Three Strongly Connected Components

Once the strongly connected components have been identified we can show a simplified view of the graph by combining all the vertices in one strongly connected component into a single larger vertex.

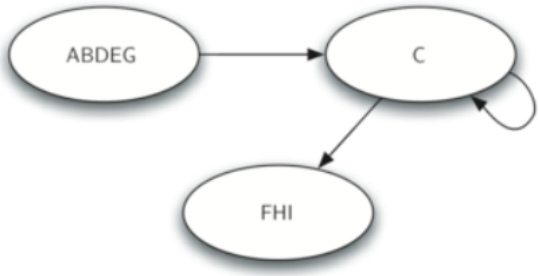


Figure 32: The Reduced Graph

The transposition of a graph G is defined as the graph G^T where all the edges in the graph have been reversed. That is, if there is a directed edge from node A to node B in the original graph then G^T will contain and edge from node B to node A.

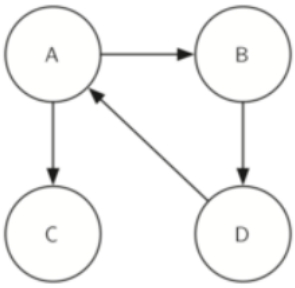


Figure 33: A Graph G

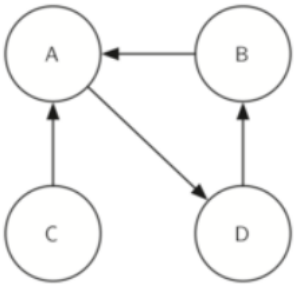


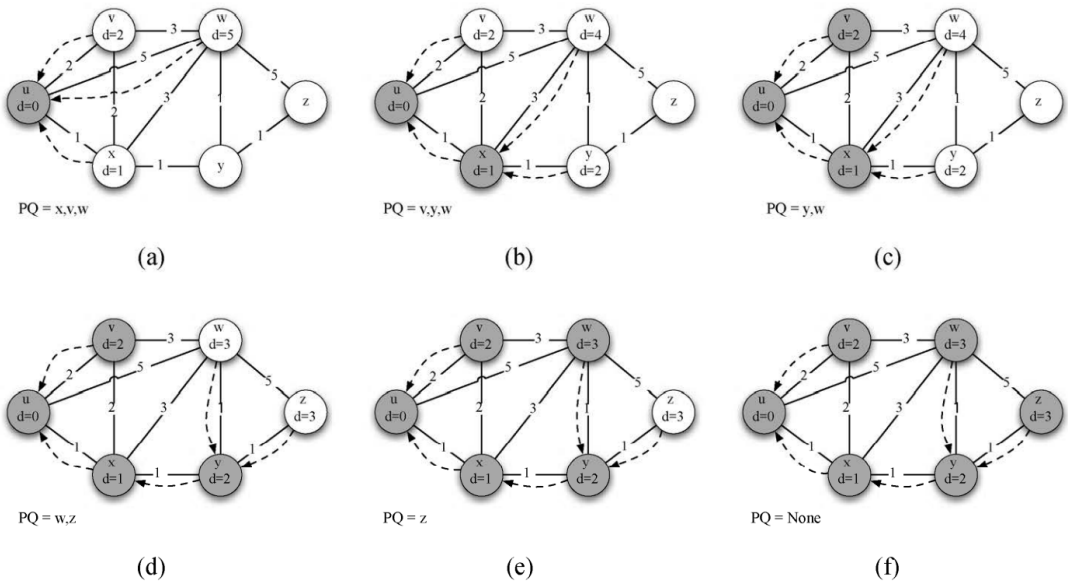
Figure 34: Its Transpose G^T

We can now describe the algorithm to compute the strongly connected components for a graph.

1. Call `dfs` for the graph GG to compute the finish times for each vertex.
2. Compute G^T.
3. Call `dfs` for the graph GTGT but in the main loop of DFS explore each vertex in decreasing order of finish time.
4. Each tree in the forest computed in step 3 is a strongly connected component. Output the vertex ids for each vertex in each tree in the forest to identify the component.

Shortest Path Problems

Dijkstra's Algorithm



It is important to note that Dijkstra's algorithm works only when the weights are all positive. You should convince yourself that if you introduced a negative weight on one of the edges to the graph that the algorithm would never exit.

Analysis of Dijkstra's Algorithm

Finally, let us look at the running time of Dijkstra's algorithm. We first note that building the priority queue takes O(V) time since we initially add every vertex in the graph to the priority queue. Once the queue is constructed the `while` loop is executed once for every vertex since vertices are all added at the beginning and only removed after that. Within that loop each call to `delMin`, takes O(logV) time. Taken together that part of the loop and the calls to delMin take O(Vlog(V)). The `for` loop is executed once for each edge in the graph, and within the `for` loop the call to `decreaseKey` takes time O(Elog(V)). So the combined running time is O((V+E)log(V)).

Prim's Spanning Tree Algorithm

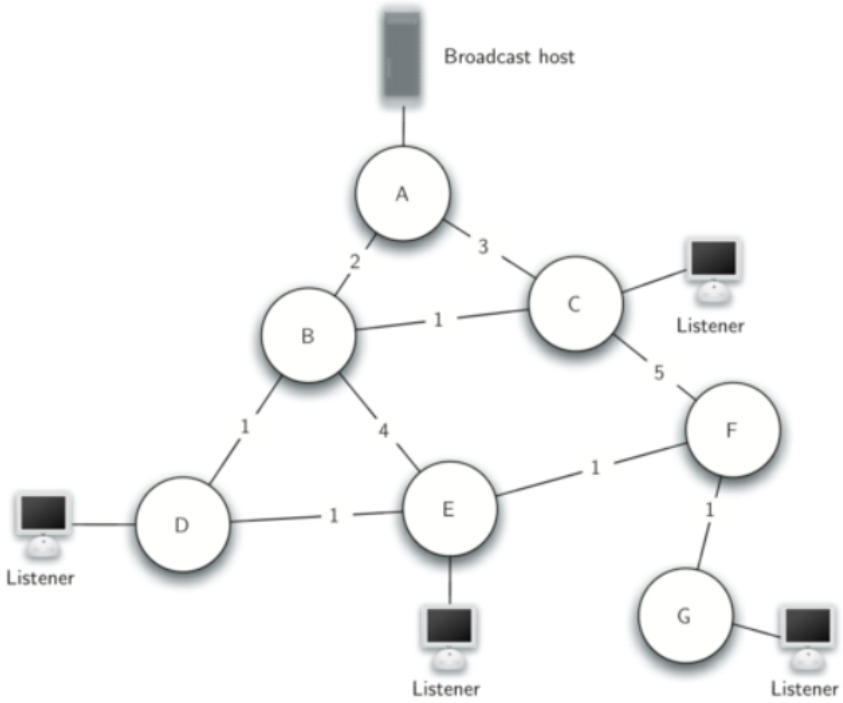


Figure 9: The Broadcast Problem

A brute force solution is for the broadcast host to send a single copy of the broadcast message and let the routers sort things out. In this case, the easiest solution is a strategy called **uncontrolled flooding**. The flooding strategy works as follows. Each message starts with a time to live (`ttl`) value set to some number greater than or equal to the number of edges between the broadcast host and its most distant listener. Each router gets a copy of the message and passes the message on to *all* of its neighboring routers. When the message is passed on the `ttl` is decreased. Each router continues to send copies of the message to all its neighbors until the `ttl` value reaches 0. It is easy to convince yourself that uncontrolled flooding generates many more unnecessary messages than our first strategy.

The solution to this problem lies in the construction of a minimum weight **spanning tree**. Formally we define the minimum spanning tree T for a graph G=(V,E) as follows. T is an acyclic subset of E that connects all the vertices in V. The sum of the weights of the edges in T is minimized.

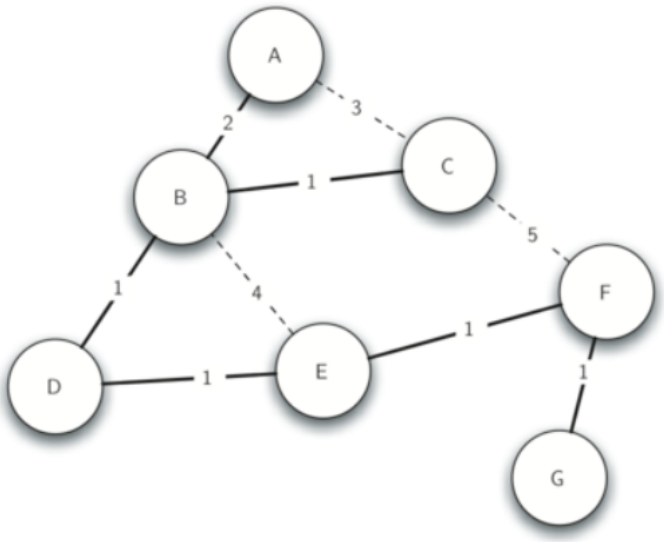


Figure 10: Minimum Spanning Tree for the Broadcast Graph

The algorithm we will use to solve this problem is called Prim's algorithm. Prim's algorithm belongs to a family of algorithms called the "greedy algorithms" because at each step we will choose the cheapest next step. In this case the cheapest next step is to follow the edge with the lowest weight. Our last step is to develop Prim's algorithm.