Computer Science Compendium

Scott Tolksdorf

December 28, 2013

Contents

1 Introduction

2 Sorting

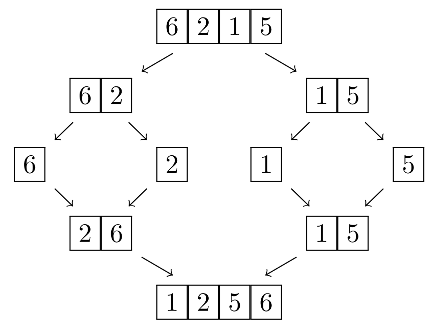
2.1 MergeSort

CLRS pg.31

MergeSort is a **Divide and Conquer algorithm**, it's recursive and can be parallelized well between multiple processors. It has **Stable Sorting**, so items with the same value perserve their order from the original list. **Best/Worst Case** $O(n \log n)$ and **Aux. Space** of $\Omega(n)$. **Best used** when accessing data sequentially is important, eg. parallel/external sorting. On average slower then HeapSort and QuickSort.

MergeSort splits its list until it's down to 1 element, then rejoins them recursively inorder.

```
function MergeSort(list)
    if length(list) <= 1
        return list
    split list -> A, B
    A = MergeSort(A), B = MergeSort(B)
    loop through each A, B and merge in sorted order
    return result
```



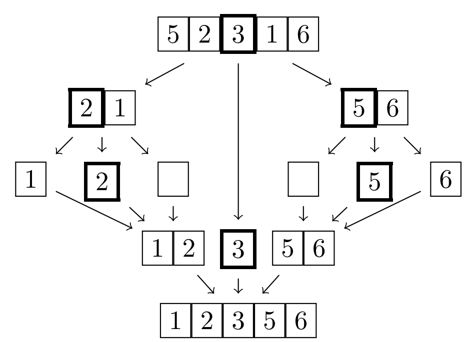
2.2 QuickSort

CLRS pg.170

QuickSort is a **Divide and Conquer algorithm**, it's recursive and can be parallelized well between multiple processors. It has **Non-Stable Sorting**, so items with the same value do not perserve their order from the original list. **Best Case** $O(n \log n)$, **Worst Case** $O(n^2)$ and **Aux. Space** $\Omega(\log n)$. **Best used** when speed is top priority. Can have cases where running time is very slow, bad for very large data sets and possible for attacks.

QuickSort chooses a pivot, then creates two new lists, one containing elements less then the pivot and one greater then the pivot. Recursively applies this to the new lists. Rejoins them with the pivot.

```
function QuickSort(list)
    if length(list) <= 1
        return list
    select and remove a pivot pivot from list
    create empty lists -> left, right
    for each x in array
        if x <= pivot
            append x to left
        else
            append x to right
    return join(QuickSort(left), pivot, QuickSort(right))
```



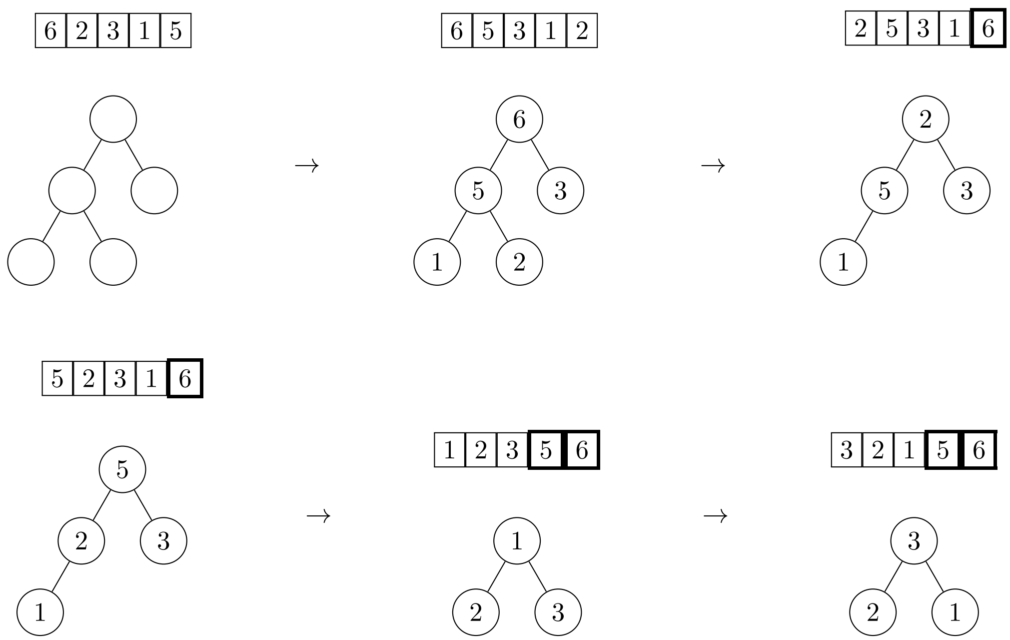
2.3 HeapSort

CLRS pg.151

Heapsort is a comparison-based sorting algorithm. It has **Non-Stable Sorting**, so items with the same value do not perserve their order from the original list. **Best/Worst Case** $O(n \log n)$ and **Aux. Space** $\Omega(1)$. **Best used** when space is a concerned, eg. embedded systems. On average runs slower then QuickSort, but faster then MergeSort.

HeapSort first builds a **heap** out of the data, the iteratively removes the largest elements from the heap and stores it in an array, then rebuilds the heap. Repeat until the heap is exhausted.

```
function HeapSort(A){
    build heap
    for length(A)
        remove and store largest, A[0] -> result
        replace with last in heap
        reduce heap size by 1
        heapify
    return result
}
```



3 Data Structures

3.1 Stacks

CLRS pg.232

A Stack is a dynamic set of data. It follows a **Last-In First-Out** ordering system. Imagine a stack of plates, where you can only add to the stack and the top and only remove plates from the top. The data structure uses **Push** to insert new data and **Pop** to remove data. It uses a **Top** pointer to keep track of the data structures position in memory.

Stacks are used ubiquitously. Used in algorithms in converting decimal numbers to binary, evaluating math expressions, maze back-tracking solutions. Most languages used stacks to resolve operations.

```
function Push(x){
    S.top = S.top + 1
    S[S.top] = x
}

function Pop(){
    S.top = S.top - 1
    return S[S.top + 1]
}
```

$$\begin{array}{|c|c|c|c|c|}\hline 4&8&9&\ &\ \\\hline \multicolumn{1}{c}{}&\multicolumn{1}{c}{}&\multicolumn{1}{c}{\uparrow}&\multicolumn{1}{c}{}&\multicolumn{1}{c}{}\end{array} \quad \to \quad \begin{array}{|c|c|c|c|c|}\hline 4&8&\ &\ &\ \\\hline \multicolumn{1}{c}{}&\multicolumn{1}{c}{\uparrow}&\multicolumn{1}{c}{}&\multicolumn{1}{c}{}&\multicolumn{1}{c}{}\end{array} \quad \to \quad \begin{array}{|c|c|c|c|c|}\hline 4&8&6&8&\ \\\hline \multicolumn{1}{c}{}&\multicolumn{1}{c}{}&\multicolumn{1}{c}{}&\multicolumn{1}{c}{\uparrow}&\multicolumn{1}{c}{}\end{array}$$

Pop() → 9 Push(6); Push(8)

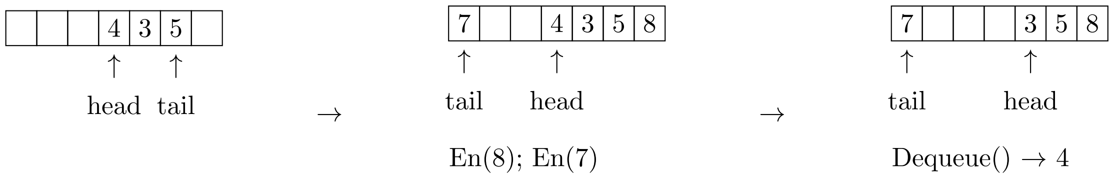
3.2 Queues

CLRS pg.232

A Queue is a dynamic set of data. It follows a **First-In First-Out** ordering system. Imagine a checkout at a store, you enter at the end of the queue and only leave at the front. The data structure uses **Enqueue** to insert new data and **Dequeue** to remove data. It uses a **Head** and **Tail** pointer to keep track of the data structures position in memory. To maximize memory use, queues are sometimes implemented cicular in nature

Queues are primarily used as **Priority Queues**, where elements are added with a priority and removed in order of their priority. Many algorithms; including Dijkstra's, and A8 use priority queues to track the most efficient ways to solve their problem.

```
function Enqueue(x){
    Q[Q.tail] = x
    if Q.tail == Q.length
        Q.tail = 1
    return Q.tail = Q.tail + 1
}

function Dequeue(){
    x = Q[Q.head]
    if Q.head == Q.length
        Q.head = 1
    else
        Q.head = Q.head + 1
    return x
}
```

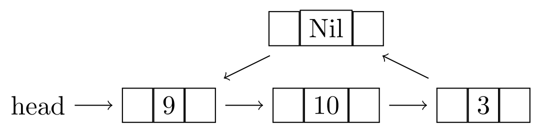


3.3 Linked Lists

CLRS pg.236

In Linked Lists data is stored linearly and sequenitially. Each node contains a pointer to the next node in the list. In a **Double Linked List** each node also has a pointer to it's parent. Linked lists are better then dynamic arrays that their inserts and deletes always take constant time, and since it uses pointers, the data structure can be spread across memory. However, you can not random access a linked list, in order to get an element you must transverse the list to get there. Also linked lists use slightly more memory per node to track the pointers.

```
function Search(k){
    x = L.head
    while x != Nil && x.key != k
        x = x.next
    return x
}
```




```
function Insert(x){
    x.next = L.head
    if L.head != Nil
        L.head.prev = x
    L.head = x
    x.prev = Nil
}

function Delete(x){
    x.prev.next = x.next
    x.next.prev = x.prev
}
```

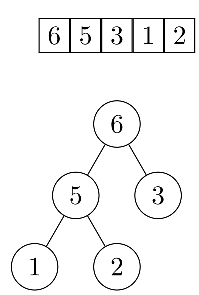
3.4 Heaps

CLRS pg.151

Heaps are a datra structure which create a tree-like structure using sequential memory. They are defined by a **Heap Property** which ditacte how the heap is formed, eg. max-heap property: Parent nodes are always larger then their children. The internal structure of a heap may be larger unordered, but the heap property ensures the top of the heap is the max element of the data structure. Heaps are useful for any scenerio where simply knowing the largest (or smallest) element is needed, eg. **Priority Queues**. Using properties of sequiential access, node navigation can be done using math on the node's index.

Heapify ensures that the node at the given index is following the heap property, if not it will "float down" the data structure until it does. When extracting the max value from a heap, replace it with the last value in the heap, and Heapify from the top.

```
Parent(i) => floor(i/2)
Left(i) => 2i
Right(i) => 2i + 1
function BuildHeap(A){
    for i = floor(A.length/2) -> 1
```



```
        Heapify(A, i)
}
function Heapify(A, i){
    if A[Left(i)] > A[i]
        largest = Left(i)
    else
        largest = i
    if A[Right(i)] > A[largest]
        largest = Right(i)
    if largest != i
        swap A[i] with A[largest]
        Heapify(A, largest)
}
function Extract(A){
    max = A[0]
    A[0] = A[A.length]
    Heapify(A,0)
    return max
}
```

3.5 Hashtables

CLRS pg.257 Hashtables is a data structure that map keys to values in an associated array using a **Hash Function**. A hash function should be choosen that limits clumping

A hash tables **Load Factor** is the ratio of the number of elements in the data structure against it maximum size. When a hash table has a high load factor it has to be resized in order to avoid collisions. Resizing can be done one of two ways; **Rebuilding** the entire hash table at once using a bigger size, or **Incremental** where a new table is allocated, new values are only inserted into the new table, as well as moving over n other elements from the old array. When the old table is empty it's removed from memory.

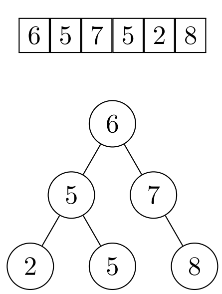
A technique to speed up deletes is to use **Tombstones**. A tombstone is a special entry that is inserted in place of an element you wish to delete. It's ignored during lookups, and replaced during inserts.

Collisions resolution could be done one of several ways: **Chaining**, **Open Addressing**, **Cuckoo**

3.6 Binary Trees

CLRS pg.287 Binary Trees are family of data structures efficient at lookups. Data is stored that $Right \leq Node \leq Left$. BSTs are **unbalanced**, meaning one part of the tree may become much larger then the other, hurting the efficiency. **Search** and **Insert** are $O(n \log n)$, **Delete** is $O(n)$.

```
function Search(node, val){
    if node == Nil or node.key == val
        return node
    if val > node.key
        return Search(node.left, val)
    else
        return Search(node.right, val)
}

//TODO: chage to recursive insert
function Insert(T, val){
    x = T.root
    while x != Nil
        if val > x.key
            x = x.left
```

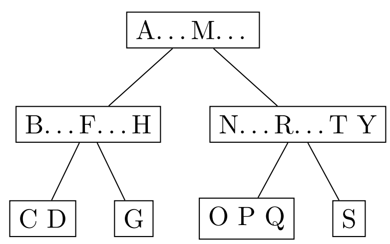


```
        else
            x = x.right
    if val > x.parent.key
        x.left = val
    else
        x.right = val
}

function Delete(T, node){
    if node.left == Nil and node.right == Nil

}
```

3.7 B-Trees

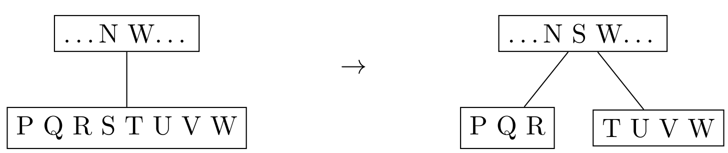
CLRS pg.488

B-Trees are an extreme form of **k-ary Trees**, where k is very large. Each node has a minimum and maximum number of children it can have; $t-1$ and $2t-1$ respectively, where t is the **order** of the tree. B-Trees are achieve extremely large tree structures with a small height when the order of the tree is high, number of nodes is $2t^h - 1$ where h is the height of the tree. This is very **useful for when lookups are very expensive** and when lookups are best done in **large sequiential blocks**. Mostly used for storing data on **physical storage**.



B-tree with order of 2

Inserting a node is slightly more complicated. When inserting a new value into a node that is full, we must split the node into smaller chunks. With deleting a value, you may have to rearrange a node's children.

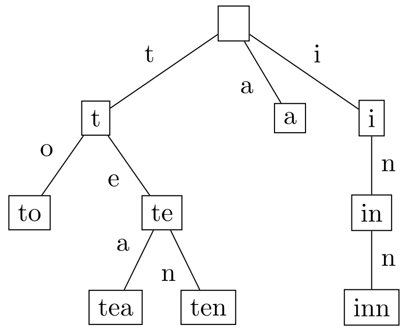


Splitting a child node into 2 on S

3.8 Tries

A Trie is an unique tree data structure, where a nodes location within the tree determines it's value. Each edge of the tree has a value given to it and as you traverse the tree to build the value of the node. Very useful for **storing strings** and **binary values**.

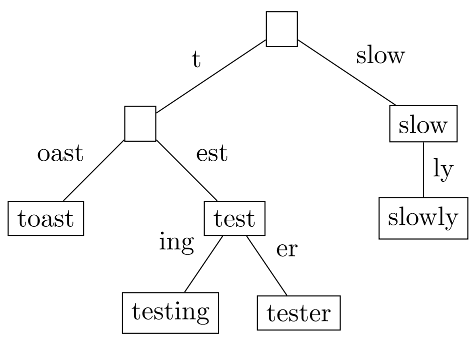
Tries have a **faster worst case than hash tables** (no rebuilds), they can produce **alpha-ordering** which may be useful for some applications like spell checking, and there is **no chance for collisions**. However, Tries may need more memory, long entires can create useless nodes, and they require **lots of random access memory** to operate. Tries are unique in that $Insert, Search, Delete \in O(M)$ where M is the length of the longest node value.



3.9 Radix Tree

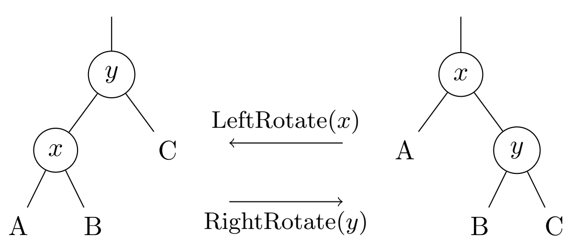
CLRS pg.304 Radix Tress are space optimized **Trie data strcutures** where each node with only one child is merged with it's parent, and it's edge is updated accordingly. They are much more efficient for small sets (especially if the strings are long) and for sets of strings that share long prefixes. $Insert, Search, Delete \in O(k)$ where k is the length of the longest edge key.

Radix Trees are used in **IP Routing**, where the ability to contain large ranges of values with a few exceptions is particularly suited to the hierarchical organization of IP addresses. They are useful for **Inverted Indexes** in text documents for very fast searching through the document.



3.10 Self-Balancing Trees

CLRS pg.308 A Self-Balancing Tree is a binary search tree that automatically keeps its height small during inserts and deletes. The time it takes for an operation on aBinary Search Tree is dependant on it's height, which in extreme cases can result in long and unexpected run times. Self-Balancing Trees attempt to correct this by using **Tree Rotations** to maintain tree properties that ensure a predictable height of the tree. Whenever a node is inserted or deleted, we need to check the node's children for consistency of the height property. Search, Insert, and Deletes all take $O(\log n)$ time in both the average and worst cases. The two most common Self-Balancing Trees are **AVL Trees** and **Red-Black Trees**.



3.10.1 AVL Trees

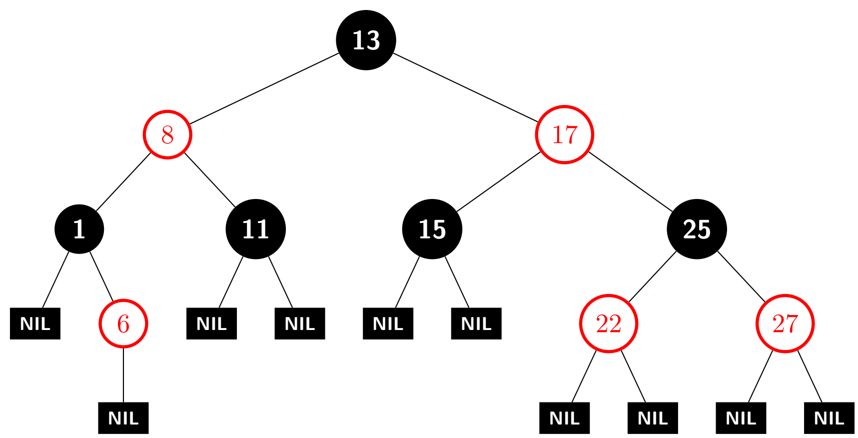
The height property of an AVL Tree is that **the heights of the two child subtrees of any node differ by at most one**. AVL Trees are more rigidly balanced than Red-Black Trees, leading to **slower inserts and deletes** but **faster search**.

```
function balanceFactor(node) => height(node.left) - height(node.right)
function Insert(node, val){
    if node is Nil
        node.key = val
        return node
    if val > node.key
        node.left = Insert(node.left, val)
        if balanceFactor(node) > 1
            if val > node.left.key
                LeftRotate(node)
            LeftRotate(node)
    if val <= node.key
        node.right = Insert(node.right, val)
        if balanceFactor(node) < -1
            if val < node.right.key
                RightRotate(node)
            RightRotate(node)
    return node
}

function Delete(z){
    //finish later, very complex
}
```

3.10.2 Red-Black Trees

Red-Black Trees are less rigidly balanced than AVL Trees, leading to **slower search** but **faster inserts and deletes**. The height properties of a Red-Black Tree are as follows:

1. A node may be red or black
2. All leaves are black
3. Every red node must have two black child nodes
4. Every path from the root to a leaf must have the same number of black nodes.



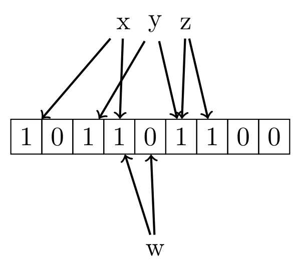

```
//TODO: Make recursive
function Insert(z){
    find leaf where node should be attached -> y
    z.parent = y
    if z.key > y.key
        y.left = z
    else
        y.right = z
    z.color = red

    //Maintain height property
    while z.parent is red
        if z.parent is a left-child
            if z.uncle is red
                z.parent.color = black
                z.uncle.color = black
                z.grandparent.color = red
                z = z.grandparent
            if z.uncle is black and z is a right-child
                z = z.parent
                LeftRotate(z)
            if z's uncle is black and z is a left-child
                z.parent.color = black
                z.grandparent.color = red
                RightRotate(z.grandparent)
        else
            Same as above but switch left and right rotate
    root.color = black
}

function Delete(z){
    //finish later, very complex
}
```

3.11 Bloom Filter

A Bloom Filter is a space efficient probalistic data structure that's used to test whether an element is in a set. False positives are possible, but **false negatives** are not. An empty Bloom Filter is a bit array of m

bits all set to 0. Whenever you add an element to the set use k hash functions and flip all the corresponing entires to 1. To test if an entry belongs to the set, simply hash it and check the corresponding bit entires. If any of them are 0 the element is not part of the set. The probability of a **false positive** is $(1 - e^{-kn/m})^k$, where n is the number of elements encoded in the set.

Bloom Filters are used in Chrome to detect bad urls from a list, in BigTable to remove unnecessary table lookups, and in url shortners.



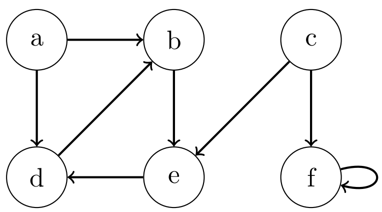
x, y, z are in the set, where w is not

4 Graph Algorithms

4.1 Representation

CLRS pg.308

A graph can either be directed or undirected. **Undirected Graphs** simply have links betwen two nodes, where **Directed Graphs** have one-way links between nodes. We can represent a graph in one of two ways; Adjacency Lists and Adjacency Matricies. **Adjacency Lists** are useful for when the graph is **sparse**, few edges compared to nodes. **Adjacency Matricies** are useful for when the graph is **dense**, many edges compared to nodes.



Directed Graph

$a \to b, d$
$b \to e$
$c \to e, f$
$d \to b$
$e \to d$
$f \to f$

Adjacency List

	a	b	c	d	e	f
a	0	1	0	1	0	0
b	0	0	0	0	1	0
c	0	0	0	0	1	1
d	0	1	0	0	0	0
e	0	0	0	1	0	0
f	0	0	0	0	0	1

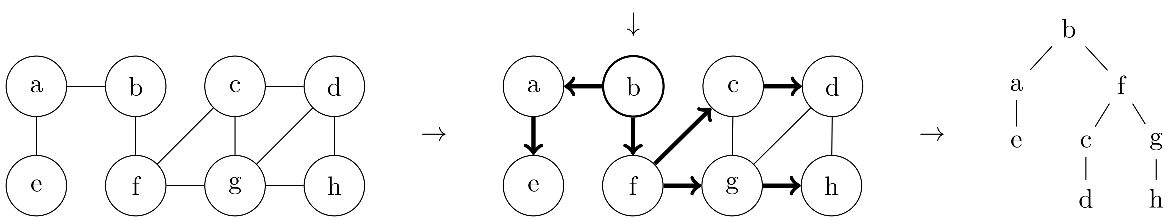
Adjacency Matrix

4.2 Breadth First Search

CLRS pg.594

Breadth-First Search is a strategy for searching a graph. It's starts at the root node, visits all of it's neighbours, then each of them, visits their neighbours. As it runs it produces a **Breadth-First Tree**, where the depth of the tree indicates how far apart the root and that node is in the graph.

Aux. Space is $\Omega(V + E)$ using an Adj. List and $\Omega(V^2)$ using an Adj. Matrix. **Worst Case Complexity** is $O(V+E)$, where $O(E)$ may vary between $O(V)$ and $O(V^2)$ depending on how dense the graph is. Used for **finding the shortest path** between nodes and asserting if two nodes are connected to eachother.



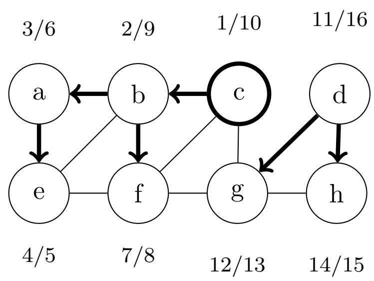

```
BFS(start, goal){
    create queue Q
    create list R
    Q.enqueue(start)
    R.push(start)

    while Q is not empty
        t = Q.dequeue()
        t.visited = true
        if t is goal
            return R
        for all neighbours v of t
            if v is not in R || v.visited == false
                v.visited = true
                R.push(v)
                Q.enqueue(v)
```

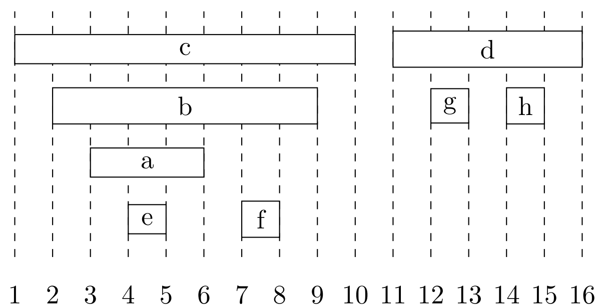
```
}
```

4.3 Depth First Search

CLRS pg.603 Depth-First Search is a strategy for searching a graph. It's starts at the root node and explores as deep as possible before backtracking. By tracking and applying a **timestamp** (increments once a node has been visitied) to each node as it's visited for the first and last time, you can sort the nodes by **preorder** and **postorder**, respectively. Postordering the nodes provides you with a **Topological Sort** of the graph.

Used in AI for limiting the depth of decisions trees and evaluating branches based on heurstics, maze generation, and topological sorting.



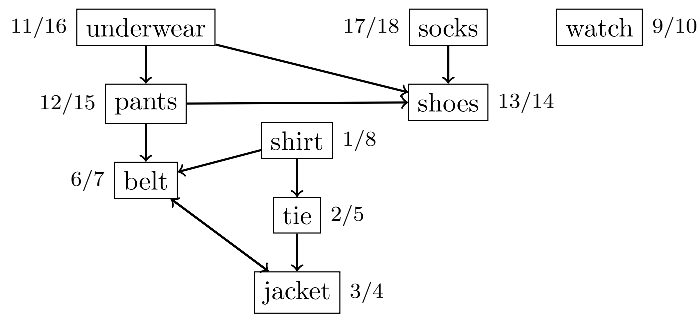


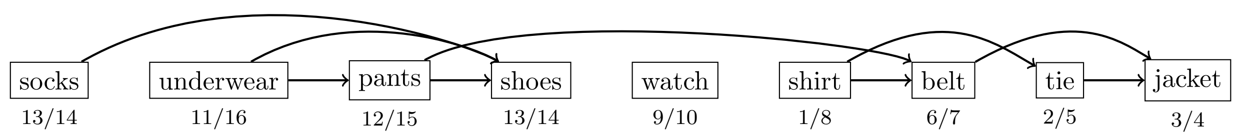

```
DFS(start){
    create stack S
    S.push(start)
    timestamp = 0
    while S is not empty
        node = S.pop()
        node.begin = timestamp
        for all neighbours v of node
            if v.visited == false
                S.push(v)
        timestamp = timestamp + 1
        node.visited = true
        node.end = timestamp
}
```

4.4 Topological Sort

CLRS pg.613 The Toplogical Sort of a **directed graph** is a linear ordering of the nodes, such that every node comes before the node it's connected to. An example is that if each node is a task and the edges between nodes represent constriants on finishing tasks before starting another, topologically sorting this graph would ensure an order to complete the tasks which would provide no conflicts.

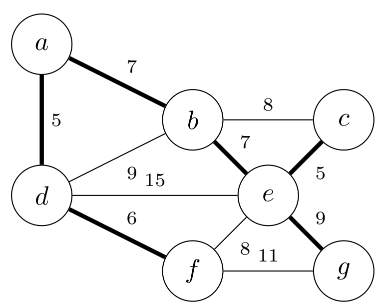
Topological Sort is a modified Depth-First Search alogithm, where we **postorder** the nodes, or sort the descending by the last when they were visited.





4.5 Minimum Spanning Tree

CLRS pg.625 A Minimum Spanning Tree is a subgraph of a graph that contains all nodes and has lowest combined weight of all of it's edges. It's possible to have multiple Minimum Spanning Trees per graph, if the edge weights are not unique.



Minimum Spanning Tree of a weighted graph

4.5.1 Prim's Alogithm

CLRS pg.634 Prim's Alogithm is a **Greedy Algorithm** using **Priority Queues**. It's running time is $O(E + V \log V)$ and best used for **dense graphs**. At each step it grows the tree by one node, selecting the smallest edge available.

```
PrimMST(G){
    add all nodes in G to Queue, Q with key = \infty
    while Q is not empty
        x = Q.extractMin()
        for all neighbours v of x
            if v is in Q and weight(x,v) < v.key
                v.parent = x
                v.key = weight(x,v)
}
```

4.5.2 Krushal's Alogithm

CLRS pg.631

Krushal's Alogithm is a **Greedy Algorithm** using **Disjoint Sets**. It's running time is $O(E \log V)$ and best used for **sparse graphs**. It first creates a set for each node, and then a set of all edges ordered by weight. For each edge, if both nodes aren't in the same set together, it merges them together and adds that edge to the solution

```
KrushalMST(G){
    make Result an empty set
    make a set for each node that contains it
    foreach (u, v) in G.Edges ordered by weight(u, v)
        if Find-Set(u) != Find-Set(v)
            add (u,v) to Result
            Join-Sets(u, v)
    return Result
}
```

4.6 Dijsktra's Algorithm

CLRS pg.658 Dijsktra's Algorithm finds the shortest path in a weighted graph from a single source. As Dijsktra's Algorithm tranverses the graph, it follows the path of lowest expected total distance. It keeps track of "how good" each branch is using a **Min-Priority Queue**. It uses a greedy algorithm to choose the next node for each branch and will switch to a different branch if it no longer is the fastest. The variable *distance* on each node in our code tracks the shortest path value to get to that node from the start.

It's running time is $O(E + V \log V)$.

```
Dijsktra(G, start, goal){
    create Priority Queue Q
    add all nodes to Q with distance of \infty
    start.distance = 0
    while Q is not empty
        current = Q.extractMin()
        if current is goal
            goal.parent = current
            return ReconstructPath(goal, empty list)
        for each neighbour of current
            temp = current.distance + weight(current, neighbour)
            if temp < neighbour.distance
                neighbour.distance = temp
                neighbour.parent = current
}
ReconstructPath(node, result){
    if node has parent
        ReconstructPath(node.parent, result)
    result.push(node)
    return result
}
```

4.7 A*

CLRS pg.308 The A* algorithm is a generalization of Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored by using an additional **heuristic** unique to the problem. The heuristic function helps guide the path in which the algorithm takes next. For example if the nodes have a coordinate location, the heuristic function could be the euclidean distance between the current node and the goal, making sure that the algorithm tries the closer node first.

Running time is $O(\log h^*(x))$, where $h^*(x)$ is the optimal heuristic function.

```
Dijsktra(G, start, goal){
    create Priority Queue Q
    add all nodes to Q with score of \infty
    start.score = 0
    while Q is not empty
        current = Q.extractMin()
        if current is goal
            goal.parent = current
            return ReconstructPath(goal, empty list)
        for each neighbour of current
            temp = current.score + weight(current, neighbour) + heuristic(neighbor,
                goal)
            if temp < neighbour.score
                neighbour.score = temp
                neighbour.parent = current
}
ReconstructPath(node, result){
    if node has parent
        ReconstructPath(node.parent, result)
    result.push(node)
    return result
}
```

4.8 k-Means Clustering

CLRS pg.308

4.9 Convex Hull

CLRS pg.1029

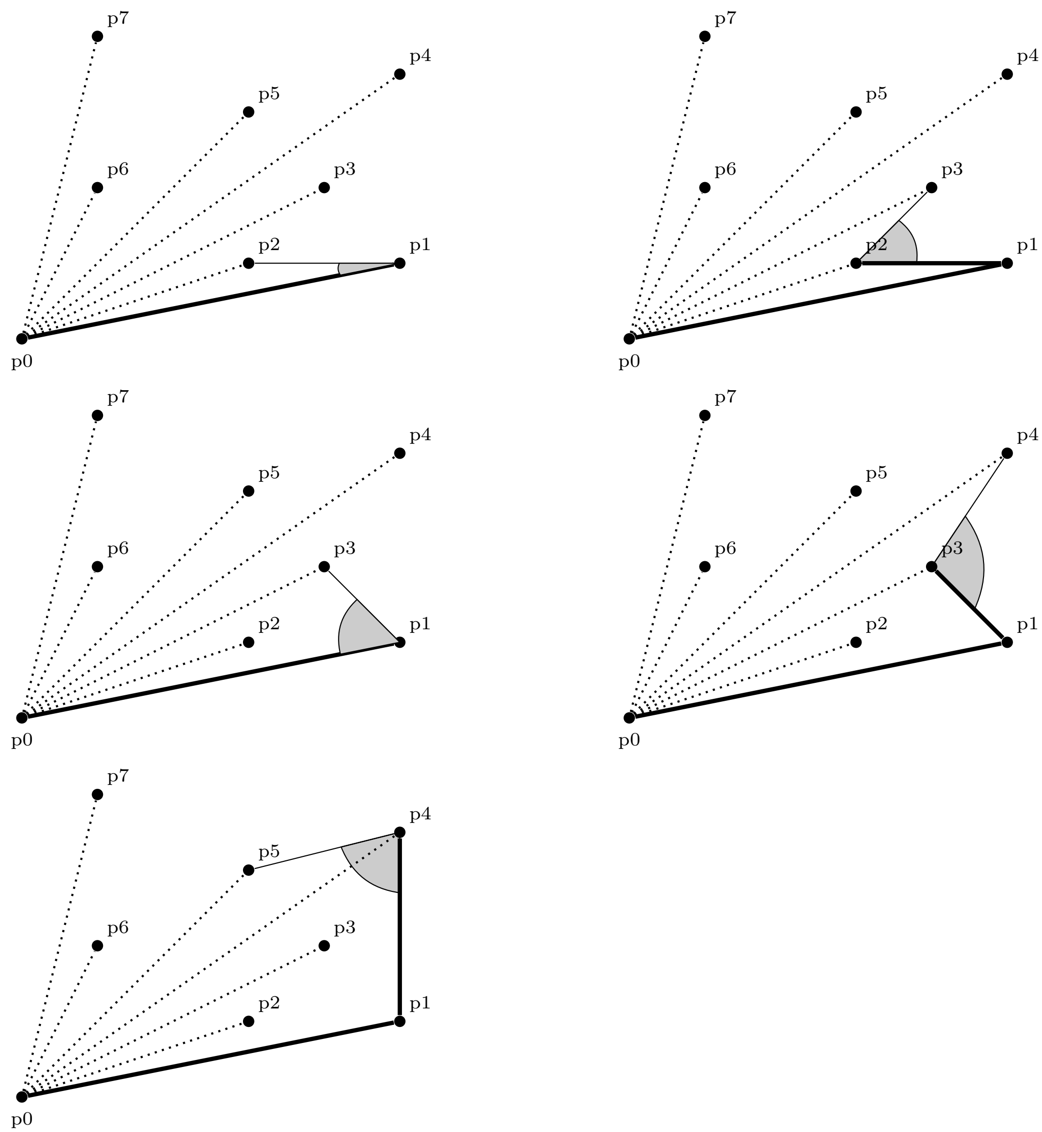
The Convex Hull of a set of points is the smallest polygon that contains all the points. Think of it like an elastic band wrapped around a number of pegs in a board. Used with bezier curves, pattern recogintion, image processing, statistics, and static code analysis.

4.9.1 Graham's Scan

Graham's Scan starts at the lowest point in the set and iterates over the points in a counterclockwise direction, relative to this point. At each step it calculates the angle between the last three points, if that angle is towards the left, we remove the last two points and try other ones. Running time of $O(n \log n)$

```
GScan(P){
    root = lowest point in P
    S = all points in P sorted by relative angle to root
    Hull = empty list

    Hull.push(root)
    Hull.push(S.pop())

    while S is not empty
        next = S.top()
        if angleDirection(Hull.secondLast(), Hull.last(), next) is left
```

```
            Hull.push(S.pop())
        else
            Hull.pop()
    return Hull
}
```



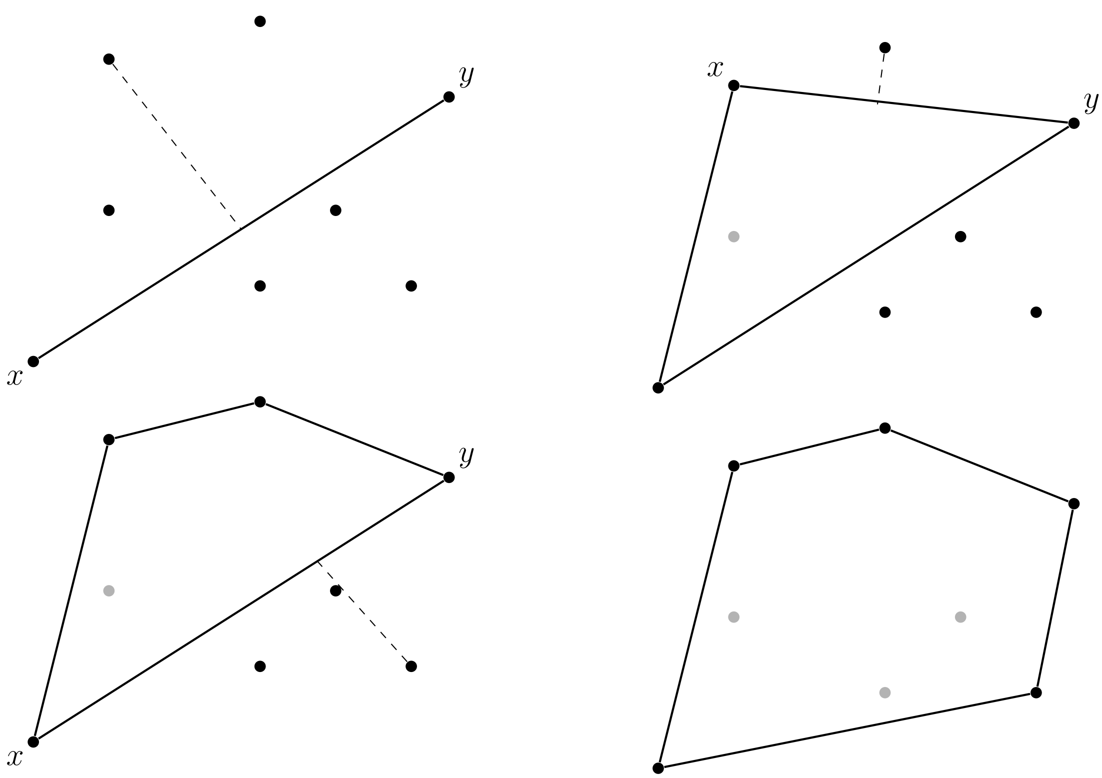
4.9.2 QuickHull

QuickHull is based off of QuickSort in that we use divide and conquer style. At each step we divide the current set by drawing a line between two points. For each side of the line we find the point farthest from the line, create a triangle between these three ppoints. We remove all the points inside the triangle from the working set, and repeat the same process on the two other triangle sideis

Just like QuickSort, QuickHull has a fast average case of $O(n \log n)$ and a Worst case of $O(n^2)$

```
QuickHull(P){
    x = right most point in P
    y = left most point in P
```

```
    Hull = P
    QH(x,y,Hull)
    return Hull
}
QH(x,y,Hull){
    s1 = all points in Hull above line(x,y)
    if s1 is not empty
        p1 = point farthest from line(x,y) in s1
        for each point r in triangle(x,y,p1)
            remove r from Hull
        QH(x,p1,Hull)
        QH(p1,y,Hull)
    s2 = all points in Hull below line(x,y)
    if s2 is not empty
        p2 = point farthest from line(x,y) in s2
        for each point r in triangle(x,y,p2)
            remove r from Hull
        QH(x,p2,Hull)
        QH(p2,y,Hull)
}
```



5 Dynamic Programming

5.1 Greedy Algorithms

5.2 Bottom-up Approach

5.3 Top-Down Approach

5.4 Bresenham's line algorithm

5.5 BoyerMoore String Search Algorithm

5.6 NP-Complete

5.6.1 Travelling Salesman

5.6.2 Knacksack Problem

6 Operating Systems

6.1 Scheduling

6.2 Threads

6.3 Semaphores

6.4 Deadlocks

7 Big Data

7.1 Hadoop

7.2 Databases?

7.3 MapReduce

8 Discrete Math

8.1 Expected Value

8.2 Conditional Probability

8.3 Counting

8.4 Combinatorics

9 Design Patterns

9.1 Abstract Factory

Creates an instance of several families of classes

9.2 Factory Method

Creates an instance of several derived classes

9.3 Prototype

A fully initialized instance to be copied or cloned

9.4 Singleton

A class of which only a single instance can exist

9.5 Adapter

Match interfaces of different classes

9.6 Decorator

Add responsibilities to objects dynamically

9.7 Facade

A single class that represents an entire subsystem

9.8 Flyweight

A fine-grained instance used for efficient sharing

9.9 Proxy

An object representing another object

9.10 Observer

A way of notifying change to a number of classes

9.11 Template Method

Defer the exact steps of an algorithm to a subclass

10 Cryptography

10.1 RSA

10.2 SHA-1

10.3 MD5

10.4 Salt

11 Web

11.1 SSL and HTTPS

11.2 EcmaScript 5

11.3 AJAX

11.4 Web RPC

11.5 Full Request

12 Questions

Prep for these, you shouldn't have to think about them hardest bug current project favourite Open source project cleverest solution what have you contributed to OSS