Chapter 1

Getting Started

Introduction

The Computer Science handbook is a handbook designed to explain algorithms and data structures in a way that anyone can understand. Many websites (eg Wikipedia) contain lengthy and wordy explanations that are full of technical jargon. We have tried our hardest to simplify all language to make it easy to read without any math or computer science background. We hope to share our knowledge with you and we ask only one thing from you. You must learn something before you leave!

Chapter 2

Fundamentals

Introduction

There are some fundamental topics about writing computer programs that we must learn before we can move on to the basic theory.

2.1 Data Types

We use closets or drawers to store our clothes and garages to store our cars. Similarly, we store different types of data in different kinds of data types. Most programming languages will support these data types and picking the right data type is important.

To define a variable of a data type we use the follow syntax:
datatype variable_name = init_val;

- datatype is the type of variable
- variable_name is the name of the variable used
- init_val is the initial value of the variable

Example:

```
int x = 3;
double y = -4.5;
```

x is an integer, and we initialize it with the value 3. y is a double floating point number (essentially a decimal number) and we initialize it with -4.5

2.1.1 Boolean

A boolean is stored in a bit that is either true or false. Booleans are usually used as flags to store if something is one one state or the other.

Data type	Number of bits	Range
bool	2 bits	true or false

2.1.2 Integer

An integer is any number that does not contain decimals. It is stored as binary number in memory. For example: 0, -5, 6 are integers.

Data type	Number of bits	Range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767
int	32 bits	2,147,483,648 to 2,147,483,647
long	64 bits	9,223,372,036,854,775,808 to 9,223,372,036,854,755,807

2.1.3 Character

A character is any letter or symbol. For example: 'a','B','8','!'.

Characters are usually stored as a number and then displayed as a character by the computer. An encoding is a computer translation from a number to a character. We store simple characters such as lower case and upper case letters, numbers and common punctuation, in 8 bits (0-255) and we can use it to encompass the English language. For these 8 bits, we use an encoding called ASCII. For example: '0' is 48 and 'B' is 66. There are other encodings like Unicode which uses more bits to convert to more languages such as Chinese or Russian. For the most part, we will just use ASCII.

Data type	Number of bits	Range
char	8 bits	256 bits

2.1.4 Float

A float is a decimal stored as binary in memory. We use scientific notation to represent the decimal. Scientific notation is a decimal number ¡ 10 times some exponent of 10. For example: 8.23 * 10^4 is in scientific notation. Decimals can be stored in 32 bits or 64 bits.

In a 32bit float, we have 1 bit for the sign (positive or negative), 23 bits for the significant figures (7 digits) and 8 bits for the exponent.

In a 64bit double we have 1 bit for the sign (positive or negative), 52 bits for the significant figures (16 digits) and 11 bits for the exponent.

Data type	Number of bits	Range
float	32 bits	3.4e038 to 3.4e+038
double	64 bits	1.7e308 to 1.7e+308

2.2 Runtime and Memory

Computers are super fast at making calculations compared to humans, but humans have much more "memory" than computers currently do. For example, computers can add two 100-digit numbers together much more quickly than any human possibly can. However, the human brain can contain much more memory than humans.

Brains have trillions of connections between neurons and the estimated storage capacity is about 2.5 petabytes. That is approximately 340 years of TV shows that you could watch! Computers on the other hand, have much less memory than human brains. Standard computers have around 8GB of RAM and some higher end machines may have 16-32GB. Although hard-disks can store terabytes of memory, we use RAM (flash memory) when analyzing computer memory because it is much faster. As an analogy, RAM can be thought of as grabbing an object in another room whereas disk memory is driving 20 min away to get that object.

The neurons and connections in our brains allow us to store an immense amount of information in our brains and allows us to easily recognize patterns much better than computers. For example it is very easy for us to identify different objects around us (e.g. we can easily differentiate between an apple and an orange), but it is difficult for a computer to do this for the countless number of objects in the world. However this is currently changing as more research is being done in "deep learning".

	Brain	Computer
Memory	2.5 petabytes (approx. 340 years of TV shows)	16GB RAM
Processing	60Hz	2.7GHz Quad Core (1,000,000,000

2.2.1 Limits

There are many different methods of implementing different things but most of the time we care most about implementations that are the fastest and use the least amount of memory. Let's go through some basic benchmarks about

computers that you should know. Adding two numbers takes a nanosecond (1 billionth of a second) for an average computer to process. For practical purposes, we'll assume that the average computer program can hold up to 1GB or RAM or about 250 million integers. We use RAM because it is flash memory and read/write is super fast in comparison to disk read/writes.

Memory	Operations (per seconds)
1 GB (250,000,000 ints)	100,000,000 operations

2.2.2 Big O Notation

When we compare how efficient algorithms or data structures are to another, we want to be able to describe them such that they can be quantified. We use something called Big O notation to do this. Many algorithms and data structures will have its time and space complexities depend on the size of the inputs. The Big O notation takes the largest factor of an input to compare computation times / memory usage. When we take the largest factor, we ignore smaller factors, coefficients and constants because they do not matter at very large values. For example: $O(3n^2 + 12n + 20)$ is simply $O(n^2)$ because it is the largest factor.

Here is a list of common Big O notations based on complexity:

Big O	Limit of N for 1 second (Standard processor)
O(1) Constant Time	runtime independent of N
O(log N) Sublinear time	a very big number
O(N) Linear Time	100,000,000
O(N log N)	5,000,000
$O(N^2)$ Quadractic Time	10000
$O(N^3)$ Cubic Time	450
$O(2^N)$ Exponential Time	27
O(N!) Factorial Time	11

Keep in mind that in a couple of years as technology improves, this chart will be outdated.

2.2.3 Runtime Analysis

Let us examine a function that takes in an array of size N and returns the maximum element. In a program we are usually concerned with two complexities: memory and time.

Code:

```
int findMax(int[] array){
  int maxVal = array[0]; //O(1) memory to store max element, O(1) time for
  for(int i=1;i<array.length;i++){ //O(n) loop runtime
       if(maxVal < array[i]){ //O(1) to compare runtime
          maxVal = array[i]; //O(1) to assign new value runtime
      }
  }
}
```

Memory: The array takes O(N) memory and storing the max value is O(1) more memory. but usually when analyzing programs, we ignore the input memory sizes and take into account additional memory that is required to produce the output. So the memory footprint of the function is O(1).

Time: The first assignment of maxVal takes O(1). The loop runs N-1 times and of each of those N-1 times it checks if the current array element is greater than the current max which takes O(1). If it is greater, then we reassign maxVal which is O(1). When analyzing time complexity, we usually take worst case so we have: O(1+(N-1)(2)) and this simplifies to O(N) since it is the largest factor.

Example

```
void zeroArrays(int[][] grid){
  for(int i=0; i<grid.length;i++){
    for(int j=0;j<grid[i].length;j++){
      grid[i][j] = 0;
    }
  }
}
```

Chapter 3

Recursion

Introduction

Next: Advanced Recursion

Recursion is process that repeats itself in a similar way. Anything that has its definition nested inside itself is considered to be recursive. For example GNU stands for GNU not Unix!. Expanding this acronym gives us ((GNU not Unix) not Unix!). As you can see this will go on forever and GNU's definition is nested inside itself so it is recursive. The Fibonacci sequence is also recursive: F(n) = F(n-1)+F(n-2). Inside the function F, we see two more F's!

In computer science infinite looping is bad because computers do not know how to terminate so we need some way to stop it. We will call a stopping point the base case. A base case is the case where the recursion will stop. Everything must eventually reduce to a base case. For the Fibonacci sequence, the base case is F(0) = 1 and F(1) = 1 and we can see that for all N¿1, the Fibonacci sequence will reach the base case.

So for something to be recursive in computer science, it needs:

- a recursive definition (contained in itself) and
- a base case that all valid inputs will eventually reach

Template for recursion:

```
recursion(parameter)
    if base\_case (parameter):
        stop
    recursion( operation(parameter) )
```

- recursion is the recursive function
- base_case is the check if the parameter has reached the base case
- operation reduces the parameters towards the base case

3.1 Advanced Recursion

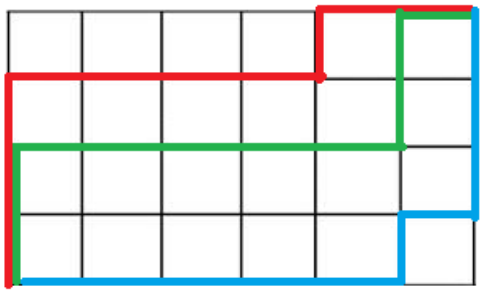
Prerequisites: Recursion

Next: Intermediate Recursion, Backtracking, Dynamic Programming

Recursion can sometimes be very difficult to wrap your head around because when you try to go deeper, it only keeps getting deeper. It is sometimes hard to figure out the recurrence relation, but once you do, the problem becomes simple to solve.

3.1.1 Number of paths

Let's say we have a grid of N rows and M columns. How many ways are there to get from the bottom left corner to the top right corner using the intersection points and only going upwards or rightwards?



We see that the only way to reach an intersection is to come from the left intersection or come from the bottom intersection. So the number of ways to reach an intersection is the number of ways to reach the intersection to the left plus the number of ways to reach the intersection on the bottom.

So more accurately, the number of ways to get to intersection N x M is the sum of the ways to get to the intersection N-1 x M and the intersection N x M-1. However, the number of ways to reach the intersection N-1 x M is the sum of the number of ways to the intersection N-2 x M and N-1 x M-1. As we can see, it is a recurrence relation.

The base case is the starting intersection in the bottom left hand corner (the intersection 1x1) which is the only way to reach that intersection. We can also see that there is only one way to reach the intersections on the left side. There is only one way to reach the intersections on the bottom side as well.

```
Let path(x,y) be the number of ways to get to the intersection at x and y

Base case:
paths(1,y) = 1
paths(x,1) = 1

Recurrence:
paths(x,y) = paths(x-1,y) + paths(x,y-1)

Example:
paths(3,5)
= paths(2,5) + paths(3,4)
= paths(1,5) + paths(2,4) + paths(2,4) + paths(3,3)
= 1 + paths(1,4) + paths(2,3) + paths(1,4) + paths(2,3) + paths(2,3) + patl
= 1 + 1 + paths(1,3) + paths(2,2) + 1 + paths(1,3) + paths(2,2) + paths(1,
= 1 + 1 + 1 + paths(1,2) + paths(2,1) + 1 + 1 + path(1,2) + paths(2,1) + 1
=
```

1	5	15	35	70	126	210
1	4	10	20	35	56	84
1	3	6	10	15	21	28
1	2	3	4	5	6	7
1	1	1	1	1	1	1

```
int paths(int n,int m){
    if(n==1||m==1){
        return 1;
    }
    return ways(n-1,m) + ways(n,m-1);
}
```

3.1.2 Towers of Hanoi

There are three poles and N disks where each disk is heaver than the next disk. In the initial configuration, the discs are stacked upon another on the first pole where the lighter discs are above the heavier discs. We want to move all the discs to the last pole with the following conditions:

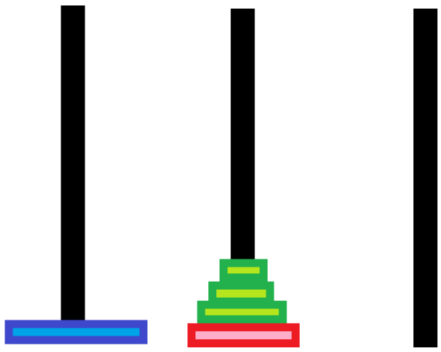
- Only be moved from one pole to another one at a time.
- The discs have to be stacked such that all the lighter discs are on top of the heavier ones.

Lets' try to make this problem simpler. To move N discs from the first pole the the last pole we need to move N-1 discs to the middle pole, then move the Nth disc to the last pole, and then move all N-1 discs from the middle pole back on to the last pole.

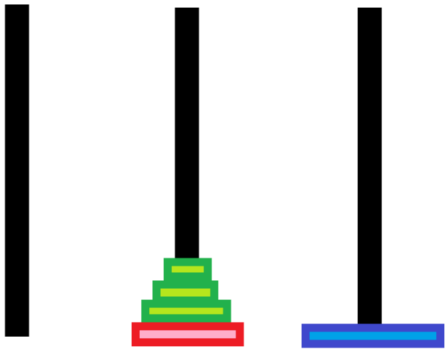
Let the starting pole be the first pole, the helper pole be the middle pole and the destination pole the third pole.

To move N discs from the starting pole to the destination pole:
Step 1:

We need to move N-1 discs from the starting pole to the helper pole.



Step 2:
We need to move the Nth disc from the starting pole to the destination



pole.

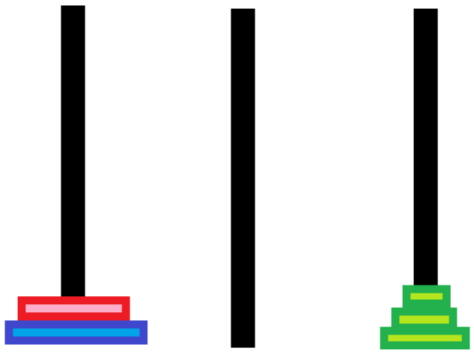
Step 3:
We need to move N-1 discs from the helper pole to the destination pole.

We can see that Step 2 is easy, all we have to do is move that one disc. But for Step 1 and Step 3, we have to move N-1 discs. How can we move N-1 discs to the middle pole?

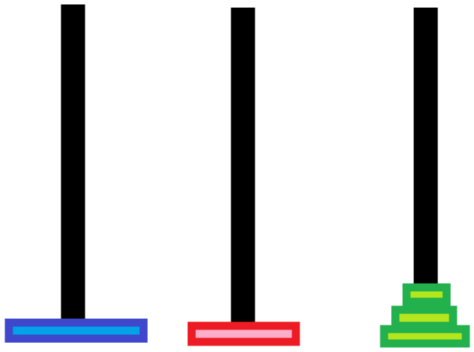
We see that we can use the same reasoning: we need to move N-2 discs to the third pole. Then we need to move the N-1 disc to the second pole and then move N-2 discs from the third pole to the second pole.

In this case, the start pole is the first pole, the helper pole is the third pole and the destination pole is the middle pole.

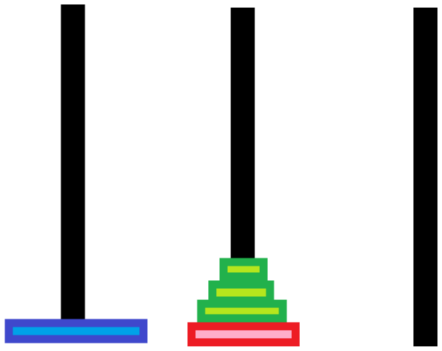
To move N-1 discs from starting pole to destination pole:
Step 1:
We need to move N-2 discs from starting pole to helper pole



Step 2:
We move the N-1th disc from the starting pole to the destination pole



Step 3:
We move N-2 discs from the helper pole to the destination pole.



As we can see, the steps to move N discs are the exact same as to move N-1 discs! The only difference is that the actual poles are different but if we reassign poles to new roles every time we move down a disc, we can build a recurrence relation.

In Step 1, when we move N-1 discs from start to helper, the new helper is the old destination and the new destination is the old helper.

In Step 3, when we move N-1 discs from the helper to the destination, the new helper is the start pole, and the new start pole is the helper

Formalization

```
Let f(N, start, helper, dest) be the steps to move N discs from the start to
```

```
Base case:
f(1, start, helper, dest)= Move from (start) to (dest)
```

```
Recurrence:
f(N, start, helper, dest)

=

f(N-1,start,dest,helper)
Move from start to dest
f(N-1,helper,start,dest)

Example:
Let A, B, C be pole 1, 2, 3

f(4,A,B,C)

=

f(3,A,C,B)
Move from A to C
f(3,B,A,C)

=

f(2,A,B,C)
Move from A to B
f(2,C,A,B)
Move from A to C
f(2,B,A,C)
Move from B to C
f(2,A,B,C)

=

f(1,A,C,B)
Move from A to C
f(1,B,A,C)
Move from A to B
f(1,C,B,A)
Move from C to B
f(1,A,C,B)
```

```
Move from A to C
f(1,B,C,A)
Move from B to C
f(1,A,B,C)
Move from B to C
f(1,A,C,B)
Move from A to C
f(1,B,A,C)
```

=

```
Move from A to B
Move from A to C
Move from B to C
Move from A to B
Move from C to A
Move from C to B
Move from A to B
Move from A to C
Move from B to A
Move from B to C
Move from A to C
Move from B to C
Move from A to B
Move from A to C
Move from B to C
```

3.1.3 Permutations

A permutation is an arrangement of the original set of elements.

For example permutations of A,B,C,D,E,F:

- D,E,F,C,B,A
- F,C,D,B,A,E
- B,D,A,E,F,C
- A,B,C,D,E,F

Given a string S of length N, how can we generate all permutations?

Let's assume that we have a list of permutations for the substring of S of N-1 characters. Then to get the permutations for the string of length N,

all we need to do is insert the Nth character in between all the positions of each permutation of N-1 characters. For example permutation of A,B,C,D.

We can manually find the permutations of A,B,C.

- A,B,C
- A,C,B
- B,A,C
- B,C,A
- C,A,B
- C,B,A

If we insert the letter D between each letter for every permutation we get:

- D,A,B,C
- A,D,B,C
- A,B,D,C
- A,B,C,D
- D,A,C,B
- A,D,C,B
- A,C,D,B
- A,C,B,D .... etc

And we can guarantee that every new permutation will be unique. (Try to prove that to yourself).

But let's look at how we can get the permutation of A,B,C. We can also get all the permutations of the string by taking the permutations of A,B and inserting C in all the positions for all substrings.

Substrings of A,B

- A, B
- B, A

Insert C for all positions for all permutations

- C,A,B
- A,C,B
- A,B,C
- C,B,A
- B,C,A
- C,B,A

We see that we have a recurrence relation. To get the permutations of a string N, we take the string[1..N-1] and we insert the Nth character at every position for each permutation of the N-1 substring. The base case of an empty string is simply an empty string. Permutation of an empty string is an empty list.

For simplicity, S[i..j] will mean the substring from including i to excluding j.

```
Let P(S) be the list of permutations for the string S of length N

Base case:
P('') = []

Recurrence:
Let N = length of string S
P(S) = ps[0..i] + S[N] + ps[i..N] for i in 0..N, for ps in P(S[0..N-1])

Example:
P('ABC')
=
```

Implementation

```
Vector<String> permutation(String s){
    int n = s.length;
    Vector<String> vec = new Vector<String>();
    if(s.length==0)return vec;
    Vector<String> subvec = permutation(s.substring(0,n-1));
```

```
    for(int i=0;i<subvec.size();i++){
        String ps = subvec.get(i);
        for(int j=0;j<n;j++){
            vec.push(ps.substring(0,j)+s.charAt(n)+ps.substring(j,n-1));
        }
    }
    return vec;
}
```

3.1.4 Exercises

1. Given a string S, write a recursive function to generate all substrings
2. Write a solution for hanoi towers but with the restriction that discs can only be moved from adjacent poles. (You can move a disc from A to B but not A to C because they are not adjacent)

Chapter 4

Sorting

Introduction

Sorting is arranging an array of n elements in either increasing or decreasing order by some property. It is very useful in computer science for efficiency in other algorithms that usually require a search.

A stable sort is a sort that can preserve sorting of other properties. For example if we have

(3,G), (1,G), (3, A), (6 K), (1,B)

and we want to sort by the first property in increasing order, we will have:

(1, G) (1,B) (3,G), (3,A) (6 K).

If we sort again by the second property we have (1,B) (3, A) (1, G) (3, G) (6,K) then the sort is stable as the first sort is preserved. However if we had (1,B) (3, A) (3, G) (1, G) (6, K), then the sort would be unstable.

4.1 Bubble Sort

Bubble sort is one of the most basic sorting algorithms, one of the first you should learn.

Its name describes how the algorithm works: bigger bubbles float to the top. This will become more clear when we see the implmentation.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

4.1.1 Implementation

```
//sorts integers from smallest to greatest
public static void BubbleSort (int a[]){
    for (int i=0;i<a.length;i++){
        for (int j=1;j<a.length;j++){
            //if the current bubble is smaller than the bubble that's
            //below then the bigger bubble floats up
            if (a[j]<a[j-1]){
                //swap the bubbles
                int temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
        }
    }
}
```

4.2 Selection Sort

Selection Sort is similar to Bubble Sort in terms of complexity and arguably more intuitive.

Selection Sort works by selecting the smallest element and setting it as the first item in the array, then selecting the second element and setting that as the second item in the array and so on. This process repeats until the least smallest element (the largest element) is set as the last item in the array.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

4.2.1 Implementation

```
//sorts integers from smallest to greatest
public static void SelectionSort (int a[]){
    //iterates through the array selecting the smallest elements
    for (int i=0;i<a.length-1;i++){
        int minIndex = i;//var for storing where the ith smallest element
        //iterates through the array looking for the ith smallest element
        for (int j=i+1;j<a.length;j++){
            //if the current element is smaller than the current smallest
            if (a[j]<a[minIndex]){
                //updated the location of the smallest element
```

```
                minIndex = j;
            }
            //swap the current i with the smallest element
            int temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
        }
    }
}
```

4.3 Insertion Sort

Insertion Sort is the another n¡sup¿2¡/sup¿ sorting algorithm.

It works by starting with an empty sorted sequence of numbers and then inserting every element in the array into the sorted sequence maintaining the order after each insert.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

4.3.1 Implementation

```
//sorts integers from smallest to greatest
public static void InsertionSort (int a[]){
    //iterates through the array selecting the smallest elements
    for (int i=0;i<a.length;i++){
        //element that needs to be inserted into the sorted list
        int val = a[i];
        //keeps track of where to insert the element
        int index = i;

        //loops through the already sorted list to find where to
        //insert the current element
        while (index > 0 \&\& a[index-1] > val){
            //switches the current item at index with the one before
            //to check if the element that needs to be inserted will
            //go in the next spot
            a[index] = a[index-1];
            index--;
        }
        a[index] = val;
```

```
    }
}
```

4.4 Heap Sort

Prerequisites: Heap

Heap sort is a sort that inserts all the element in an array into a min heap and then pops all the element outs. Since a heap guarantees that the root node will be the smallest element, we can store the entire array in a heap and the order that they are popped out of the heap will be in order.

Implementation

```
public void heapSort(int[] arr){
    PriorityQueue<Integer> pq; //built in heap
    for(int i=0;i<arr.length;i++){
        pq.push(arr[i]);
    }
    for(int i=0;i<arr.length;i++){
        arr[i] = pq.pop(i);
    }
}
```

4.5 Merge Sort

Preqrequistes: Recursion

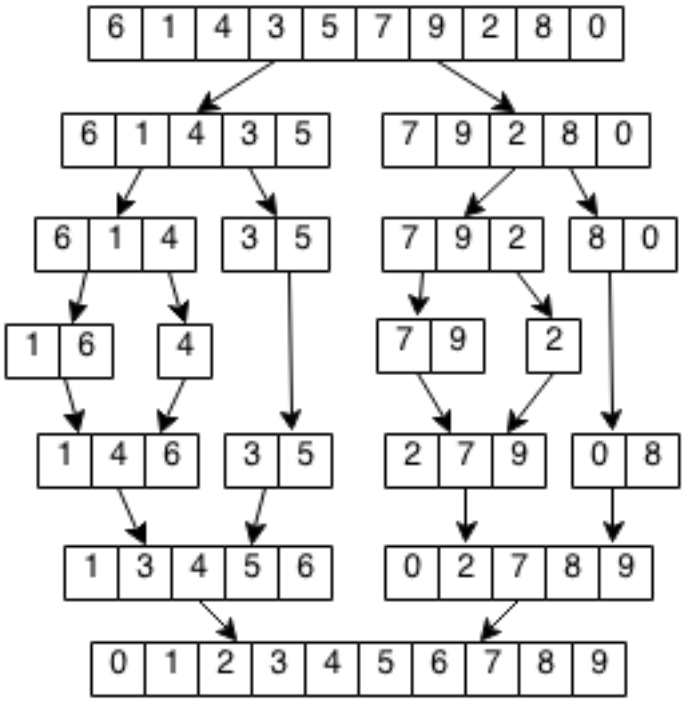
Merge sort works by breaking down the sorting into smaller pieces. If we want to sort N elements, we can sort the first half of the elements, sort the second half and then merge the results together. To sort the first half, we can do the exact same thing of sorting the first quarter and the second quarter and merging the results.

4.5.1 Implementation

Merge sort work be breaking down the problem into smaller and smaller parts and then combing those parts to solve the big problem.

We can keep splitting the list into half until there each piece has 1 element or no elements. We can then combine the results of each piece repeatedly until the entire list is sorted.

Example:



Formalization

```
Let merge(arr1,arr2) combine two sorted arrays into one sorted array

merge(arr1, arr2)= { arr2 if arr1 is empty
                   { arr1 if arr2 is empty
                   { arr1.first + merge(arr1[1..arr1.length],arr2) if ar
                   { arr2.first + merge(arr1[0..arr.length],arr2[1..lengt

Let sort(arr) sort an array
Let middle be (0+arr.length)/2
sort(arr) = merge(sort(arr[0..middle]) ,sort(arr[middle..arr.length])
```

Code

For our Java implementation, instead of returning a new array every time we merge two arrays, we can create a temporary array to store the merged results and then move it back.

```
/**
 * Merges two segments of the same array. The two segments should be adjac
```

```
 * @param arr Array containing segments
 * @param start1 Start index of first segment
 * @param end1 End index of first segment
 * @param start2 Start index of second segment
 * @param end2 End index of second segment
 */

public static void merge(int arr[],int start1,int end1, int start2,int end2
    int arr2[] = new int[end2-start1+1];
    int begin = start1;
    int n = 0;

    //Pick smallest element one by one
    while(start1 <= end1 \&\& start2 <= end2){
        if(arr[start1] <= arr[start2]){
            arr2[n] = arr[start1];
            start1++;
        }else{
            arr2[n] = arr[start2];
            start2++;
        }
        n++;
    }
    //If first segment still has elements
    while(start1 <= end1){
        arr2[n] = arr[start1];
        n++;
        start1++;
    }
    //if second segment still has elements
    while(start2 <= end2){
        arr2[n] = arr[start2];
        start2++;
        n++;
    }
    //Copy merged array back
    for(int i=0;i<n;i++){
        arr[begin+i]=arr2[i];
    }
}
```

```
/**
 * Sorts an array using merge sort
 * @param arr Array to be sorted
 * @param start Index at beginning of array
 * @param end Index at end of array
 */
public static void mergeSort(int arr[],int start,int end){
    if(end-start<=0){
        return;
    }
    int mid = (start+end)/2;
    mergeSort(arr,start,mid);
    mergeSort(arr,mid+1,end);
    merge(arr,start,mid,mid+1,end);
}

public static void main(){
    int arr[] = {5,7,2,6,8,5};
    mergeSort(arr,0,arr.length-1);
}
```

4.6 Quick Sort

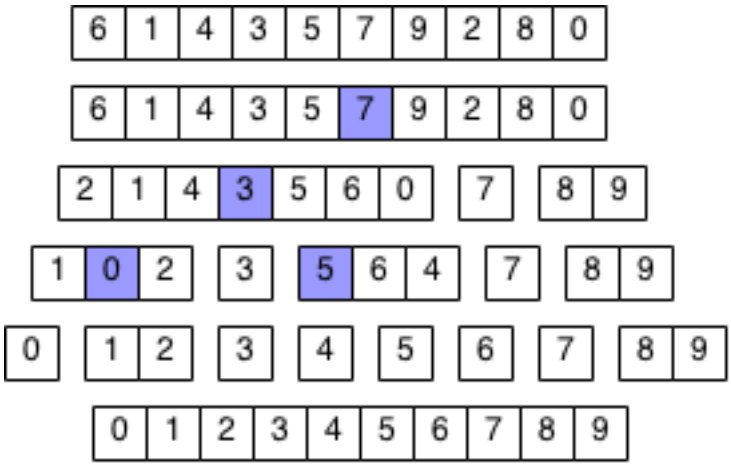
Quick Sort is another fast sorting algorithm that uses divide and conquer. It is also known as partition-exchange sort or as Hoare's quicksort (named after the author).

In Quick Sort an element is selected as a "pivot". The list is then divided into two sublists: a list of elements less than (or equal to) the pivot and a list of elements greater than the pivot. Each sublist is sorted (conquered) and then appended together along with the origin pivot.

In the best case (if the pivot that is chosen is exactly the middle element), then the runtime is O(n log n). However, in the worst case, the runtime for Quick Sort is n^2. Despite its worst case behaviour, quick sort is still popular and in widespread use. The average case behaviour is quite good and there are straightforward methods that can be used to improve the selection of the pivot.

The sorting algorithms are implemented to sort integer arrays but can be adapted to sort arrays of any data structure that is comparable.

4.6.1 Implementation




```
//sorts integers from smallest to greatest
public static void QuickSort (int a[], int first, int last){
    //first is the starting index of the list
    //last is the last index of the list
    //QuickSort can be called by QuickSort(a, 0, a.length-1)

    if (last <= first) return; //if the array size <= 1

    int pivot = a[first]; //picks the first element as the pivot
    // the location to put the next integer that's larger than the pivot
    int index = last;

    //iterate through the list, sorting it by the first pivot
    for (int i=last;i>=first+1;i--){
        if (a[i]>=pivot){
            //swap the current element to be on the side that's
            //larger than the pivot
            int temp = a[index];
            a[index] = a[i];
            a[i] = temp;

            //decrements index for the next switch
            index--;
        }
```

```
    }

    //swap the element at index with the pivot so that it's
    //in the right place
    int temp = pivot;
    a[first] = a[index];
    a[index] = pivot;

    //recursively sort the lists less than and greater than the pivot
    QuickSort (a, first, index-1);
    QuickSort (a, index+1, last);
}
```

4.7 Bozo Sort

Bozo sort is a sort that keeps randomly arranging an array until it is sorted. If you are very lucky bozo sort will be very fast! But for the most part this is a horrible sort.

4.7.1 Implementation

```
public void bozoSort(int[] arr){

   boolean sorted = false;
   int i = 0;

   while(!sorted){
       sorted = true;
       for(i=1;i<arr.length;i++){
          if(arr[i]<arr[i-1]){
             sorted = false;
             break;
          }
       }
       if(sorted)return;

       for(i=0;i<arr.length;i++){
           int x = Math.randInt(arr.length);
           int y = Math.randInt(arr.length);
```

```
            int temp = arr[x];
            arr[x] = arr[y];
            arr[y] = temp;
        }

    }
}
```

4.8 Permutation Sort

Permutation sort is a sort that keeps permutating the array until it is sorted. It is the slowest sort that will guarantee that the array will be sorted.

Implementation

```
void permuteSort(int[] arr){
  while(!sorted(arr)){
    permute(arr);
  }
}
```

4.9 Miracle Sort

Prerequisites: Miracles, Sense of Humour

Miracle sort is a sort that truly requires a miracle. We keep checking the array until it is sorted. It requires that some external force (a miracle?) changes some bits in the computer in a way that it becomes sorted.

4.9.1 Implementation

```
public void miracleSort(int[] arr){
   boolean sorted = false;
   do{
      sorted = true;
      for(int i=1;i<arr.length;i++){
         if(arr[i]<arr[i-1]){
            sorted = false;
            break;
         }
      }
```

```
    }while(!sorted);
}
```

Chapter 5

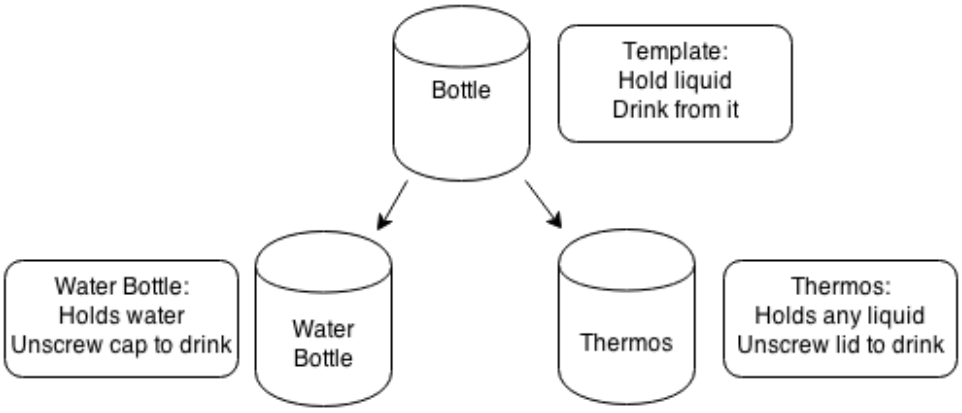
Data Structures

Introduction

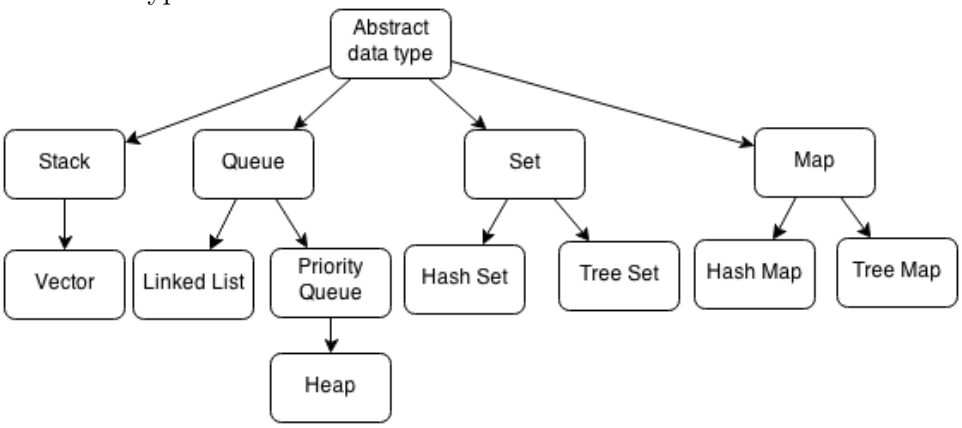
Data structures are a way of storing data such that it can be used in an efficient way. Although many of these data structures are already built into various languages, it is important to understand how they work. By understanding the implementations we can have a sense of which data structure to use in different problems as well as determine how efficient they are.

An abstract data type is a conceptual model for representing data. An abstract data type tells what it should do as opposed to how it should work. It will tell us what operations it should have but should not tell us how to implement them.

For example, a bottle should be able to hold water and allow us to drink from. This tells us what it should do but we don't need to know how it works or how it is made. A plastic water bottle is an implementation of a bottle. It holds water in its interior and allows us to drink by unscrewing the cap and letting us pour water down our throat. A thermos is also an implementation of the bottle, it holds fluid inside it, but this thermos has a lid that can be popped open and water can come from it. A thermos and plastic water bottle are different implementations as they are made differently and used differently, but they fundamentally do what a bottle is supposed to do: store liquid, and provide a way to drink. A bottle does not actually exist, but types of bottles do.



Some implementations of abstract data types are better than others for different purposes. For example plastic water bottles are very cheap whereas a thermos is more expensive but a thermos can hold hot water and keep it warm for a long period of time. When thinking of a implementation for an abstract data type we need to know what we need it for.



For more intermediate data structures, read the advanced data structures page.

5.1 Arrays

Source on Github

Imagine you had a row of parking spaces where each was labelled with a number. If you wanted to know what the license plate of a car at parking space 4 was, all you have to do is go to the parking space and read off the license plate. If you wanted to park you car at parking space 5, you would go to parking space 5 and put your car in there if there was nothing there.

Let's say that you had cars at parking spaces 1, 2, and 3. If you wanted to insert a new car at parking space 1 and keep the rest of the cars in the same order, you would have to shift the cars in the parking spaces from 1, 2 and 3 to 2, 3 and 4 by getting in each car and parking them in the new spaces which would take some time. This type of structure is an array.

An array is the most basic data structure that stores elements of the same type in a fixed block. The fact that it is in one block and the same type is important because it allows accessing elements very quickly if you have the index. All you have to do is go to the index and retrieve the element. However, inserting elements in the array is slow because you would have to shift all the elements and also if you want to shift past the fixed size you will get an error. (Imagine the parking spaces are full and you wanted to insert a car somewhere, there will still be one car that will have no parking space).

Arrays can be multidimensional meaning you can have an array of array of objects. (Imagine a parking lot with multiple rows of parking spaces).

Operation	Create	Get i	Set i
Time Complexity	O(n)	O(1)	O(1)

5.1.1 Implementation

Implementation of a very simple array in Java

Create array

Creating an array allocates a block of memory for us to store the elements.

```
public static int[] createArray(int size){
     int[] array = new int[size];
     return array;
}
```

Get

When we want to get the element of an index all we have to do is offset the index by the array. However we need to check that the index is within bounds of the array.

```
public static int getElemenent(int[] arr, int i){
    if(i<0 || i>=arr.length){
        throw new ArrayIndexOutOfBoundsException();
```

```
    }
    return arr[i];
}
```

Set

Setting is very similar to getting but instead we change the value in the array.

```
public static void setElemenent(int[] arr, int i,int value){
    if(i<0 || i>=arr.length){
        throw new ArrayIndexOutOfBoundsException();
    }
    arr[i] = value;
}
```

5.1.2 See Also

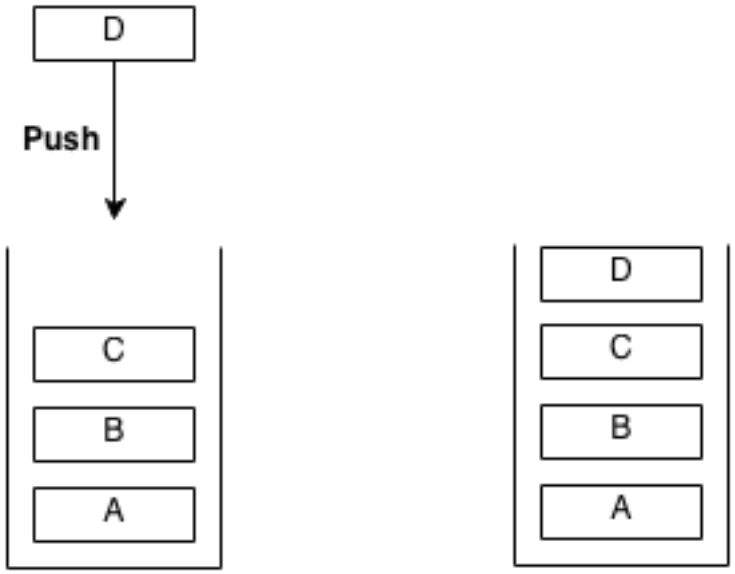
- Vector
- Stack

5.2 Stack

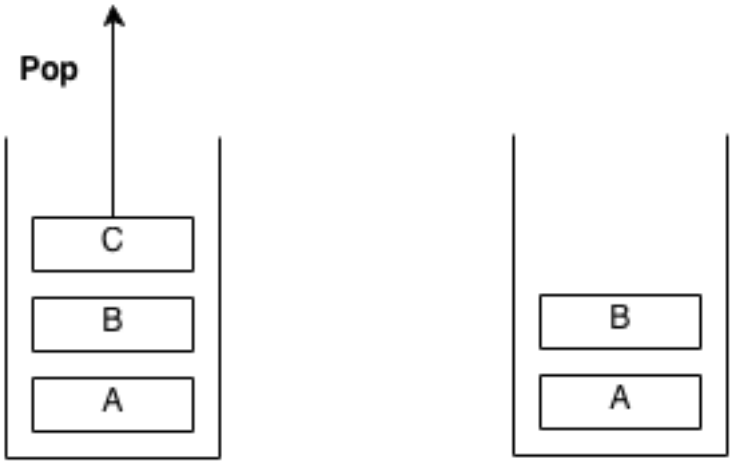
Imagine a stack of plates at a buffet, the plates are taken from the top and are also replaced from the top. The first plate to go in will be the last plate to come out. The last plate to go in will be the first to come out. This structure is called a stack.

A stack is an abstract data type with the property that it can remove and insert elements following a FILO (First In Last Out) structure. The first element to be inserted must be the last element to be removed and the last element to be inserted must be the first element to be removed. Sometimes, removal is called "pop" and insertion is called "push".

Example of push:



Example of pop:



Stacks are used for function calls on the memory stack. Whenever a function is called, it is placed on the memory stack with its variables and when it is returning a value, it is popped off the stack.

A stack is usually implemented as a vector.

5.2.1 Implementation

Implementation	Pop	Push
Vector	O(1)	O(1)

5.2.2 Exercises

1. Given a string of brackets of either () or [], determine if the bracket syntax is legal (every opening bracket has a closing bracket from left to right).

   Legal syntax:

   - ([( ) [ ]])
   - ( ) ( ) [ ] ( ) ( )

   Illegal syntax:

   - (( )]
   - ( ) [( ])

5.3 Vector

Prerequisites: Arrays, Stack

Source on Github

A vector is a stack that is implemented as an array. It is very similar to an array, but it is more flexible in terms of size. Elements are added and removed only from the end of the array. When more elements are added to the vector and the vector is at full capacity, the vector resizes itself and reallocates for 2*N space. When using an vector we can keep adding elements and let the data structure handle all the memory allocation.

Operation	Get	Push	Pop	Insert	Delete
Time Complexity	O(1)	O(1)	O(1)	O(n)	O(n)

5.3.1 Implementation

There is a builtin Vector class already, but we will go through the implementation of a simple integer vector class to understand how the data structure works.

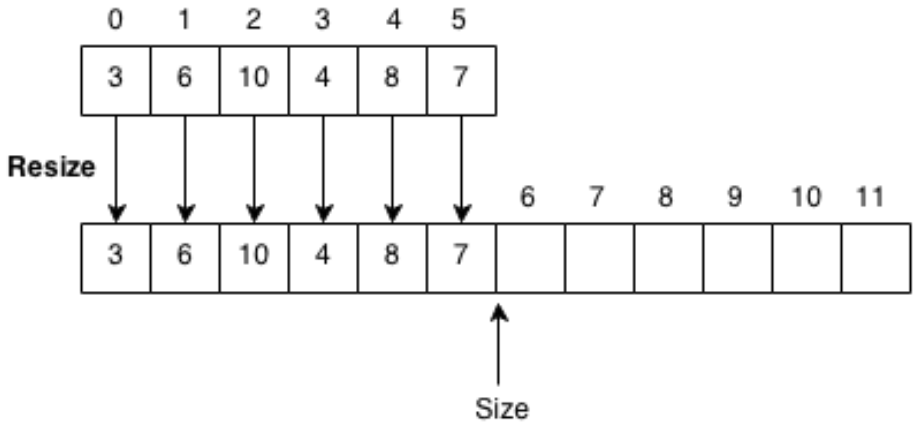
Class

In our vector class, we need to store the element and the size of the current vector.

```
class Vec{
    private int[] arr; //Storage of elements
    private int size; //Current size
}

//Constructor
publc Vec(int startSize){
    arr = new int[startSize];
    this.size = 0;
}
```

Resize

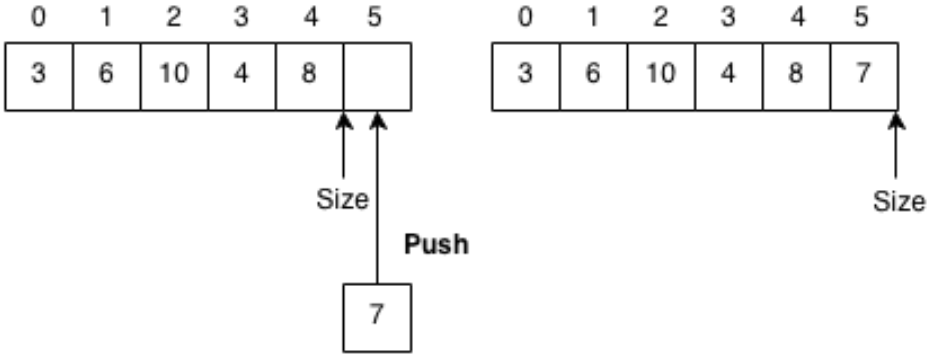
Resize will be used to resize the current size of elements. We create a new array of two times the size of the old one and copy the old array over.




```
public void resize(){
    int[] newArr = new int[2*arr.length];
    for(int i=0;i<end;i++){
        newArr[i] = arr[i];
    }
    arr = newArr;
}
```

5.3.2 Add Element

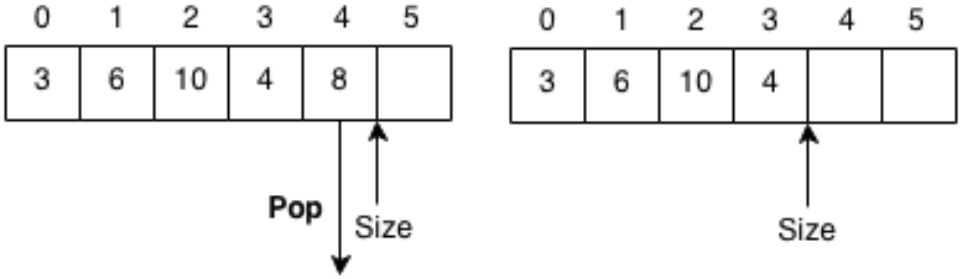
Add element will add elements to the end of the vector. If the array is full, the vector will resize itself.




```
public void add(int x){
        if(size>=arr.length){
        resize();
    }
    arr[size] = x;
    end++;
}
```

5.3.3 Pop

Removes the element at the end of the vector. We decrease the size of the vector and return the last element.



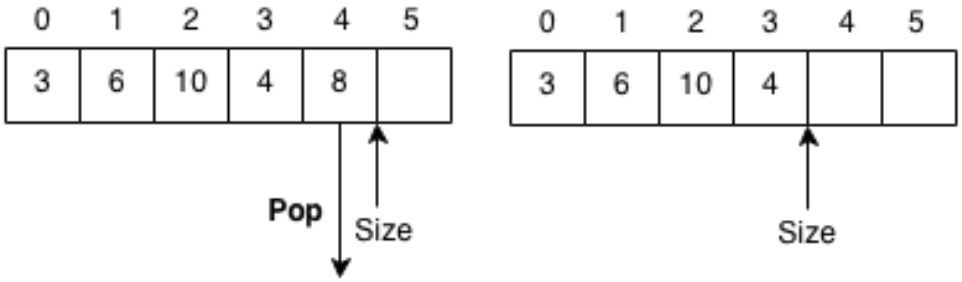

```
public int pop(){
    if(size==0){
        throw new NoSuchElementException();
    }
```

```
    int ret = arr[size];
    size--;
    return ret;
}
```

5.3.4 Remove

Removes element at the index idx. It will throw an exception if the index is out of bounds.

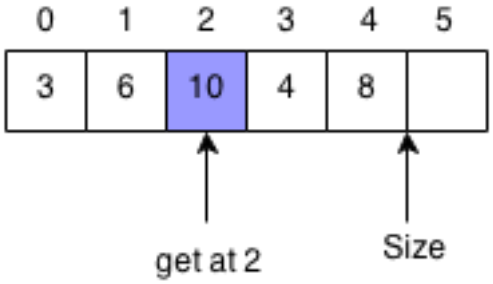
We shift everything to the right of the index to the left by one to fill in the missing element.




```
public int remove(int idx){
    if(idx<0||idx>=size){
        throw new ArrayIndexOutOfBoundsException();
    }
    int ret = arr[idx];
    while(idx+1<size){
        arr[idx]=arr[idx+1];
        idx++;
    }
    size--;
    return ret;
}
```

5.3.5 Get Element

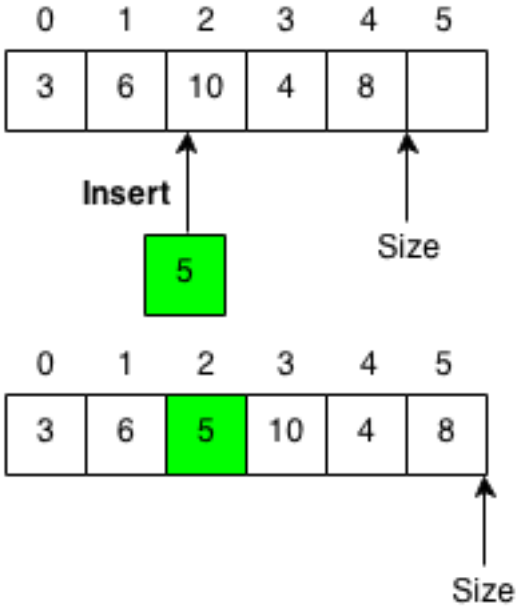
Returns the element at the specified index. It will throw an exception if the index is out of bounds. Note this function is exactly as the same as in the array




```
public int get(int idx){
    if(idx<0||idx>=size){
        throw new ArrayIndexOutOfBoundsException();
    }
    return arr[idx];
}
```

5.3.6 Insert Element

Insert the new number x at the index. We need to make space at the index for the new element so we shift everything to the right of the index by 1.



```
public void insert(int idx,int x){
    if(idx<0||idx>size){
        throw new ArrayIndexOutOfBoundsException();
    }
    size++;
    if(size>=arr.length){
        resize();
    }
    //Shift elements to the right or idx by 1
    int idx2 = size;
    while(idx2>idx){
        arr[idx2]=arr[idx2-1];
        idx2--;
    }
    arr[idx] = x;
    }
```

5.3.7 Exercises

1. Implement removeAtIndex(int index) for Vector

5.4 Queue

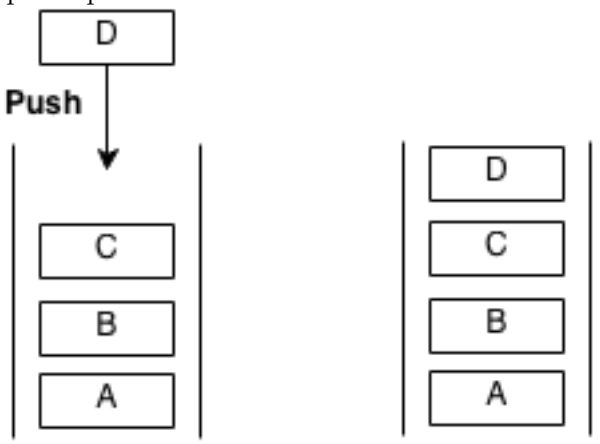
Imagine you are standing in line for a restaurant. Whoever is first in line will be served first and whoever is last in line will be served last. People can be served while more people join the line and the line may get very long because it takes a while to serve one person while more people join the queue. This is called a queue.

A queue is an abstract data type with two functions, pop and push. Removal from the front is called "pop" or "dequeue". Insertion from the back is called "push" or "enqueue". A queue follows a First In First Out (FIFO) structure meaning the first element pushed should be the first element popped and the last element pushed should be the last element popped.

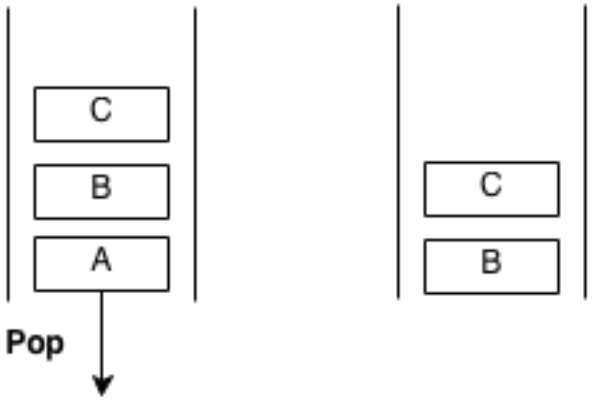
Queues are often used for buffer systems, for example a text message service. The messages that arrive at the server first are relayed first and the messages that arrive later are relayed later. If there are too many text messages in the system such that the rate texts are received overwhelm the number of texts that are sent the buffer may overflow and messages will get dropped. Most of the time this won't happen because the systems are

designed to handle large loads, but if there were an emergency that caused everyone to start texting many texts could be dropped.

Example of push:



Example of pop:



5.4.1 Implementation

We can implement a queue most efficiently using a linked list because it has an efficient memory allocation.

Implementation	Pop	Push
Linked List	O(1)	O(1)

5.4.2 Exercises

1. Given a list of letters representing instructions where the first instruction is executed, output what the final list should look like after N instructions are executed.

   First instruction:

   - A. Add B to the end of the list of instructions
   - B. Do nothing
   - C. Add two A's to the front of the list of instructions

   Example:

   - ABC
   - BCB
   - CB
   - AAB
   - ABB
   - BBB
   - BB
   - B

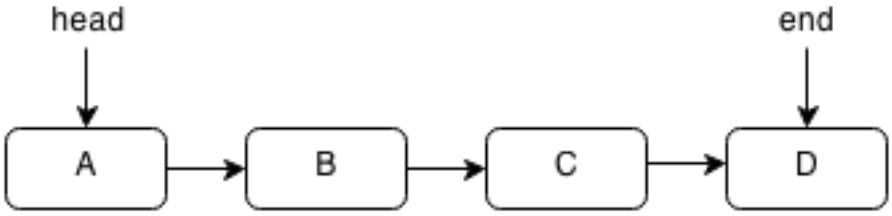
5.5 Linked List

Prerequisites: Queue

Source on Github

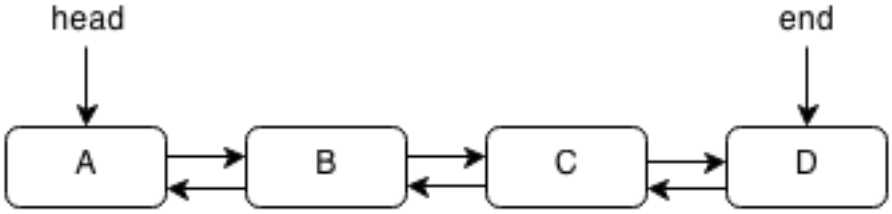
Imagine you had some train cars that were linked together where each was labelled on the inside with a different letter. If you had to find a specific letter you would have to start at the first train car and look inside to check the letter and then walk into the next train car to check the letter and so forth until you found the train car you wanted. If you wanted to insert a train car somewhere all you would have to do it unlink the position where you wanted to insert it and then relink the new train car with the other cars. If you wanted to remove a train car all you would have to do is unlink that car from the other cars and then create a new link to the cars that were adjacent to it. This sort of structure is called a linked list.

A pointer is something that holds the memory location of another object.



A linked list is similar to an array but it is different such that it is not stored in one block of data. Each element can be stored in a random place in memory but each element contains a pointer to the next element thus forming a chain of pointers. Think of a pointer as a link that links two train cars. Since the elements aren't in a block, accessing an element must be done by traversing the entire linked list by following each pointer to the next. However, this also allows insertion to be done more quickly by simply changing the point of the previous element and setting to the pointer of the current element to the next element. Deletion is also done by taking the previous element and changing its pointer to two elements ahead. In a linked list the links only go forward and you cannot move backward.

A doubly linked list is a linked list that has pointers going backwards as well as forwards.



Operation	Get	Push	Delete	Insert
Time Complexity	O(n)	O(1)	O(1)	O(1)

5.5.1 Implementation

In Java, there already exists a LinkedList class but we will implement our own.

Link Class

The Link class for each "link" in the Linked List. In each Link we only need the value and location of the previous and next node.

```
class Link{
```

```
    int value;
    Link next;
    public Link(int value){
        this.value = value;
        this.next = null;
    }
}
```

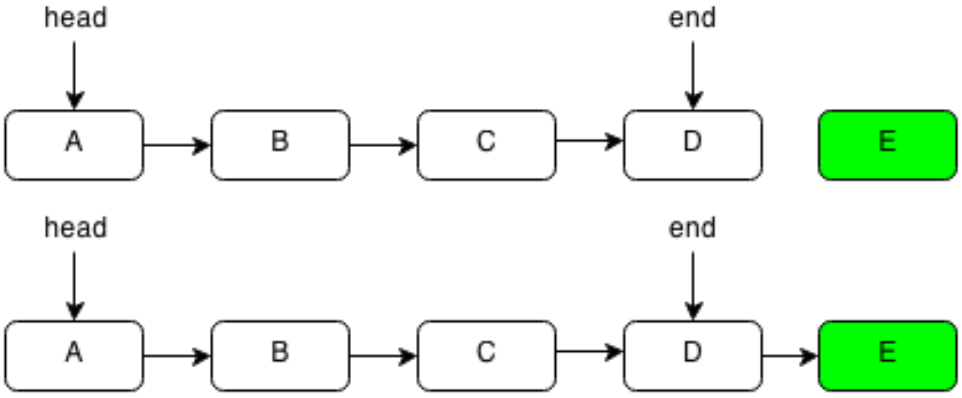
Class

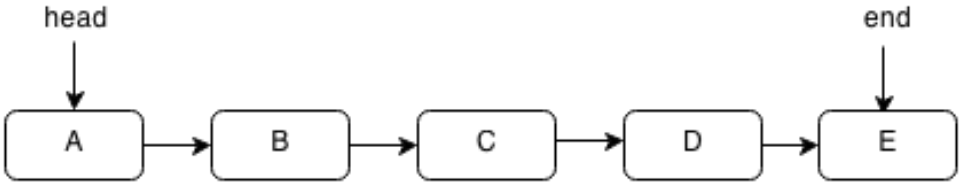
Create the linked list by initializing the starting node as null and setting the size to empty.

```
class LinkedList{
    Link head;
    Link end;
    int size;
    public LinkedList(){
        start = end = null;
        size = 0;
    }
}
```

Push

Create a new node with the value given and add it to the end. We have to set the current head previous node to the new node and the new next next to the last node.

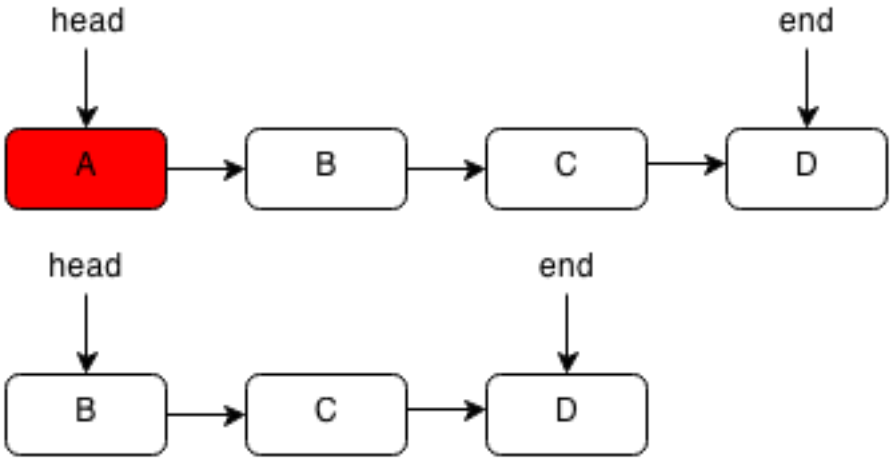





```
/*
 * Adds new node to head of linked list
 */
public void push(int value){
    Link newLink = new Link(value);
    if(size==0){
        head = end = newLink;
    }else{
        end.next = newLink;
        end = newLink;
    }
    size++;
}
```

Pop

Pops off the node at the head.

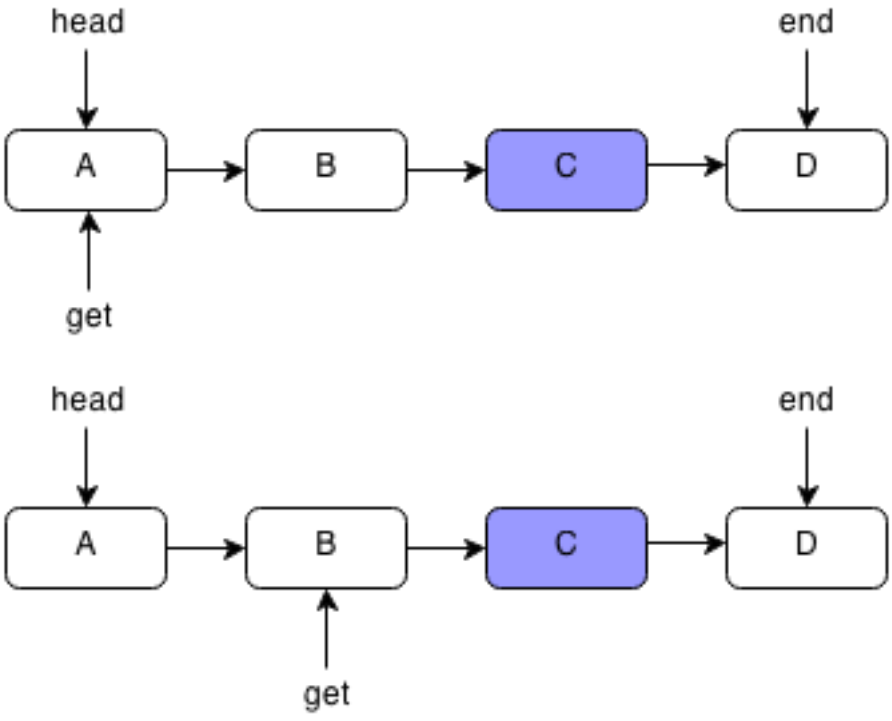


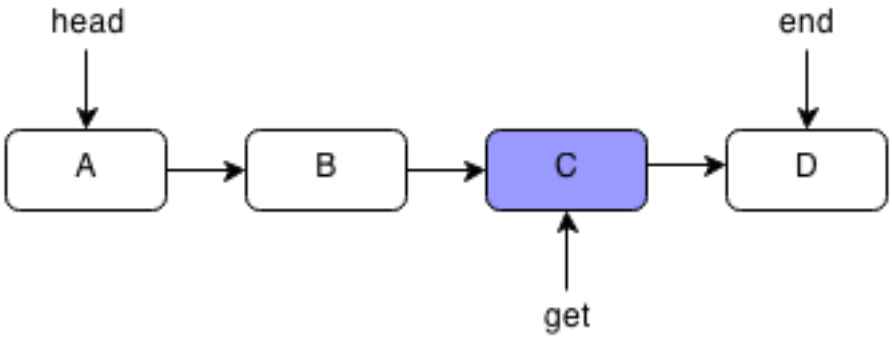

```
/*
 * Adds new node to head of linked list
```

```
 */
public int pop(){
    if(head==null){
        throw new NoSuchElementException();
    }
    int ret = head.value;
    head = head.next;
    size--;
    if(size==0){
        end = null;
    }
    return ret;
}
```

Get

Get retrieves the value at the specified index. We have to loop through the entire list to get the index we want because the nodes are not in the same block of memory.

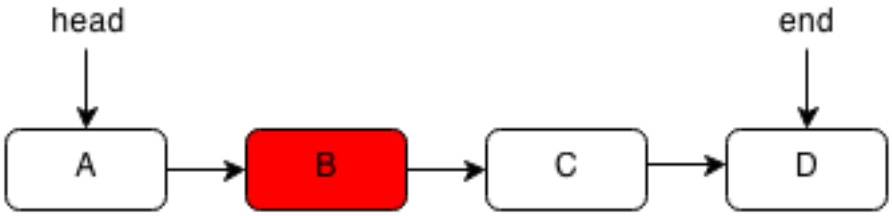


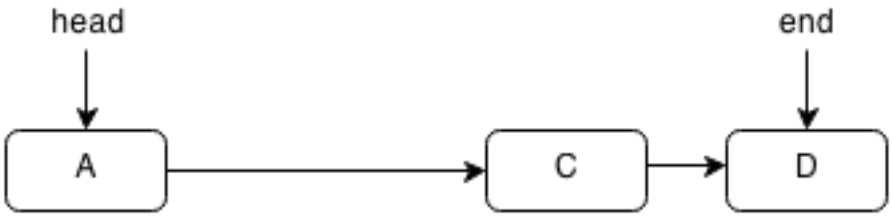



```
/*
* Gets the value at index
*/
public int get(int index){
    int i = 0;
    Link curNode = head;
    while(curNode!=null){
        if(index==i){
            return curNode.value;
        }
        curNode = curNode.next;
        i++;
    }
    throw new NoSuchElementException();
}
```

Delete

To delete the current node we set the previous node next link to the link after.






```
/*
 * Deletes node after specified
 */
public void deleteNext(Link node){
    if(node.next==end){
        end = node;
    }
    node.next = node.next.next;
    size--;
}
```

5.5.2 Exercises

1. Implement a doubly linked list.

2. A game is player by always eliminating the kth player from the last elimination and played until one player is left. Given N players each assigned to a number, find the number of the last player.

   For example, you have 5 players (1,2,3,4,5) and the 3rd player is eliminated.

   - 1, 2, 3, 4, 5
   - 1, 2, 4, 5 (1, 2, 3 is eliminated)
   - 2, 4, 5 (4, 5, 1 is eliminated)
   - 2, 4 (2, 4, 5 is eliminated)
   - 2 (2, 4, 2 is eliminated)
   - Player 2 is the last one standing.

3. Given two linked lists which may share tails, determine the point at which they converge.

5.6 Trees

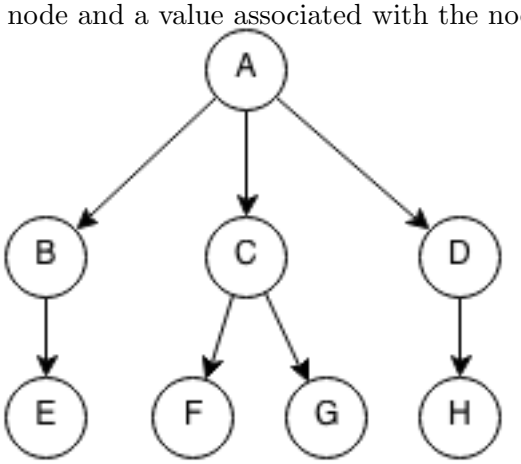
Trees are data structures that follow a hierarchy, each node has exactly one or zero parents and each node has children. Trees are recursive structures meaning that each child of a tree is also a tree. A tree within another tree is called a subtree.

A child is a node that is below another node.

A parent is a node that is above another node.

The element at the top of the tree with no parents is called a root. The node at the bottom of the tree with no children is called a leaf.

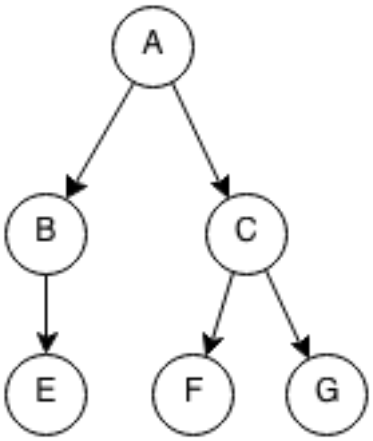
Each node can hold different kinds of information depending on the tree. A node can hold the children it has, the parent it has, a key associated with the node and a value associated with the node.



- A is the root of the tree and E,F,G,H are leaves.
- The parent of E is B.
- C,F,G is a subtree of the original tree.

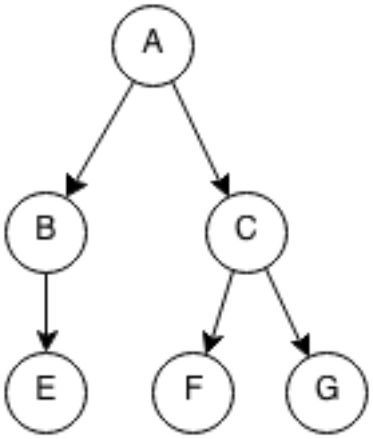
5.6.1 Binary Tree

A binary tree is a tree where every node has at max two children.



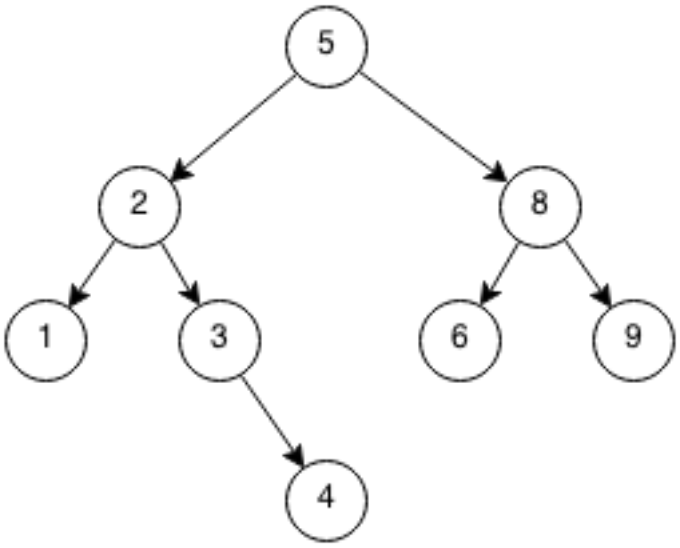
5.7 Binary Tree

A binary tree is a tree such that each node has at most 2 children.



5.7.1 Binary Search Tree

A binary search tree is a type of binary tree where all the nodes in a left subtree will be smaller than the node and all the nodes in a right subtree will be greater than the node. It has a recursive structure such that each subtree is also a binary search tree.



5.8 Sets

Imagine you have a grocery list that you use to keep tracking of things you need to buy. You want to make sure there are no duplicate items in the list, you can add items to the list and that you can remove items from your list. This structure is similar to what a set does.

Sets are abstract data structures which are able to store values and are used for three operations: insertion, deletion and membership test.

Insertion places an element into the set, deletion removes an element from the set and a membership test is checking whether an element exists within the set.

5.8.1 Implementation

Type	Membership	Insertion	Deletion
Tree Set	O(log n)	O(log n)	O(log n)
Hash Set	O(1)	O(1)	O(1)

5.8.2 Exercises

1. Given a list of words, determine how many of them are anagrams of each other. An anagram is a word that can have its letters scrambled into another word.

- For example silent and listen are anagrams but banana and orange are not.

2. Given two lists of friends, find the number of mutual friends.

3. Given a list of numbers, find the number of tuples of size 4 that add to 0.

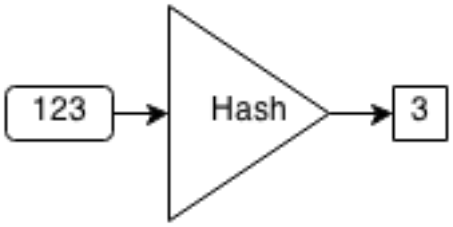
   - For example in the list (10,5,-1, 3, 4, -6) the tuple of size 4 (-1,3,4-6) adds to 0.

5.9 Hash Set

Prerequisites: Sets

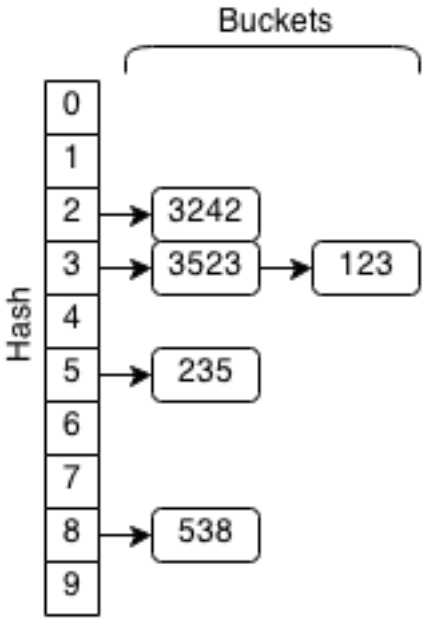
Source on Github

Hash sets are sets that use hashes to store elements. A hashing algorithm is an algorithm that takes an element and converts it to a smaller chunk called a hash. For example let our hashing algorithm be (x mod 10). So the hashes of 232, 217 and 19 are 2,7, and 9 respectively.



For every element in a hash set, the hash is computed and elements with the same hash are grouped together and stored in a linked list. The linked list is called a bucket.

If we want to check if an element already exists within the set, we first compute the hash of the element and then search through the linked list associated with the hash to see if the element is contained.



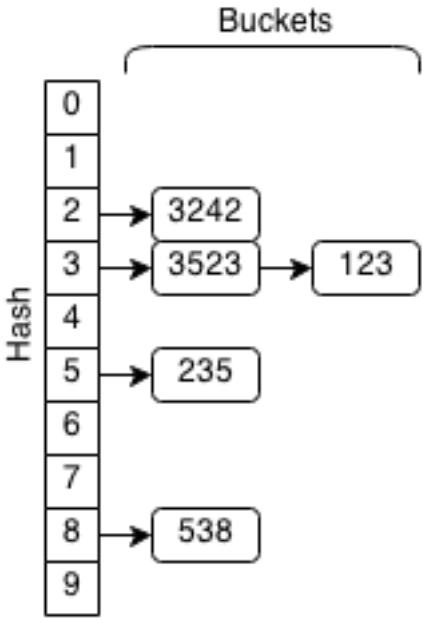
Operation	Membership	Insertion	Deletion
Time Complexity	O(1)	O(1)	O(1)

Prerequisites

- Sets
- Linked List

5.9.1 Implementation

Let use the example of the hashset of the elements of 3242, 3523, 123, 235 and 538. The hash set looks like this when computed:



If we wanted to check if 7238 was in the hash set, we would get the hash (7238 mod 10 = 8). So we get the bucket associated with the hash 8 and we get the list of (538). When we iterate through this short list, we see that 7238 is not a member of the set.

Similarly, if we wanted to insert 7238 into the hash set, we would check if it exists and if it did not we would append the element to the end of the bucket. For deletion we would find 7238 check if it existed in the set and remove it from the bucket.

Hash sets are very efficient in all three set operations if a good hashing algorithm is used. When the objects are that being stored are large then hash sets are effective as a set.

Class

Inside our implementation of a hash set we will store the buckets using an array of linked lists, the number of buckets, and the number of elements in the set.

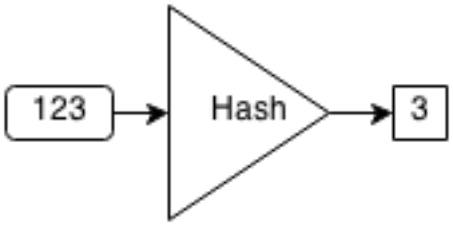
The collision chance is the threshold for resizing the hash set. When the ratio of elements in the set to number of buckets is greater than the threshold, then the chance of collision will be high enough that it will slow down the operations. The lower this ratio, the better performing a hash set

will be.

```
public class HashSet {

    public LinkedList<Integer>[] buckets;
    public int bucketsSize = 10;
    public int size = 0;
    public static final double COLLISION\_CHANCE = 0.3;

    public HashSet(){
        buckets = new LinkedList[10];
        for(int i=0;i<bucketsSize;i++){
            buckets[i] = new LinkedList<Integer>();
        }
        size = 0;
    }
```

Hash code

The hash code is the result of the hashing algorithm for an element. In our hash set implementation, we will use a simple hash: modulus of the integer by the number of buckets.



For the most part if the numbers are all random then the hash function is fine. However, if the number of buckets was 10 and we added the elements 20,30,40,50,60,70, they will all end up in the same bucket and results in poor performance.

```
public int getHash(int x,int hashSize){
    return x \% hashSize;
}
```

Resize

A hash set must be able to resize. When the ratio of number of elements to number of buckets, the chance of collision will increase more and more.

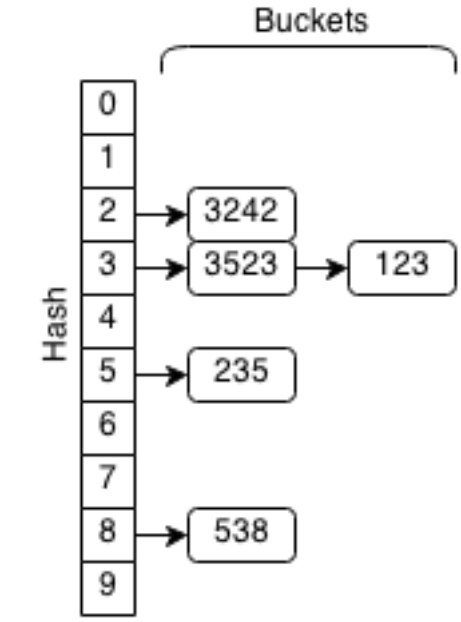
So we must able to resize the number of buckets to support the number of elements to lower the chance of collision.

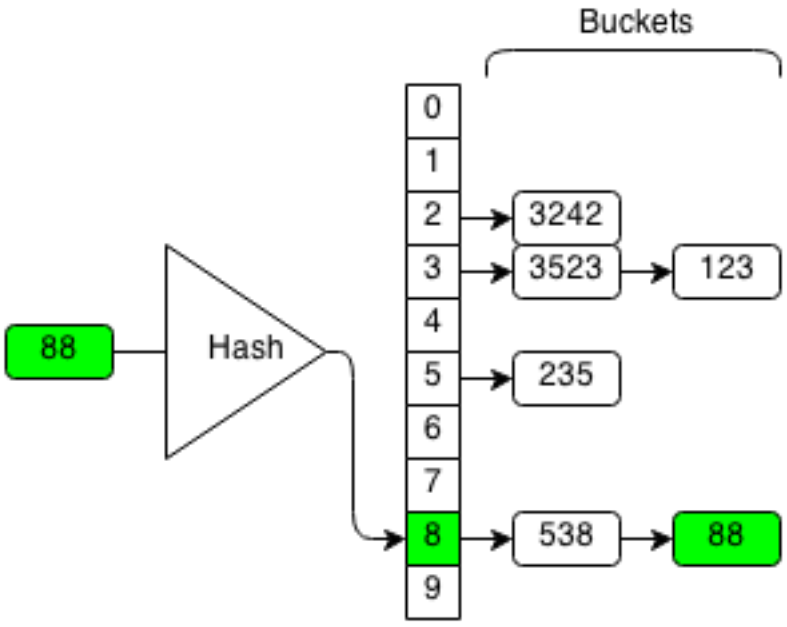
To resize efficiently, we can create two times the number of buckets and set them to empty and then insert all the elements in the old buckets to the new buckets.

```
public void resize(){
    int newBucketsSize = bucketsSize*2;
    LinkedList<Integer>[] newBuckets = new LinkedList[newBucketsSize];
    for(int i=0;i<newBucketsSize;i++){
        newBuckets[i] = new LinkedList<Integer>();
    }
    for(int i=0;i<bucketsSize;i++){
        for(Integer y:buckets[i]){
            int hash = getHash(y,newBucketsSize);
            newBuckets[hash].push(y);
        }
    }
    buckets = newBuckets;
    bucketsSize = newBucketsSize;
}
```

Insert

To insert an element in a hash set, we get the hash code from our hashing algorithm and insert the element into the corresponding bucket.





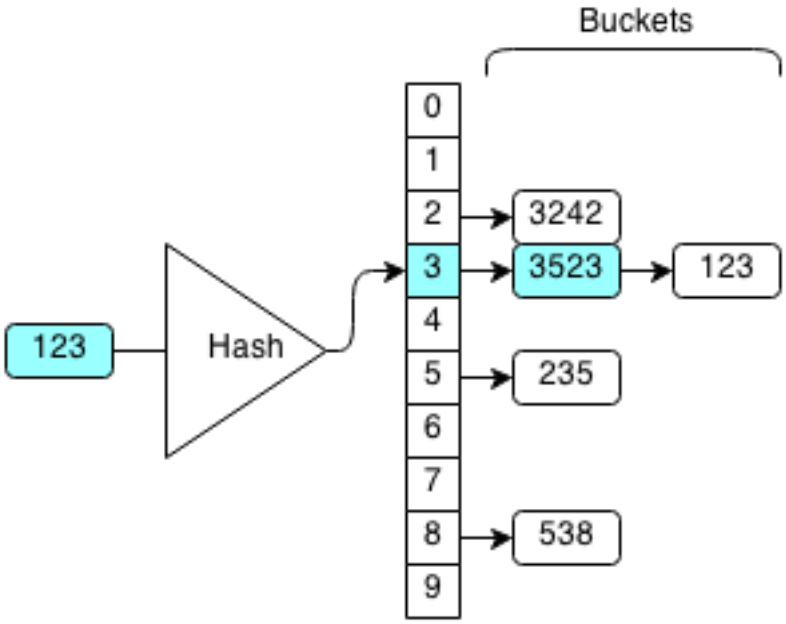
The function will return method or not the operation was successful. If the bucket already contains the element the operation will stop because we

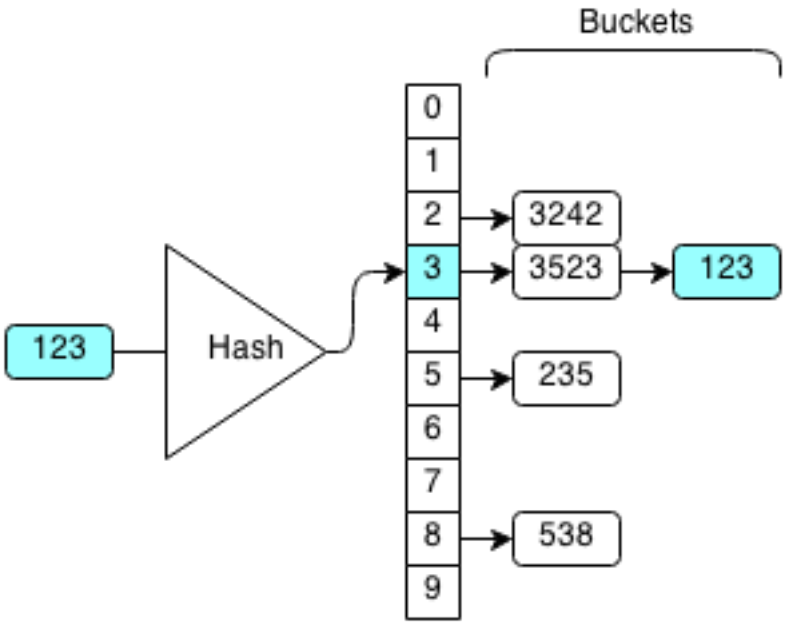
do not want to add duplicate elements into the set. If the bucket does not contain the element, we will insert it into the bucket and the operation is successful.

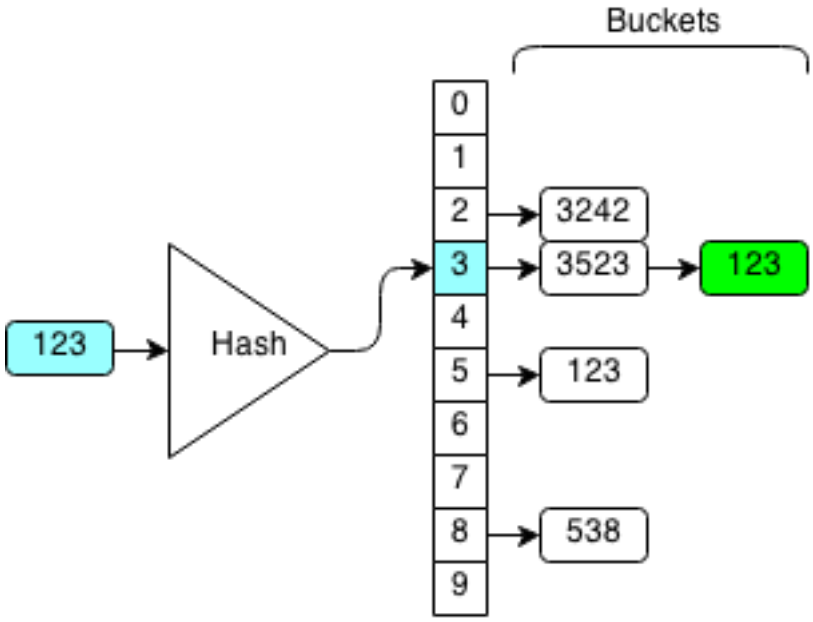
```
public boolean insert(int x){
    int hash = getHash(x, bucketsSize);

    LinkedList<Integer> curBucket = buckets[hash];
    if(curBucket.contains(x)){
        return false;
    }
    curBucket.push(x);
    if( (float)size/bucketsSize>COLLISION\_CHANCE){
        resize();
    }
    size++;
    return true;
}
```

Contains

To check if a hash set contains an element, we get the hash code from our hashing algorithm and check if the corresponding bucket contains the element.


Buckets
0
1
2
3
4
5
6
7
8
9
3242
3523
123
235
538
123
Hash


Buckets
0
1
2
3
4
5
6
7
8
9
3242
3523
123
235
538
123
Hash

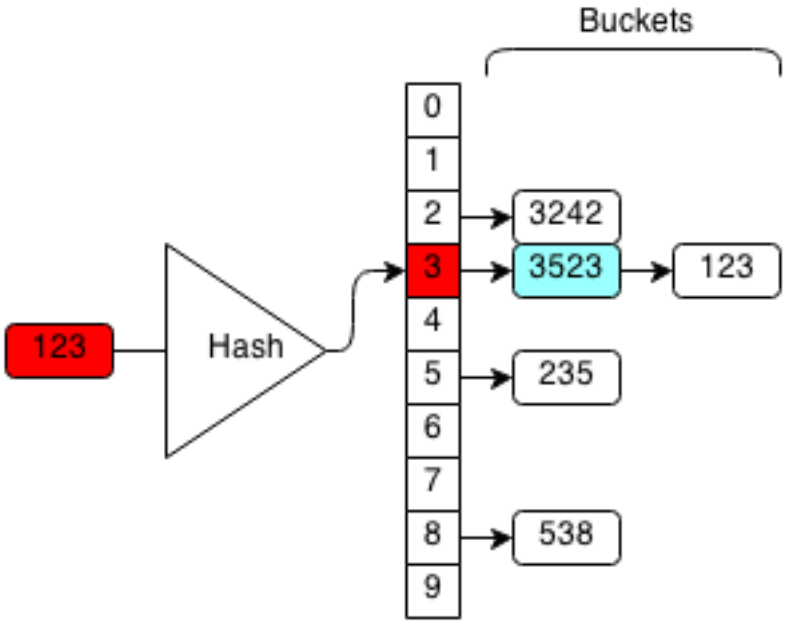


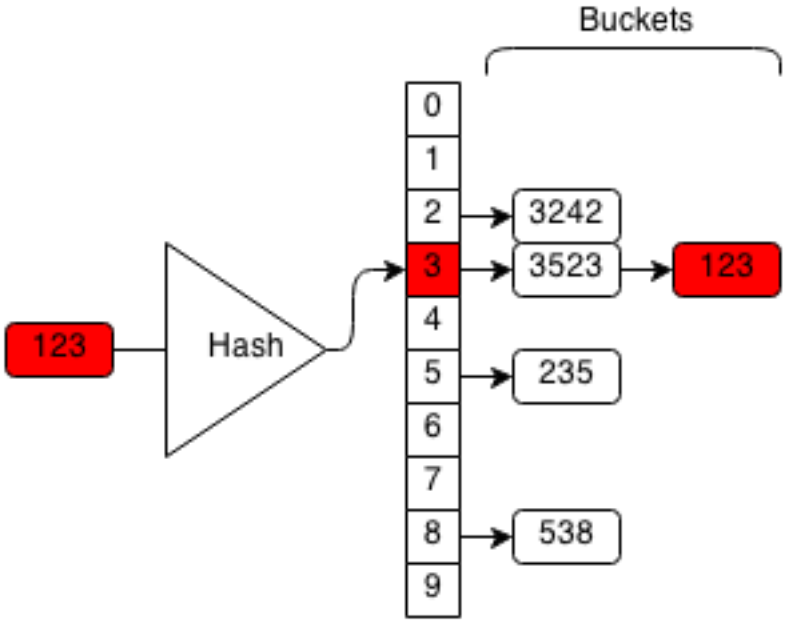

```
public boolean contains(int x){
    int hash = getHash(x, bucketsSize);
    LinkedList<Integer> curBucket = buckets[hash];
    return curBucket.contains(x);
}
```

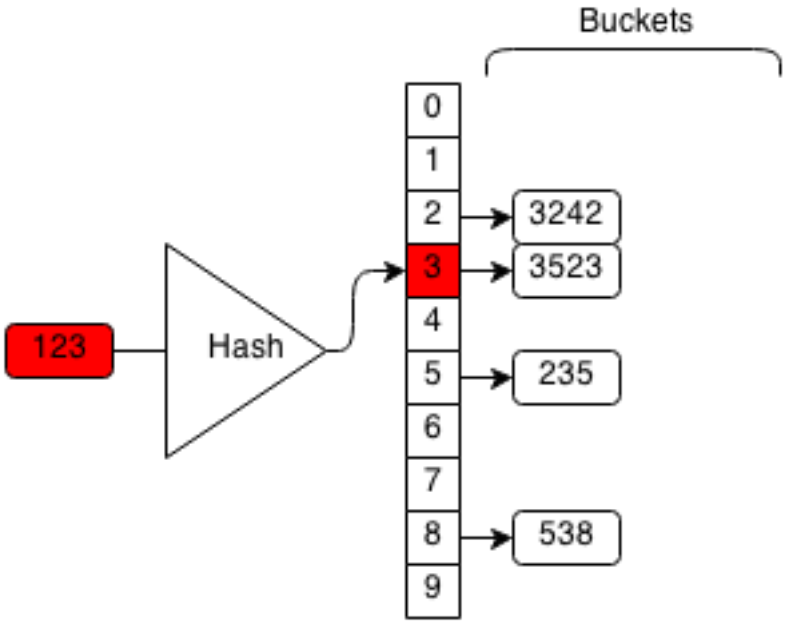
Remove

To remove an element from a hash set, we get the hash code from our hashing algorithm and remove the element from the corresponding bucket.

The function will return whether or not the operation was successful. If the bucket contains the element we can remove it from the linked list and the operation is successful. If the element is not in the bucket then the operation fails because we cannot remove something that is not there.


Buckets
0
1
2
3
4
5
6
7
8
9
3242
3523
123
235
538
123
Hash


Buckets
0
1
2
3
4
5
6
7
8
9
3242
3523
123
235
538
123
Hash




```
public boolean remove(int x){
    int hash = getHash(x,bucketsSize);

    LinkedList<Integer> curBucket = buckets[hash];
    if(curBucket.remove((Integer)x)){
        return true;
    }
    return false;
}
```

5.9.2 Exercises

1. Try to come up with a better hashing algorithm
2. Calculate the probability of a collision occurring given the number of buckets and number of elements in the hash set
3. Given an array of numbers, find the number of pairs of numbers that sum to 0.
4. Given an array of numbers and a number A, find the number of pairs of numbers that sum to A.

5. Given an array of numbers and a number A, find the number of quadruples that sum to A.

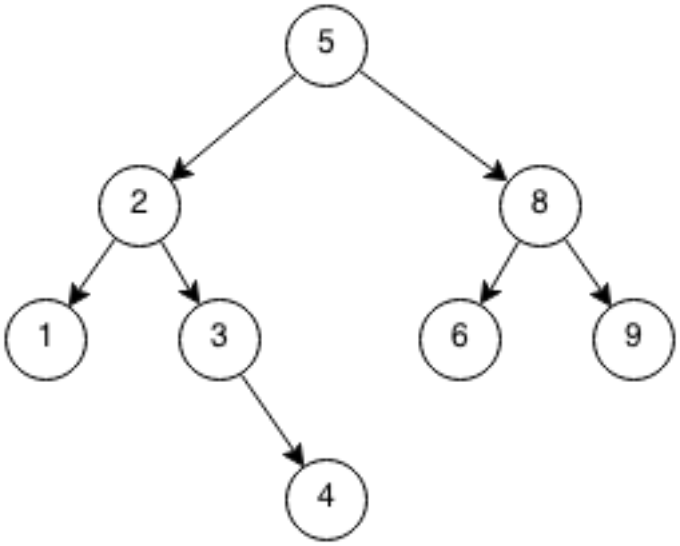
5.10 Tree Set

Prerequeisites: Sets, Binary Search Tree

Source on Github

A tree set is a set which stores the values in a binary search tree. To store elements in a tree set, they must be able to be sorted by a property. To insert an element, it is added to the binary tree. To delete an element, it is removed from the binary tree. To check for membership, we do a binary search for the element in the binary tree.

The advantage of tree sets is that they are maintained in a sorted order.



Operation	Membership	Insertion	Deletion
Time Complexity	O(log n)	O(log n)	O(log n)

5.10.1 Implementation

Tree Sets are implemented using binary search trees.

5.10.2 Exercises

1. Given a list of names, output all the unique names in alphabetical order

5.11 Maps

A map is an abstract data type that stores key-value pairs.

Imagine you had a English dictionary. If you look up a word, you can find it's definition and read it out. For example if you looked up the word 'cat' in the English dictionary, you would look through the dictionary alphabetically until you found the word 'cat' and then you would look at the definition: 'a feline animal'. If you really wanted to, you could also add your own words into the dictionary and the definitions of your words. This type of structure is called a map.

Maps (also called dictionaries) are abstract data types that store pairs of key-values and can be used to look up values from the keys. The keys are like the words in an English dictionary and the definitions can be seen as the values. Maps are able to support insertion of key-value pairs, retrieve values from keys, and delete key-value pairs.

Prerequisites

- Sets

5.11.1 Implementation

Type	Get	Insertion	Deletion
Tree Map	O(log n)	O(log n)	O(log n)
Hash Map	O(1)	O(1)	O(1)

5.11.2 Exercises

1. Given a list of N strings, output the strings in alphabetical order and the number of times they appear in the list. 2, Given two list of N strings, output the

5.12 Hash Map

Prerequisites: Map, Hash Set

Hash maps are maps that use hash sets to store the keys.

5.12.1 Implementation

Here is a Java implementation of a hash map which is a modified version of a hash set.

Class

Inside our implementation of a hash map we will store the buckets using an array of linked lists, the number of buckets, and the number of elements in the set.

The collision chance is the threshold for resizing the hash set. When the ratio of elements in the set to number of buckets is greater than the threshold, then the chance of collision will be high enough that it will slow down the operations. The lower this ratio, the better performing a hash set will be.

```
public class HashMap {

    public LinkedList<Pair>[] buckets;
    public int bucketsSize = 10;
    public int size = 0;
    public static final double COLLISION\_CHANCE = 0.3;

    public HashMap(){
        buckets = new LinkedList[10];
        for(int i=0;i<bucketsSize;i++){
            buckets[i] = new LinkedList<Pair>();
        }
        size = 0;
    }
}
```

Hash

The hash code is the result of the hashing algorithm for an element. In our hash set implementation, we will use a simple hash: modulus of the integer by the number of buckets.

For the most part if the numbers are all random then the hash function is fine. However, if the number of buckets was 10 and we added the elements 20,30,40,50,60,70, they will all end up in the same bucket and results in poor performance.

```
public int getHash(int x,int hashSize){
    return x \% hashSize;
}
```

Resize

A hash map must be able to resize. When the ratio of number of elements to number of buckets, the chance of collision will increase more and more. So we must able to resize the number of buckets to support the number of elements to lower the chance of collision.

To resize efficiently, we can create two times the number of buckets and set them to empty and then insert all the elements in the old buckets to the new buckets.

```
public void resize(){
    int newBucketsSize = bucketsSize*2;
    LinkedList<Pair>[] newBuckets = new LinkedList[newBucketsSize];
    for(int i=0;i<newBucketsSize;i++){
        newBuckets[i] = new LinkedList<Pair>();
    }
    for(int i=0;i<bucketsSize;i++){
        for(Pair p:buckets[i]){
            int hash = getHash(p.key,newBucketsSize);
            newBuckets[hash].push(p);
        }
    }
    buckets = newBuckets;
    bucketsSize = newBucketsSize;
}
```

Insert

To insert an element in a hash set, we get the hash code from our hashing algorithm and insert the element into the corresponding bucket.

The function will return method or not the operation was successful. If the bucket already contains the element the operation will stop because we do not want to add duplicate elements into the set. If the bucket does not contain the element, we will insert it into the bucket and the operation is successful.

```
public boolean insert(Pair p){
```

```
    int hash = getHash(p.key, bucketsSize);

    LinkedList<Pair> curBucket = buckets[hash];
    if(curBucket.contains(p.key)){
        return false;
    }
    curBucket.push(p);
    if( (float)size/bucketsSize>COLLISION\_CHANCE){
        resize();
    }
    size++;
    return true;
}
```

Get

To get the value from a hash set from a key, we get the hash code from our hashing algorithm of the key and find the key-value pair in the corresponding bucket.

```
public Pair get(int key){
    int hash = getHash(key, bucketsSize);
    LinkedList<Pair> curBucket = buckets[hash];
    for(Pair p:curBucket){
        if(p.key==key){
            return p;
        }
    }
    return null;
}
```

Remove

To remove an element from a hash set, we get the hash code from our hashing algorithm and remove the element from the corresponding bucket.

The function will return whether or not the operation was successful. If the bucket contains the element we can remove it from the linked list and the operation is successful. If the element is not in the bucket then the operation fails because we cannot remove something that is not there.

```
public boolean remove(int key){
```

```
        int hash = getHash(key,bucketsSize);

        LinkedList<Pair> curBucket = buckets[hash];
        for(Pair p: curBucket){
            if(p.key==key){
                curBucket.remove(p);
                return true;
            }
        }
        return false;
    }
```

5.12.2 Exercises

1. Create a hash map for the English dictionary (word as keys, definition as value). You will need to create a hash function for strings.

5.13 Tree Map

A tree map is a map that stores the key value pairs in a tree set.

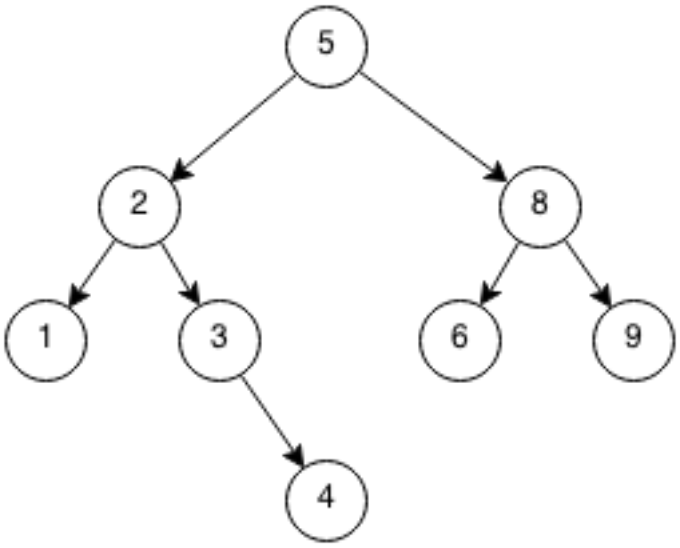
Operation	Membership	Insertion	Deletion
Complexity	O(log n)	O(log n)	O(log n)

Prerequisites

- Tree Set
- Map

5.13.1 Implementation

Here is a Java implementation of a tree map:



Class

A pair is a key with a value. In this implementation we will use the value as a string.

```
class Pair{
    int key;
    String value;
    public Pair(int key,String value){
        this.key = key;
        this.value = value;
    }
}
```

A node is a node contained in the binary search tree. The node must store the child nodes and for simplicity of the implementation, we will store the parent node as well. Each node will also have a key value pair associated with it.

```
class Node{
    Pair pair;
    Node left;
    Node right;
    Node parent;
    public Node(Pair p){
```

```
        this.pair = p;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
    public void replaceChild(Node child,Node replacement){
        if(left==child){
            left = replacement;
                if(replacement!= null){
                    replacement.parent = this;
                }
            }
            if(right==child){
                right = replacement;
                if(replacement!= null){
                    replacement.parent = this;
                }
            }
        }
    }
}
```

In our tree map, we will store the root node (ancestor of all nodes) and the number of numbers.

```
public class TreeMap {

    int size;
    Node root;

    public TreeMap(){
        size = 0;
        root = null;
    }
}
```

Insert

To insert a key-value pair into the tree set, we first find where the key should be. If the key already exists,

```
public boolean insert(Pair p){
```

```
    if(root==null){
        root = new Node(p);
        return true;
    }
    Node curTree = root;
    while(curTree != null){
        if(p.key == curTree.pair.key){
            return false;
        }else if(p.key < curTree.pair.key){
            if(curTree.left == null){
                Node newTree = new Node(p);
                newTree.parent = curTree;
                curTree.left = newTree;
                return true;
            }
            curTree = curTree.left;
        }else {
            if(curTree.right == null){
                Node newTree = new Node(p);
                newTree.parent = curTree;
                curTree.right = newTree;
                return true;
            }
            curTree = curTree.right;
        }
    }
    return false;
}
```

Get

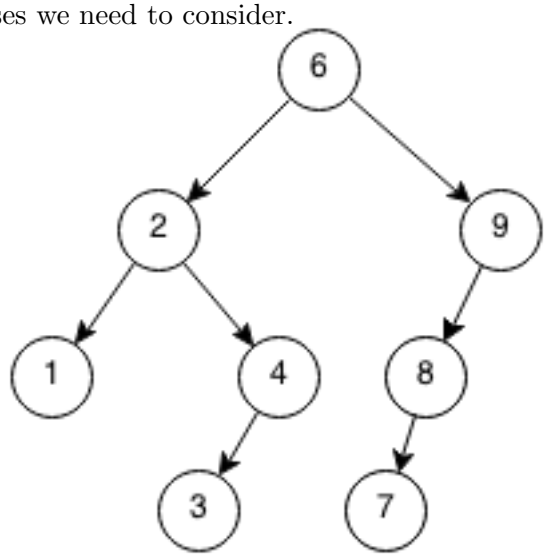
To get the value from a key stored in a tree set, we binary search for the key and then retrieve the key-value pair located at the node.

```
public Pair get(int key){
    Node curTree = root;
    while(curTree!=null){
        if(key==curTree.pair.key){
            return curTree.pair;
        }else if(key<curTree.pair.key){
```

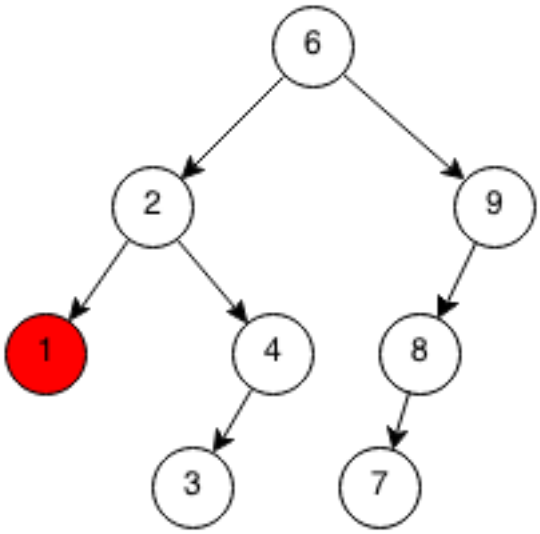
```
                curTree = curTree.left;
            }else{
                curTree = curTree.right;
            }
        }
        return null;
}
```

Remove

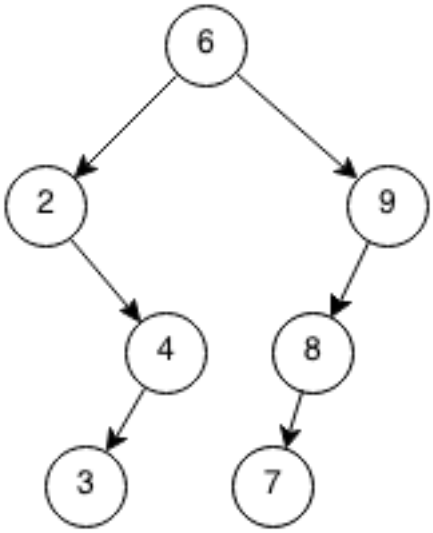
Removing an element is a much more complex because we need to maintain the tree structure of the tree set when removing elements. First we locate the element that we want to remove. If the element is not there then the operation failed and we return false. If the element is there then are three cases we need to consider.



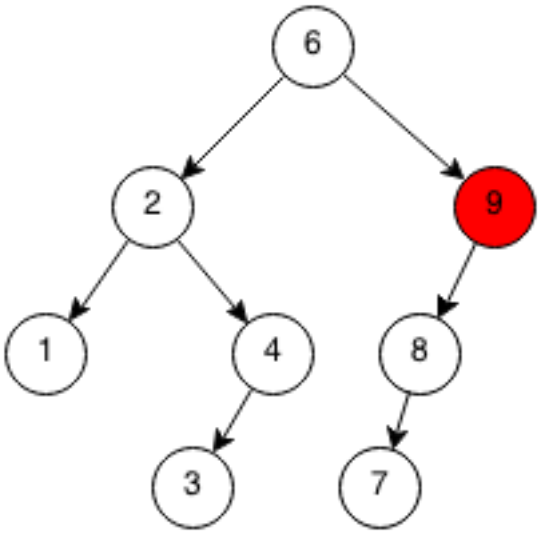
Case 1: Node is a leaf node



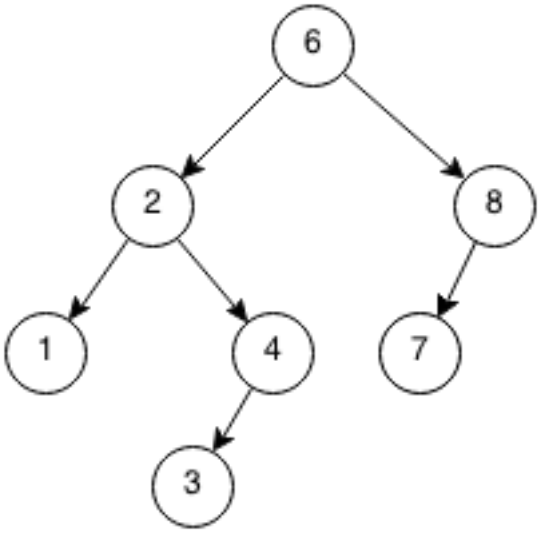
If the node we want to remove is the leaf node, we can simply remove it.



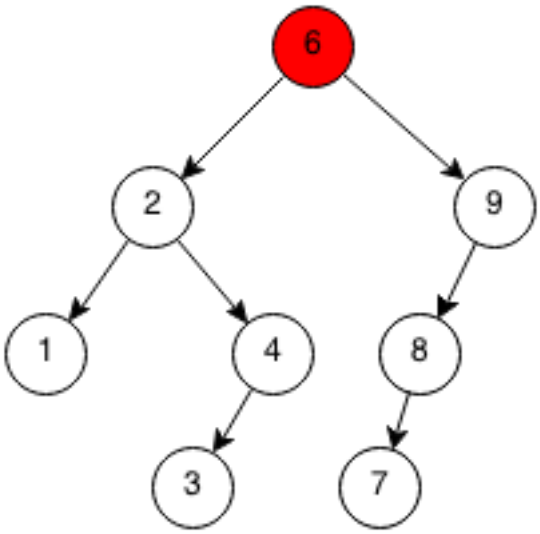
Case 2: Node has one child



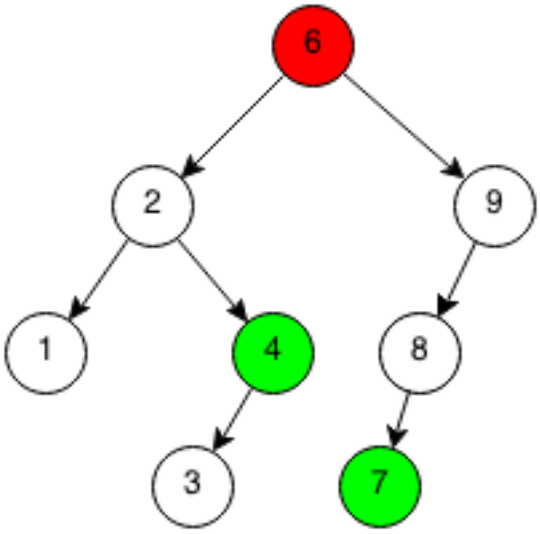
If the node we want to remove has a child, we can replace that node with its' only child.



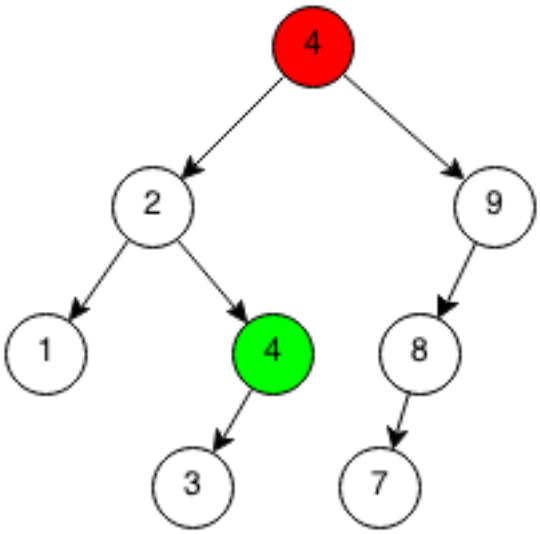
Case 3: Node has two children



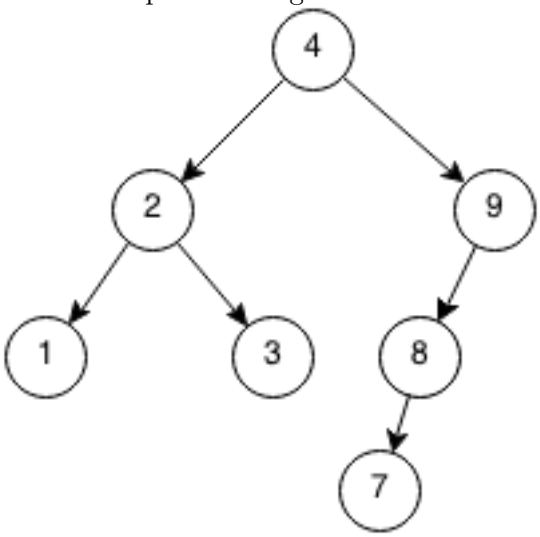
We need to replace the node with the rightmost of the left subtree or the leftmost of the right subtree to maintain the order.



It does not matter which side we pick so we will use the rightmost of the left subtree. First we copy the value of the rightmost of the left subtree into the node that will be deleted.



Then we replace the rightmost of the left subtree with its left subtree.




```
public boolean remove(int key){
    //Get node to remove
    Node curNode = root;
    while(curNode!=null){
        if(key==curNode.pair.key){
            break;
        }else if(key<curNode.pair.key){
            curNode = curNode.left;
```

```
        }else{
            curNode = curNode.right;
        }
    }
    if(curNode==null){
        return false;
    }
    //Case 1: leaf node
    if(curNode.left==null \&\& curNode.right==null){
        //if root
        if(curNode==root){
            this.root = null;
        }else{
            curNode.parent.replaceChild(curNode,null);
        }
    }
    //Case 2: one child
    else if(curNode.left==null){
        //If root
        if(curNode==root){
            root = curNode.right;
            root.parent = null;
        }else {
            curNode.parent.replaceChild(curNode,curNode.right);
        }
    }
    else if(curNode.right==null){
        //If root
        if(curNode==root){
            root = curNode.left;
            root.parent = null;
        }else {
            curNode.parent.replaceChild(curNode,curNode.left);
        }
    }
    //Case 3: two children
    else {
        //Get rightmost of left subtree
        Node rightmost = curNode.left;
        while(rightmost.right!=null){
```

```
            rightmost = rightmost.right;
        }
        curNode.pair = rightmost.pair;
        rightmost.parent.replaceChild(rightmost, rightmost.left);
    }
    size--;
    return true;
}
```

5.13.2 Exercises

1. Given a list of N numbers, output the first M unique numbers.

5.14 Priority Queue

Prerequisites: Queue, Heap

Consider a waiting list for lung donors. The patients are given a score when they are placed on the waiting list by how much they need a lung based on their whether they smoke, risk factors, age, expected time left etc. When a lung is available, the patient with the highest score will get removed from the waiting list. During this time, more patients could be added to the queue. The behaviour is similar to a queue but instead of the first person getting in the queue getting a lung first, the person with the highest score will get it. This means that if Sam has a score of 60 and gets placed in the queue after Bob who has a score of 40, Sam will get the lung first even though Bob was in the queue before him.

A priority queue is an abstract data structure with two operations: push and pop. Push adds an element into the priority queue and pop removes the highest or lowest element.

A priority queue is usually implemented as a heap because it is the most efficient because of its structure.

5.14.1 Implementation

Implementation	Push	Pop
Heap	O(log n)	O(log n)

5.14.2 Applications

Priority queues are very efficient in its opertions O(log n) and it is used in many other algorithms such as:

- Dijkstra's
- Prim's
- Kruskal
- Line Sweeping

5.14.3 Exercises

1. Given a list of N numbers, find the M largest numbers. (Note you can do better than O(N log N))
2. Given N lists of N numbers, find the N largest numbers.

5.15 Heap

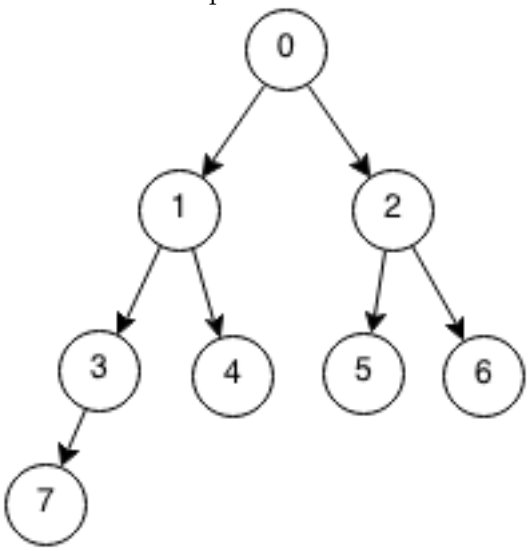
Prerequisites: Queue

Heaps are data structures that are able to pop the maximum or minimum value or push a value very quickly. Heaps are implemented as trees which have the property that a parent node must either be greater than all the elements in its left and right subtrees (a max heap) or less than all the elements in its left and right subtrees (a min heap). Priority queue's are most efficiently implemented as heaps. This guarantees that the maximum or minimum element is the root node.

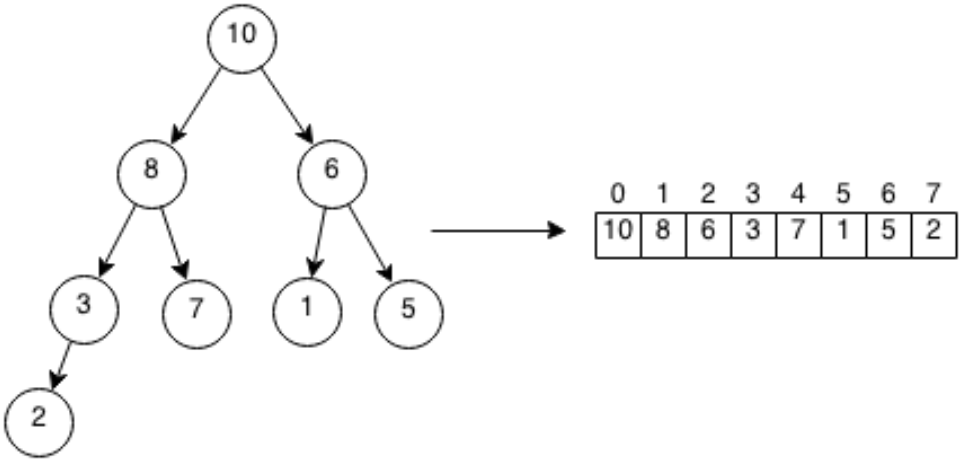
Heaps store their data level by level in a binary tree. This allows us to store heaps in an array. The root index is 0. For every node, the left index can be found by using the formula 2=ind+1 and the right index can be found by using the formula 2=ind + 2. The parent of a node can be found by integer division with (ind-1)/2.

```
root = 0
left = index * 2 + 1
right = index * 2 + 2
parent = (index-1)/2
```

Indexes of a heap



Example Heap:



A heap has two operations: push and pop. Pushing an element into a heap adds it into the heap and the heap needs to ensure that the properties of the heap still hold. Popping removes an element from the top of the heap and the heap needs to ensure that the properties of the heap still hold.

Operation	Heapify	Resize	Push	Pop
Time Complexity	O(n)	O(n)	O(log n)	O(log n)

5.15.1 Implementation

Here is an implementation of a max heap. A heap needs to be able to resize, push an element and pop an element.

Class

```
public class Heap {

    public int[] arr;
    public int size;

    public Heap(int startSize){
        arr = new int[startSize];
        size = 0;
    }
}
```

Heapify

Heapify takes a random array of N elements and transforms it into a heap. The runtime of heapify is O(N).

```
public void heapify(int arr[]){
    this.arr = arr;
    for(int i=0;i<Math.floor(arr.length/2.0);i++){
        int idx = i;
        while(idx<size){
            int left = idx*2+1;
            int right = idx*2+2;
            if(left<size \&\& arr[left]>arr[idx]){
                int swap = arr[left];
                arr[left] = arr[idx];
                arr[idx] = swap;
                idx = left;
            }else if(right<size \&\& arr[right]>arr[idx]){
                int swap = arr[right];
                arr[right]=arr[idx];
                arr[idx] = swap;
                idx = right;
            }else {
                break;
```

```
            }
        }
    }
}
```

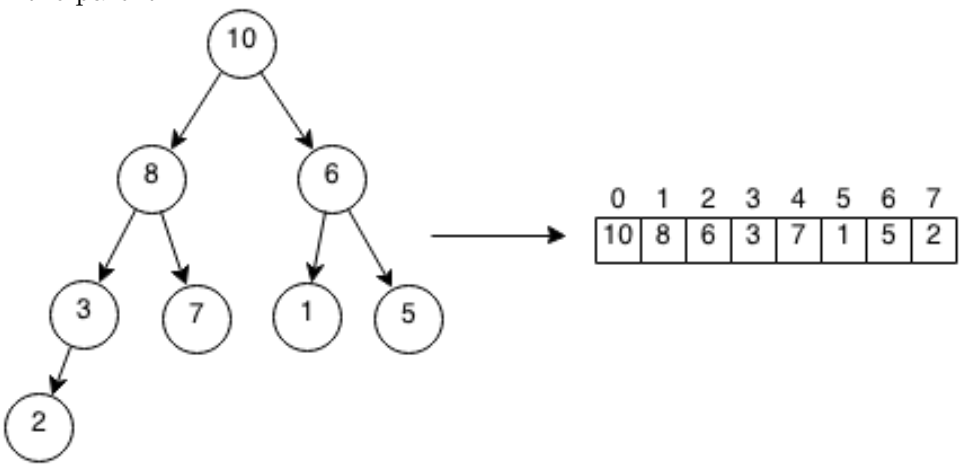
Resize

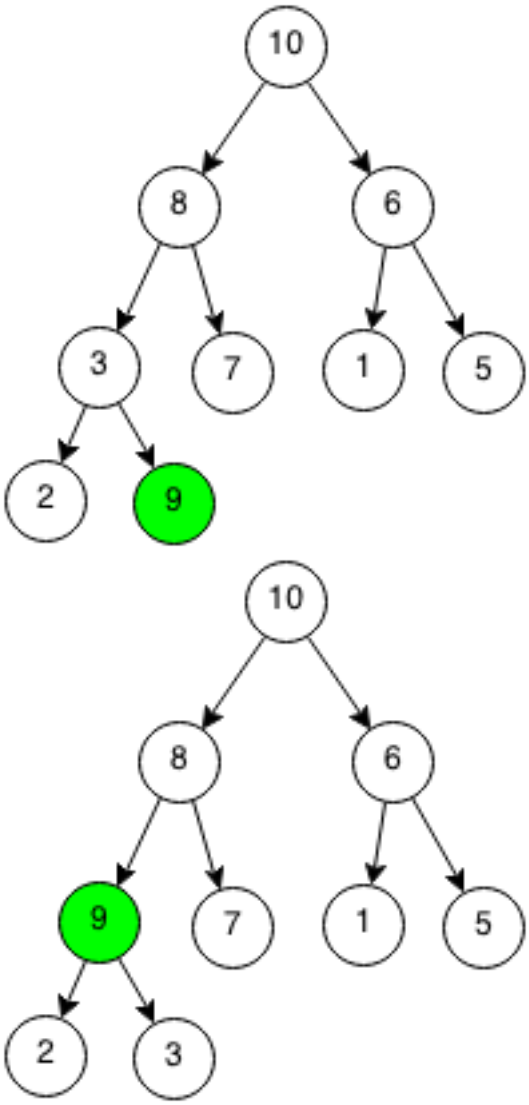
When the heap gets too full, we can resize it to make it bigger

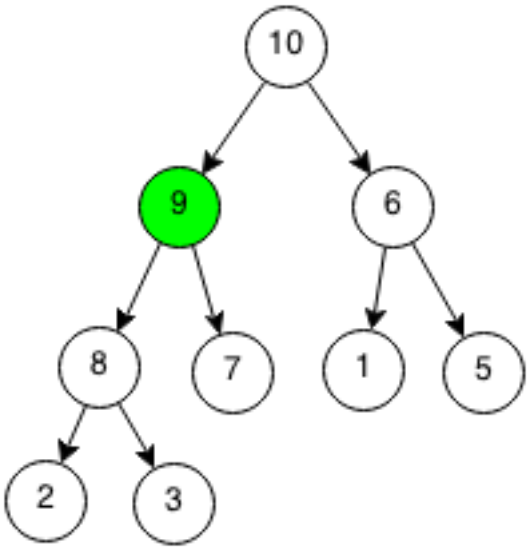
```
public void resize(){
    int[] newArr = new int[arr.length*2];
    for(int i=0;i<size;i++){
        newArr[i] = arr[i];
    }
    arr = newArr;
}
```

Push

Pushes the number x into the priority queue. We can do this by adding it to the bottom of the heap and then keep swapping it upwards if it is greater than the parent.




10
8
6
3
7
1
5
2
9
10
8
6
9
7
1
5
2
3

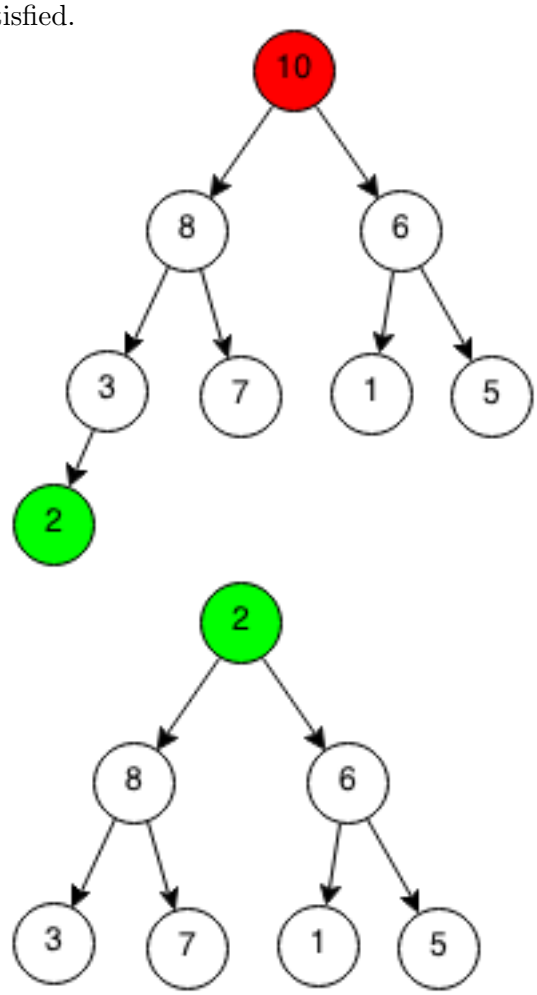


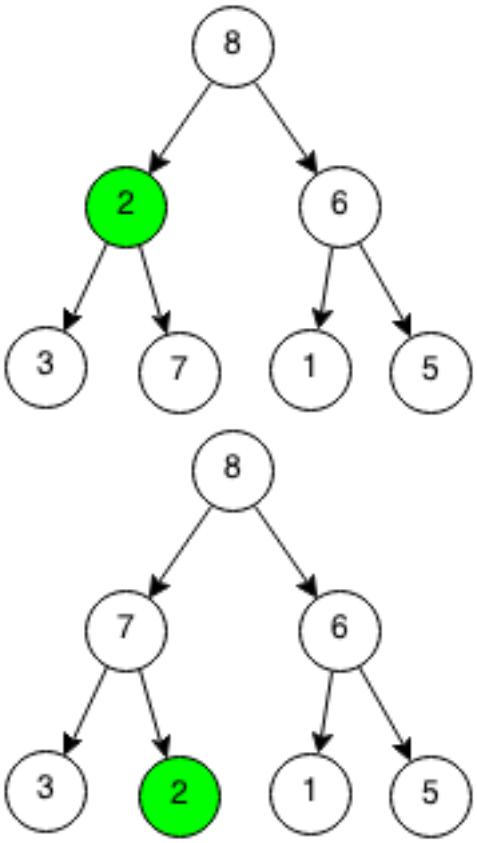

```
public void push(int x){

    if(size>=arr.length){
        resize();
    }
    arr[size] = x;
    size++;

    //Make sure parent is > child from the last element
    int idx = size-1;
    int parent = (idx-1)/2;
    while(idx>0 \&\& arr[parent]<arr[idx]){
        int swap = arr[parent];
        arr[parent] = arr[idx];
        arr[idx] = swap;
        idx = parent;
        parent = (idx-1)/2;
    }
}
```

Pop

Popping removes the greatest element in the priority queue by removing the root which is guaranteed to be the greatest as property of a heap. After

removing the root, we replace it with the element at the bottom of the heap and we can keep swapping it with its children until the heap property is satisfied.






```
public int pop(){
    if(size==0)return 0;
    int ret = arr[0];
    arr[0] = arr[size-1];
    size--;

    int idx = 0;

    while(idx<size){
        int left = idx*2+1;
        int right = idx*2+2;
        if(left<size \&\& arr[left]>arr[idx]){
            int swap = arr[left];
            arr[left] = arr[idx];
            arr[idx] = swap;
            idx = left;
```

```
        }else if(right<size \&\& arr[right]>arr[idx]){
            int swap = arr[right];
            arr[right]=arr[idx];
            arr[idx] = swap;
            idx = right;
        }else {
            break;
        }
    }
}
```

5.15.2 Applications

Heaps are very efficient in its opertions O(log n) and it is used in many other algorithms such as:

- Dijkstra's
- Prim's
- Kruskal
- Line Sweeping

5.15.3 Exercises

1. Implement a min heap
2. Prove heaps work

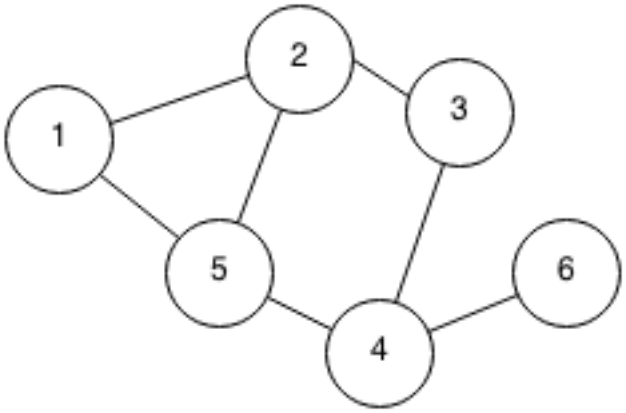
Chapter 6

Graph Theory

Introduction

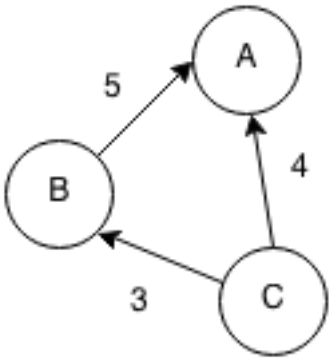
Next: Advanced Graph Theory

Graphs are a set of objects where some pairs of objects called nodes or verticies are usually connected by links called edges. The nodes here can be seen numbered from 1 to 6. There are edges connecting these various nodes.



A undirected graph is a graph where an edge from A to B is the same as the edge from B to A for all edges. The above graph is undirected.

A directed or bidirectional graph is a graph where edges have direction meaning if there is an edge from A to B then there may not be an edge from B to A.



A subgraph is a subset of edges and vertices within a graph.

A directed acyclic graph (DAG) is a graph with no directed cycles (see topological sorting).

A weighted graph is a graph that contains weights or values assigned to each edge or node. Usually these weights act as the cost to reach/use that node.

6.1 Advanced Graph Theory

Prerequisites: Graph Theory

Advanced topics on graph theory.

6.1.1 Bipartite Graph

A bipartite graph is a graph which can be partitioned into two sets such that no nodes in a set connect to another node in the same set.

6.1.2 Special Paths

A path is a certain order of visiting objects.

Hamiltonian Path

A Hamiltonian Path is a path that visits every node exactly once.

Eulerian Path

A Eulerian Path is a path that visited every edge exactly once.

6.1.3 Special Cycles

A cycle is a path that ends up at the same starting position.

Hamiltonian Cycle

A Hamiltonian cycle is a cycle that visits every node exactly once at ends back at the start.

Eulerian Cycle

A Eulerian Cycle is a cycle that visited every edge exactly once at ends back at the start.

Travelling Salesman Problem

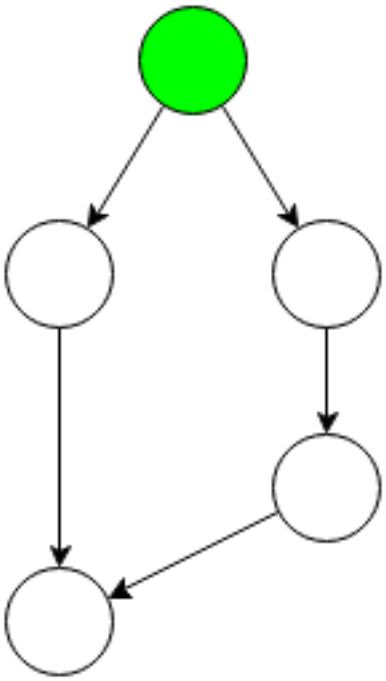
The Travelling salesman is the problem where a salesman wants to find a cycle that minimizes the total cost of weights used of edges.

6.1.4 Special Nodes

Some graphs may have nodes that have special properties.

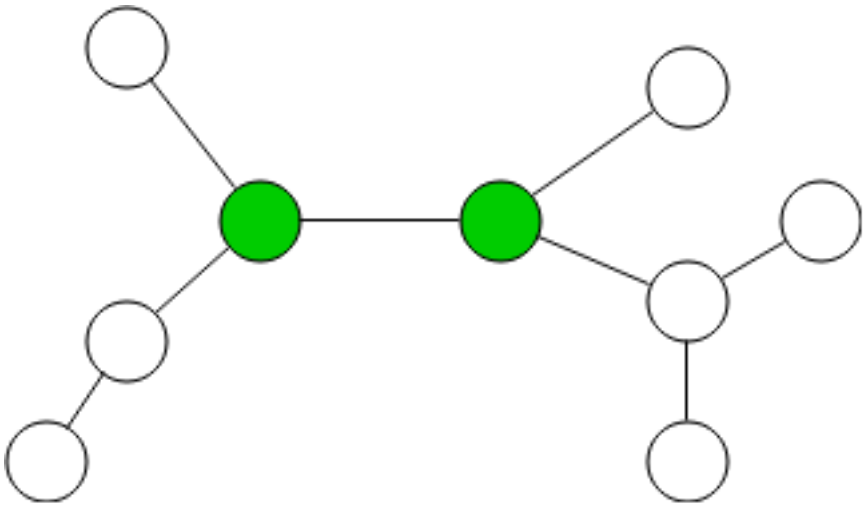
Root

A node in a directed acyclic graph that has no ancestors is a root.



Center

The center of a undirected tree is the node that minimizes the sum of the distance to every other node. The center can be found be continuously stripping away leaf nodes (nodes with only one edge) layer by layer until either 1 or 2 nodes remain. The longest path in a tree will contain the center.



6.1.5 Network Flow

Max Flow Problem

Min Cut Problem

Ford-Fulkerson

6.2 Adjacency Matrix

An adjacency matrix is a method of storing a graph using a two dimensional array. Given n nodes, the adjacency matrix can be stored in a n x n matrix.

Array[i][j] represents the weight between the node i and node j.

Example

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	0	1	0
3	0	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	0
6	0	0	0	1	0	0

6.2.1 Implementation

```
class edge{
```

```
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
}
public static int[][] getAdjMatrix(Vector<edge> edges){
    int n = 0;
    int adjMatrix[][] = new int[n][n];

    for(int i=0;i<n;i++)for(int j=0;j<n;j++)adjMatrix[i][j] = 0;

    for(int i=0;i<edges.size();i++){
        edge e = edges.get(i);
        adjMatrix[e.source][e.dest] = e.weight;
        adjMatrix[e.dest][e.source] = e.weight;
    }
    return adjMatrix;
}
```

6.3 Prim's

Prerequisites: Priority Queue, Minimum Spanning Tree

A minimum spanning tree is a tree in a graph that connects all the nodes using the smallest cost total cost of edges.

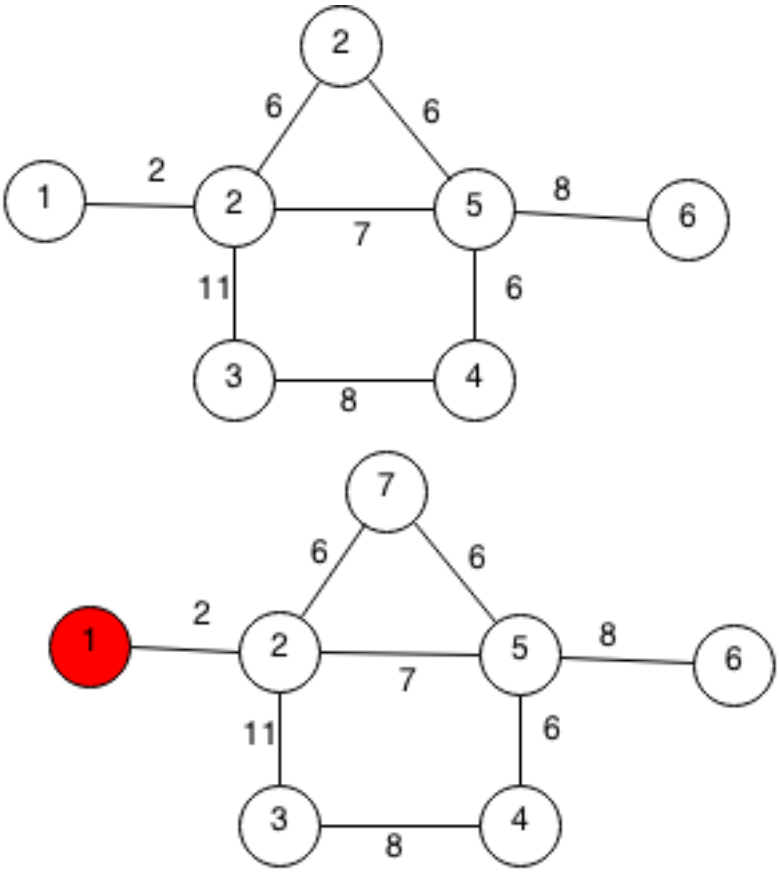
6.3.1 Implementation

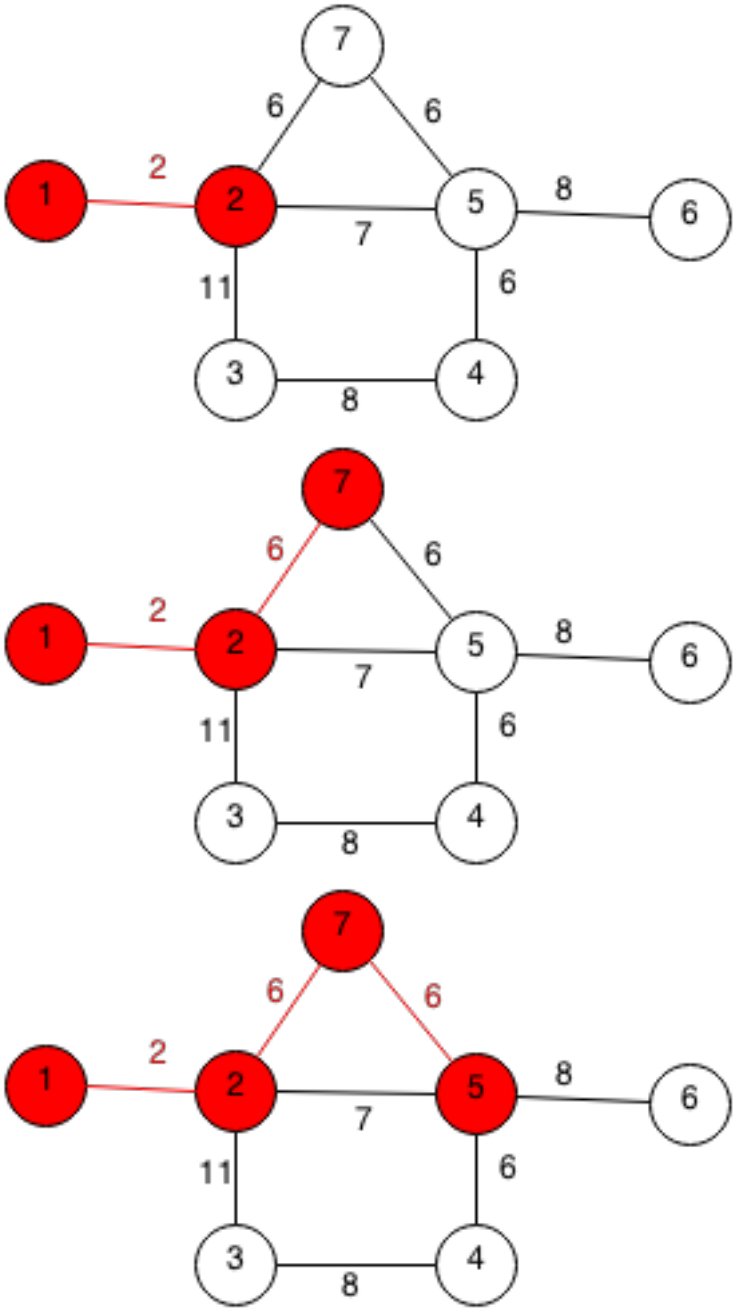
Prim's algorithm finds the minimum spanning tree using a greedy fashion. It works as such:

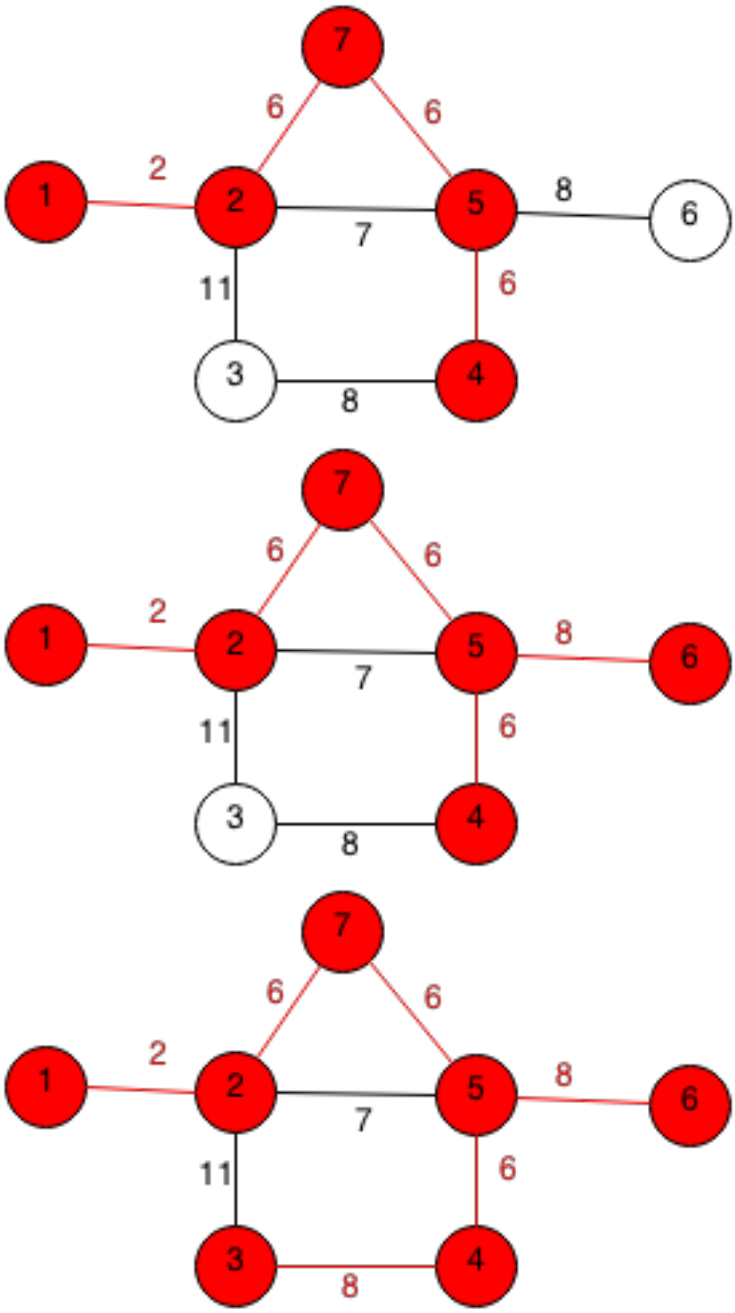
1. Pick an arbitrary node
2. Find the closest node to that node
3. Find the closest node to the 2 nodes
4. Find the closest node to the 3 nodes
5. ...
6. Find the closest node to n-1 nodes

The closest node is the node with the lowest cost edge to the already connected nodes.

Example




7
6
6
1
2
2
5
8
6
7
11
6
3
4
8
7
6
6
1
2
2
5
8
6
7
11
6
3
4
8
7
6
6
1
2
2
5
8
6
7
11
6
3
4
8


7
6
6
2
1
2
5
8
6
7
11
6
3
4
8
7
6
6
2
1
2
5
8
6
7
11
6
3
4
8
7
6
6
2
1
2
5
8
6
7
11
6
3
4
8

Java Code

In Java, we need to specify a comparison for the Priority Queue to order. We do this by implementing the Comparable class and overriding the compareTo method.

adjList is an adjacency list that is an array of arrays that store the graph.

```
class node implements Comparable<node>{
    int weight,index;
    public node(int weight,int index){
        this.weight = weight;
        this.index = index;
    }
    public int compareTo(node e){
        return weight-e.weight;
    }
}
public static int Prims(Vector<Vector<node>> adjList){
    int cost = 0;
    int n = adjList.size();
    PriorityQueue<node> pq = new PriorityQueue<node>();
    boolean visited[] = new boolean[n];
    for(int i=0;i<n;i++){
        visited[i] = false;
    }
    int inTree = 1;
    visited[0] = true;
    for(int i=0;i<adjList.get(0).size();i++){
        pq.add(adjList.get(0).get(i));
    }
    while(!pq.isEmpty()\&\&inTree<n){
        node cur = pq.poll();
        if(visited[cur.index])continue;
        inTree++;
        visited[cur.index]=true;
        cost+=cur.weight;
        for(int i=0;i<adjList.get(cur.index).size();i++){
            pq.add(adjList.get(cur.index).get(i));
                }
    }
```

```
    //Graph is not connected
    if(inTree<n)return -1;
    return cost;
}
```

6.3.2 Applications

In networking, if a cable company wanted to connect all the houses with the least amount of wiring, the minimum spanning tree can be found that finds the least total cost of wire.

Minimum spanning trees can also be used in generating mazes.

6.3.3 Exercises

1. Prove that Prim's algorithm works
2. Extends Prim's to output all the edges used
3. Given a weighted graph with n nodes, find the smallest total cost to connect all nodes into 3 separate groups. (A single node can be a group)
4. Same as 3, but a group must contain at least 3 other nodes.

6.4 Kruskal

Prerequisites: Sorting, Minimum Spanning Tree

A minimum spanning tree is a tree in a graph that connects all the nodes using the smallest cost total cost of edges.

Kruskal's algorithm finds the minimum spanning tree using connected components.

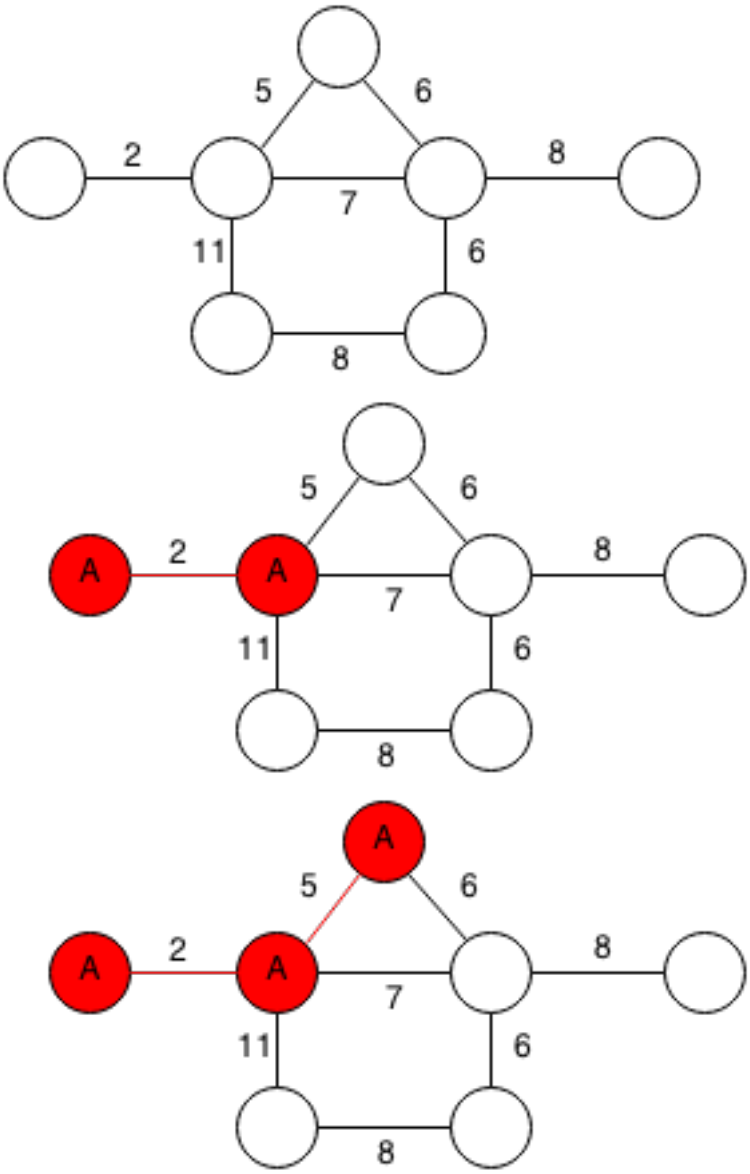
If implemented efficiently using a priority queue to get the edges with minimum weight or a sorting the edges, the runtime is O(n log n).

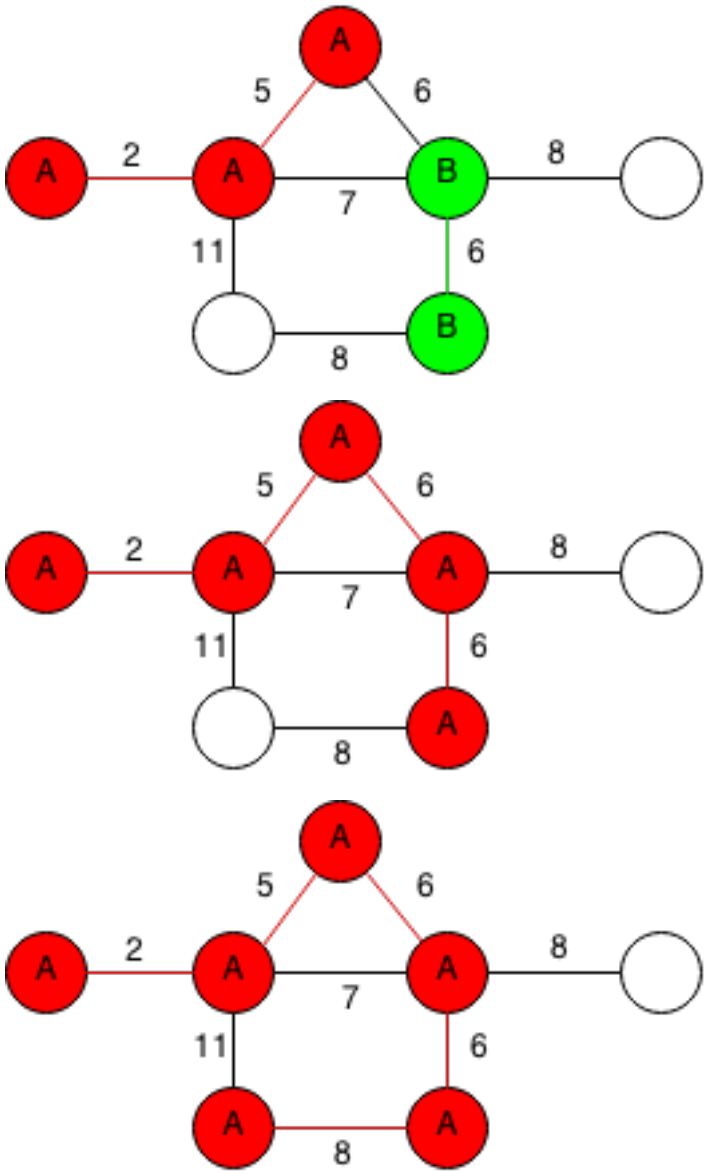
6.4.1 Implementation

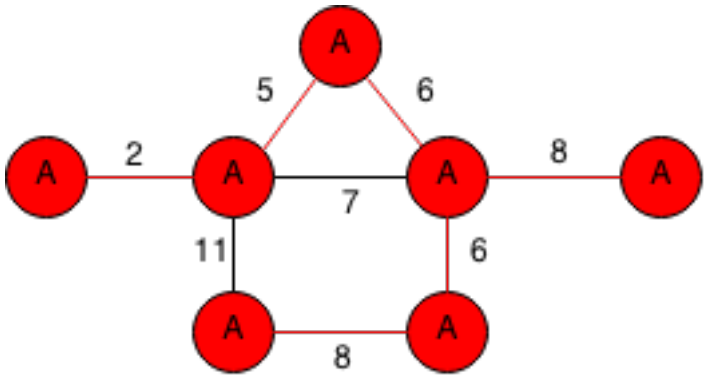
1. Uniquely label each node
2. Take the edge with the minimum weight
3. If the edge connects nodes A and B with different labels, all nodes with label B will be labeled with A. Otherwise, throw the edge away

4. Repeat 2-3 until all the nodes have the same label

Example




A
5
6
A
2
A
B
8
7
11
6
B
8
A
5
6
A
2
A
A
8
7
11
6
A
8
A
5
6
A
2
A
A
8
7
11
6
A
A
8



Code

```
class edge implements Comparable<edge>{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
    public int compareTo(edge e){
        return weight-e.weight;
    }
}
public static int getParent(int parents[],int x){
    if(parents[x]==x)return x;
    parents[x] = getParent(parents,parents[x]);
    return parents[x];
}
public static int Kruskal(Vector<Vector<edge>> adjList){
    int n = adjList.size();
    int parents[] = new int[n];
    for(int i=0;i<n;i++)parents[i] = i;
    int sum = 0;

    PriorityQueue<edge> edges = new PriorityQueue<edge>();

    for(int i=0;i<n;i++){
```

```
            for(int j=0;j<adjList.get(i).size();j++){
                edges.add(adjList.get(i).get(j));
            }
        }

        while(!edges.isEmpty()){
            edge e = edges.poll();
            if(getParent(parents,e.source)!=getParent(parents,e.dest)){
                parents[e.source] = getParent(parents,e.dest);
                sum+=e.weight;
            }
        }

        return sum;
    }
```

6.4.2 Applications

6.4.3 Exercises

1. Prove Kruskal's Algorithm works
2. Extends Kruskals's to output all the edges used
3. Given a weighted graph with n nodes, find the smallest total cost to connect all nodes into 3 separate groups. (A single node can be a group)
4. Same as 3, but a group must contain at least 3 other nodes.

6.5 Floyd Warshall

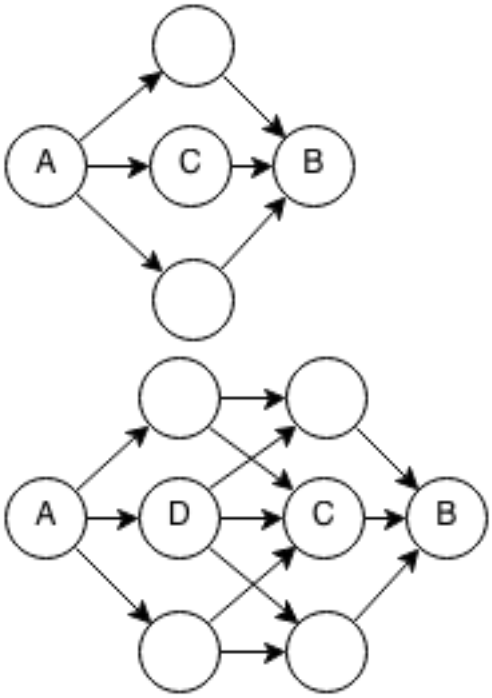
Prerequisites: Shortest Path, Dynamic Programming

Floyd Warshall is a algorithm for finding the shortest distances between all pairs of nodes in a graph. Floyd Warshall can be used to find negative cycles in the graph.

Description	Time	Space	Detect cycles?
Computes shortest path between all pairs of nodes	$O(n^3)$	$O(n^2)$	Yes

6.5.1 Implementation

Floyd-Warshall uses a dynamic programming approach to finding the shortest path between node A and node B. Every path from node A to node B can be rewritten as a path from A to some node in between plus the path from the node in between to node B. The shortest path from A to B can be found by finding a node C such the shortest path from A to C plus the shortest path from C to B is minimized.



Formalization

Recursion

```
Given a directed graph with N nodes and edges between nodes:
Let edge(i,j) be the weight of the edge from node i to node j in the graph
Let shortestPath(i,j) be the shortest path from i to j

Base Case:
shortestPath(i,i) = 0
shortestPath(i,j) = edge(i,j)
```

Recursion:
shortestPath(i,j) = mimmum of shortestPath(i,k) + shortestPath(k,j) for a

Code

```
class edge{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
}
public static final int UNDEFINED = Integer.MIN\_VALUE;

public static int[][] FloydWarshall(Vector<Vector<edge>> adjList){
    int n = adjList.size();
    //dist[i][j] is the minimum distance from i to j
    int[][] dist = new int[n][n];

    //initialize dist[i][j]
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            dist[i][j] = UNDEFINED;
        }
    }

    //dist[i][i] = 0
    for(int i=0;i<n;i++){
        dist[i][i]=0;
    }

    //initialize weights, dist[i][j] = edge from i to j
    for(int i=0;i<n;i++){
        for(int j=0;j<adjList.get(i).size();j++){

            edge e = adjList.get(i).get(j);
            dist[e.source][e.dest] = e.weight;
```

```
                System.out.println(e.source+" "+e.dest);
            }
        }

        for(int k=0;k<n;k++){
            for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                    //If dist[i][k] and dist[k][j] have been set then use those
                    if(dist[i][k]!=UNDEFINED\&\&dist[k][j]!=UNDEFINED){
                        //If the new distance is less than current or not used
                        int newDist = dist[i][k]+dist[k][j];
                        if(dist[i][j] > newDist || dist[i][j]==UNDEFINED){
                            dist[i][j] = newDist;
                        }
                    }
                }
            }
        }
        for(int i=0;i<n;i++){
            if(dist[i][i]<0){
                System.out.println("negative cycle");
            }
        }

        return dist;
    }
```

6.5.2 Applications

Floyd Warshall is useful when you want to find the shortest distance between all possible pairs of nodes.

6.5.3 Exercises

1. Prove Floyd Warshall works
2. Extend Floyd Warshall to reconstruct the paths from each pair of nodes

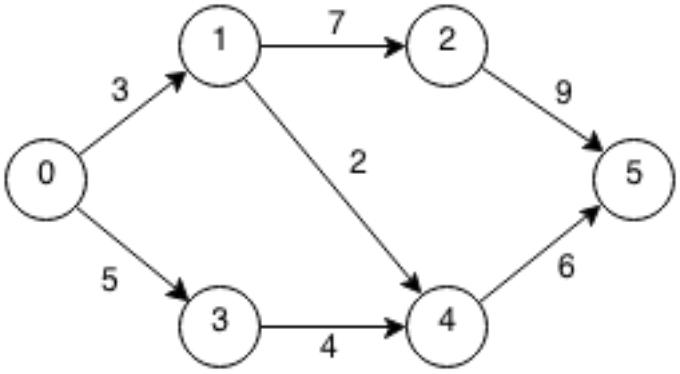
6.6 Bellman Ford

Prerequisites: Shortest Path

Bellman Ford is an algorithm that finds the shortest path from one source node to every other node in the graph. The running time is $O(n^2)$ and is slower than Dijkstra's but it is able to find negative cycles.

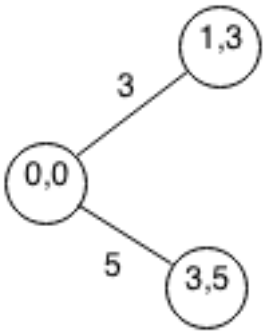
6.6.1 Implementation

Bellman Ford can be done using backtracking to find the shortest path in a graph. We first start at the starting node with starting cost of 0 and 0 edges used. For each node thats connected to that node, we repeat and add to the cost of the node.



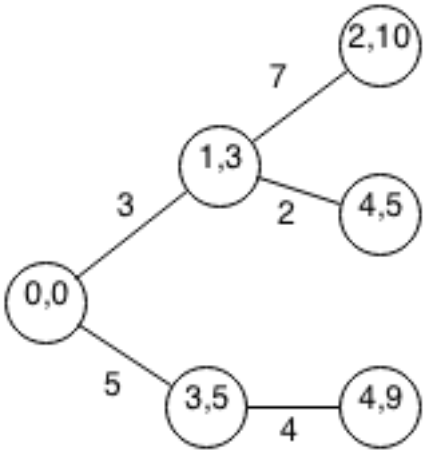
We will do an example of the Bellman Ford algorithm on the above graph. At each node we have the node index and the current weight to reach that node. We start at node 0 with a weight of 0.

From node 0, we can reach node 1 and node 3. At node 1, we have an accumulative weight of 3. At node , we have an accumulative weight of 5.



From node 1, we can reach node 2 and node 4 with respective accumulative weights of 10 and 5.

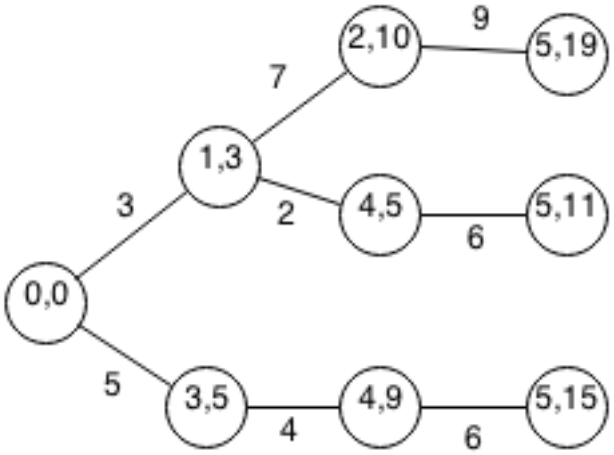
From node 3 we can reach node 4 with an accumulative weight of 9.



From node 2, we can reach node 5 with an accumulative weight of 19.

From node 4, we can reach node 5 with an accumulative weight of 11.

From node 4, we can reach node 5 with an accumulative weight of 15.




```
Let N be the number of nodes in the graph
Let edges an adjacency list of the graph where:
    edges[source] contains all edges of the graph where source is the sourc
An edge is represented as an object where:
    edge.weight is the weight of the edge
    edge.target is the target node of the edge
    edge.source is the source node of the edge
Let start be the starting node
Let shortestPath[target] be the shortest path from the source node to the

Let bellmanFord(target,n,w) be the shortest path from the source node to th

bellmanFord(target, n , w)

Base Case:
bellmanFord(target, N , w):
    stop

Recurrence:
bellmanFord(source, n, w):
    shortestPath[source] = min(shortestPath[source], w)
    bellmanFord(edge.dest, n + 1, w + edge.weight) for edge in edges[source

Init:
shortestPath = [0] * N
```

```
bellmanFord(start,0,0)
```

We can rewrite this solution using dynamic programming without recursion.

```
class edge{
    int weight,source,dest;
    public edge(int source,int dest,int weight){
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }
}

public static int BellmanFord(Vector<Vector<edge>> adjList,int startNode,i

    int n = adjList.size();
    //dist[i] is minimum distance from start to i
    int[] dist=new int[n];

    //used[i] is if dist[i] has been initialized
    boolean[] used = new boolean[n];

    //initialize dist[i]=0 and used[i]=false
    for(int i=0;i<n;i++){
        dist[i] = 0;
        used[i] = false;
    }
    used[startNode] = true;
    dist[startNode] = 0;
    for(int i=0;i<n-1;i++){
        //Iterate through adjacency list
        for(int j=0;j<n;j++){
            for(int k=0;k<adjList.get(j).size();k++){
                if(!used[j])continue;
                edge e = adjList.get(j).get(k);
                //If dist[e.source] has been used
                if(used[e.source]){
                    //If new dist < cur dist or not used, then update
                    int newDist = dist[e.source]+e.weight;
                    if(newDist<dist[e.dest] || !used[e.dest]){
```

```
                        used[e.dest]= true;
                        dist[e.dest] = newDist;
                    }
                }
            }
        }
    }

    for(int j=0;j<n;j++){
        for(int k=0;k<adjList.get(j).size();k++){
            edge e = adjList.get(j).get(k);
            //If negative cycle
            if(dist[e.source]+e.weight < dist[e.dest]){
                System.out.println("Negative cycle");
            }
        }
    }

    //If no path exists
    if(!used[endNode]){
        System.out.println("No path from start to end");
    }

    //Return distance from start to end
    return dist[endNode];
}
```

6.6.2 Applications

Arbitrage occurs when you can exchange currencies for another and make a profit. For example given a currency exchange table:

	USD	CAD	EURO
USD	/	1.12	0.72
CAD	0.90	/	0.64
EURO	1.38	1.56	/

Notice that 1 USD -¿ 1.12 CAD -¿ 1.008 USD. Bellman Ford can be used to find methods of arbitrage by using the vertex as currency and edges as transactions, and the weight as the exchange rate. All that is needed is to find a path that maximizes product of weights and finding a negative cycle.

6.6.3 Exercises

1. Write a program that detects a path for arbitrage to occur
2. Prove Bellmand-Ford works

6.7 Dijkstra's

Prerequisites: Shortest Path, Priority Queue, Greedy Algorithm

Dijkstra's is a greedy approach to find the shortest path in a graph with positive weights. It has many useful applications in networking and it can be extended to a variety of problems.

Dijkstra works by beginning at the starting node repeatedly picking the next closest node of those already visited.

If Dijkstra's is implemented using a priority queue, the run time is O(n log n).

If a negative cycle exists within the graph, then the algorithm breaks as it will repeatedly try to take the negative edges. See Bellman Ford to find negative cycles in a graph.

A naive fix for negative cycles would be to offset all edges by the largest negative edge and then subtract it from the resulting total but this does not work. Consider an example where you have start node A and end node B. The first route from A-¿B has length 2 and the second route has lengths 1,1,-2. Clearly the second route has less cost. If we try to make the length positive by adding all costs by 2, we will have the first path of length 4 and the second path of lengths 3,3,0 and the first route becomes the smallest total cost which is wrong.

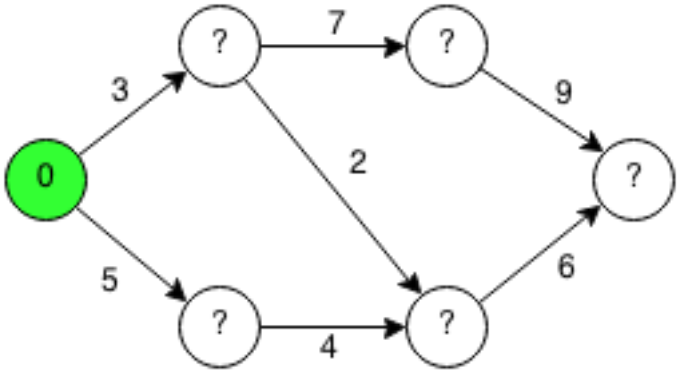
6.7.1 Implementation

At each node we visit we keep track of the minimum cost it takes to reach to reach that node from the starting node.

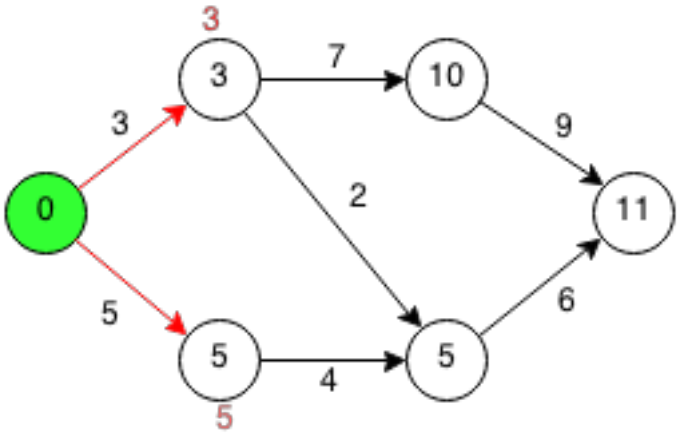
1. Start at the starting node
2. Find an unvisited node that has the least cost to reach from the visited nodes.
3. Mark that node as visited
4. Repeat until all nodes are visited

When we reach a node for the first time, it will be the shortest path to that node (Try to prove this to yourself).

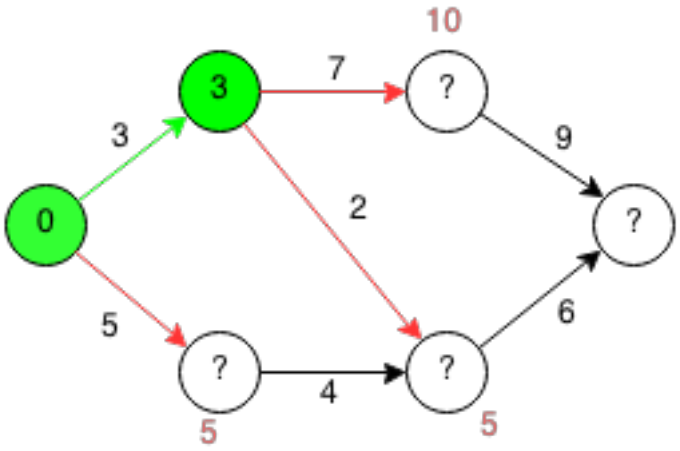
We first start at the starting node. The distance from the starting node to the starting node is obviously 0.



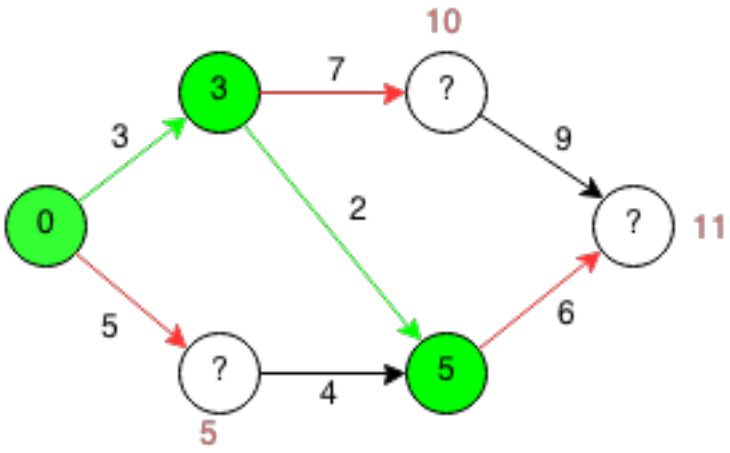
From the starting node we have two nodes we can reach. The top node has a cost of 3 to reach and the bottom node has a cost of 5 to reach.



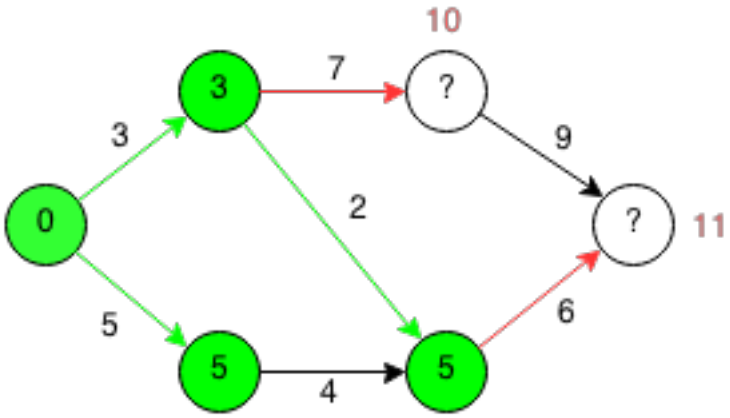
We pick the smallest node the reach and we mark it as visited. Once we visit a node, we can guarantee that it is the smallest cost to reach it. The next nodes minimum cost to reach is 10, 5 and 5.



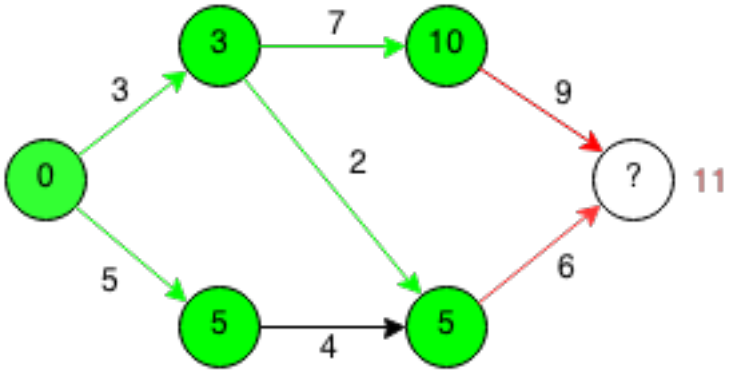
We are indifferent to both 5's as they are both the minimum and we can choose either. We mark the node as visited and we find the minimum costs to other nodes which are 5,10,11.



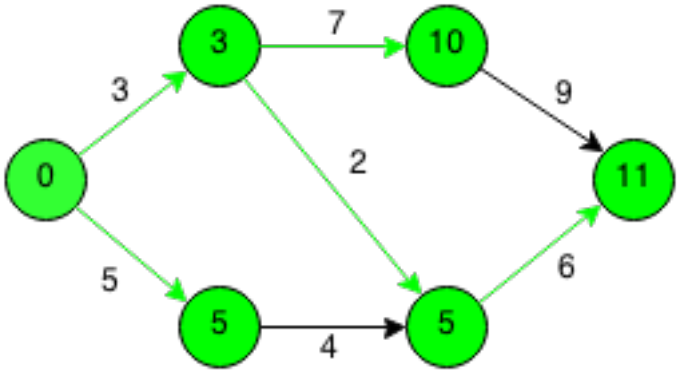
We take the next smallest which is 5 and we mark the node as visited. The next costs are 10 and 11.



We take the smallest which is 10 and we now only have one last node to reach at a cost of 11.



At each node we have the minimum cost to get from the start node to each node.



Java

Implementation of Dijkstra in Java using a priority queue.

```
class node implements Comparable<node>{
    int weight,index;
    public node(int weight,int index){
        this.weight = weight;
        this.index = index;
    }
    public int compareTo(node e){
        return weight-e.weight;
    }
}

public static int dijkstra(int[][] adjMatrix,int start,int end){
    int n = adjMatrix.length;
    PriorityQueue <node> pq = new PriorityQueue<node>();
    boolean visited[] = new boolean[n];
    for(int i=0;i<n;i++)visited[i] = false;
    pq.add(new node(0,start));
    while(!visited[end] \&\& !pq.isEmpty()){
        node curNode = pq.poll();

        if(visited[curNode.index])continue;
        visited[curNode.index] = true;
        if(curNode.index==end){
            return curNode.weight;
        }
        for(int i=0;i<n;i++){
            if(adjMatrix[curNode.index][i]>0 \&\& !visited[i]){
                int newWeight = curNode.weight+adjMatrix[curNode.index][i]
                pq.add(new node(newWeight,i));
            }
        }
    }
    return -1;
}
```

6.7.2 Applications

In general, Dijkstra is usually the goto method for finding the minimum cost between two nodes in any kind of network. For example, Dijkstra can be used in networking to find the shortest path between two hosts. It can also be used in flight networking to find the cheapest cost to get from one airport to another airport.

6.7.3 Practice Exercises

1. Extend Dijkstra's to find the exact path from start to end (the order of nodes of the shortest path A-¿B-¿C)
2. Extend Dijkstra's to find the best three minimal costs with unique paths from the start node to the end node
3. Prove that Dijkstra's algorithm works

6.8 Cycle detection

A cycle occurs in a graph when a duplicate node is encountered when traversing a tree using a depth first search. In other words, a cycle occurs when you can reach the same node again.

An undirected graph where the number of edges is greater than or equal to the number of nodes will always have cycles.

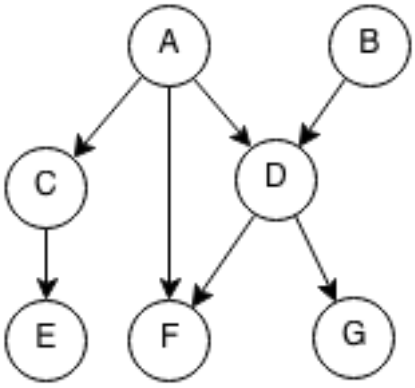
6.8.1 Implementation

```
public static boolean hasCycle(int [][] adjMatrix){
        int[] visited = new int[adjMatrix.length];
        for(int i=0;i<adjMatrix.length;i++)visited[i] = 0;
        for(int i=0;i<adjMatrix.length;i++){
            if(hasCycleAt(adjMatrix,i,visited))return true;
        }
        return false;
    }

    public static boolean hasCycleAt(int[][] adjMatrix,int i, int visited[]
        if(visited[i] == 1)return true;
        visited[i] = 1;
        for(int j = 0; j < adjMatrix.length;i++){
```

```
            if(adjMatrix[i][j] > 0){
                if(hasCycleAt(adjMatrix,j,visited))return true;
                visited[j] = 0;
            }
        }
        visited[i] = 2;
        return false;
    }
```

6.9 Topological Sorting

A topological sort or topological order of a directed graph is an order in which every node will come after its ancestors.



For example topological orders could be:

- (A, B, C, D, E, F, G)
- (B, A, D, C, F, E, G)
- (B, A, D, G, F, C, E)

But (B, A, C, F, D, E, G) is not a topological ordering because D is an ancestor of F and it comes after F.

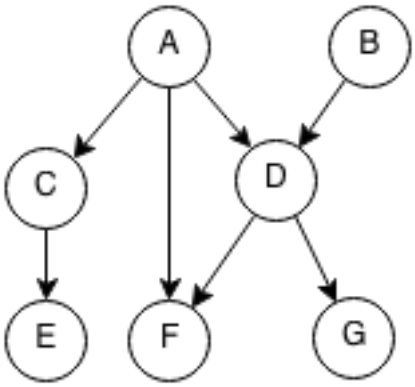
Prerequisites

- Graph Theory
- Depth First Search

6.9.1 Implementation

Topological sort can implemented in O(n) time using DFS for a directed acyclic graph (a digraph with no cycles). How it works:

1. Start with an empty top order
2. Pick any unmarked node
3. Get the DFS preordering from that node for unvisited nodes
4. Add the DFS to the head of the current order
5. Mark every node that has been visited



Example:

- Pick C
- DFS preorder from C is (C,E)
- Add DFS preorder to head [C,E]
- Pick F
- DFS preorder from F is (F)
- Add DFS preorder from F to head [F,C,E]
- Pick B
- DFS preorder from B is (B,D,G)
- Add DFS preorder from B to head [B,D,G,F,C,E]

- Pick A
- DFS preorder from A is (A)
- Add DFS preorder from A to head [A,B,D,G,F,C,E]
- Done, all nodes visited

A DFS order from a node is guaranteed to be a topological order. Since we add everything to the head of the order, a child of a node cannot appear before it.

6.10 Connected Components

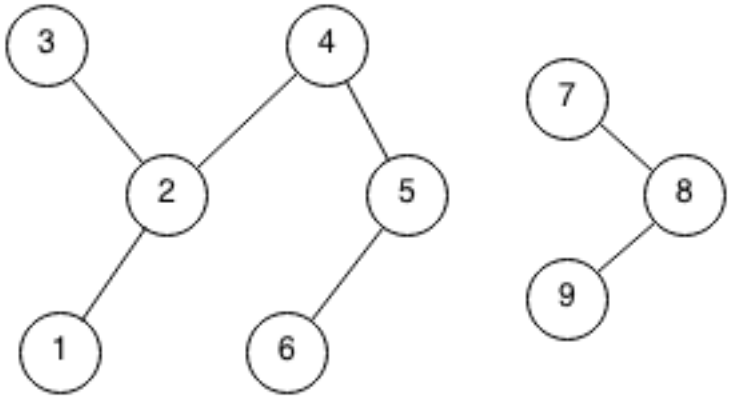
A connected component is a subgraph where all the vertices in the subgraph connect to each other.

Finding the number of distinct connected components can be done using a breadth first search or a depth first search.

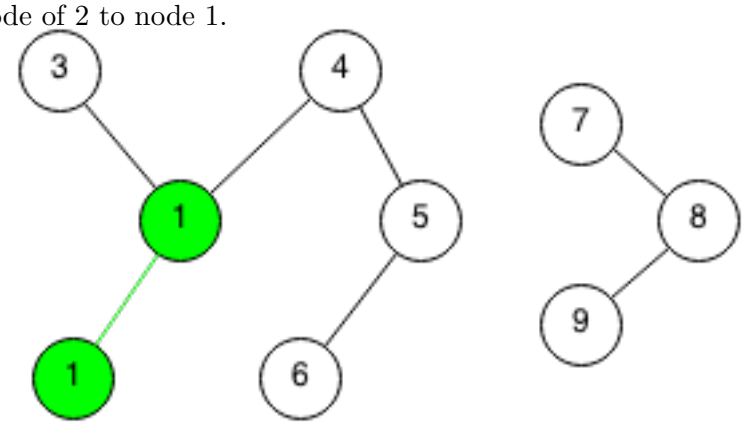
6.10.1 Implementation

1. Set each node's parent to itself
2. Pick an unused edge in the graph that is from node A to node B, if the parent of A is not the same as parent of B then set all nodes whose parent is B to parents of A
3. Repeat for all edges

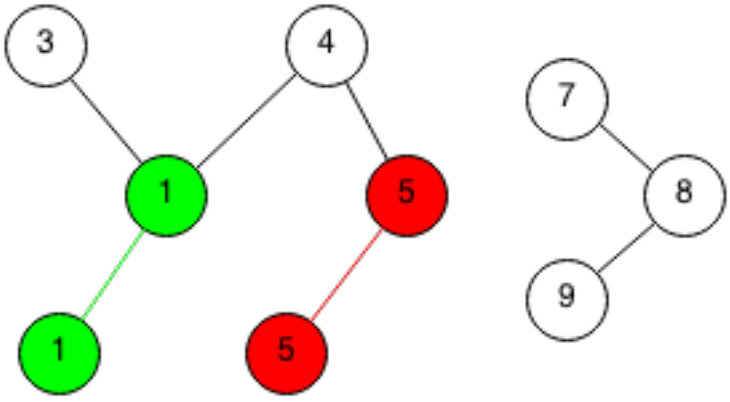
Example



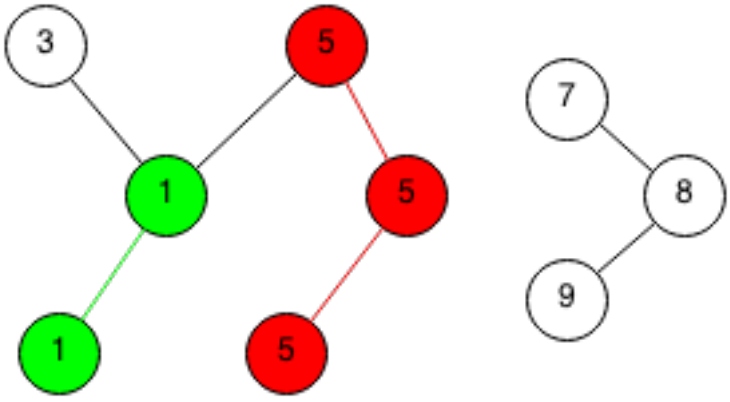
We select a random edge from node 1 and node 2. We set the parent node of 2 to node 1.



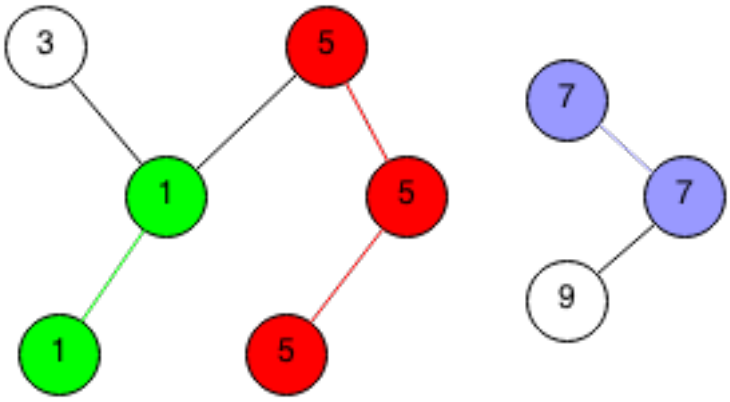
We select another random edge from node 5 to node 6.



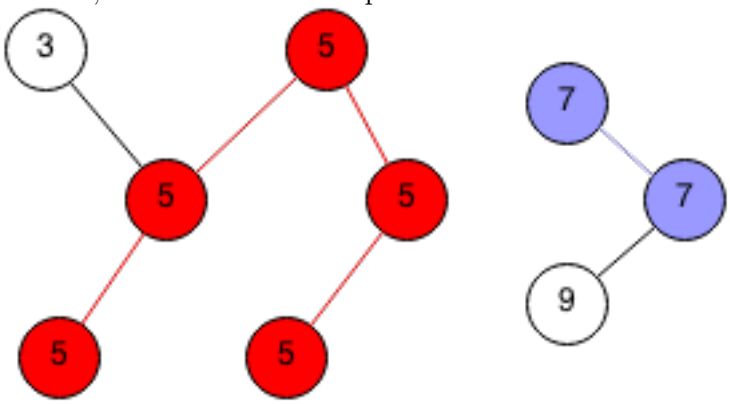
We pick another random edge and we set the parent to 5.



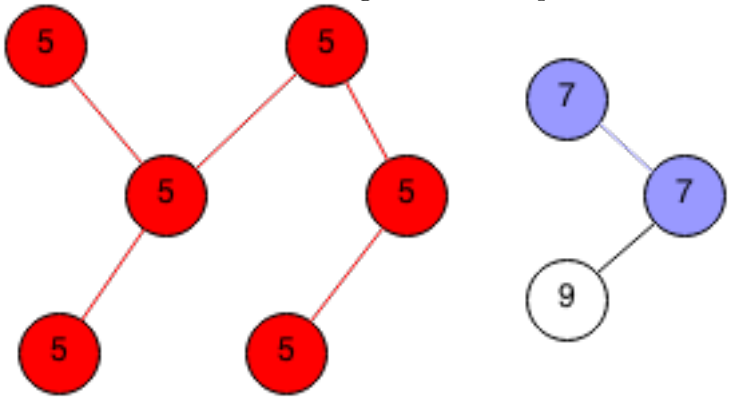
We take another random edge and we set the parents to 7.



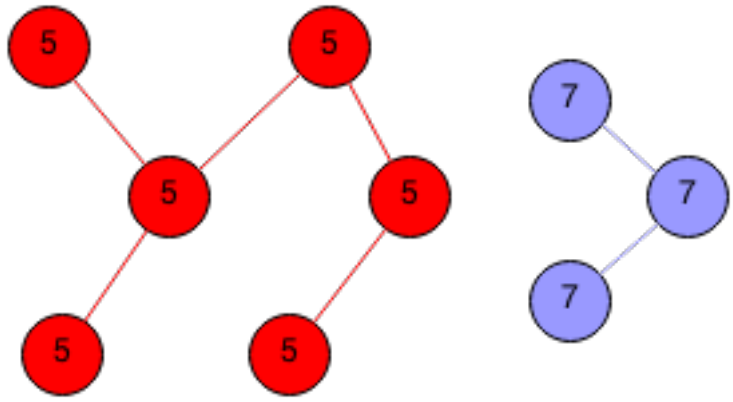
We take another random edge and we have an interesting case, two connected components are to be connected. We take all the nodes whose parent is 1, and we set the new parent to 5.



We take another random edge and set its parent to 1.



We take the last edge and set the last node parent to 7.



Java Code

```
public static int getParent(int x,int[] parent){
    if(parent[x]==x)return x;
    parent[x] = getParent(parent[x],parent);
    return parent[x];
}

public static void connected(int adjMatrix[][]){
    int n = adjMatrix.length;
    int[] parent = new int[n];
    int i,j;
    for(i=0;i<n;i++){
        parent[i] = i;
    }
    for(i=0;i<n;i++){
        for(j=0;j<n;j++){
            if(adjMatrix[i][j]>0){
                int pi = getParent(i,parent);
                int pj = getParent(j,parent);
                if(pi!=pj){
                    parent[pj] = pi;
                }
            }
        }
    }
}
```

Chapter 7

Searches

Introduction

Searches are used to find solutions to problems and there many ways to search for a solution. Here are some generic searches that can be applied to many different problems.

7.1 Binary Search

7.1.1 Binary Search

Binary search is a type of search that is able to find an object in a sorted list in O(log n). In binary search we first start at the middle element and we keep trying to halve the problem until we find the number.

Example

For example if I told you I had a number form 1 to 100 and I told you if your guess was higher or lower than my number you could use binary search to find it.

Eg: my number = 17

- You guess: 50
- I say lower.

So we know that:1number¡50. Since the number is ¡50 then we know we can eliminate all the numbers above 50. We just made the problem half as hard! The reason we picked 50 is important because it is the middle and it tells us the most information. If we picked 80 and the reply was higher it would narrow down the problem a lot, but if the reply was lower it would barely reduce the problem. Picking the middle works best because it tells us the most information if we get a "lower" or "higher" reply. So we should also guess the middle between 1 and 50.

- You guess: 25.
- I say lower.

So we know that 1number¡25. Once again we made the problem half as hard again. Note that at every step we will make the problem half as hard. We need to pick the next middle number which is either 12 or 13, but we are indifferent because it will still tell us the most information (unless you get lucky).

- You guess 13.
- I say higher.

So we know that 13¡number¡25.

- You guess 19.
- I say lower.

So we know that 13¡number¡19.

- You guess 16.
- I say higher.

So we know that 16¡number¡19

- You guess 17.
- I say correct!

7.1.2 Implementation

This is a generic implementation of a binary search.

Generic Binary Search

```
void binarySearch(int ans,int minBound,int maxBound){
    while(maxVal>=minVal){
        int mid = (minVal+maxVal)/2;
        if(mid==ans)return;
        if(mid<ans)minVal = mid;
        else maxVal = mid;
    }
}
```

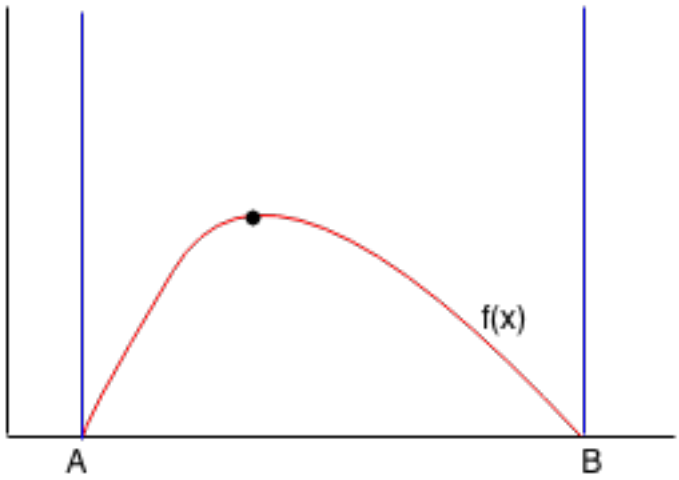
7.1.3 Exercises

1. Given a sorted array, find whether or not the number N exists
2. Given a sorted array, find the the number of elements between the number A and number B inclusive. Example: 1,2 4, 6, 8, 10, 16, 20. Given A=5 and B=15, the number of elements between A and B is 3 (6, 8, 10)
3. Given two sorted arrays, find the number of duplicate elements.
4. Given two decimal numbers A and B, how do you find A/B without using division?

7.2 Ternary Search

Ternary search is a search that find local minimum or maximum values in a function given the interval between A and B.

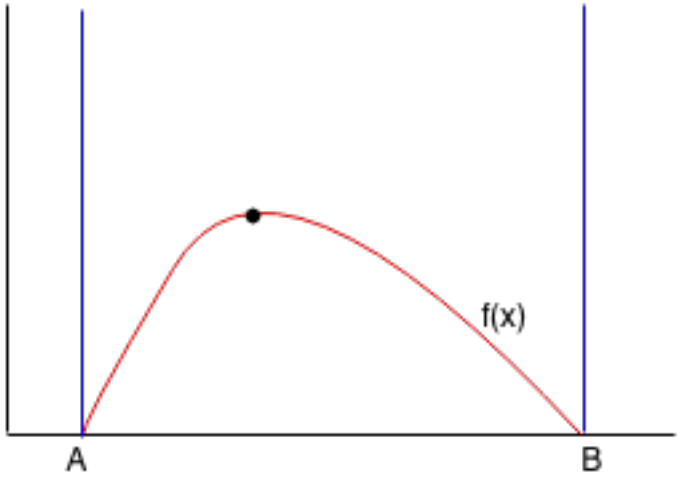
If there are multiple local minimum and maximum values, ternary search will find one of them but not necessarily the maximum value of all points.



7.2.1 Implementation

Let's say we have a function f(x) with only one max point between A and B.

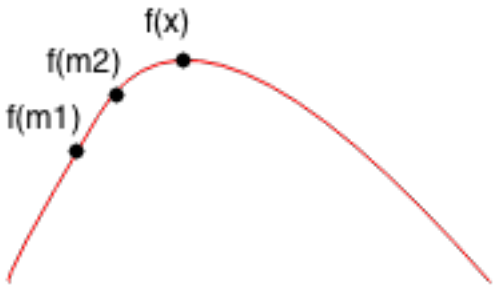
Let m1 by 1/3 of the way from A and B and let m2 be 2/3 of the way from B.



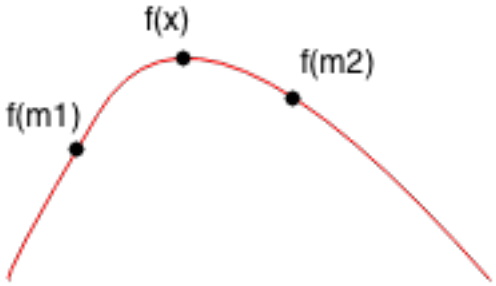
Proof

Case 1 : f(m1) ¡ f(m2)

- Case 1.1: m1 ¡ m2 ¡ x

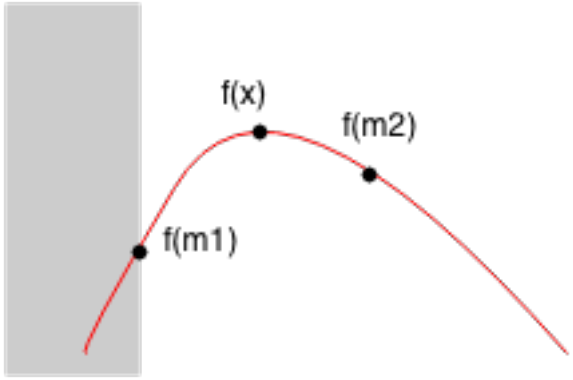


- Case 1.2: m1¡x¡m2



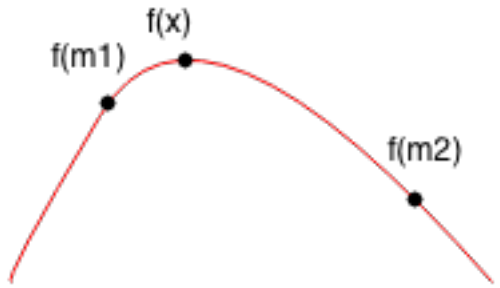
- x ¡ m1 ¡ m2 is not possible.

If f(m1) ¡ f(m2) then the max point must be in the middle third or last third.

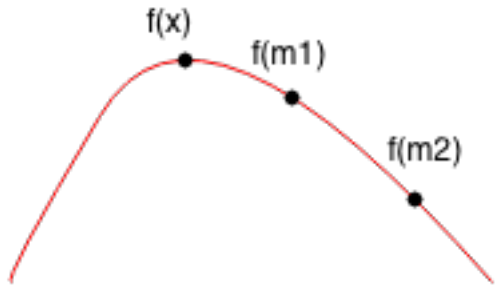


Case 2: f(m1) ¿= f(m2)

- Case 2.1: m1 ¡ x ¡ m2

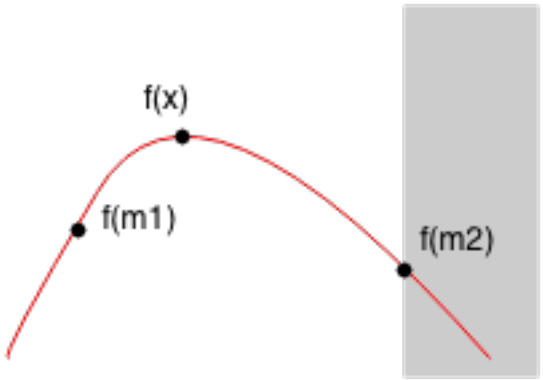


- Case 2.2: x ¡ m1 ¡ m2



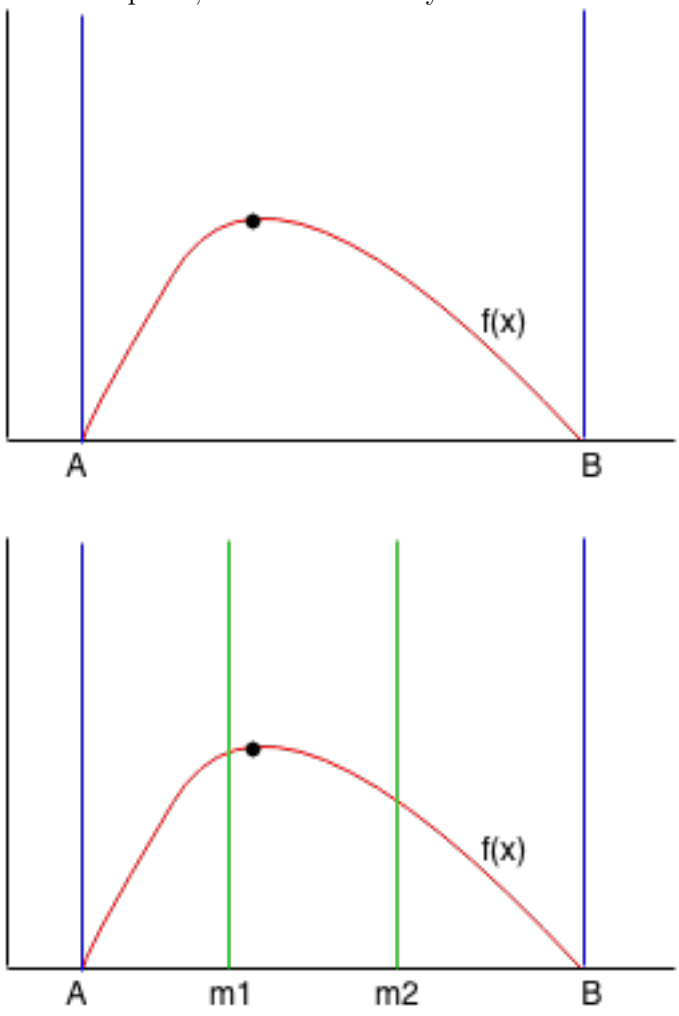
- m1 ¡ m2 ¡ x is not possible

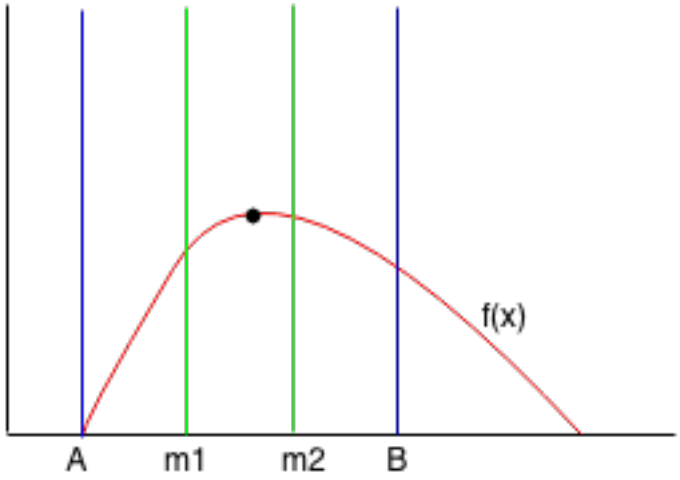
If f(m1) ¿= f(m2) then the max point must be in the first third or middle third.

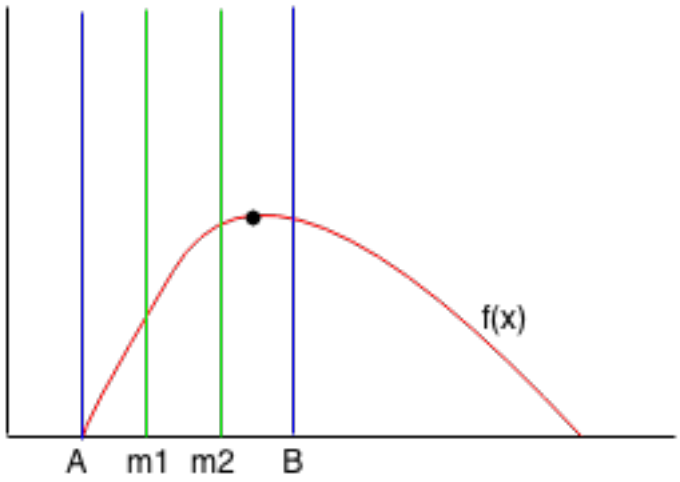


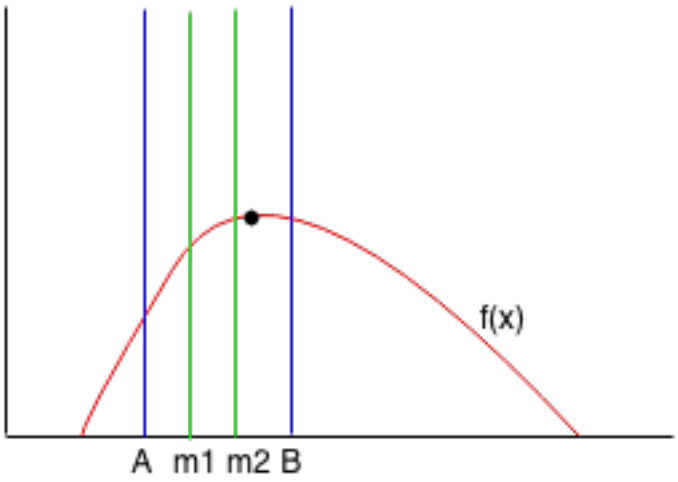
Example

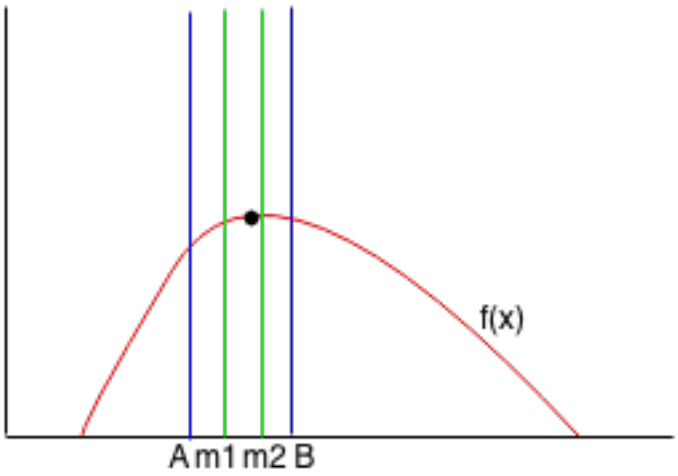
Using the above proof, we can use ternary search to find the maximum point.

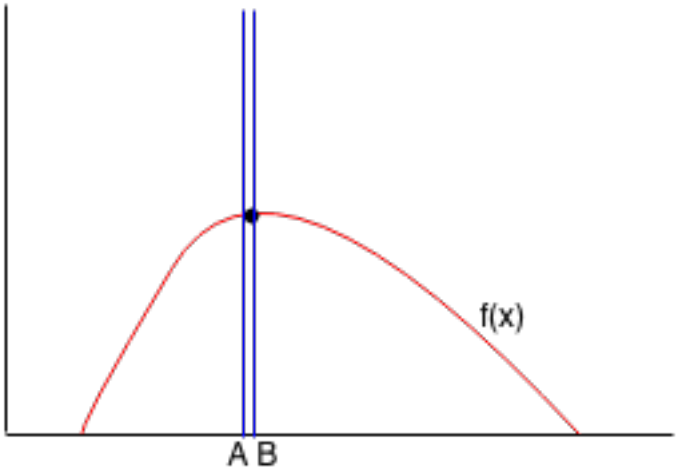



f(x)
A
m1
m2
B


f(x)
A
m1
m2
B


f(x)
A m1 m2 B


f(x)
A m1 m2 B



Formalization

```
Let f(x) be the function
Let A,B be interval

Let tern(a,b) return x where f(x) = maximum

Let m1 = a+(b-a)/3,
m2 = a+(b-a)*2/3

tern(a,b) = { if |f(a)-f(b)|<epsilon      (a+b)/2
            { if f(a) < f(b)               tern(m1,b)
            { else                         tern(a,m2)
```

Code

```
public double tern(double a,double b){
    if(Math.abs(f(a)-f(b))<0.0001){
        return (a+b)/2.0;
    }
    double m1 = a+(b-a)/3.0;
    double m2 = a+(b-a)*2/3;
    if(f(a)<f(b)){
        return tern(m1,b);
    }else {
```

```
            return tern(a,m2);
        }
}
```

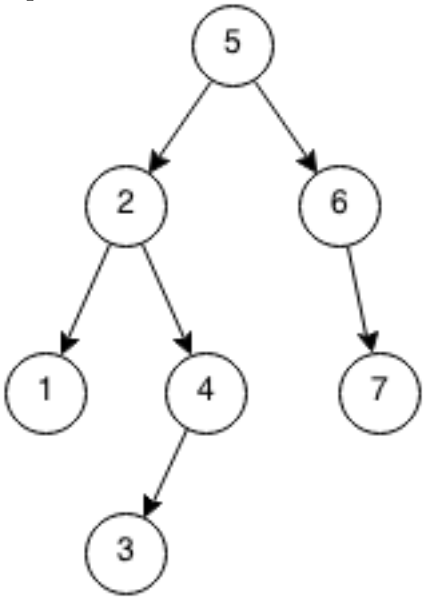
7.3 Depth First Search

Prerequisites: Recursion, Stack

A depth first search on a tree is a traversal of a tree that goes as far down as a branch as possible and then back.

A DFS can also be used on a graph to transverse it by ignoring nodes that have already been visited.

A DFS requires a stack but most the time DFS is implemented with recursion which uses the system stack. So most of the time you do not need an explicit stack.



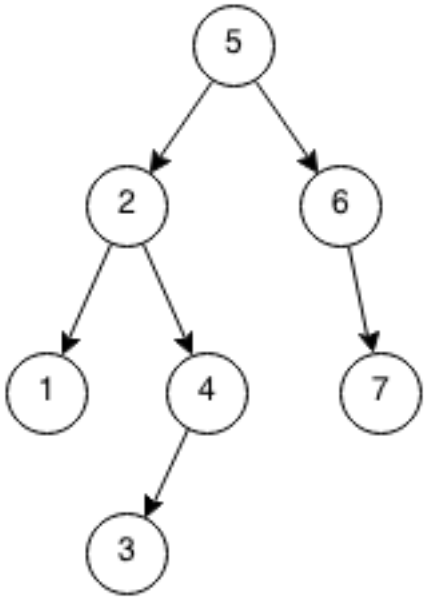
7.3.1 Implementation

Most of the time, DFS is implemented using recursion and it is very short and simple to code.

```
public class Tree {
    int value;
    Tree left;
```

```
    Tree right;
}
```

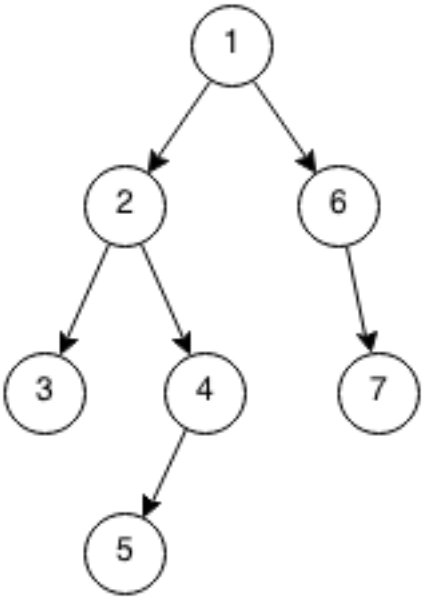
Binary Tree Transversal



Implementation for outputting a binary tree in order from left to right using DFS

```
/**
 * Performs a DFS on a binary tree
 * tree - current tree DFS is at
 */
public static void DFS(Tree cur){
    if(cur==null)return;
    DFS(cur.left);
    System.out.println(cur.value);
    DFS(cur.right);
}
```

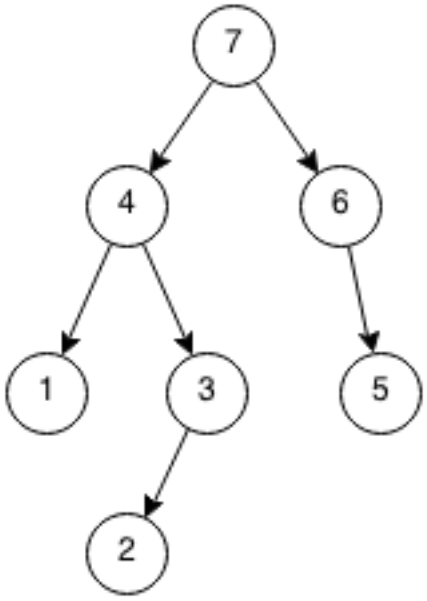
Binary Tree Preorder



Implementation for outputting a binary tree in DFS pre order

```
/**
 * Performs a DFS on a binary tree
 * tree - current tree DFS is at
 */
public static void DFS(Tree cur){
    if(cur==null)return;
    System.out.println(cur.value);
    DFS(cur.left);
    DFS(cur.right);
}
```

Binary Tree Postorder



Implementation for outputting a binary tree in DFS post order

```
/**
 * Performs a DFS on a binary tree
 * tree - current tree DFS is at
 */
public static void DFS(Tree cur){
    if(cur==null)return;
    DFS(cur.left);
    DFS(cur.right);
    System.out.println(cur.value);
}
```

Graph Transversal

This is a DFS implementation for traversing a bidirectional graph with positive weights.

```
/**
 * Performs a DFS on a graph
 * adjMatrix - Adjacency matrix for a graph with positive edges. 0 indicat
 * cur - Current node the DFS is at
```

```
 * visited - Mutable array which keeps track of which nodes have been visit
 */
public static void DFS\_graph(int[][] adjMatrix,int cur,boolean[] visited)
    if(visited[cur])return;
    visited[cur] = true;
    System.out.println(cur);
    for(int i=0;i<adjMatrix.length;i++){
        if(adjMatrix[cur][i]>0)DFS\_graph(adjMatrix,i,visited);
    }
    return;
}
```

7.3.2 Memory

Since DFS goes as deep as possible before going back, the stack we use will need to store at least the depth of the tree.

7.3.3 Exercises

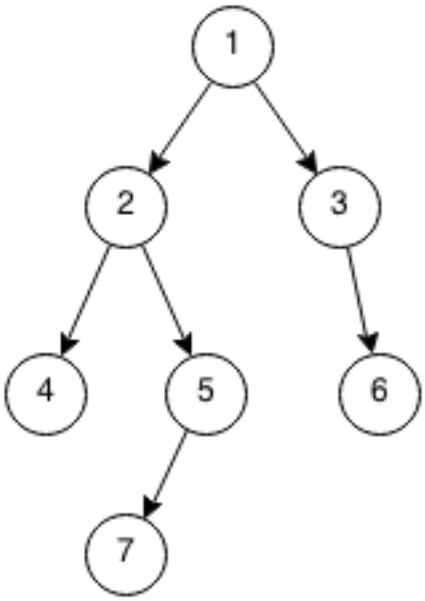
1. Given a binary tree, find the height of it (the longest path from root to leaf)
2. Given a graph, determine if it contains a cycle.
3. Given a node and a binary tree, find the next node for post order, pre order and normal order.

7.4 Breadth First Search

Prerequisites: Recursion, Queue

A breadth first search is a search that transverses level by level. For example in a tree, it will transverse everything from the first layer, to the second layer, the third layer and all the way down to the last layer. BFS is implemented with a queue.

1. Push root into queue
2. Pop element from queue and push all non visited neighbours
3. Repeat 2 until queue is empty



7.4.1 Implementation

Printing a binary tree using BFS:

```
void bfs(Node root){
    Queue<Node> q = new Queue<Node>();
    q.push(root);
    while(q.isEmpty()==false){
        Node cur = q.pop();
        System.out.println(cur.value);
        if(cur.left)q.push(cur.left);
        if(cur.right)q.push(cur.right);
    }
}
```

7.4.2 Exercises

1. Given a grid of squares with walls at certain locations and two locations A and B, find the minimum distance (going up/left/right/down) between the locations or impossible otherwise. For example if A is at (1,1) and B is at (3,1) but there is a wall at (2,1) then the minimum distance would be 4 (down, left, left, up).

2. Given a tree of letters (A is the root), output the tree using BFS with separators between levels

   - Example: A-¿B, B-¿D,B-¿C, C-¿G will output A — B — C D — G

3. Given a tree of letters, and two letters X and Y determine if X is an ancestor of Y or Y is a ancestor of X or neither. An ancestor of a node is another node that is the root of a subtree that contains that node. Or simply parent of the node, grand parent, great grandparents etc.

   - Example: A-¿B, B-¿C, B-¿D, D-¿G, A is a parent of both C and D but G and C are not ancestors of each other

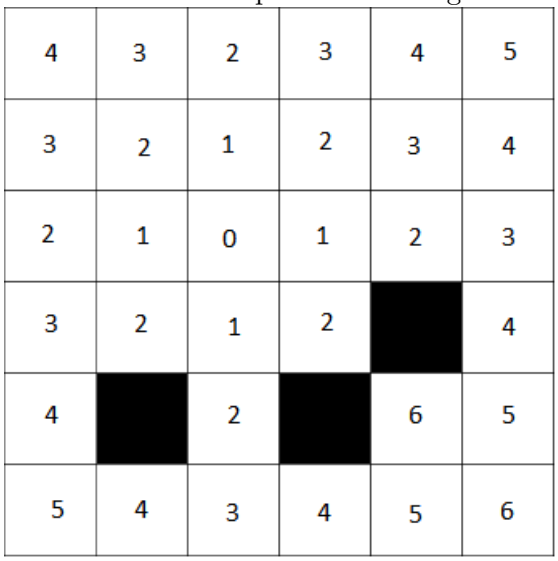
4. Given a binary tree and a node in the binary tree, find the next node in BFS order of the tree.

7.5 Flood Fill

Prerequisites: Depth First Search, Breadth First Search

Flood fill is a search that fills a grid from a start point to find the areas connected to the start point. For example the "bucket fill" in Photoshop or MS Paint uses flood fill to fill in the connecting areas with the same colour.

Flood fill can be implemented using a BFS or DFS.



7.5.1 General solution

Most flood fill solutions follow the same basic layout. There is a general DFS solution and a general BFS solution.

General DFS

```
//marked is initially a n by m boolean array of false
void floodFill(int i,int j){
    if(outOfBounds(i,j))return;
    if(visited(i,j))return;
    markVisited(i,j);
    floodFill(i+1,j);
    floodFill(i,j+1);
    floodFill(i-1,j);
    floodFill(i,j-1);
}
```

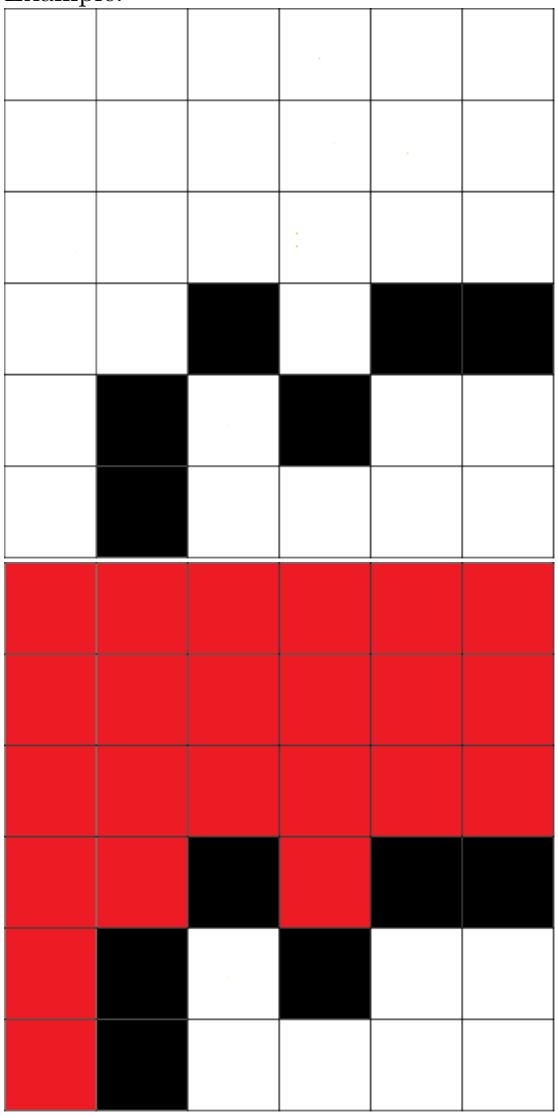
General BFS

```
void floodFill(int i,int j){
    Queue<Point> q;
    q.push(new Point(i,j));
    while(!q.isEmpty()){
        Point cur = q.pop();
        if(outOfBounds(cur))continue;
        if(visited(cur))continue;
        markVisited(cur);
        q.push(new Point(cur.x+1,cur.y));
        q.push(new Point(cur.x-1,cur.y));
        q.push(new Point(cur.x,cur.y+1));
        q.push(new Point(cur.x,cur.y-1));
    }

}
```

7.5.2 Bucket Fill

Given an n x m matrix and a start point and two colors (src and dst), we want to replace all the cells in the matrix that are connected to the start point with the color src and change to color dst as well as output the number of cells changed.

Example:



In a numeric representation of the colors:

1	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1
1	1	0	1	0	0
1	0	1	0	1	1
1	0	1	1	1	1
2	2	2	2	2	2
2	2	2	2	2	2
2	2	2	2	2	2
2	2	0	2	0	0
2	0	1	0	1	1
2	0	1	1	1	1

DFS Solution

This is the DFS approach to the problem.

- Assume that n,m are global integers that are the width and height of the image
- Assume that image is a global integer n by m matrix for the image

- Assume that visited is a global boolean n by m matrix that is initially all false

```
/* Changes all pixels that are equal to src to tar that are connected to s
 * and returns the number of pixels changed
 */
public int FloodFillDFS(int x,int y,int src,int tar){
    if(x<0 || x>=n || y<0 || y>=m)return 0;
    if(visited[x][y])return 0;
    visited[x][y] = true;
    if(image[x][y]!=src)return 0;
    image[x][y] = tar;
    int sum = 0;
    sum+=FloodFillDFS(x+1,y,src,tar);
    sum+=FloodFillDFS(x-1,y,src,tar);
    sum+=FloodFillDFS(x,y+1,src,tar);
    sum+=FloodFillDFS(x,y-1,src,tar);
    return sum;
}

floodFillDFS(1,1,1,2);
```

BFS Solution

This is the BFS approach to the problem.

- Assume that n,m are global integers that are the width and height of the image
- Assume that image is a global integer n by m matrix for the image
- Assume that visited is a global boolean n by m matrix that is initially all false

```
/* Changes all pixels that are equal to src to tar that are connected to s
 * and returns the number of pixels changes
 */
public int FloodFillBFS(int x,int y,int src,int tar){
    LinkedList<Point> q = new LinkedList<Point>();
    q.push(new Point(x,y));
```

```
    int total = 0;
    while(q.isEmpty()==false){
        Point cur = q.pop();
        if(cur.x<0||cur.x>=n||cur.y<0||cur.y>=m)continue;
        if(visited[cur.x][cur.y])continue;
                visited[cur.x][cur.y] = true;
        if(image[cur.x][cur.y]!=src)continue;
        image[cur.x][cur.y] = tar;
        total++;
        q.push(new Point(cur.x+1,cur.y));
        q.push(new Point(cur.x-1,cur.y));
        q.push(new Point(cur.x,cur.y+1));
        q.push(new Point(cur.x,cur.y-1));
    }
    return total;
}
FloodFillBFS(1,1,1,2);
```

7.5.3 Exercises

1. Given a grid and list of start points and walls, find the distance to the closest start point for every non-wall grid space without passing through a wall.

2. Given a grid and list of cell coordinates, find the perimeter around the cells

    - Example: a single cell has a perimeter of 4, two cells joined side by side have a perimeter of 6

7.6 Backtracking

Backtracking is a search that find all possible solutions by enumerating on a partial solution. Backtracking can be done using DFS or BFS. Generally, DFS will be better than BFS because backtracking is used to enumerate a large amount of solution. Since BFS requires storing each "level" of solutions and DFS requires storing each "height" of the solution.

Backtracking is similar to recursion, but instead of generating all the solutions, we will generate each solution one by one. In this way, we only

need to store the current solution in memory whereas in normal recursion we need to store all the solutions into memory.

Since backtracking requires enumerating through all solutions, it is usually slow with runtimes usually O(n!) or $O(2^n)$.

7.6.1 General Solution

```
Base case:
When a solution has been generated

Reject:
Check if partial solution needs to be rejected

Recurrence:
Generate next partial solution thats growing to full solution
```

```
backtrack(solution):
    if reject(solution)
        stop

    if base case
       stop

    backtrack( next\_solution ) for next\_solution from solution
```

```
Initial:
backtrack( empty\_solution)
```

7.6.2 List all sets

Given a set of numbers S of length N, output all subsets of S.

For example S=[1,2,3,4]. The subsets encodings of [1,2,3,4]:

-
- 1, 2, 3, 4

- 1,2, 1,3, 1,4, 2,3, 2,4, 3,4
- 1,2,3, 1,2,4,1,3,4,2,3,4
- 1,2,3,4

We want to be able to enumerate all the subsets of S so we need to find a way to encode a subset of an array. We can use a binary number of length N to encode a subset of an array of length N. For example a 1 represents we use a number in the set and a 0 means we don't use a number in the set.

So above:

- = 0000
- 1 = 1000
- 2 = 0100
- 3 = 0010
- 4 = 0001
- 1,2 = 1100
- 1,3 = 1010
- 1,4 = 1001
- 2,3 = 0110
- 2,4 = 0101
- 3,4 = 0011
- 1,2,3 = 1110
- 1,2,4 = 1101
- 1,3,4 = 1011
- 2,3,4 = 0111
- 1,2,3,4 = 1111

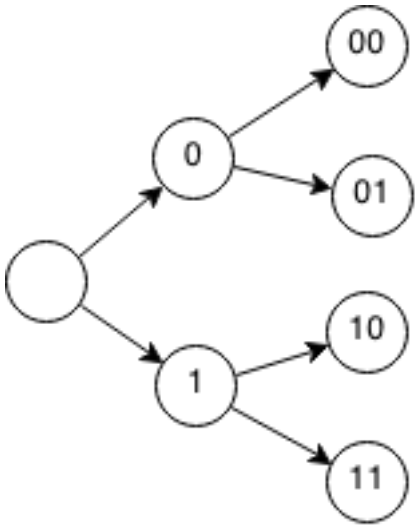
At each position we either take or don't take the number in the set and we can do this for each number. We can enumerate through all these encoding by first starting with 0 or 1 and appending more 0's or 1's.

Recursive Method

Here is the way we would do this problem with recursion:

```
Start with []
Add 1 and 0 to the right of each binary number in the array
Repeat until N
[0,1]
[00,01,10,11]
[000,001,010,011,100,101,110,111]
[0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,111
Let S be an array of N integers
Let subsets(arr,n) be subsets of S from 1 to n

Base case
S(set,0) = set

Recurrence
S(set,n) = subsets([sub+0 for sub in set] + [sub+1 for sub in set],n)

Example

subsets([],4)
subsets([0,1],3)
subsets([00,01,10,11],2)
subsets([000,001,010,011,100,101,110,111],1)
subsets([0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,
=
[0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,111
```

Here is the way we would do this problem with backtracking, however it takes a lot of memory to store ALL the solutions. Instead of building all the solutions, we can build each solution one by one.



Formalization

```
Base case
subset(0, binary): print binary

Recurrence
subset(n-1,binary+'0')
subset(n-1,binary+'1')

Example
subset(4,'')

subset(3,'0')
subset(3,'1')

subset(2,'00')
subset(2,'01')
subset(2,'10')
subset(2,'11')

subset(1,'000')
subset(1,'001')
subset(1,'010')
subset(1,'011')
```

```
subset(1,'100')
subset(1,'101')
subset(1,'110')
subset(1,'111')

subset(0,'0000')
subset(0,'0001')
subset(0,'0010')
subset(0,'0011')
subset(0,'0100')
subset(0,'0101')
subset(0,'0110')
subset(0,'0111')
subset(0,'1000')
subset(0,'1001')
subset(0,'1010')
subset(0,'1011')
subset(0,'1100')
subset(0,'1101')
subset(0,'1110')
subset(0,'1111')
```

Implementation

```
void subsets(int arr[],bool use[],int i){
    if(i>=arr.length){
        for(int j=0;j<n;j++){
            System.out.print("\%d ",arr[i]);
        }
        System.out.println();
        return;
     }
     use[i] = false;
     subsets(arr,use,i+1);
     use[i] = true;
     subsets(arr,use,i+1);
}
subsets([1,2,3,4],[0,0,0],0);

subsets([1,2,3,4],[0,0,0],0);
```

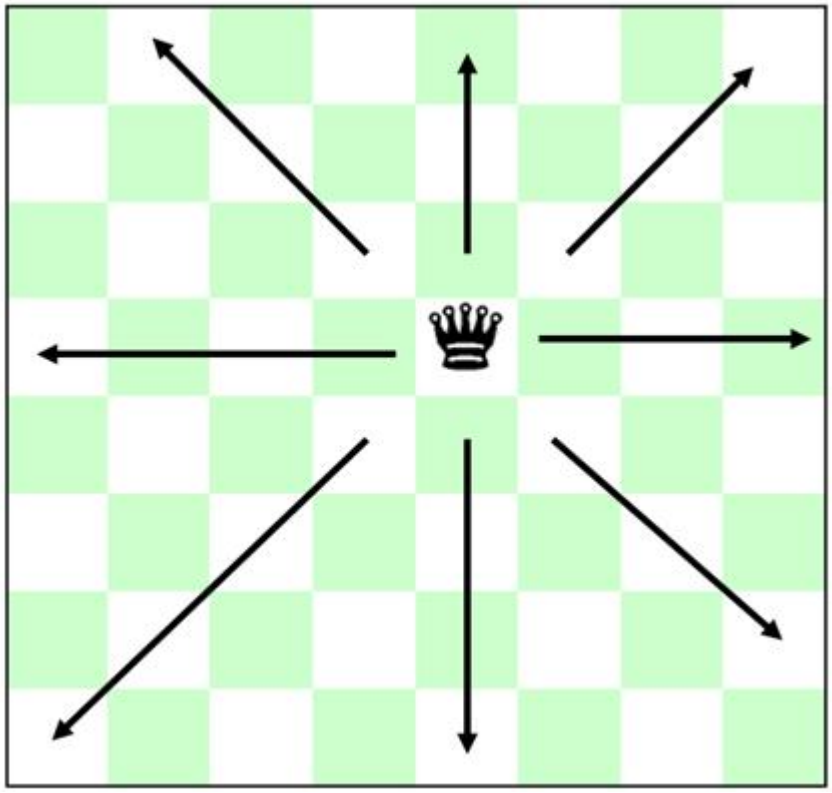
```
subsets([1,2,3,4],[1,0,0],0);
```

7.6.3 Permutation

Permutation can be

7.6.4 N Queen Problem

Find the number of ways to place N queens on a NxN board without any of them attacking each other. Queens attack each other by being along the same row, column or diagonal.



First we need to be able to encode a solution. There must be only one queen in each row, each column, each positive diagonal and each negative diagonal. We need a way to encode each row, column and diagonal to make it easy to check that they are all unique.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	1	1	1	1	1	1
2	2	2	2	2	2	2
3	3	3	3	3	3	3
4	4	4	4	4	4	4
5	5	5	5	5	5	5

If we guarantee that all rows are unique and we have all columns filled, then we can guarantee that all the columns are unique as well.

To encode a diagonal we can use a clever equation to represent the diagonal. We can notice that every positive diagonal has the same value with row+col.

d1 = row+column

N=6:

	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	6
2	2	3	4	5	6	7
3	3	4	5	6	7	8
4	4	5	6	7	8	9
5	5	6	7	8	9	10

Every square with the same d1 is in the same diagonal (/). For example on a 6x6 board: (0,2), (1,1), (2,0) all have d1 = 2 and are all along the same diagonal.

We can also notice that every negative diagonal has the same value of (N-row)+column. The second diagonal can also be found using another equation:

d2 = (N-row-1)+column

N=6:

	0	1	2	3	4	5
0	5	4	3	2	1	0
1	6	5	4	3	2	1
2	7	6	5	4	3	2
3	8	7	6	5	4	3
4	9	8	7	6	5	4
5	10	9	8	7	6	5

Everything with the same d2 will be on the same diagonal. For example on a 6x6 board (0,0),(1,1),(2,2) have d2 = 5 and are all along the same diagonal.

Now that we get can encode rows, columns, and diagonals, we can use it to make checking solutions more easy. We can keep sets that track with rows/columns/diagonals are currently filled and check our solution to make sure we do not have anything in the same row/column/diagonal.

We can place a queen in each row and make sure that each column / diagonals is unfilled.

Formalization

```
Let N be a NxN board where we want to place N queens
Let queen(n,columns,d1,d2) be a placing of N queens across a board

Base case
queen(0,cols,d1,d2): print solution

Reject solution
reject(cols,d1,d2):    { false if duplicates in cols or d1 or d2
                       { true otherwise
Recurrence
queen(row,cols,d1,d2) = queen(row-1,cols with col,d1 with row+col,d2 with
```

Examples
N=4

```
queen(4,[],[],[])

queen(3,[0],[3],[6])
queen(3,[1],[4],[5])
queen(3,[2],[5],[4])
queen(3,[3],[6],[3])

queen(2,[0,0],[3,2],[6,5]) x— reject
queen(2,[0,1],[3,3],[6,4]) x— reject
queen(2,[0,2],[3,4],[6,3])
queen(2,[0,3],[3,5],[6,2])
queen(2,[1,0],[4,2],[5,5]) x— reject
queen(2,[1,1],[4,3],[5,4]) x— reject
queen(2,[1,2],[4,4],[5,3]) x— reject
queen(2,[1,3],[4,5],[5,2])
queen(2,[2,0],[5,2],[4,5])
queen(2,[2,1],[5,3],[4,4]) x— reject
queen(2,[2,2],[5,4],[4,3]) x— reject
queen(2,[2,3],[5,5],[4,2]) x— reject
queen(2,[3,0],[6,2],[3,5])
queen(2,[3,1],[6,3],[3,4])
queen(2,[3,2],[6,4],[3,3]) x— reject
queen(2,[3,3],[6,5],[3,2]) x— reject

queen(1,[0,2,0],[3,4,1],[6,3,4]) x— reject
queen(1,[0,2,1],[3,4,2],[6,3,3]) x— reject
queen(1,[0,2,2],[3,4,3],[6,3,2]) x— reject
queen(1,[0,2,3],[3,4,4],[6,3,1]) x— reject
queen(1,[0,3,0],[3,5,1],[6,2,4]) x— reject
queen(1,[0,3,1],[3,5,2],[6,2,3])
queen(1,[0,3,2],[3,5,3],[6,2,2]) x— reject
queen(1,[0,3,3],[3,5,4],[6,2,1]) x— reject
queen(1,[1,3,0],[4,5,1],[5,2,4])
queen(1,[1,3,1],[4,5,2],[5,2,3]) x— reject
queen(1,[1,3,2],[4,5,3],[5,2,2]) x— reject
queen(1,[1,3,3],[4,5,4],[5,2,1]) x— reject
queen(1,[2,0,0],[5,2,1],[4,5,4]) x— reject
```

```
queen(1,[2,0,1],[5,2,2],[4,5,3]) x— reject
queen(1,[2,0,2],[5,2,3],[4,5,2]) x— reject
queen(1,[2,0,3],[5,2,4],[4,5,1])
queen(1,[3,0,0],[6,2,1],[3,5,4]) x— reject
queen(1,[3,0,1],[6,2,2],[3,5,3]) x— reject
queen(1,[3,0,2],[6,2,3],[3,5,2])
queen(1,[3,0,3],[6,2,4],[3,5,1]) x— reject
queen(1,[3,1,0],[6,3,1],[3,4,4]) x— reject
queen(1,[3,1,1],[6,3,2],[3,4,3]) x— reject
queen(1,[3,1,2],[6,3,3],[3,4,2]) x— reject
queen(1,[3,1,3],[6,3,4],[3,4,1]) x— reject

queen(0,[0,3,1,0],[3,5,2,0],[6,2,3,3]) x— reject
queen(0,[0,3,1,1],[3,5,2,1],[6,2,3,2]) x— reject
queen(0,[0,3,1,2],[3,5,2,2],[6,2,3,1]) x— reject
queen(0,[0,3,1,3],[3,5,2,3],[6,2,3,0]) x— reject
queen(0,[1,3,0,0],[4,5,1,0],[5,2,4,3]) x— reject
queen(0,[1,3,0,1],[4,5,1,1],[5,2,4,2]) x— reject
queen(0,[1,3,0,2],[4,5,1,2],[5,2,4,1]) SOLUTION
queen(0,[1,3,0,3],[4,5,1,3],[5,2,4,0]) x— reject
queen(0,[2,0,3,0],[5,2,4,0],[4,5,1,3]) x— reject
queen(0,[2,0,3,1],[5,2,4,1],[4,5,1,2]) SOLUTION
queen(0,[2,0,3,2],[5,2,4,2],[4,5,1,1]) x— reject
queen(0,[2,0,3,3],[5,2,4,3],[4,5,1,0]) x— reject
queen(0,[3,0,2,0],[6,2,3,0],[3,5,2,3]) x— reject
queen(0,[3,0,2,1],[6,2,3,1],[3,5,2,2]) x— reject
queen(0,[3,0,2,2],[6,2,3,2],[3,5,2,1]) x— reject
queen(0,[3,0,2,3],[6,2,3,3],[3,5,2,0]) x— reject
```

7.6.5 Exercises

1. Given a sequence of numbers, output all increasing subsequences

2. Given a NxN chessboard with certain squares that can have no pieces placed, output the number of configurations that can be made from rooks without attacking each other.

3. Given a sudoku grid, output a solution for the grid if it exists. Note: this question is popular for technical interviews.

Chapter 8

Dynamic Programming

Introduction

Prerequisites: Advanced Recursion

Next: Advanced Dynamic Programming

Dynamic programming uses memoization by solving subproblems to solve the more complex problem. Dynamic programming uses recursion but instead of working backwards, it builds up the answer and reduces the number of duplicate computations.

Like recursion, dynamic programming requires two things:

- A base case and
- A subproblem that can be reduced into smaller subproblems

8.1 Advanced Dynamic Programming

Dynamic programming is very powerful and more efficient than recursion for problems that recompute multiple values. However sometimes it is difficult to find an efficient dynamic programming solution and we will examine problems where we will need higher dimensions.

Prerequistes

- Dynamic Programming

8.1.1 Longest Common Subsequence

A subsequence is a subset of the original sequence that is in the same order. For example in the string "abcdefghi", "aeg" is a subsequence but "eaq" is not because it is not in order.

The longest common subsequence between two strings A and B is the longest subsequence in A that is also in B.

For example given A="xyaaaabcdeg", B="bcaaaaefgxy" the longest common subsequence is "aaaaeg"

```
xyaaaabcde\_g
bcaaaa\_\_\_efgxy
```

If we try to use greedy we will see it doesn't work. For example if we use try to take as much as B as we can, we see that we will get BCEG or if we try to take as much as A we get XY.

Let first write a formal definition of the problem, given two strings A and B each with lengths N and M respectively we want to find the longest common subsequence between them.

[Note to make reading easier I have used short forms for substring and index. For example A[3] means the 3rd character of A and A[1..4] means the substring of A from the first character to the fourth character inclusive. A[0] represents the null substring of A]

The base is simple. The LCS of A[1..x] (where x is from 1 to N) and B[0] = 0. The LCS of B[1..x] and A[0] = 0.

We need to break this problem into subproblems.

If A[N] = B[M] and then the new LCS is the LCS of A[1..N-1] and B[1..M-1]. Note that if two strings have the same character at the end of their string it has to be part of the LCS. Try to prove this to yourself.

```
LCS of matched = (LCS of A from 1 to N-1, B from 1 to M-1) + A[N]
```

If A[N] != B[M] then we try to match A[N] with B[M-1] or A[N-1] with B[M]. Thus we take the LCS of A[1..N-1] and B[1..M] and the LCS of A[1..N] and B[1..M-1].

```
LCS of not matched = max ( (LCS of A[1..N], B[1..M-1]) , (LCS of A[1..N-1]
```

Putting it all together we have:

```
Let LCS[i][j] be the length of longest common subsequence of A[1..i] and B

Base case:
LCS[i][0] = 0 where 0< i       n
```

```
LCS[0][j] = 0 where 0< j       m

Subproblem:
if A[i] == B[j], LCS[i][j] = LCS[i-1][j-1]+1 where 0< i       n and 0 < j
if A[i] != B[j], LCS[i][j] = max( LCS[i-1][j], LCS[i][j-1] ) where 0< i
```

8.1.2 Zero-one Knapsack Problem

In the Dynamic Programming section, we examined the knapsack problem:

Given an unlimited amount of N items with positive weights and values, we want to find the maximum value we can hold with a capacity.

Let's change the problem slightly such that there is only one of each object. The problem becomes slightly more difficult because we to take into account whether or not we have used an object before.

Given one of each N items with positive weights and values, we want to find the maximum value we can hold with a capacity W.

```
Let knapsack(N,W) be the maximum value of iterating through N items with a
Let weights be an array of weights
Let values be an array of values

knapsack(i,0) = 0
knapsack(i,w) = 0 if no items can fit
knapsack(i,w) = 0 where i <= 0

knapsack(n,w) = max(knapsack(n-1,w-weight[n])+value[n], knapsack(n-1,w))

Let knapsack[N][W] be the maximum value of iterating through N items with n
Let weights be an array of weights where weights[i] is the weight of the it
Let values be an array of values where values[i] is the value of the ith it

for i from 0 to W
    knapsack[0][i] = 0

for n from 1 to N
    knapsack[n][0] = 0
    for w from 1 to W
        knapsack[n][w] = max(knapsack[n-1][w-weight[n]]+value[n], knapsack
```

8.2 Coin Problem

Let's say that you wanted to make change for $51 using the smallest amount of bills ($1, $2, $5, $10, $20). We can use a greedy approach by always taking the highest bill that can be subtracted to find the smallest amount of change. 51 - 20 = 31 - 20 = 11 - 10 = 1. So the smallest amount of change would be comprised of 2 x $20 + 1 x $10 + 1 for a total of 5 bills. This solution seems very easy to implement, but what if the bills were not so nice?

Imagine that an alien currency was in denominations of $3, $5, $7 and $11. What would be the smallest amount of bills to make change for $13? Note that a greedy approach does not work for this alien currency. For example: 13 - 11 = 2. It is impossible to make change using the greedy approach.

8.2.1 Solution

Let's define the problem more formally: Given a list of bills each with a positive denomination d, find the lowest amount of bills required to make C dollars or return impossible if it cannot be done.

The base case 0 for this is very simple. There are 0 bills to make 0 dollars.

We can reduce this problem into subproblems. Let's assume that we have found out the lowest amount of bills required to make all the dollar amounts from 0 to C-1 or determined if it is impossible to do so. Lets take a look at an arbitrary bill b with denomination d. We know the minimum number of bills to make C-d (or if its impossible) based on our assumption that we have solved from 0 to C-1. Thus if use the bill b to make C then it is just the minimum number of bills to make C-d with 1 more bill so we add 1 more to that value. If we take the minimum value for all bills (if its possible to make C-d), we will get the lowest amount of bills required to make C.

Putting it all together:

```
Let bills[C] be the smallest amount of bills to make the amount C, or impo

Base case:
bills[0] = 0

Subproblem:
bills[C] = min(bills[C–d]+1) for all bills where d is the denominator of t
if bill[C–d] = impossible for all bills, then bills[C] is impossible
```

Example of previous problem where the bills are ($3, $5, $7, $11) and we want to find the minimum number of bills to make 13.

Let -1 be "impossible".

C	0	1	2	3	4	5	6	7	8	9	10	11	12	13
bills[C]	0	-1	-1	1	0	1	2	-1	2	3	2	1	4	3

8.2.2 Implementation

8.2.3 Exercises

1. Given a list of bills each with unique denomination d, find the number of ways to make C dollars.

    - For example given bills: $2, $3, $5, there are 2 ways to make $7 (2+5, 2+2+3)

2. Given a list of N integers, separate the list into two sets such that the difference is minimized and output the difference.

    - For example given integers: 1, 4, 10, 12, we can separate them into (4+10=14) and (1+12=13) so the minimum difference is 1.

3. Given a list of lengths, find the smallest area that can be created if the lengths are used to make a triangle.

    - For example, given lengths: 2,4,6,8,10 we can make a triangle with minimum area 43.3 if we use the sides (2+8=10, 4+6 = 10, 10).

8.3 Knapsack Problem

Imagine you are a robber and you have found a large stash of valuables. Each valuable has a value and a weight. You can only hold 10kg in your bag and you want to find the highest valued haul you can get away with.

- Necklace: $10, 1kg
- Stack of cash: $270, 3kg
- Jewelry: $665, 7kg
- Rare painting: $900, 9kg

Let's try a greedy approach: we will take the items with the highest value to weight ratio.

- Necklace: $10/kg
- Stack of cash: $90/kg
- Jewelry: $95/kg
- Rare painting: $100/kg

The greedy approach will choose the rare painting and the necklace for a total of $910. However if we take the jewelry and the stack of cash we will get $935 and still fit it into the bag. How can we solve this problem? The answer is dynamic programming.

8.3.1 Solution

Let's first write a more formal definition of the problem:

Given n objects, each associated with a positive weight and value, and a maximum total weight W that we can hold, what is the maximum value we can hold. In the zero/one knapsack problem, there is only one of each object so we either take it or leave it.

Let's write a more specific version of the problem: we want to find the maximum value that a bag with maximum weight W can hold of N objects with positive weight and value which we can either take or not take.

The base case for this is trivial. With zero weight, the maximum value you can have is 0.

We now have to break this problem down into subproblems.

Now we want to find the maximum value for a bag of maximum weight W and assessing all N objects. Since we have already assessed up to N-1 objects we only need to assess the Nth object. For the Nth object we can either take it or leave it.

If we leave it, then it is the same as a bag of maximum weight W with N-1 objects because we are just ignoring the Nth object.

If we take it, then we need to find the maximum value thats possible while making room for that object and add that to the value of the object.

If we want the maximum value of assessing N objects and maximum weight W then we want the max of leaving the Nth object and taking the Nth object so:

```
max value = max( maximum value taking , maximum value leaving )
max value = max( maximum value of N–1 objects with weight W, (maximum value
```

Putting it all together we have:

```
Let weight[i] be the weight of object i
Let value[i] be the value of object i

Let knapsack[i][j] be the maximum value that a knapsack of maximum weight

Base case:
knapsack[0][0] = 0

Subproblem:
knapsack[i][j] = max(knapsack[i-1][j-weight[i]], knapsack[i-1][j-1])
```

8.3.2 Implementation

8.3.3 Exercises

1. Given a list of n objects with a positive weight and value, find the maximum value that can be obtained with maximum weight W. However, each object can be used more than once.

Chapter 9

Strings

Introduction

String problems are more important than ever before with the enormous amount of text and information that is now available. For example, if we search for keywords on Google out of the millions of articles, how can we do it in such a way that the retrieval is relevant, accurate and efficient? If we misspell the word "shooting" as "sohoting" how can we come up with a list of autocorrected words?

9.1 Pattern Matching

Pattern matching is finding if a certain sequence of elements exists in another larger sequence of elements. For example we want to find if the string "abc" is in the string "abdabdbdacbaabcasd" (which the answer is yes).

9.1.1 Knuth Morris Pratt

If we have the needle string "abcxabcy" and haystack string "abcyabcxabcy" then our first search will be putting the needle at position 0 as we see the search fails at 'x'. However we note that we do not need to set the needle at position 1 because we have already done the search for the prefix "abc". Thus we can search starting by setting the needle at the next 'a'.

KMP uses this type of optimization for pattern matching by precomputing a table for the needle string.

9.1.2 Rabin Karp

9.1.3 Finite State Automata

9.1.4 Boyer Moore