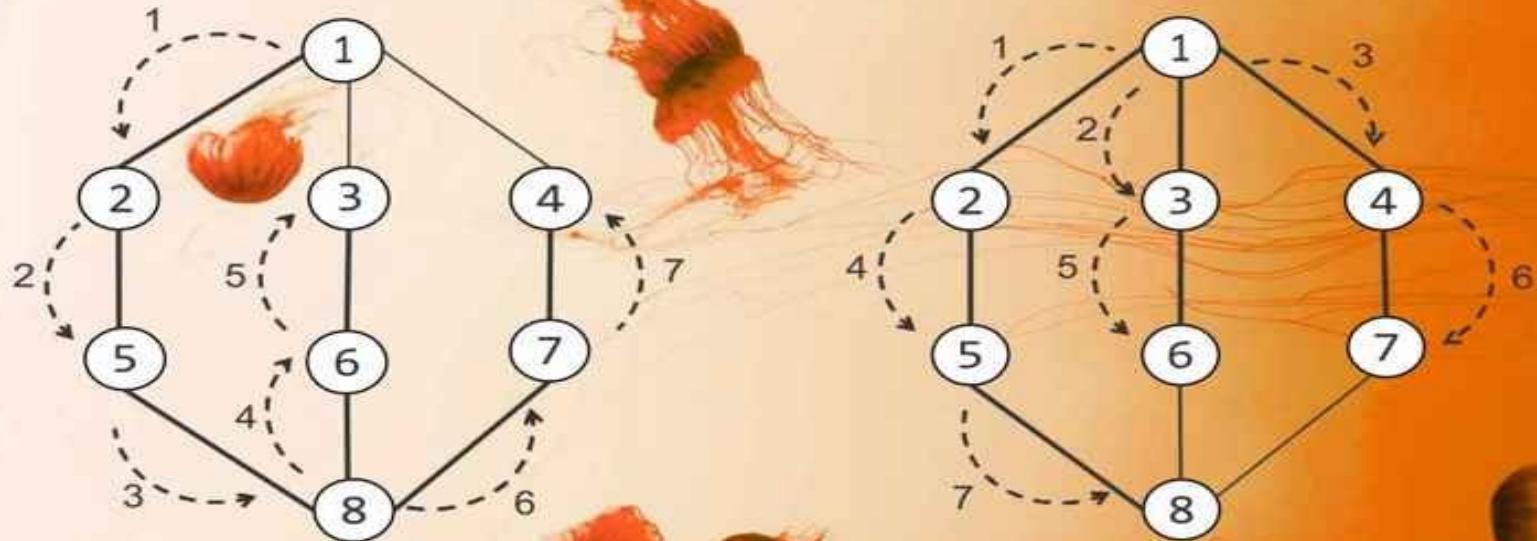


Problem Solving in Data Structures & Algorithms Using Java

Second Edition



Hemant Jain

Problem Solving in
Data Structures &
Algorithms Using
Java
Second Edition

By Hemant Jain

Problems Solving in Data Structures & Algorithms Using Java

Hemant Jain

Copyright © Hemant Jain 2018. All Right Reserved.

Hemant Jain asserts the moral right to be identified as the author of this work.

All rights reserved. No part of this publication may be reproduced, stored in or introduced into a retrieval system, or transmitted, in any form, or by any means (electrical, mechanical, photocopying, recording or otherwise) without the prior written permission of the author, except in the case of very brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law. Any person who does any unauthorized act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

ACKNOWLEDGEMENT

The author is very grateful to GOD ALMIGHTY for his grace and blessing. My deepest gratitude to my elder brother Dr. Sumant Jain for his help and support. This book would not have been possible without the support and encouragement he provided.

I would like to express profound gratitude to my friends Naveen Kaushik, Love Singhal, Anand Rajashekaran and Harvinder Bholowasia for their invaluable encouragement, supervision and useful suggestion throughout this book writing work. Their support and continuous guidance enable me to complete my work successfully.

Hemant Jain

TABLE OF CONTENTS

[TABLE OF CONTENTS](#)

[TABLE OF CONTENTS](#)

[CHAPTER 0: HOW TO USE THIS BOOK](#)

[WHAT THIS BOOK IS ABOUT](#)

[PREPARATION PLANS](#)

[CODE DOWNLOADS](#)

[SUMMARY](#)

[CHAPTER 1: ALGORITHMS ANALYSIS](#)

[INTRODUCTION](#)

[ASYMPTOTIC ANALYSIS](#)

[BIG-O NOTATION](#)

[OMEGA-Ω NOTATION](#)

[THETA-Θ NOTATION](#)

[COMPLEXITY ANALYSIS OF ALGORITHMS](#)

[TIME COMPLEXITY ORDER](#)

[DERIVING THE RUNTIME FUNCTION OF AN ALGORITHM](#)

[TIME COMPLEXITY EXAMPLES](#)

[MASTER THEOREM](#)

[ARRAY BASED QUESTIONS](#)

[RECURSIVE FUNCTION](#)

[EXERCISE](#)

[CHAPTER 2: APPROACH TO SOLVE ALGORITHM DESIGN PROBLEMS](#)

[INTRODUCTION](#)

[CONSTRAINTS](#)

[IDEA GENERATION](#)

[COMPLEXITIES](#)

[CODING](#)

[TESTING](#)

[EXAMPLE](#)

[SUMMARY](#)

CHAPTER 3: ABSTRACT DATA TYPE & JAVA COLLECTIONS

ABSTRACT DATA TYPE (ADT)

DATA-STRUCTURE

JAVA COLLECTION FRAMEWORK

ARRAY

LINKED LIST

STACK

QUEUE

TREE

BINARY TREE

BINARY SEARCH TREES (BST)

PRIORITY QUEUE (HEAP)

HASH-TABLE

DICTIONARY / SYMBOL TABLE

GRAPHS

GRAPH ALGORITHMS

SORTING ALGORITHMS

COUNTING SORT

END NOTE

CHAPTER 4: SORTING

INTRODUCTION

TYPE OF SORTING

BUBBLE-SORT

MODIFIED (IMPROVED) BUBBLE-SORT

INSERTION-SORT

SELECTION-SORT

MERGE-SORT

QUICK-SORT

QUICK SELECT

BUCKET SORT

GENERALIZED BUCKET SORT

HEAP-SORT

TREE SORTING

EXTERNAL SORT (EXTERNAL MERGE-SORT)

STABLE SORTING

COMPARISONS OF THE VARIOUS SORTING ALGORITHMS.

SELECTION OF BEST SORTING ALGORITHM PROBLEMS BASED ON SORTING EXERCISE

CHAPTER 5: SEARCHING

INTRODUCTION
WHY SEARCHING?
DIFFERENT SEARCHING ALGORITHMS
LINEAR SEARCH – UNSORTED INPUT
LINEAR SEARCH – SORTED
BINARY SEARCH
STRING SEARCHING ALGORITHMS
HASHING AND SYMBOL TABLES
HOW SORTING IS USEFUL IN SELECTION ALGORITHM?
PROBLEMS IN SEARCHING
EXERCISE

CHAPTER 6: LINKED LIST

INTRODUCTION
LINKED LIST
TYPES OF LINKED LIST
SINGLY LINKED LIST
DOUBLY LINKED LIST
CIRCULAR LINKED LIST
DOUBLY CIRCULAR LIST
EXERCISE

CHAPTER 7: STACK

INTRODUCTION
THE STACK ABSTRACT DATA TYPE
STACK USING ARRAY
STACK USING ARRAY WITH MEMORY MANAGEMENT
STACK USING LINKED LIST
SYSTEM STACK AND METHOD CALLS
PROBLEMS IN STACK
USES OF STACK
EXERCISE

CHAPTER 8: QUEUE

[INTRODUCTION](#)

[THE QUEUE ABSTRACT DATA TYPE](#)

[QUEUE USING ARRAY](#)

[QUEUE USING LINKED LIST](#)

[PROBLEMS IN QUEUE](#)

[EXERCISE](#)

[CHAPTER 9: TREE](#)

[INTRODUCTION](#)

[TERMINOLOGY IN TREE](#)

[BINARY TREE](#)

[TYPES OF BINARY TREES](#)

[PROBLEMS IN BINARY TREE](#)

[BINARY SEARCH TREE \(BST\)](#)

[PROBLEMS IN BINARY SEARCH TREE \(BST\)](#)

[SEGMENT TREE](#)

[AVL TREES](#)

[RED-BLACK TREE](#)

[SPLAY TREE](#)

[B-TREE](#)

[B+ TREE](#)

[B* TREE](#)

[EXERCISE](#)

[CHAPTER 10: PRIORITY QUEUE / HEAPS](#)

[INTRODUCTION](#)

[TYPES OF HEAP](#)

[HEAP ADT OPERATIONS](#)

[OPERATION ON HEAP](#)

[HEAP-SORT](#)

[USES OF HEAP](#)

[PROBLEMS IN HEAP](#)

[EXERCISE](#)

[CHAPTER 11: HASH-TABLE](#)

[INTRODUCTION](#)

[HASH-TABLE](#)

[HASHING WITH SEPARATE CHAINING](#)

PROBLEMS IN HASHING

EXERCISE

CHAPTER 12: GRAPHS

INTRODUCTION

GRAPH TERMINOLOGY

GRAPH REPRESENTATION

GRAPH TRAVERSALS

DEPTH FIRST TRAVERSAL

BREADTH FIRST TRAVERSAL

USES OF BFS AND DFS

DFS & BFS BASED PROBLEMS

MINIMUM SPANNING TREES (MST)

SHORTEST PATH ALGORITHMS IN GRAPH

HAMILTONIAN PATH AND HAMILTONIAN CIRCUIT

EULER PATH AND EULER CIRCUIT

TRAVELLING SALESMAN PROBLEM (TSP)

EXERCISE

CHAPTER 13: STRING ALGORITHMS

INTRODUCTION

STRING MATCHING

DICTIONARY / SYMBOL TABLE

PROBLEMS IN STRING

EXERCISE

CHAPTER 14: ALGORITHM DESIGN TECHNIQUES

INTRODUCTION

BRUTE FORCE ALGORITHM

GREEDY ALGORITHM

DIVIDE-AND-CONQUER, DECREASE-AND-CONQUER

DYNAMIC PROGRAMMING

REDUCTION / TRANSFORM-AND-CONQUER

BACKTRACKING

BRANCH-AND-BOUND

A* ALGORITHM

CONCLUSION

CHAPTER 15: BRUTE FORCE ALGORITHM

[INTRODUCTION](#)

[PROBLEMS IN BRUTE FORCE ALGORITHM](#)

[CONCLUSION](#)

[CHAPTER 16: GREEDY ALGORITHM](#)

[INTRODUCTION](#)

[PROBLEMS ON GREEDY ALGORITHM](#)

[CHAPTER 17: DIVIDE-AND-CONQUER, DECREASE-AND-CONQUER](#)

[INTRODUCTION](#)

[GENERAL DIVIDE-AND-CONQUER RECURRENCE](#)

[PROBLEMS ON DIVIDE-AND-CONQUER ALGORITHM](#)

[CHAPTER 18: DYNAMIC PROGRAMMING](#)

[INTRODUCTION](#)

[PROBLEMS ON DYNAMIC PROGRAMMING ALGORITHM](#)

[CHAPTER 19: BACKTRACKING](#)

[INTRODUCTION](#)

[PROBLEMS ON BACKTRACKING ALGORITHM](#)

[CHAPTER 20: COMPLEXITY THEORY](#)

[INTRODUCTION](#)

[DECISION PROBLEM](#)

[COMPLEXITY CLASSES](#)

[CLASS P PROBLEMS](#)

[CLASS NP PROBLEMS](#)

[CLASS CO-NP](#)

[NP-HARD:](#)

[NP-COMPLETE PROBLEMS](#)

[REDUCTION](#)

[END NOTE](#)

[APPENDIX](#)

[APPENDIX A](#)

[INDEX](#)

CHAPTER 0: HOW TO USE THIS BOOK

What this book is about

This book introduces you to the world of data structures and algorithms. Data structures defines the way in which data is arranged in memory for fast and efficient access while algorithms are a set of instruction to solve problems by manipulating these data structures.

Designing an efficient algorithm is a very important skill that all software companies, e.g. Microsoft, Google, Facebook etc. pursues. Most of the interviews for these companies are focused on knowledge of data-structures and algorithms. They look for how candidates use concepts of data structures and algorithms to solve complex problems efficiently. Apart from knowing, a programming language you also need to have good command of these key computer fundamentals to not only qualify the interview but also excel in your jobs as a software engineer.

This book assumes that you are a Java language developer. You are not an expert in Java language, but you are well familiar with concepts of classes, functions, arrays, pointers and recursion. At the start of this book, we will be looking into Complexity Analysis followed by the various data structures and their algorithms. We will be looking into a Linked-List, Stack, Queue, Trees, Heap, Hash-Table and Graphs. We will also be looking into Sorting, Searching techniques.

In last few chapters, we will be looking into various algorithmic techniques. Such as, Brute-Force algorithms, Greedy algorithms, Divide and Conquer algorithms, Dynamic Programming, Reduction and Backtracking.

Preparation Plans

Generally, you have few months' time before appearing for a next interview, so it is important to have a solid preparation plan. The preparation plan depends upon the preparation duration and companies that you are planning to target. Below are the three-preparation plan for 1 Month, 3 Month and 5 Month durations.

1 Month Preparation Plans

This preparation plan is for someone who is well familiar with the concepts of data structures and algorithms and just want to revisit these concepts and appear for an interview in a month. Below are the list of topics that we need to study and approximate time to finish them to complete preparation in a month. These are the most important chapters that must be prepared before appearing for an interview.

Time	Chapters	Explanation
Week 1	Chapter 1: Algorithms Analysis Chapter 2: Approach To Solve Algorithm Design Problems Chapter 3: Abstract Data Type	You will get a basic understanding of how to find complexity of a solution. You will come to know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 2	Chapter 4: Sorting Chapter 5: Searching Chapter 13: String Algorithms	Searching, Sorting and String algorithm consists of a major portion of the interviews.
Week 3	Chapter 6: Linked List Chapter 7: Stack Chapter 8: Queue	Linked list, Stack and Queue are some of the favorites in an interview.
Week 4	Chapter 9: Tree	In this portion, you will read about Trees. Now you are well versed to go for interviews. Best of luck.

3 Month Preparation Plan

This plan should be used when you have at least three months' time to prepare for an interview. This preparation plan includes nearly everything in this book except algorithm techniques like “dynamic programming”, “divide & conquer” etc. Which are asked by specific companies like Google, Facebook, etc.

Therefore, until you are planning to face interview with these companies you can withhold these chapters for some time and should focus on the rest of the chapters.

Time	Chapters	Explanation
Week 1	Chapter 1: Algorithms Analysis Chapter 2: Approach To Solve Algorithm Design Problems Chapter 3: Abstract Data Type	You will get a basic understanding of how to find complexity of a solution. You will know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 2 Week 3	Chapter 4: Sorting Chapter 5: Searching Chapter 13: String Algorithms	Searching, sorting and string algorithm consists of a major portion of the interviews.
Week 4 Week 5	Chapter 6: Linked List Chapter 7: Stack Chapter 8: Queue	Linked list, Stack and Queue are some of the favorites in an interview.
Week 6 Week 7	Chapter 9: Tree Chapter 10: Priority Queue / Heap	In this portion, you will read about trees and heap data structures.
Week 8 Week 9	Chapter 11: Hash-Table Chapter 12: Graphs	Hash-Table is used throughout this book in various places, but now it is time to understand how Hash-Table is actually implemented. Graphs are used to propose a solution many real life problems.
Week 10 Week 11 Week 12	Revision of the chapters listed above.	At this time, you need to revise all the chapters that we have gone through in this book. Whatever remains needs to be completed and the exercise that remain unsolved need to be solved at this time

5 Month Preparation Plan

This plan should be used when we have at least 5 months of time. In this plan, we are going to study the whole book. In addition to this, we need to practice more and more from www.topcoder.com and other resources. If you are targeting for google, Facebook, etc., Then it is highly recommended to join topcoder and make practice as much as possible.

Time	Chapters	Explanation
Week 1	Chapter 1: Algorithms Analysis	You will get a basic understanding of

Week 2	Chapter 2: Approach To Solve Algorithm Design Problems Chapter 3: Abstract Data Type	how to find complexity of a solution. You will know how to handle unseen problems. You will read about a variety of datatypes and their uses.
Week 3	Chapter 4: Sorting	Searching, sorting and string
Week 4	Chapter 5: Searching	algorithm consists of a major portion
Week 5	Chapter 13: String Algorithms	of the interviews.
Week 6	Chapter 6: Linked List	Linked list, Stack and Queue are some
Week 7	Chapter 7: Stack	of the favorites in an interview.
Week 8	Chapter 8: Queue	
Week 9	Chapter 9: Tree	This portion you will read about trees
Week 10	Chapter 10: Heap	and priority queue.
Week 11	Chapter 11: Hash-Table	Hash-Table is used throughout this
Week 12	Chapter 12: Graphs	book in various places, but now it is time to understand how Hash-Table are actually implemented.
		Graphs are used to propose a solution in many real life problems.
Week 13	Chapter 14: Algorithm Design Techniques	These chapters contain various
Week 14	Chapter 15: Brute Force	algorithms types and their usage. Once
Week 15		the user is familiar with most of these
Week 16	Chapter 16: Greedy Algorithm Chapter 17: Divide-And-Conquer, Decrease-And-Conquer Chapter 18: Dynamic Programming Chapter 19: Backtracking And Branch-And-Bound Chapter 20: Complexity Theory And Np Completeness	algorithms. Then the next step is to start solving topcoder problems from topcoder .
Week 17	Revision of the chapters listed above.	At this time, you need to revise all the chapters that we have gone through in this book. Whatever remains needs to be completed and the exercise that may remain needs to be solved at this time
Week 18		
Week 19		
Week 20		

Code downloads

You can download the code of solved examples in the book from author's github repositories at <https://github.com/Hemant-Jain-Author/>. At this location the author had solved examples in various programming languages like Java, C#, C++, C, Swift, GoLang, Python 2.7, Python 3, JavaScript ES5, JavaScript ES6, VB.net, PHP and Ruby.

Summary

These are few preparation plans that can be followed to complete this book while preparing for the interview. It is highly recommended that you should read the problem statement, try to solve the problems by yourself and then only you should look into the solution to find the approach of this book. Practicing more and more problems will increase your thinking power and you will be able to handle unseen problems in an interview. We recommend you to make practicing all the problems given in this book, then solve more and more problems from online resources like www.topcoder.com, www.careercup.com etc.

CHAPTER 1: ALGORITHMS ANALYSIS

Introduction

We learn by experience. With experience, it becomes easy to solve new problems. By looking into various problem-solving algorithms or techniques, we begin to develop a pattern that will help us in solving similar problems.

An **algorithm** is a set of steps to accomplish a task. An algorithm is a computer program is a set of steps that are applied over a set of input to produce a set of output.

Knowledge of algorithm helps us to get desired result faster by applying the appropriate algorithm.

The most important properties of an algorithm are:

1. **Correctness:** The algorithm should be correct. It should be able to process all the given inputs and provide correct output.
2. **Efficiency:** The algorithm should be efficient in solving problems. Efficiency is measured in two parameters. First is Time-Complexity, how quick result is provided by an algorithm. Second is Space-Complexity, how much RAM or memory that an algorithm is going to consume to give desired result.

Time-Complexity is represented by function $T(n)$ - time required versus the input size n .

Space-Complexity is represented by function $S(n)$ - memory used versus the input size n .

Asymptotic analysis

Asymptotic analysis is used to compare the efficiency of algorithm independent of any particular data set or programming language.

We are generally interested in the order of growth of an algorithm and not interested in the exact time required for running an algorithm. This time is also called Asymptotic-running time.

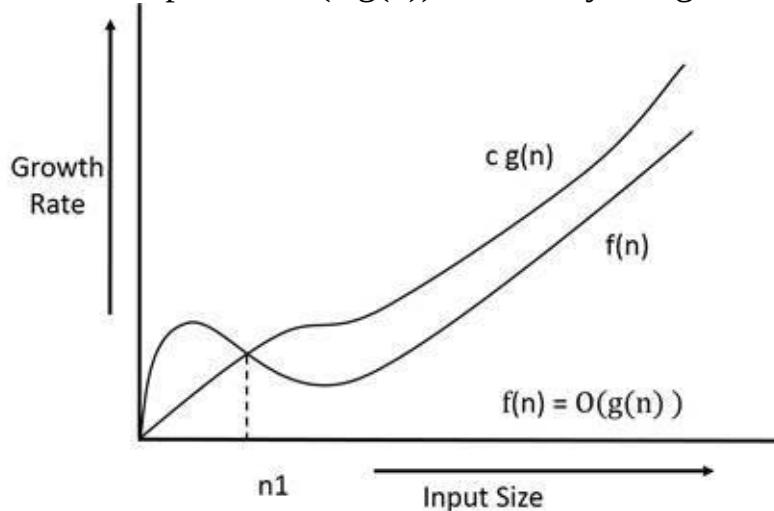
Big-O Notation

Definition: “ $f(n)$ is big-O of $g(n)$ ” or $f(n) = O(g(n))$, if there are two +ve constants c and n_0 such that

$f(n) \leq c g(n)$ for all $n \geq n_0$,

In other words, $c g(n)$ is an upper bound for $f(n)$ for all $n \geq n_0$

The function $f(n)$ growth is slower than $c g(n)$ We can simply say that after a sufficient large value of input N the ($c.g(n)$) will always be greater than $f(n)$.

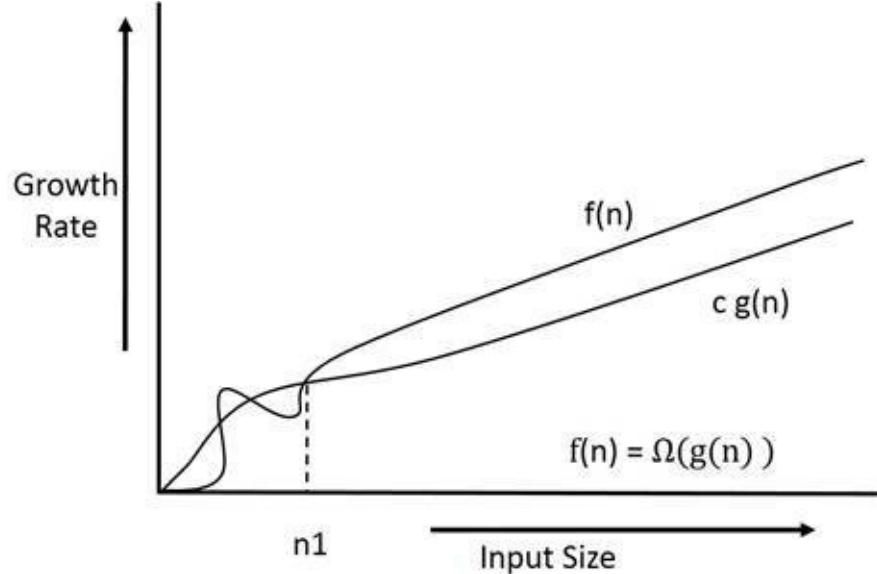


Example: $n^2 + n = O(n^2)$

Omega- Ω Notation

Definition: “ $f(n)$ is omega of $g(n)$.” or $f(n) = \Omega(g(n))$ if there are two +ve constants c and n_0 such that $c g(n) \leq f(n)$ for all $n \geq n_0$

In other words, $c g(n)$ is lower bound for $f(n)$ Function $f(n)$ growth is faster than $c g(n)$



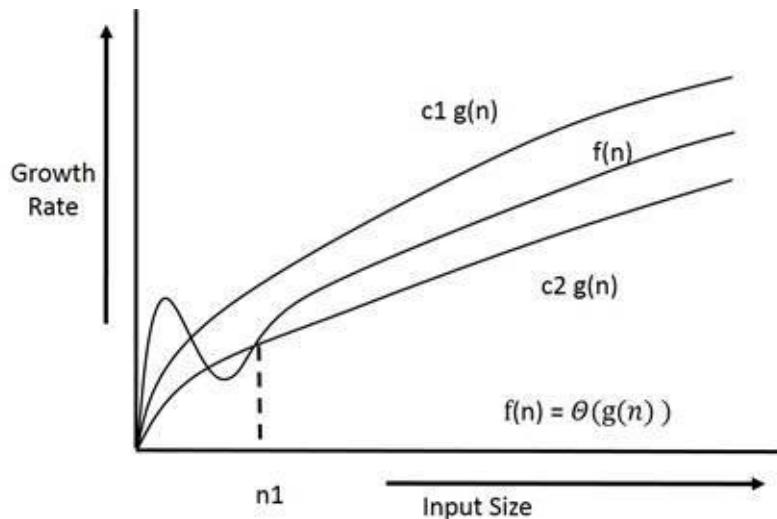
Find relationship of $f(n) = n^c$ and $g(n) = c^n$

$$f(n) = \Omega(g(n))$$

Theta- Θ Notation

Definition: “ $f(n)$ is theta of $g(n)$.” or $f(n) = \Theta(g(n))$ if there are three +ve constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$

Function $g(n)$ is an asymptotically tight bound on $f(n)$. Function $f(n)$ grows at the same rate as $g(n)$.



Example: $n^3 + n^2 + n = \Theta(n^3)$

Example: $n^2 + n = \Theta(n^2)$

Find relationship of $f(n) = 2n^2 + n$ and $g(n) = n^2$

$f(n) = O(g(n))$

$f(n) = \Theta(g(n))$

$f(n) = \Omega(g(n))$

Note: - Asymptotic Analysis is not perfect, but that is the best way available for analyzing algorithms.

For example, say there are two sorting algorithms first take $f(n) = 10000 * n * \log(n)$ and second $f(n) = n^2$ time. The asymptotic analysis says that the first algorithm is better (as it ignores constants) but, actually for a small set of data when n is smaller than 10000, the first algorithm will perform better. To consider this drawback of asymptotic analysis case analysis of the algorithm is

introduced.

Complexity analysis of algorithms

1. **Worst Case complexity:** It is the complexity of solving the problem for the worst input of size n. It provides the upper bound for the algorithm. This is the most common analysis used.
2. **Average Case complexity:** It is the complexity of solving the problem on an average. We calculate the time for all the possible inputs and then take an average of it.
3. **Best Case complexity:** It is the complexity of solving the problem for the best input of size n.

Time Complexity Order

A list of commonly occurring algorithm Time Complexity in increasing order:

Name	Notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
N-LogN	$O(n \log n)$
Quadratic	$O(n^2)$
Polynomial	$O(n^c)$ c is a constant & $c > 1$
Exponential	$O(c^m)$ c is a constant & $c > 1$
Factorial or N-power-N	$O(n!)$ or $O(n^n)$

Below diagram shows growth rate of some of the commonly occurring complexities.

N	Function Growth Rate (Approximate)						
	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	1	3	10	30	10^2	10^3	10^3
10^2	1	6	10^2	6×10^2	10^4	10^6	10^{30}
10^3	1	9	10^3	9×10^3	10^6	10^9	10^{300}
10^4	1	13	10^4	13×10^4	10^8	10^{12}	10^{3000}
10^5	1	16	10^5	16×10^5	10^{10}	10^{15}	10^{30000}
10^6	1	19	10^6	19×10^6	10^{12}	10^{18}	10^{300000}

From the above table it is clear that the time required for completing some algorithms changes drastically with the growth rate. For same data set, some algorithms will give result in minutes if not seconds, while there are other algorithms that will not be able to complete in days.

Constant Time $O(1)$

An algorithm is said to run in constant time if the output is produced in constant time regardless of the input size.

Examples:

1. Accessing n^{th} element of an Array
2. Push and pop of a stack.

3. Add and remove of a queue.
4. Accessing an element of Hash-Table.

Linear Time $O(n)$

An algorithm is said to run in linear time if the execution time of the algorithm is directly proportional to the input size.

Examples:

1. Array operations like search element, find min, find max etc.
2. Linked list operations like traversal, find min, find max etc.

Note: when we need to see/ traverse all the nodes of a data-structure for some task then complexity is no less than $O(n)$

Logarithmic Time, $O(\log n)$

An algorithm is said to run in logarithmic time if the execution time of the algorithm is proportional to the logarithm of the input size. Each step of an algorithm, a significant portion (eg. half portion) of the input is pruned / rejected out without traversing it.

Example: Binary search algorithm, we will read about this algorithm in this book.

$N \cdot \log N$ Time, $O(n \log n)$

An algorithm is said to run in $n \log n$ time if execution time of an algorithm is proportional to the product of input size and logarithm of the input size. In these algorithms, each time the input is divided into half (or some proportion) and each portion is processed independently.

Example:

1. Merge-Sort
2. Quick-Sort (Average case)
3. Heap-Sort

Note: Quicksort is a special kind of algorithm to sort an Array of numbers. Its worst-case complexity is $O(n^2)$ and average case complexity is $O(n \log n)$.

Quadratic Time, $O(n^2)$

An algorithm is said to run in quadratic time if the execution time of an algorithm is proportional to the square of the input size. In these algorithms each element are compared with all the other elements.

Examples:

1. Bubble-Sort
2. Selection-Sort
3. Insertion-Sort

Exponential Time $O(2^n)$

In these algorithms, all possible subsets of elements of input date are generated.

Factorial Time $O(n!)$

In these algorithms, all possible permutation of elements of input date are generated.

Deriving the Runtime Function of an Algorithm

Constants

Each statement takes a constant time to run. Time Complexity is $O(1)$

Loops

The running time of a loop is a product of running time of the statement inside a loop and number of iterations in the loop. Time Complexity is $O(n)$

Nested Loop

The running time of a nested loop is a product of running time of the statements inside loop multiplied by a product of the size of all the loops. Time Complexity is $O(n^c)$. Where c is a number of loops. For two loops, it will be $O(n^2)$

Consecutive Statements

Just add the running times of all the consecutive statements

If-Else Statement

Consider the running time of the larger of if block or else block and ignore the other block.

Logarithmic statement

If each iteration the input size is decreased by a constant factors. Time Complexity = $O(\log n)$.

Time Complexity Examples

Example 1.1

```
int fun1(int n) {  
    int m = 0;  
    for (int i = 0; i < n; i++) {  
        m += 1;  
    }  
    return m;  
}
```

Time Complexity: $O(n)$, single for loop takes linear time.

Example 1.2

```
int fun2(int n) {  
    int i, j, m = 0; for (i = 0; i < n; i++) {  
        for (j = 0; j < n; j++) {  
            m += 1;  
        }  
    }  
    return m;  
}
```

Time Complexity: $O(n^2)$, two nested for loop takes quadratic time.

Example 1.3

```
int fun3(int n) {  
    int i, j, m = 0; for (i = 0; i < n; i++) {  
        for (j = 0; j < i; j++) {  
            m += 1;  
        }  
    }  
    return m;  
}
```

Time Complexity: $O(N+(N-1)+(N-2)+\dots) == O(N(N+1)/2) == O(n^2)$

Example 1.4

```
int fun4(int n) {  
    int i, m = 0; i = 1;  
    while (i < n) {  
        m += 1;  
        i = i * 2;  
    }  
    return m;  
}
```

In each iteration, the value of I is doubled. Each time problem space is divided into half.

Time Complexity: $O(\log(n))$

Example 1.5

```
int fun5(int n) {  
    int i, m = 0; i = n;  
    while (i > 0) {  
        m += 1;  
        i = i / 2;  
    }  
    return m;  
}
```

In each iteration, the value of “I” is halved. Same as above each time problem space is divided into half.

Time Complexity: $O(\log(n))$

Example 1.6

```
int fun6(int n) {  
    int i, j, k, m = 0;  
    for (i = 0; i < n; i++) {  
        for (j = 0; j < n; j++) {  
            for (k = 0; k < n; k++) {  
                m += 1;  
            }  
        }  
    }  
}
```

```
    }
    return m;
}
```

Outer loop will run for n number of iterations. In each iteration of the outer loop, inner loop will run for n iterations of its own. Three nested loops running n number of times.

Time Complexity: $O(n^3)$

Example 1.7

```
int fun7(int n) {
    int i, j, k, m = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            m += 1;
        }
    }
    for (i = 0; i < n; i++) {
        for (k = 0; k < n; k++) {
            m += 1;
        }
    }
    return m;
}
```

These two groups of loop are in consecutive so their complexity will add up to form the final complexity of the program. Time Complexity: $O(n^2) + O(n^2) = O(n^2)$

Example 1.8

```
int fun8(int n) {
    int i, j, m = 0;
    for (i = 0; i < n; i++) {
        for (j = 0; j < Math.sqrt(n); j++) {
            m += 1;
        }
    }
    return m;
}
```

```
}
```

Time Complexity: $O(n * \sqrt{n}) = O(n^{3/2})$

Example 1.9

```
int fun9(int n) {
    int i, j, m = 0; for (i = n; i > 0; i /= 2) {
        for (j = 0; j < i; j++) {
            m += 1;
        }
    }
    return m;
}
```

For nested loops look for inner loop iterations. Time complexity will be calculated by looking into inner loop. First, it will run for n number of time then $n/2$ and so on. ($n+n/2+n/4+n/8+n/16 \dots$)

Time Complexity: $O(n)$

Example 1.10

```
int fun10(int n) {
    int i, j, m = 0; for (i = 0; i < n; i++) {
        for (j = i; j > 0; j--) {
            m += 1;
        }
    }
    return m;
}
```

$O(N+(N-1)+(N-2)+\dots) = O(N(N+1)/2)$ // arithmetic progression.

Time Complexity: $O(n^2)$

Example 1.11

```
int fun11(int n) {
    int i, j, k, m = 0;
    for (i = 0; i < n; i++) {
        for (j = i; j < n; j++) {
```

```
for (k = j + 1; k < n; k++) {  
    m += 1;  
}  
}  
}  
}  
return m;  
}
```

Time Complexity: $O(n^3)$

Example 1.12

```
int fun12(int n) {  
    int i, j = 0, m = 0;  
    for (i = 0; i < n; i++) {  
        for (; j < n; j++) {  
            m += 1;  
        }  
    }  
    return m;  
}
```

Think carefully once again before finding a solution, j value is not reset at each iteration.

Time Complexity: $O(n)$

Example 1.13

```
int fun13(int n) {  
    int i, j, m = 0; for (i = 1; i <= n; i *= 2) {  
        for (j = 0; j <= i; j++) {  
            m += 1;  
        }  
    }  
    return m;  
}
```

The inner loop will run for 1, 2, 4, 8,... n times in successive iteration of the outer loop.

Time Complexity: $T(n) = O(1+ 2+ 4+ \dots+n/2+n) = O(n)$

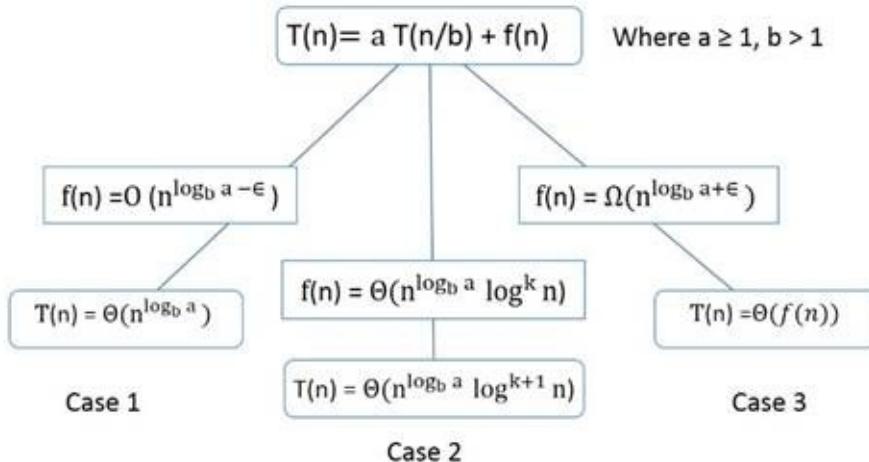
Master Theorem

The master theorem solves recurrence relations of the form: $T(n) = a T(n/b) + f(n)$, Where $a \geq 1$ and $b > 1$.

"n" is the size of the problem. "a" is a number of sub problem in the recursion. " n/b " is the size of each sub-problem. "f(n)" is the cost of the division of the problem into sub problem and merger of results of sub-problems to get the final result.

It is possible to determine an asymptotic tight bound in these three cases:

- Case 1: when $f(n) = O(n^{\log_b a - \epsilon})$ and constant $\epsilon > 1$, then the final Time complexity is: $T(n) = \Theta(n^{\log_b a})$
- Case 2: when $f(n) = \Theta(n^{\log_b a} \log^k n)$ and constant $k \geq 0$, then the final Time complexity is: $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
- Case 3: when $f(n) = \Omega(n^{\log_b a + \epsilon})$ and constant $\epsilon > 1$, Then the final Time complexity is:
$$T(n) = \Theta(f(n))$$



Example 1.14: Take an example of Merge-Sort, $T(n) = 2 T(n/2) + n$

$$\text{Sol:- } \log_b a = \log_2 2 = 1$$

$$f(n) = n = \Theta(n^{\log_2 2} \log^0 n)$$

Case 2 applies and $T(n) = \Theta(n^{\log_2 2} \log^{0+1} n)$

$$T(n) = \Theta(n \log(n))$$

Example 1.15: Binary Search $T(n) = T(n/2) + O(1)$ Sol:- $\log_b a = \log_2 1 = 0$

$$f(n) = 1 = \Theta(n^{\log_2 1} \log^0 n)$$

Case 2 applies and $T(n) = \Theta(n^{\log_2 1} \log^{0+1} n)$

$$T(n) = \Theta(\log(n))$$

Example 1.16: Binary tree traversal $T(n) = 2T(n/2) + O(1)$

Sol:- $\log_b a = \log_2 2 = 1$

$$f(n) = 1 = O(n^{\log_2 2-1})$$

Case 1 applies and $T(n) = \Theta(n^{\log_2 2})$

$$T(n) = \Theta(n)$$

Example 1.17: $T(n) = 2 T(n/2) + n^2$

Sol:- $\log_b a = \log_2 2 = 1$

$$f(n) = n^2 = \Omega(n^{\log_2 2+1})$$

Case 3 applies and $T(n) = \Theta(f(n))$

$$T(n) = \Theta(n^2)$$

Example 1.18: $T(n) = 4 T(n/2) + n^2$

Sol:- $\log_b a = \log_2 4 = 2$

$$f(n) = n^2 = \Theta(n^{\log_2 4} \log^0 n)$$

Case 2 applies and $T(n) = \Theta(n^{\log_2 4} \log^{0+1} n)$

$$T(n) = \Theta(n^2 \log n)$$

Example 1.19: $T(n) = T(n/2) + 2n$ Sol:- $\log_b a = \log_2 1 = 0$

Case 3

$$T(n) = \Theta(n)$$

Example 1.20: $T(n) = 16T(n/4) + n$ Sol:- $\log_b a = \log_4 16 = 2$

Case 1

$$T(n) = \Theta(n^2)$$

Example 1.21: $T(n) = 2T(n/2) + n \log n$ Sol:- $\log_b a = \log_2 2 = 1$

$$f(n) = n \log(n) = \Theta(n^{\log_2 2} \log^1 n)$$

$$T(n) = \Theta(n^{\log_2 2} \log^{0+1} n) = \Theta(n \log(n))$$

Example 1.22: $T(n) = 2 T(n/4) + n^{0.5}$

Sol:- $\log_b a = \log_4 2 = 0.5$

Case 2:

$$T(n) = \Theta(n^{\log_4 2} \log^{0.5+1} n) = \Theta(n^{0.5} \log^{1.5} n)$$

Example 1.23: $T(n) = 2 T(n/4) + n^{0.49}$

Sol:- $\log_b a = \log_4 2 = 0.5$

Case 1:

$$T(n) = \Theta(n^{\log_4 2}) = \Theta(n^{0.5})$$

Example 1.24: $T(n) = 3T(n/3) + \sqrt{n}$ Sol:- $\log_b a = \log_3 3 = 1$

Case 1

$$T(n) = \Theta(n)$$

Example 1.25: $T(n) = 3T(n/4) + n \log n$ Sol:-

$$f(n) = n \log n = \Omega(n^{\log_4 3} \log^1 n)$$

Case 3:

$$T(n) = \Theta(n \log(n))$$

Example 1.26: $T(n) = 3T(n/3) + n/2$

Sol:- $\log_b a = \log_3 3 = 1$

Case 2:

$$T(n) = \Theta(n \log(n))$$

Array Based Questions

The following section will discuss the various algorithms that are applicable to Arrays and will follow by list of practice problems with similar approaches.

Sum Array

Problem: Write a method that will return the sum of all the elements of the integer Array, given Array as an input argument.

Example 1.27:

```
public static int SumArray(int arr[]) {  
    int size = arr.length;  
    int total = 0;  
    for (int index = 0; index < size; index++) {  
        total = total + arr[index];  
    }  
    return total;  
}  
  
public static void main(String[] args) {  
    int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
    System.out.println("Sum of values in array:" + SumArray(arr));  
}
```

Output:

Sum of values in array: 45

Analysis: All the elements of array are traversed and added. Finally, the result is returned. Time complexity is $O(n)$.

Sequential Search

Problem: Write a method, which will search an Array for some given value.

Example 1.28:

```

public static int SequentialSearch(int arr[], int size, int value) {
    for (int i = 0; i < size; i++) {
        if (value == arr[i]) {
            {
                return i;
            }
        }
    }
    return -1;
}

```

Analysis:

- Since we have no idea about the data stored in the array, or if the data is not sorted then we have to search the array in sequential manner one by one.
- If we find the value, we are looking for we return its index.
- Else, we return index -1 in the end, as we did not find the value we are looking for.
- The elements of the array are traversed sequentially so the Time complexity is $O(n)$

Binary Search

If we want to search some value in sorted array, a binary search can be used. We examine the middle position at each step. Depending upon the data that we are searching is greater or smaller than the middle value. We will search either the left or the right portion of the array. At each step, we are eliminating half of the search space, thereby, making this algorithm efficient as compared with the linear search.

Example 1.29: Binary search in a sorted array.

```

public static int BinarySearch(int arr[], int size, int value) {
    int mid;
    int low = 0;
    int high = size - 1;
    while (low <= high) {
        mid = low + (high - low) / 2; // To avoid the overflow
        if (arr[mid] == value) {

```

```

        return mid;
    } else {
        if (arr[mid] < value) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

```

Analysis:

- Since we have data sorted in increasing / decreasing order, we can search efficiently by applying binary search. At each step, we reduce our search space by half.
- At each step, we compare the middle value with the value we are searching. If mid value is equal to the value we are searching for then we return the middle index.
- If the value is smaller than the middle value, we search the left half of the array.
- If the value is greater than the middle value then we search the right half of the array.
- If we find the value we are looking for then its index is returned otherwise -1 is returned.
- Time complexity of this algorithm is $O(\log n)$.

Rotating an Array by K positions.

Problem: Given an Array, you need to rotate its elements K number of times. For example, an Array [10,20,30,40,50,60] rotate by 2 positions to [30,40,50,60,10,20]

Example 1.30:

```

public static void rotateArray(int[] a, int n, int k) {
    reverseArray(a, 0, k - 1);
    reverseArray(a, k, n - 1);
    reverseArray(a, 0, n - 1);
}

```

```

}

public static void reverseArray(int[] a, int start, int end) {
    for (int i = start, j = end; i < j; i++, j--) {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}

```

Analysis:

- Rotating list is done in two parts trick. In the first part, we first reverse elements of list first half and then second half.
 $1,2,3,4,5,6,7,8,9,10 \Rightarrow 5,6,7,8,9,10,1,2,3,4$
 $1,2,3,4,5,6,7,8,9,10 \Rightarrow 4,3,2,1,10,9,8,7,6,5 \Rightarrow 5,6,7,8,9,10,1,2,3,4$
- Then we reverse the whole list thereby completing the whole rotation.
- First reversal of the two parts of list is done in $O(n)$ time and the final list reversal is also done in $O(n)$ so the total time complexity of this algorithms is $O(n)$

Find the largest sum contiguous subarray.

Problem: Given an Array of positive and negative integers, find a contiguous subarray whose sum (sum of elements) is maximum.

Example 1.31:

```

public static int maxSubArraySum(int[] a, int size) {
    int maxSoFar = 0, maxEndingHere = 0;

    for (int i = 0; i < size; i++) {
        maxEndingHere = maxEndingHere + a[i];
        if (maxEndingHere < 0) {
            maxEndingHere = 0;
        }
        if (maxSoFar < maxEndingHere) {
            maxSoFar = maxEndingHere;
        }
    }
}

```

```

    }
    return maxSoFar;
}

public static void main(String[] args) {
    int[] arr = {1,-2,3,4,-4,6,-4,3,2};
    System.out.println("Max sub array sum :" + maxSubArraySum(arr, 9));
}

```

Analysis:

- Maximum subarray in an Array is found in a single scan. We keep track of global maximum sum so far and the maximum sum, which include the current element.
- When we find global maximum value so far is less than the maximum value containing current value we update the global maximum value.
- Finally return the global maximum value.
- Time complexity is O(n).

Array wave form

Problem: Given an array, arrange its elements in wave form such that odd elements are lesser then its neighbouring even elements.

First solution: Compare even index values with its neighbour odd index values and swap if odd index is not smaller than even index.

Example 1.32:

```

public static void WaveArray2(int[] arr){
    int size = arr.length;
    /* Odd elements are lesser then even elements. */
    for ( int i = 1 ;i < size; i+= 2){
        if ((i - 1) >= 0 && arr[i] > arr[i - 1]){
            swap(arr, i, i-1);
        }
        if ((i + 1) < size && arr[i] > arr[i + 1]){
            swap(arr, i, i+1);
        }
    }
}

```

```

    }
}

public static void main(String[] args) {
    int[] arr = {8, 1, 2, 3, 4, 5, 6, 4, 2};
    WaveArray2(arr);
    printArray(arr, arr.length);
}

```

Time Complexity: O(n)

Second solution: Sort the array then swap ith and i+1th index value so the array will form a wave.

Example 1.33:

```

public static void WaveArray(int[] arr){
    int size = arr.length;
    QuickSort srt = new QuickSort(arr);
    srt.sort();
    for(int i = 0 ; i < size - 1 ; i+= 2) /* Swap adjacent elements in array */
        swap(arr, i, i+1);
}

```

Time Complexity: O(n)

Index Array

Problem: Given array of size N, containing elements from 0 to N-1. All values from 0 to N-1 are present in array and if they are not there than -1 is there to take its place. Arrange values of array so that value i is stored at arr[i].

Input: [8, -1, 6, 1, 9, 3, 2, 7, 4, -1]

Output: [-1, 1, 2, 3, 4, -1, 6, 7, 8, 9]

First solution: For each index value pick the element and then put it into its proper place and the element which is in its proper position then pick it and repeat the process again.

Example 1.34:

```
public static void indexArray(int[] arr, int size){  
    for( int i = 0; i < size; i++){  
        int curr = i;  
        int value = -1;  
  
        /* swaps to move elements in proper position. */  
        while (arr[curr] != -1 && arr[curr] != curr){  
            int temp = arr[curr];  
            arr[curr] = value;  
            value = curr = temp;  
        }  
        /* check if some swaps happened.*/  
        if (value != -1){  
            arr[curr] = value;  
        }  
    }  
}
```

Time Complexity: $O(n)$, it looks like quadratic time complexity but the inner loop traverse elements only one. Once inner loop elements are processed then the elements at that position will be either that index value or it will contain -1.

Second solution: For each index, we will pick the value at an index and put that value at its proper position.

Example 1.35:

```
public static void indexArray2(int[] arr, int size)  
{  
    int temp;  
    for (int i = 0; i < size; i++)  
    {  
        while (arr[i] != -1 && arr[i] != i)  
        {  
            /* swap arr[i] and arr[arr[i]] */  
            temp = arr[i];  
            arr[i] = arr[arr[i]];  
            arr[arr[i]] = temp;  
        }  
    }  
}
```

```

        arr[i] = arr[temp];
        arr[temp] = temp;
    }
}
}

/* Testing code */
public static void main(String[] args) {
    int[] arr = {8, -1, 6, 1, 9, 3, 2, 7, 4, -1};
    int size = arr.length;
    indexArray2(arr, size);
    printArray(arr, size);
}

```

Time Complexity: $O(n)$, it looks like quadratic time complexity but the inner loop swaps elements once no matter how many time the outer loop run. Once inner loop elements are processed then the elements at that position will be either that index value or it will contain -1.

Sort 1toN

Problem: Given an array of length N. It contains unique elements from 1 to N. Sort the elements of the array.

First solution: For each index value pick the element and then put it into its proper place and the element which is in its proper position then pick it and repeat the process again.

Example 1.36:

```

public static void Sort1toN(int[] arr, int size)
{
    int curr, value, next;
    for (int i = 0; i < size; i++)
    {
        curr = i;
        value = -1;
        /* swaps to move elements in proper position.*/
    }
}

```

```

while (curr >= 0 && curr < size && arr[curr] != curr + 1)
{
    next = arr[curr];
    arr[curr] = value;
    value = next;
    curr = next - 1;
}
}

public static void main(String[] args) {
    int[] arr = {8, 5, 6, 1, 9, 3, 2, 7, 4, 10};
    int size = arr.length;
    Sort1toN(arr, size);
    printArray(arr, size);
}

```

Time Complexity: $O(n)$, it looks like quadratic time complexity but the inner loop traverse elements only one. Once inner loop elements are processed then the elements at that position will be either that index value or it will contain -1.

Second solution: For each index, we will pick the value at that index and put that value at its proper position.

Example 1.37: Swapping the elements

```

public static void Sort1toN2(int[] arr, int size)
{
    int temp;
    for (int i = 0; i < size; i++)
    {
        while (arr[i] != i + 1 && arr[i] > 1)
        {
            temp = arr[i];
            arr[i] = arr[temp - 1];
            arr[temp - 1] = temp;
        }
    }
}

```

```
}
```

Time Complexity: $O(n)$, it looks like quadratic time complexity but the inner loop swaps elements once no matter how many time the outer loop run. Once inner loop elements are processed then the elements at that position will be either that index value or it will contain -1.

Smallest Positive Missing Number

Problem: Given an unsorted array, find smallest positive number missing in the array.

First solution: Brute force approach, for each number in range we find if the number is present in the array.

Example 1.38:

```
public static int SmallestPositiveMissingNumber(int[] arr, int size)
{
    int found;
    for (int i = 1; i < size + 1; i++)
    {
        found = 0;
        for (int j = 0; j < size; j++)
        {
            if (arr[j] == i)
            {
                found = 1;
                break;
            }
        }
        if (found == 0)
            return i;
    }
    return -1;
}

public static void main(String[] args) {
    int[] arr = {8, 5, 6, 1, 9, 11, 2, 7, 4, 10};
```

```

int size = arr.length;
System.out.println("Max sub array sum:" +
SmallestPositiveMissingNumber(arr, size));
}

```

Time Complexity: $O(n^2)$

Second solution: Using hash table to keep track of elements.

Example 1.39:

```

public static int SmallestPositiveMissingNumber2(int[] arr, int size){
    HashMap<Integer, Integer> hs = new HashMap<Integer, Integer>();
    for(int i = 0;i< size; i++) {
        hs.put(arr[i], 1);
    }
    for(int i=1;i< size+1; i++) {
        if (hs.containsKey(i) == false){
            return i;
        }
    }
    return -1;
}

```

Time Complexity: $O(n)$, Space Complexity: $O(n)$ for hash table.

Third solution: Using an auxiliary array to store values. We know the range so we had created an auxiliary array then traverse the input array. Mark the values that are found in array.

Example 1.40:

```

public static int SmallestPositiveMissingNumber3(int[] arr, int size)
{
    int[] aux = new int[20];
    Arrays.fill(aux, -1);

    for (int i = 0; i < size; i++)
    {

```

```

if (arr[i] > 0 && arr[i] <= size)
aux[arr[i] - 1] = arr[i];
}
for (int i = 0; i < size; i++)
{
if (aux[i] != i + 1)
{
return i + 1;
}
}
return -1;
}

```

Time Complexity: O(n), Space Complexity: O(n) for auxiliary array.

Forth Solution: Rearranging the elements in the given array. By rearranging the elements, we can find the missing element with constant space complexity in linear time.

Example 1.41:

```

public static int SmallestPositiveMissingNumber4(int[] arr, int size)
{
    int temp;
    for (int i = 0; i < size; i++)
    {
        while (arr[i] != i + 1 && arr[i] > 0 && arr[i] <= size)
        {
            temp = arr[i];
            arr[i] = arr[temp - 1];
            arr[temp - 1] = temp;
        }
    }
    for (int i = 0; i < size; i++)
    {
        if (arr[i] != i + 1)
            return i + 1;
    }
}

```

```
    return -1;  
}
```

Time Complexity: O(n), Space Complexity: O(1).

Maximum Minimum Array

Problem: Given a sorted array, rearrange it in maximum-minimum form.

Input: [1, 2, 3, 4, 5, 6, 7]

Output: [7, 1, 6, 2, 5, 3, 4]

First solution: Using auxiliary array, create a copy of the input array. Traverse from start and end of array, and put these values in input array in alternatively.

Example 1.42:

```
public static void MaxMinArr(int arr[], int size) {  
    int[] aux = new int[size];  
    for (int i = 0; i < size; i++) {  
        aux[i] = arr[i];  
    }  
  
    int start = 0;  
    int stop = size - 1;  
    for (int i = 0; i < size; i++) {  
        if (i % 2 == 0) {  
            arr[i] = aux[stop];  
            stop -= 1;  
        } else {  
            arr[i] = aux[start];  
            start += 1;  
        }  
    }  
}  
  
public static void main(String[] args) {  
    int[] arr = { 1, 2, 3, 4, 5, 6, 7 };  
    int size = arr.length;
```

```

MaxMinArr(arr, size);
printArray(arr, size);
}

```

Time Complexity: O(n), Space Complexity: O(n).

Second solution: Without using any auxiliary array. Use reverse array operation on the array and change the array as follows:

```

1, 2, 3, 4, 5, 6, 7
7, 6, 5, 4, 3, 2, 1
7, 1, 2, 3, 4, 5, 6
7, 1, 6, 5, 4, 3, 2
7, 1, 6, 2, 3, 4, 5
7, 1, 6, 2, 5, 4, 3
7, 1, 6, 2, 5, 3, 4

```

Example 1.43:

```

public static void ReverseArr(int arr[], int start, int stop) {
    while (start < stop) {
        swap(arr, start, stop);
        start += 1;
        stop -= 1;
    }
}

public static void MaxMinArr2(int arr[], int size) {
    for (int i = 0; i < (size - 1); i++) {
        ReverseArr(arr, i, size - 1);
    }
}

```

Time Complexity: O(n^2), Space Complexity: O(1).

Max Circular Sum

Problem: Given an array you need to find maximum sum of $\text{arr}[i] * (i+1)$ for all element such that you are allowed to rotate the array.

Example 1.44:

```
public static int maxCircularSum(int[] arr, int size){  
    int sumAll = 0;  
    int currVal = 0;  
    int maxVal;  
  
    for(int i = 0; i < size ; i++){  
        sumAll += arr[i];  
        currVal += (i*arr[i]);  
    }  
    maxVal = currVal;  
    for(int i= 1; i < size; i++){  
        currVal = ( currVal + sumAll ) - ( size * arr[size-i] );  
        if (currVal > maxVal){  
            maxVal = currVal;  
        }  
    }  
    return maxVal;  
}  
  
/* Testing code */  
public static void main(String[] args) {  
    int[] arr = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};  
    System.out.println("MaxCircularSum: " + maxCircularSum(arr, arr.length));  
}
```

Time Complexity: O(n), Space Complexity: O(1).

Array Index Max Diff

Problem: Given an array arr[], find maximum distance of index j and i, such that arr[j] > arr[i]

First solution: Brute force, for each index call it i find index j such that arr[j] > arr[i]. We will need two loops one to select index i and another to traverse index i+1 to size of array.

Example 1.45:

```
public static int ArrayIndexMaxDiff(int[] arr, int size)
{
    int maxDiff = -1;
    int j;
    for (int i = 0; i < size; i++)
    {
        j = size - 1;
        while (j > i)
        {
            if (arr[j] > arr[i])
            {
                maxDiff = Math.max(maxDiff, j - i);
                break;
            }
            j -= 1;
        }
    }
    return maxDiff;
}

public static void main(String[] args) {
    int[] arr = {33, 9, 10, 3, 2, 60, 30, 33, 1};
    System.out.println("MaxDiff : " + ArrayIndexMaxDiff(arr, arr.length));
}
```

Time Complexity: $O(n^2)$, Space Complexity: $O(1)$.

Second solution: Pre-processing and creating two auxiliary arrays. Always keep decreasing minimum values in decreasing order.

Example 1.46:

```
public static int ArrayIndexMaxDiff2(int[] arr, int size)
{
    int[] leftMin = new int[size];
    int[] rightMax = new int[size];
    leftMin[0] = arr[0];
```

```
int i, j;
int maxDiff;
for (i = 1; i < size; i++)
{
if (leftMin[i - 1] < arr[i])
leftMin[i] = leftMin[i - 1];
else
leftMin[i] = arr[i];
}
rightMax[size - 1] = arr[size - 1];
for (i = size - 2; i >= 0; i--)
{
if (rightMax[i + 1] > arr[i])
{
rightMax[i] = rightMax[i + 1];
}
else
{
rightMax[i] = arr[i];
}
}
i = 0;
j = 0;
maxDiff = -1;
while (j < size && i < size)
{
if (leftMin[i] < rightMax[j])
{
maxDiff = Math.max(maxDiff, j - i);
j = j + 1;
}
else
{
i = i + 1;
}
}
return maxDiff;
```

}

Time Complexity: O(n), Space Complexity: O(n).

Max Path Sum

Problem: Given two array in increasing order you need to find maximum sum by choosing few consecutive elements from one array then few elements form other array. The elements switching can happened at transition points only when elements value is same in both the array.

```
arr1 = [12, 13, 18, 20, 22, 26, 70]
arr2 = [11, 15, 18, 19, 20, 26, 30, 31]
```

Max Sum elements: 11, 15, 18, 19, 20, 22, 26, 70
Max Sum: 201

Example 1.47:

```
public static int maxPathSum(int[] arr1, int size1, int[] arr2, int size2) {
    int i = 0, j = 0, result = 0, sum1 = 0, sum2 = 0;

    while (i < size1 && j < size2) {
        if (arr1[i] < arr2[j]) {
            sum1 += arr1[i];
            i += 1;
        } else if (arr1[i] > arr2[j]) {
            sum2 += arr2[j];
            j += 1;
        } else {
            result += Math.max(sum1, sum2);
            result = result + arr1[i];
            sum1 = 0;
            sum2 = 0;
            i += 1;
            j += 1;
        }
    }
}
```

```
while (i < size1) {  
    sum1 += arr1[i];  
    i += 1;  
}  
  
while (j < size2) {  
    sum2 += arr2[j];  
    j += 1;  
}  
  
result += Math.max(sum1, sum2);  
return result;  
}  
  
/* Testing code */  
public static void main(String[] args) {  
    int[] arr1 = { 12, 13, 18, 20, 22, 26, 70 };  
    int[] arr2 = { 11, 15, 18, 19, 20, 26, 30, 31 };  
    System.out.println("Max Path Sum :: " + maxPathSum(arr1, arr1.length,  
arr2, arr2.length));  
}
```

Time Complexity: O(n), Space Complexity: O(n).

Recursive Function

A recursive function is a function that calls itself, directly or indirectly.

A recursive method consists of two parts: Termination Condition and Body (which includes recursive expansion).

1. **Termination Condition:** A recursive method always contains one or more terminating condition. A condition in which recursive method processes a simple case and does not call itself.
2. **Body** (including recursive expansion): The main logic of the recursive method is contained in the body of the method. It also contains the recursion expansion statement that, in turn, calls the method itself.

Three important properties of recursive algorithm are:

- 1) A recursive algorithm must have a termination condition.
- 2) A recursive algorithm must change its state, and move towards the termination condition.
- 3) A recursive algorithm must call itself.

Note: The speed of a recursive program is slower because of stack overheads. If the same task can be done using an iterative solution (using loops), then we should prefer an iterative solution in place of recursion to avoid stack overhead.

Note: Without termination condition, the recursive method may run forever and will finally consume all the stack memory.

Factorial

Problem: Given a value N find N! , $N! = N * (N-1) \dots 2 * 1$.

Example 1. 48: Factorial Calculation.

```
public int factorial(int i) {  
    // Termination Condition  
    if (i <= 1) {  
        return 1;  
    }  
    // Body, Recursive Expansion
```

```
    return i * factorial(i - 1);
}
```

Analysis: Each time method fn is calling fn-1. Time Complexity is **O(N)**

Print Base 16 Integers

Problem: Given an integer in decimal form print its hexadecimal form.

Example 1.49: Generic print to some specific base method.

```
public void printInt(int number) {
    String conversion = "0123456789ABCDEF";
    int base = 16;
    char digit = (char) (number % base);
    number = number / base;
    if (number != 0) {
        printInt(number);
    }
    System.out.print(conversion.charAt(digit));
}
```

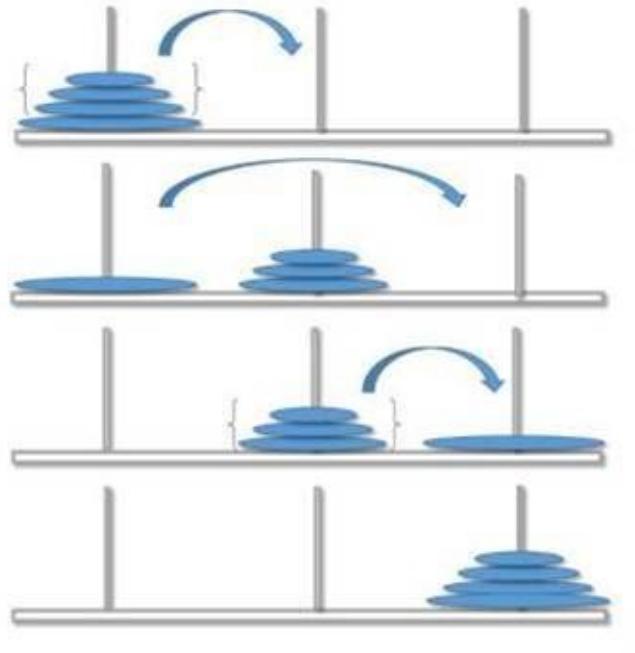
Analysis:

- Base value is provided along with the number in the function parameter.
- Remainder of the number is calculated and stored in digit.
- If the number is greater than base then, number divided by base is passed as an argument to the printInt() method recursively.
- Number will be printed with higher order first then the lower order digits.

Time Complexity is **O(N)**

Tower of Hanoi

Problem: The **Tower of Hanoi**, we are given three rods and N number of disks, initially all the disks are added to first rod (the leftmost one) is in decreasing size order. The objective is to transfer the entire stack of disks from first tower to third tower (the rightmost one), moving only one disk at a time and never a larger one onto a smaller.



Example 1.50:

```

public static void towerOfHanoi(int num, char src, char dst, char temp) {
    if (num < 1) {
        return;
    }

    towerOfHanoi(num - 1, src, temp, dst);
    System.out.println("Move " + num + " disk from peg " + src + " to peg " +
dst);
    towerOfHanoi(num - 1, temp, dst, src);
}

public static void main(String[] args) {
    int num = 4;
    System.out.println("The sequence of moves are :\n");
    towerOfHanoi(num, 'A', 'C', 'B');
}

```

Analysis: If we want to move N disks from source to destination, then we first move N-1 disks from source to temp, and then move the lowest Nth disk from source to destination. Then it will move N-1 disks from temp to destination.

Greatest common divisor (GCD)

Problem: Find the greatest common divisor.

Example 1.51:

```
public static int GCD(int m, int n) {  
    if (m < n) {  
        return (GCD(n, m));  
    }  
  
    if (m % n == 0) {  
        return (n);  
    }  
    return (GCD(n, m % n));  
}
```

Analysis: Euclid's algorithm is used to find gcd. $\text{GCD}(n, m) == \text{GCD}(m, n \bmod m)$.

Fibonacci number

Problem: Given N, find the Nth number in the Fibonacci series.

Example 1.52:

```
public static int fibonacci(int n) {  
    if (n <= 1) {  
        return n;  
    }  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

Analysis: Fibonacci numbers are calculated by adding sum of the previous two numbers.

Note: - There is an inefficiency in the solution we will look better solution in coming chapters.

All permutations of an integer list

Problem: Generate all permutations of an integer array.

Example 1.53:

```
public static void permutation(int[] arr, int i, int length) {  
    if (length == i) {  
        printArray(arr, length);  
        return;  
    }  
    int j = i;  
  
    for (j = i; j < length; j++) {  
        swap(arr, i, j);  
        permutation(arr, i + 1, length);  
        swap(arr, i, j);  
    }  
    return;  
}  
  
public static void main(String[] args) {  
    int[] arr = new int[5];  
    for (int i = 0; i < 5; i++) {  
        arr[i] = i;  
    }  
    permutation(arr, 0, 5);  
}
```

Analysis: In permutation method at each recursive call number at index, “i” is swapped with all the numbers that are right of it. Since the number is swapped with all the numbers in its right one by one it will produce all the permutation possible.

Binary search using recursion

Problem: Given an array of integers in increasing order, you need to find if some particular value is present in array using recursion.

Example 1.54: Search a value in an increasing order sorted list of integers.

```
// Binary Search Algorithm - Recursive
public static int BinarySearchRecursive(int[] arr, int low, int high, int value) {
    if(low > high)
        return -1;
    int mid = (low + high) / 2;

    if (arr[mid] == value) {
        return mid;
    } else if (arr[mid] < value) {
        return BinarySearchRecursive(arr, mid + 1, high, value);
    } else {
        return BinarySearchRecursive(arr, low, mid - 1, value);
    }
}

/* Testing code */
public static void main(String[] args) {
    int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    System.out.println(BinarySearchRecursive(arr, 0, arr.length - 1, 6));
    System.out.println(BinarySearchRecursive(arr, 0, arr.length - 1, 16));
}
```

Analysis: Similar iterative solution we have already seen. Now let us look into the recursive solution of the same problem. In this solution, we are diving the search space into half and discarding the rest. This solution is very efficient as at each step we are rejecting half the search space/ list.

Exercise

1. True or false
 - a. $5n + 10n^2 = O(n^2)$
 - b. $n \log n + 4n = O(n)$
 - c. $\log(n^2) + 4\log(\log n) = O(\log n)$
 - d. $12n^{1/2} + 3 = O(n^2)$
 - e. $3^n + 11n^2 + n^{20} = O(2^n)$
2. What is the best-case runtime complexity of searching an Array?
3. What is the average-case runtime complexity of searching an Array?
4. Given array of positive numbers, you need to find the maximum sum under constraint that no two elements should be adjacent.

CHAPTER 2: APPROACH TO SOLVE ALGORITHM DESIGN PROBLEMS

Introduction

Theoretical knowledge of the algorithm is essential, yet it is not sufficient. When an interviewer asks to develop a program in an interview, than interviewee should follow our five-step approach to solve it. Master this approach and you will perform better than most of the candidates in interviews.

Five steps for solving algorithm design questions are: 1. Constraints 2. Ideas Generation 3. Complexities 4. Coding 5. Testing

Constraints

Solving a technical question is not just about knowing the algorithms and designing a good software system. The interviewer wants to know you approach towards any given problem. Many people make mistakes, as they do not ask clarifying questions about a given problem. They assume many things simultaneously and begin working with that. There is lot of data that is missing that you need to collect from your interviewer before beginning to solve a problem.

In this step, you will capture all the constraints about the problem. We should never try to solve a problem that is not completely defined. Interview questions are not like exam paper where all the details about a problem are well defined. In the interview, the interviewer actually expects you to ask questions and clarify the problem.

For example, when the problem statement says that write an algorithm to sort numbers.

1. The first information you need to capture is what kind of data is provided.
Let us assume interviewer respond with the answer Integer.
2. The second information that you need to know what is the size of data.
Your algorithm differs if the input data size if 100 integers or 1 billion integers.

Basic guideline for the Constraints for an array of numbers:

1. How many numbers of elements are there in an array?

2. What is the range of value in each element? What is the min and max value?
3. What is the kind of data in each element? Is it an integer or a floating point?
4. Does the array contain unique data or not?

Basic guideline for the Constraints for an array of string:

1. How many numbers of elements are there in the array?

2. What is the length of each string? What is the min and max length?
3. Does the array contain unique data or not?

Basic guideline for the Constraints for a Graph

1. How many nodes are there in the graph?
2. How many edges are there in the graph?
3. Is it a weighted graph? What is the range of weights?
4. Is the graph directed or undirected?
5. Is there is a loop in the graph?
6. Is there negative sum loop in the graph?
7. Does the graph have self-loops?

We will see this in graph chapter that depending upon the constraints the algorithm applied changes and so is the complexity of the solution.

Idea Generation

We will cover a lot of theoretical knowledge in this book. It is impossible to cover all the questions as new ones are created every day. Therefore, you should know how to handle new problems. Even if you know the solution of a problem asked by the interviewer then also you need to have a discussion with the interviewer and try to reach to the solution. You need to analyze the problem also because the interviewer may modify a question a little bit so the approach to solve it will vary.

How to solve an unseen problem? The solution to this problem is that you need to do a lot of practice and the more you will practise the more you will be able to solve any unseen question, which come before you. When you have solved enough problems, you will be able to see a pattern in the questions and will be able to solve unseen problems easily.

Following is the strategy that you need to follow to solve an unknown problem:

1. Try to simplify the task in hand.
2. Try a few examples
3. Think of a suitable data-structure.
4. Think about similar problems that you have already solved.

Try to simplify the task in hand

Let us look into the following problem: Husbands and their wives are standing in random in a line. They have been numbered, for husbands H₁, H₂, H₃ and so on. Their corresponding wives have been numbered, W₁, W₂, W₃ and so on. You need to arrange them so that H₁ will stand first, followed by W₁, then H₂ followed by W₂ and so on.

At the first look, it looks difficult, but it is a simple problem. Try to find a relation of the final position.

$$P(H_i) = i^2 - 1, P(W_i) = i^2$$

For rest of the algorithm we are leaving you to do something like Insertion-Sort and you are done.

Try a few examples

In the above problem if you have tried it with some example for 3 husband-wife pair then you will reach to the same formula that we have shown in the previous section. Sometime applying some more examples will help you to solve the

problem.

Think of a suitable data-structure

For some problems, it is straightforward to choose which data structure will be most suitable. For example, if we have a problem finding min / max of some given value, then probably heap is the data structure we are looking for. We have seen a number of data structure throughout this book. We have to figure out which data-structure will suite our need.

Let us look into a problem: We are given a stream of data, at any time we can be asked to tell the median value of the data and maybe we can be asked to pop median data.

We can think about some sort of tree, may be balanced tree where the root is the median. Wait! It is not so easy to make sure the tree root to be a median.

A heap can give us minimum or maximum so we cannot get the desired result from it too. However, what if we use two heap a max heap and a min heap. The smaller values will go to max heap and the bigger values will go to min heap. In addition, we can keep the count of how many elements are there in the heap. The rest of the algorithm you can think yourself.

For every unseen problem think about the data structures, you know and may be one of them or some combination of them will solve your problem.

Think about similar problems you have already solved. Let us suppose you are given, two linked list head pointer and they meet at some point and need to find the point of intersection. However, in place of the end of both the linked list to be a null pointer, there is a loop.

You know how to find intersection point of two intersecting linked-list, you know how to find if a linked list have a loop (three-pointer solution). Therefore, you can apply both of these solutions to find the solution of the problem in hand.

Complexities

Solving a problem is not just finding a correct solution. The solution should be fast and should have reasonable memory requirement. You have already read about Big-O notation in the previous chapters. You should be able to do Big-O analysis. In case you think the solution you have provided is not optimal and maybe there is a better solution, then try to figure it out.

Most interviewers expect that you should be able to find the time and Space Complexity of the algorithms. You should be able to compute the time and Space Complexity instantly. Whenever you are solving any problem, you should find the complexity associated with it from this you would be able to choose the best solutions. In some problems there is some trade-offs between Space and Time Complexity, so you should know these trade-offs. Sometime taking some bit more space saves a lot of time and make your algorithm much faster.

Coding

At this point, you have already captured all the constraints of the problem, proposed few solutions, evaluated the complexities of the various solutions and picked the solution to do final coding. Never ever, jump into coding before discussing constraints, Idea generation and complexity with the interviewer.

We are accustomed to coding in an IDE like visual studio. So many people struggle when asked to write code on a whiteboard or some blank sheet. Therefore, we should have a little practice to the coding on a sheet of paper. You should think before coding because there is no back button in sheet of paper. Always try to write modular code. Small functions need to be created so that the code is clean and managed. If there is a swap function so just use this function and tell the interviewer that you will write it later. Everybody knows that you can write swap code.

Testing

Once the code is written, you are not yet done. It is most important that you test your code with several small test cases. It shows that you understand the importance of testing. It also gives confidence to your interviewer that you are not going to write a buggy code. Once you are done with coding, you go through your code line-by-line with some small test cases. This is just to make sure that your code is working as it is supposed to work.

You should test few test cases.

Normal test cases: These are the positive test cases, which contain the most common scenario, and focus is on the working of the base logic of the code. For example, if we are solving some problems for linked list, then this test may contain, what will happen when a linked list with 3 or 4 nodes is given as input. These test cases you should always contemplate before stating that the code is done.

Edge cases: These are the test cases, which are designed to test the boundaries of the code. For the same linked list algorithm, edge cases may be created to test how the code behaves when an empty list is passed or just one node is passed. Edge cases may help to make your code more robust. Just few checks need to be added to the code to take care of these conditions.

Load testing: In this kind of test, your code will be tested with a huge data. This will allow us to test if your code is slow or too much memory intensive.

Always follow these five steps never jump to coding before doing constraint analysis, idea generation, and Complexity Analysis: At least, never miss the testing phase.

Example

Let us suppose the interviewer ask you to give a best sorting algorithm. Some interviewee will directly jump to Quick-Sort **O(nlogn)**. Oops, mistake! You need to ask many questions before beginning to solve this problem.

Questions 1: What is the kind of data? Are they integers?

Answer: Yes, they are integers.

Questions 2: How much data are we going to sort?

Answer: May be thousands.

Questions 3: What exactly is this data about?

Answer: They store a person's age

Questions 4: What kind of data-structure is used to hold this data?

Answer: Data are given in the form of some list Questions 5: Can we modify the given data-structure? In addition, many, many more...?

Answer: No, you cannot modify the data structure provided

Ok, from the first answer, we will deduce that the data is integer. The data is not very big it just contains a few thousand entries. The third answer is interesting; from this, we deduce that the range of data is 1-150. Data is provided in a list. From fifth answer we deduce that we have to create our own data structure and we cannot modify the array provided. So finally, we conclude, we can just use bucket sort to sort the data. The range is just 1-150 so we need just 151-capacity integral list. Data is under thousands so we do not have to worry about data overflow and we get the solution in linear time **O(N)**.

Note: We will read sorting in the coming chapters.

Summary

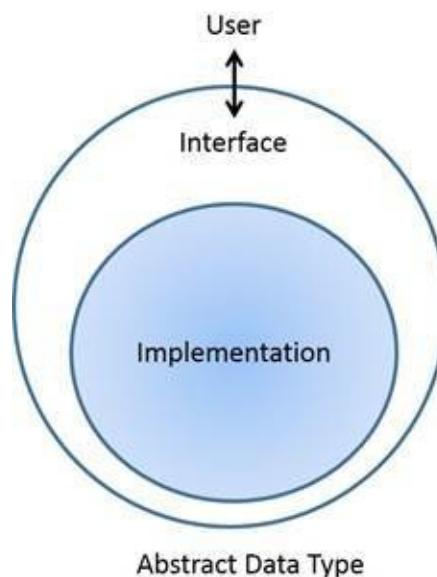
At this point, you know the process of handling unseen problems very well. In the coming chapter we will be looking into a lot of various data structures and the problems they solve. It may be possible that the user is not able to understand some portion of this chapter as knowledge of rest of the book is needed, so they can read this chapter again after they have read the rest of the data structures portion. A huge number of problems are solved in this book. However, it is recommended that first try to solve them by yourself, and then look for the solution. Always think about the complexity of the problem. In the interview interaction is the key to get problem described completely and discuss your approach with the interviewer.

CHAPTER 3: ABSTRACT DATA TYPE & JAVA COLLECTIONS

Abstract data type (ADT)

An abstract data type (ADT) is a logical description of the data and the operations that are allowed on it. ADT is defined as a user point of view of a data. ADT concerns about the possible values of the data and the interface exposed by it. ADT does not concern about the actual implementation of the data structure.

For example, a user wants to store some integers and find their mean value. ADT for this data structure will support two function one for adding integers and other to get mean value. ADT for this data structure does not talk about how exactly it will be implemented.



Data-Structure

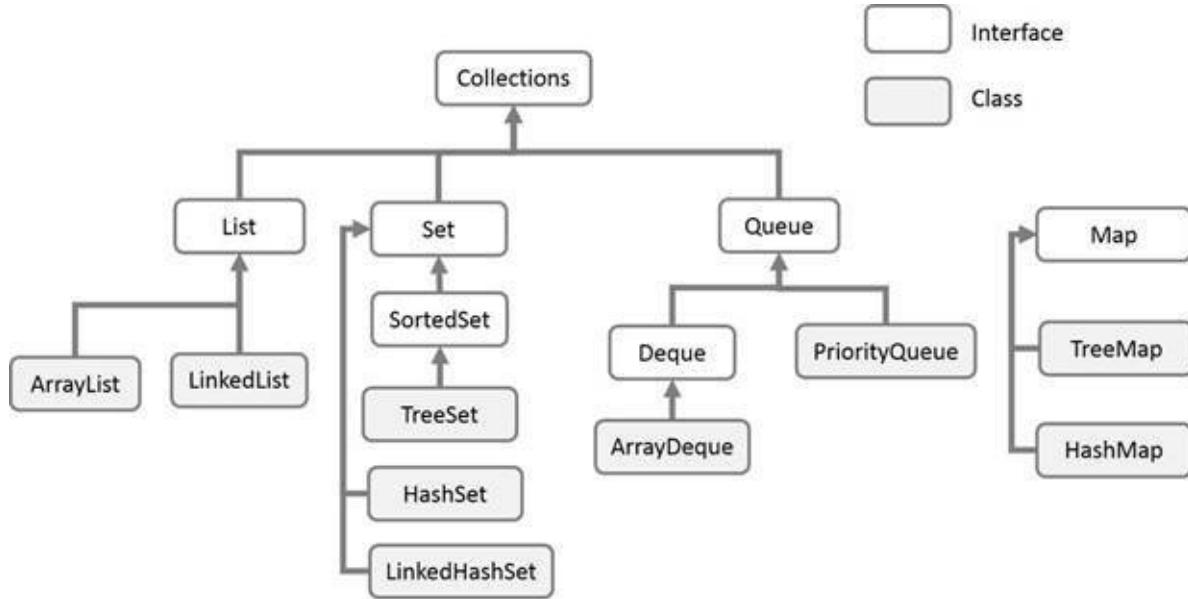
Data structures are concrete representations of data and is defined as a programmer point of view of data. Data-structure represents how data will be stored in memory. All data-structures have their own pros and cons. Depending upon the type of problem we choose a data-structure that is best suited for it.

For example, we can store data in an array, a linked-list, stack, queue, tree, etc.

Note: - In this chapter, we will be studying various data structures and their API. So that the user can use them without knowing their internal implementation.

JAVA Collection Framework

JAVA programming language provides a JAVA Collection Framework, which is a set of high quality, high performance & reusable data-structures and algorithms.



The following advantages of using a JAVA collection framework:

1. Programmers do not have to implement basic data structures and algorithms repeatedly. Thereby it prevents the reinvention the wheel. Thus, the programmer can devote more effort in business logic
2. The JAVA Collection Framework code is well-tested, high quality, high performance code. Using them increase quality of the programs.
3. Development cost is reduced as basic data structures and algorithms are implemented in Collections framework are reused.
4. Easy to review and understand programs written by other developers as most of Java developers uses the Collection framework. In addition, collection framework is well documented.

Array

Array represents a collection of multiple elements of the same datatypes.

Array ADT Operations

Below is the API of array:

1. Adds an element at kth position. Value can be stored in array at Kth position in **O(1)** constant time. We just need to store value at arr[k].
2. Reading the value stored at kth position. Accessing the value stored at some index in array is also **O(1)** constant time. We just need to read value stored at arr[k].
3. Substitution of value stored in kth position with a new value. Time complexity: **O(1)** constant time.

Example 3.1:

```
public class ArrayDemo {  
    public static void main(String[] args) {  
        int[] arr = new int[10];  
        for (int i = 0; i < 10; i++)  
        {  
            arr[i] = i;  
        }  
    }  
}
```

JAVA standard arrays are of fixed length. Sometime we do not know how much memory we need so we create a bigger size array. There by wasting space. If an array is already full and we want to add more values to it than we need to create a new array, which have sufficient space and then copy the old array to the new array. To avoid this manual reallocation and copy we can use ArrayList of JAVA Collection framework or Linked Lists to store data.

ArrayList implementation in JAVA Collections

ArrayList<E> in JAVA Collections is a data structure which implements List<E>

interface which means that it can have duplicate elements in it. ArrayList is an implementation as dynamic array that can grow or shrink as needed. (Internally array is used when it is full a bigger array is allocated and the old array values are copied to it.)

Example 3.2:

```
import java.util.ArrayList;

public class ArrayListDemo {
    public static void main(String[] args) {
        ArrayList<Integer> al = new ArrayList<Integer>();

        al.add(1); // add 1 to the end of the list
        al.add(2); // add 2 to the end of the list
        System.out.println("Contents of Array: " + al);
        System.out.println("Array Size: " + al.size());
        System.out.println("Array IsEmpty: " + al.isEmpty());
        al.remove(al.size() -1); // last element of array is removed.
        al.removeAll(al); // all the elements of array are removed.
        System.out.println("Array IsEmpty: " + al.isEmpty());
    }
}
```

Output:

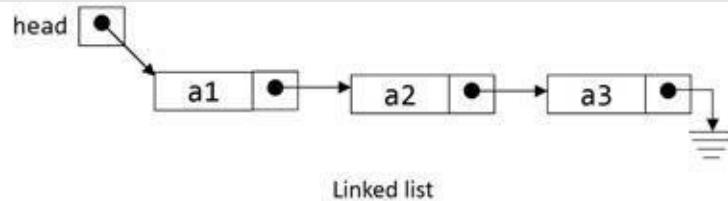
Contents of Array: [1, 2]

Array Size: 2

Array IsEmpty: false

Array IsEmpty: true

Linked List



Linked lists are dynamic data structure and memory is allocated at run time. The concept of linked list is not to store data contiguously. Nodes of linked list contains link that point to the next elements in the list.

Performance wise linked lists are slower than arrays because there is no direct access to linked list elements. Linked list is a useful data structure when we do not know the number of elements to be stored ahead of time. There are many types of linked list: linear, circular, doubly, doubly circular etc.

Linked list ADT Operations

Below is the API of Linked list.

Insert(k): adds k to the start of the list

Insert an element at the start of the list. Just create a new element and move pointers. So that this new element becomes the first element of the list. This operation will take **O(1)** constant time.

Delete(): delete element at the start of the list

Delete an element at the start of the list. We just need to move one pointer. This operation will also take **O(1)** constant time.

PrintList(): display all the elements of the list.

Start with the first element and then follow the pointers. This operation will take **O(N)** time.

Find(k): find the position of element with value k

Start with the first element and follow the pointer until we get the value we are looking for or reach the end of the list. This operation will take **O(N)** time.

Note: binary search does not work on linked lists.

FindKth(k): find element at position k

Start from the first element and follow the links until you reach the kth element. This operation will take **O(N)** time.

IsEmpty(): check if the number of elements in the list are zero.

Just check the head pointer of the list, if it is Null then the list is empty otherwise not empty. This operation will take **O(1)** time.

LinkedList implementation in JAVA Collections

LinkedList<E> in by JAVA Collections is a data structure that also implements List<E> interface.

Example 3.3:

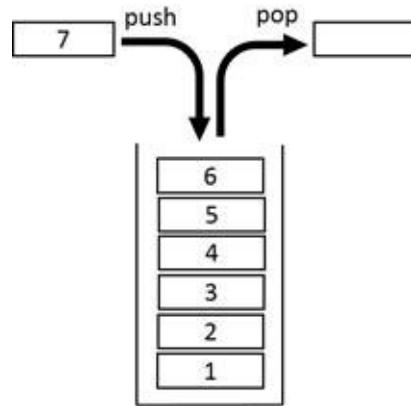
```
import java.util.LinkedList;
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.addFirst(2); // 8 is added to the list
        ll.addLast(10); // 9 is added to last of the list.
        ll.addFirst(1); // 7 is added to first of the list.
        ll.addLast(11); // 20 is added to last of the list
        System.out.println("Contents of Linked List: " + ll);
        ll.removeFirst();
        ll.removeLast();
        System.out.println("Contents of Linked List: " + ll);
    }
}
```

Output:

Contents of Linked List: [1, 2, 10, 11]

Contents of Linked List: [2, 10]

Stack



Stack is a special kind of data structure that follows Last-In-First-Out (LIFO) strategy. This means that the element that is added last will be the first to be removed.

The various applications of stack are:

1. Recursion: recursive calls are implemented using system stack.
2. Postfix evaluation of expression.
3. Backtracking implemented using stack.
4. Depth-first search of trees and graphs.
5. Converting a decimal number into a binary number etc.

Stack ADT Operations

Push(k): Adds value k to the top of the stack

Pop(): Remove element from the top of the stack and return its value.

Top(): Returns the value of the element at the top of the stack

Size(): Returns the number of elements in the stack

IsEmpty(): determines whether the stack is empty. It return true if the stack is empty otherwise return false.

Note: All the above stack operations are implemented in **O(1)** Time Complexity.

Stack implementation in JAVA Collection

Stack is implemented by calling push and pop methods of Stack <T> class.

Example 3.4:

```
public class StackDemo {  
    public static void main(String[] args) {  
        Stack<Integer> stack = new Stack<Integer>();  
        int temp;  
        stack.push(1); stack.push(2); stack.push(3); System.out.println("Stack : "+stack);  
        System.out.println("Stack size : "+stack.size());  
        System.out.println("Stack pop : "+stack.pop());  
        System.out.println("Stack top : "+stack.peek());  
        System.out.println("Stack isEmpty : "+stack.isEmpty());  
    }  
}
```

Output:

```
Stack : [1, 2, 3]  
Stack size : 3  
Stack pop : 3  
Stack top : 2  
Stack isEmpty : false
```

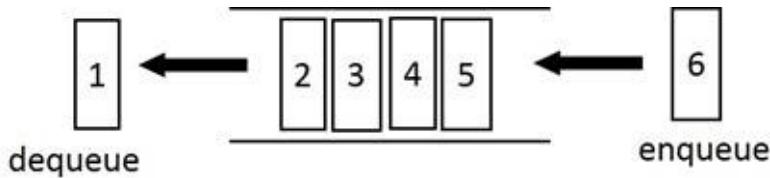
Stack is also implemented by calling push and pop methods of ArrayDeque<T> class.

JDK provides both ArrayDeque<T> and Stack<T>. We can use both of these classes. But there are some advantages of ArrayDeque<T>.

1. First reason is that Stack<T> does not drive from Collection interface.
2. Second Stack<T> drives from Vector<T> so random access is possible so it brakes abstraction of a stack.
3. Third ArrayDeque is more efficient as compared to Stack<T>.

Queue

A queue is a First-In-First-Out (FIFO) kind of data structure. The element that is added to the queue first will be the first to be removed and so on.



Queue has the following application uses:

1. Access to shared resources (e.g., printer)
2. Multiprogramming
3. Message queue
4. BFS, breadth first traversal of graph or tree are implemented using queue.

Queue ADT Operations:

Add(K): Adds a new element k to the back of the queue.

Remove(): Removes an element from the front of the queue and return its value.

Front(): Returns the value of the element at the front of the queue.

Size(): Returns the number of elements inside the queue.

IsEmpty(): Returns 1 if the queue is empty otherwise returns 0

Note: All the above queue operations are implemented in **O(1)** Time Complexity.

Queue implementation in JAVA Collection

ArrayDeque<T> is the class implementation of doubly ended queue. If we use add(), remove() and peek () it will behave as a queue. (Moreover, if we use push(), pop() and peekLast() it behave as a stack.)

Example 3.5:

```
import java.util.ArrayDeque;
```

```
public class QueueDemo {
```

```
public static void main(String[] args) {  
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();  
    que.add(1); que.add(2); que.add(3); System.out.println("Queue : "+que);  
    System.out.println("Queue size : "+que.size());  
    System.out.println("Queue peek : "+que.peek());  
    System.out.println("Queue remove : "+que.remove());  
    System.out.println("Queue isEmpty : "+que.isEmpty());  
}  
}
```

Output:

```
Queue : [1, 2, 3]  
Queue size : 3  
Queue peek : 1  
Queue remove : 1  
Queue isEmpty : false
```

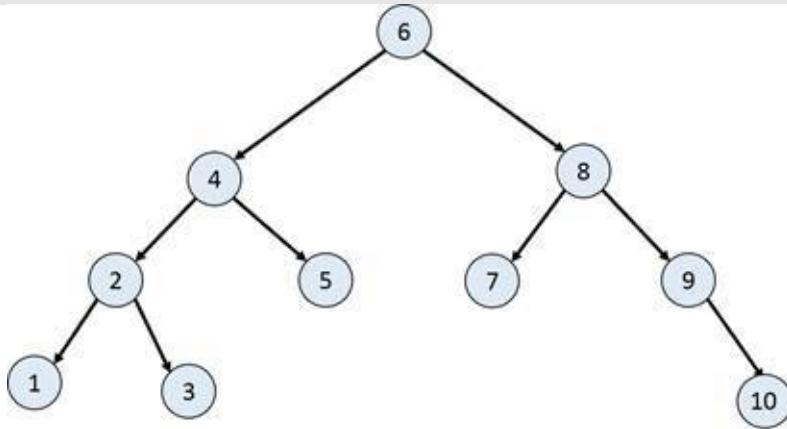
Tree

Tree is a hierarchical data structure. The top element of a tree is called the root of the tree. Except the root element, every element in a tree has a parent element, and zero or more child elements. The tree is the most useful data structure when you have hierarchical information to store.

Binary Tree

A binary tree is a type of tree in which each node has at most two children (0, 1 or 2) which are referred as left child and right child.

Binary Search Trees (BST)



A binary search tree (BST) is a binary tree on which nodes are ordered in the following way:

1. The key in the left subtree is less than or equal to the key in its parent node.
2. The key in the right subtree is greater than the key in its parent node.

Binary Search Tree ADT Operations

Insert(k): Inserts an element k into the tree.

Delete(k): Deletes an element k from the tree.

Search(k): Searches a particular value k into the tree if it is present or not.

FindMax(): Finds the maximum value stored in the tree.

FindMin(): Finds the minimum value stored in the tree.

The Average time complexity of all the above operations on a binary search tree is $O(\log n)$, the case when the tree is balanced. The worst-case time complexity is $O(n)$ when the tree is not balanced.

There are few binary search trees, which always keep themselves balanced. Most important among them are Red-Black Tree (RB-Tree) and AVL tree. Ordered dictionary in collections is implemented using RB-Tree.

TreeSet implementation in JAVA Collections

TreeSet<> is a class which implements Set<> interface which means that it stores

only unique elements. TreeSet<> is implemented using a red-black balanced binary search tree in JAVA Collections. Since TreeSet<> is implemented using a binary search tree its elements are stored in sequential order.

Example 3.6:

```
import java.util.TreeSet;
public class TreeSetDemo {
    public static void main(String[] args) {
        // Create a tree set.
        TreeSet<String> ts = new TreeSet<String>();
        // Add elements to the tree set.
        ts.add("India");
        ts.add("USA");
        ts.add("Brazil");
        System.out.println(ts);
    }
}
```

Output:

```
[Brazil, India, USA]
```

Note: TreeSet is implemented using a binary search tree so add, remove, and contains methods have logarithmic time complexity $O(\log(n))$, where n is the number of elements in the set.

TreeMap implementation in JAVA Collection

A Map<> is an interface that maps keys to values. A TreeMap<> is an implementation of Map<> and is implemented using red-black balanced binary tree so the key value pairs are stored in sorted order.

Example 3.7:

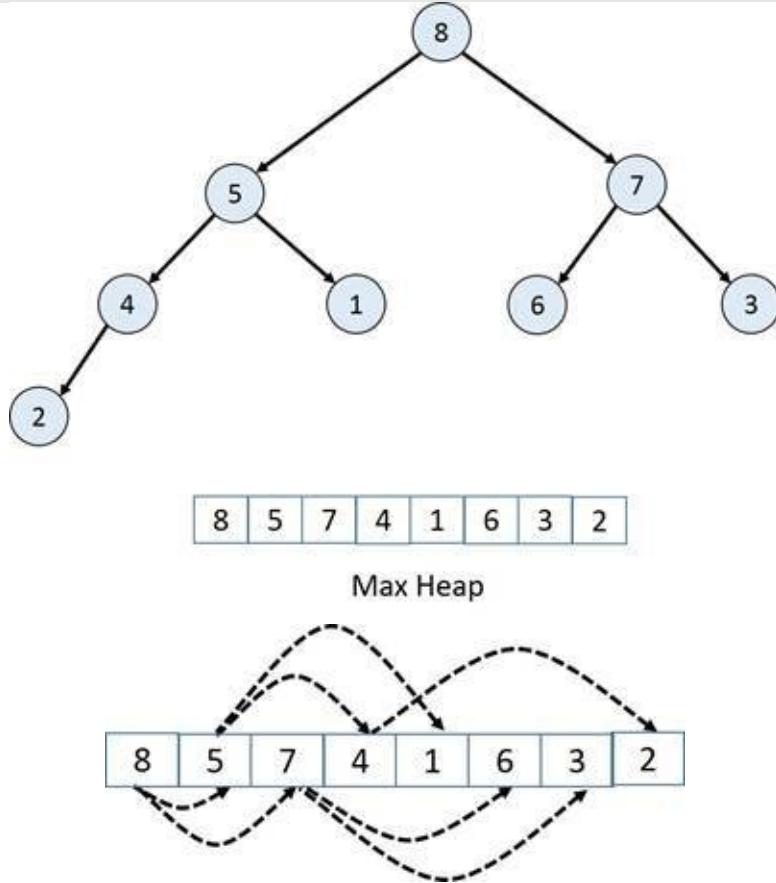
```
import java.util.TreeMap;
public class TreeMapDemo {
    public static void main(String[] args) {
        // create a tree map.
        TreeMap<String, Integer> tm = new TreeMap<String, Integer>();
        // Put elements into the map
    }
}
```

```
tm.put("Mason", new Integer(55));
tm.put("Jacob", new Integer(77));
tm.put("William", new Integer(99));
tm.put("Emma", new Integer(65));
System.out.println("Students count :: " + tm.size());
for(String key : tm.keySet()){
    System.out.println(key + " score marks :" + tm.get(key));
}
System.out.println("Emma score present::" + tm.containsKey("Emma"));
System.out.println("John score present:: " + tm.containsKey("John"));
tm.remove("Emma");
System.out.println("Emma score present::" + tm.containsKey("Emma"));
}
}
```

Output:

```
Students count :: 4
Emma score marks :65
Jacob score marks :77
Mason score marks :55
William score marks :99
Emma score present::true
John score present:: false
Emma score present::false
```

Priority Queue (Heap)



Priority queue is a special kind of queue in which elements are extracted out of it in special order according to their priority. Priority queue is implemented using a binary heap data structure. In a heap, the records are stored in an array. Each node in the heap follows the same rule that the parent value is greater (or smaller) than its children.

There are two types of the heap data structure:

1. Max heap: Value of each node should be greater than or equal to the values of each of its children.
2. Min heap: Value of each node should be smaller than or equal to the values of each of its children.

A heap is a useful data structure when you want to get max/min value one by one from data. Heap-Sort uses heap to sort data in increasing or decreasing order.

Heap ADT Operations

Insert(k) - Adds a new element k to the heap. Time Complexity of this operation is $O(\log(n))$

Remove() - Extracts max for max heap case (or min for min heap case). Time Complexity of this operation is $O(\log(n))$

Heapify() – Converts an array of numbers into a heap. This operation has a Time Complexity **O(n)**

PriorityQueue implementation in JAVA Collection

Min heap implementation of Priority Queue

Example 3.8:

```
import java.util.PriorityQueue;
public class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
        int[] arr = {1,2,10,8,7,3,4,6,5,9};
        for(int i: arr){
            pq.add(i);
        }
        System.out.println("Printing Priority Queue Heap : " + pq);

        System.out.print("remove elements of Priority Queue ::");
        while(pq.isEmpty() == false){
            System.out.print(" " + pq.remove());
        }
    }
}
```

Output

Printing Priority Queue Heap : [1, 2, 3, 5, 7, 10, 4, 8, 6, 9]

Dequeue elements of Priority Queue :: 1 2 3 4 5 6 7 8 9 10

Max heap implementation of Priority Queue

We just need to change collection order to make max heap from PriorityQueue<> collection.

Example 3.9:

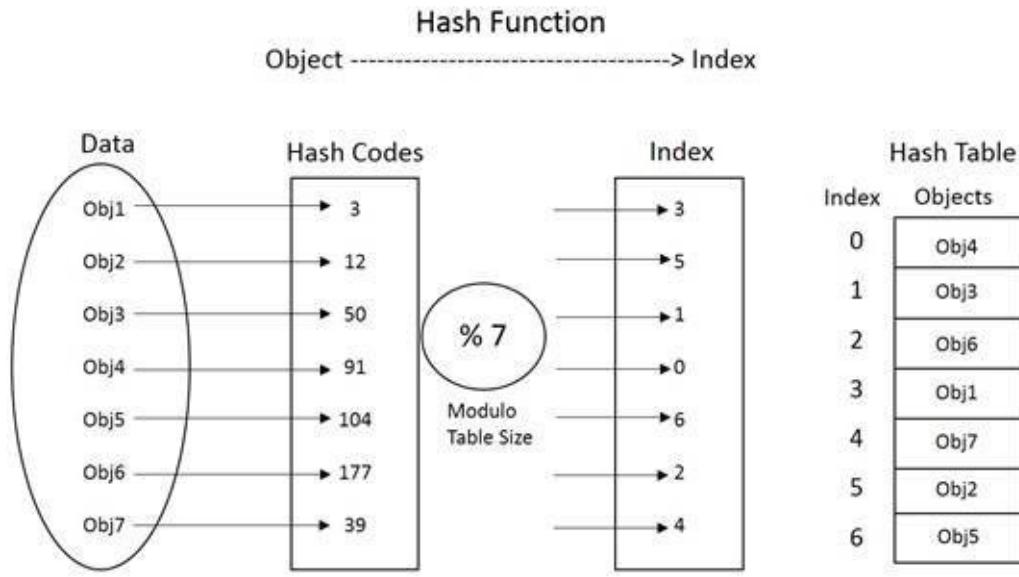
```
PriorityQueue<Integer> pq = new PriorityQueue<Integer>()
(Collections.reverseOrder());
```

Output

Printing Priority Queue Heap : [10, 9, 4, 6, 8, 2, 3, 1, 5, 7]

Dequeue elements of Priority Queue :: 10 9 8 7 6 5 4 3 2 1

Hash-Table



A Hash-Table is a data structure that maps keys to values. Each position of the Hash-Table is called a slot. The Hash-Table uses a hash function to calculate an index of data in an array. We use the Hash-Table when the number of key-value pair actually stored are small relatively to the number of possible keys.

The process of storing data using a hash function is as follows:

1. Create an array of size M to store data, this array is called Hash-Table.
2. Find a hash code of a key by passing it through a hash function.
3. Take module of hash code by the size of Hash-Table to get the index where data will be stored.
4. Finally store this data in the designated index.

The process of searching data in Hash-Table using a hash function is as follows:

1. Find a hash code of the key which we are searching, by passing it through a hash function.
2. Take module of hash code by the size of Hash-Table to get the index of the table where data is stored.
3. Finally, retrieve the value from the designated index.

Hash-Table Abstract Data Type (ADT)

ADT of Hash-Table contains the following functions:

Insert(x): Add x to the data set.

Delete(x): Delete x from the data set.

Search(x): Searches x in data set.

The Hash-Table is a useful data structure for implementing dictionary. The average time to search for an element in a Hash-Table is **O(1)**.

HashSet implementation of JAVA Collections

HashSet <> is a class which implements Set<> interface which means that it store only unique elements. HashSet <> is implemented using a hash table. Since HashSet<> is implemented using a hash table so its elements are not stored in sequential order.

Example 3.10:

```
import java.util.HashSet;
public class HashSetDemo {
    public static void main(String[] args) {
        // Create a hash set.
        HashSet<String> hs = new HashSet<String>();
        // Add elements to the hash set.
        hs.add("India"); hs.add("USA"); hs.add("Brazil"); System.out.println(hs);
        System.out.println("Hash Table contains USA : " + hs.contains("USA"));
        System.out.println("Hash Table contains UK : " + hs.contains("UK"));
        hs.remove("USA"); System.out.println(hs); System.out.println("Hash Table
contains USA : " + hs.contains("USA"));
    }
}
```

Output

```
[USA, Brazil, India]
```

```
Hash Table contains USA : true
```

```
Hash Table contains UK : false
```

```
[Brazil, India]
```

```
Hash Table contains USA : false
```

LinkedHashSet implementation of JAVA Collections

LinkedHashSet <> is a class which implements Set<> interface which means that it store only unique elements. LinkedHashSet <> is implemented using a hash table and a linked list. Linked list is used to preserver order of elements based on insertion. Traversing the elements of the hash table is done in order of insertion.

Example 3.11

```
import java.util.HashSet; import java.util.LinkedHashSet;
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        // Create a hash set.
        HashSet<String> hs = new HashSet<String>();
        // Add elements to the hash set.
        hs.add("India"); hs.add("USA"); hs.add("Brazil");
        System.out.println("HashSet value:: " + hs);

        // Create a linked hash set.
        LinkedHashSet<String> lhs = new LinkedHashSet<String>();
        // Add elements to the linked hash set.
        lhs.add("India"); lhs.add("USA"); lhs.add("Brazil");
        System.out.println("LinkedHashSet value:: " + lhs);
    }
}
```

Output

```
HashSet value:: [USA, Brazil, India]
LinkedHashSet value:: [India, USA, Brazil]
```

Comparison of various Set classes.

	TreeSet	HashSet	LinkedHashSet
Storage	Red-Black Tree	Hash Table	Hash Table with Linked List.
Performance	Slower than HashSet, O(log(N))	Fastest, constant time	More expensive than HashSet, have to maintain linked list.
Order of	Increasing Order	No order guarantee	Order of insertion

HashMap implementation in JAVA Collection

A Map<> is a data structure that maps keys to values. A HashMap<> is an implementation of Map<> and is implemented using a hash table so the key value pairs are not stored in sorted order. Map<> does not allow duplicate keys but values can be duplicate.

Example 3.12

```
import java.util.HashMap;
public class HashMapDemo {
    public static void main(String[] args) {
        // Create a hash map.
        HashMap<String, Integer> hm = new HashMap<String, Integer>();

        // Put elements into the map
        hm.put("Mason", new Integer(55));
        hm.put("Jacob", new Integer(77));
        hm.put("William", new Integer(99));
        hm.put("Emma", new Integer(65));
        System.out.println("Students count :: " + hm.size());
        for(String key : hm.keySet()){
            System.out.println(key + " score marks :" + hm.get(key));
        }
        System.out.println("Emma score available ::" + hm.containsKey("Emma"));
        System.out.println("John score available:: " + hm.containsKey("John"));
    }
}
```

Output

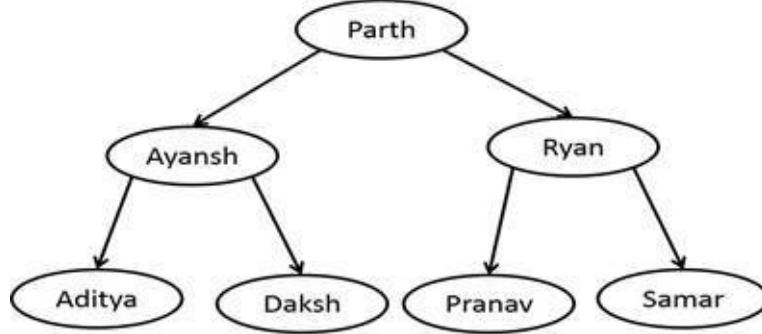
```
Mason score marks :55
William score marks :99
Emma score marks :65
Jacob score marks :77
Emma score available ::true
John score available :: false
```


Dictionary / Symbol Table

A symbol table is a mapping between a string (key) and a value, which can be of any data type. A value can be an integer such as occurrence count, dictionary meaning of a word and so on.

Binary Search Tree (BST) for Strings

Binary Search Tree (BST) is the simplest way to implement symbol table. Simple string compare function can be used to compare two strings. If all the keys are random, and the tree is balanced. Then on an average key lookup can be done in logarithmic time.



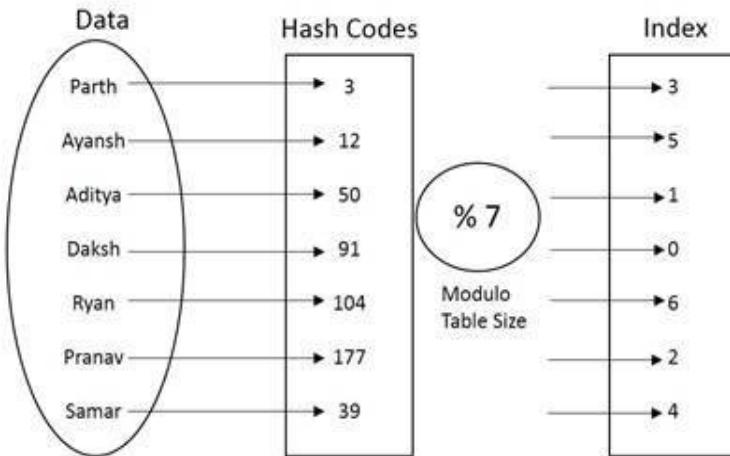
Binary Search Tree as Dictionary

Hash-Table

The Hash-Table is another data structure, which can be used for symbol table implementation. Below is the Hash-Table diagram, we can see that the name of a person is taken as key, and their meaning is the value of the search. The first key is converted into a hash code by passing it to appropriate hash function. Inside hash function, the size of Hash-Table is also passed, which is used to find the actual index where values will be stored. Finally, the value that is meaning of name is stored in the Hash-Table.

Hash Function

String -----> Index

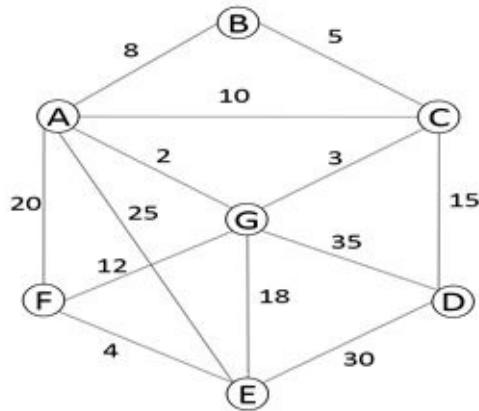


Hash Table

Index	String
0	Daksh meaning...
1	Aditya meaning...
2	Pranav meaning...
3	Parth meaning...
4	Samar meaning...
5	Ayansh meaning...
6	Ryan meaning...

Hash-Table has an excellent lookup of constant time.

Graphs

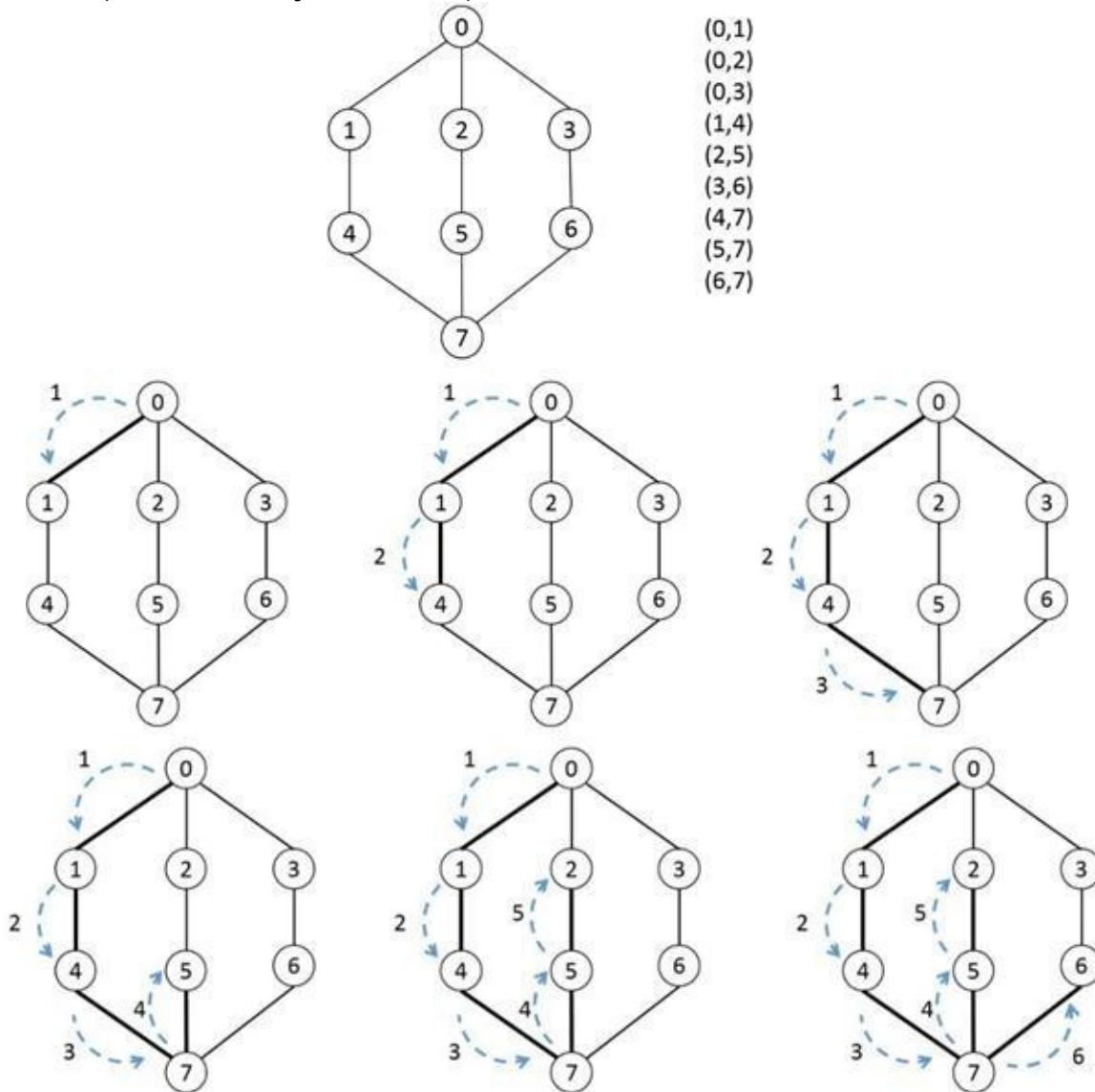


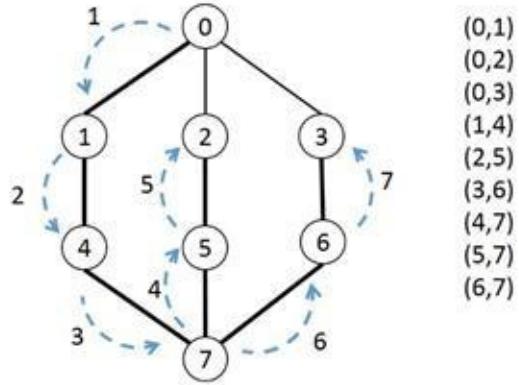
A graph is a data structure that represents a network. It contains a collection of nodes called vertices, and their connections called edges. An edge can be seen as a path between two nodes. These edges can be either directed or undirected. If a path is directed then you can move only in one direction, while in an undirected path you can move in both the directions. There can be a cost associated with the edges.

Graph Algorithms

Depth-First Search (DFS)

In the DFS algorithm, we start from starting point and go into depth of graph until we reach a dead end and then move up to parent node (Backtrack). In DFS, we use stack to get the next vertex to start a search. Alternatively, we can use recursion (which uses system stack) to do the same.

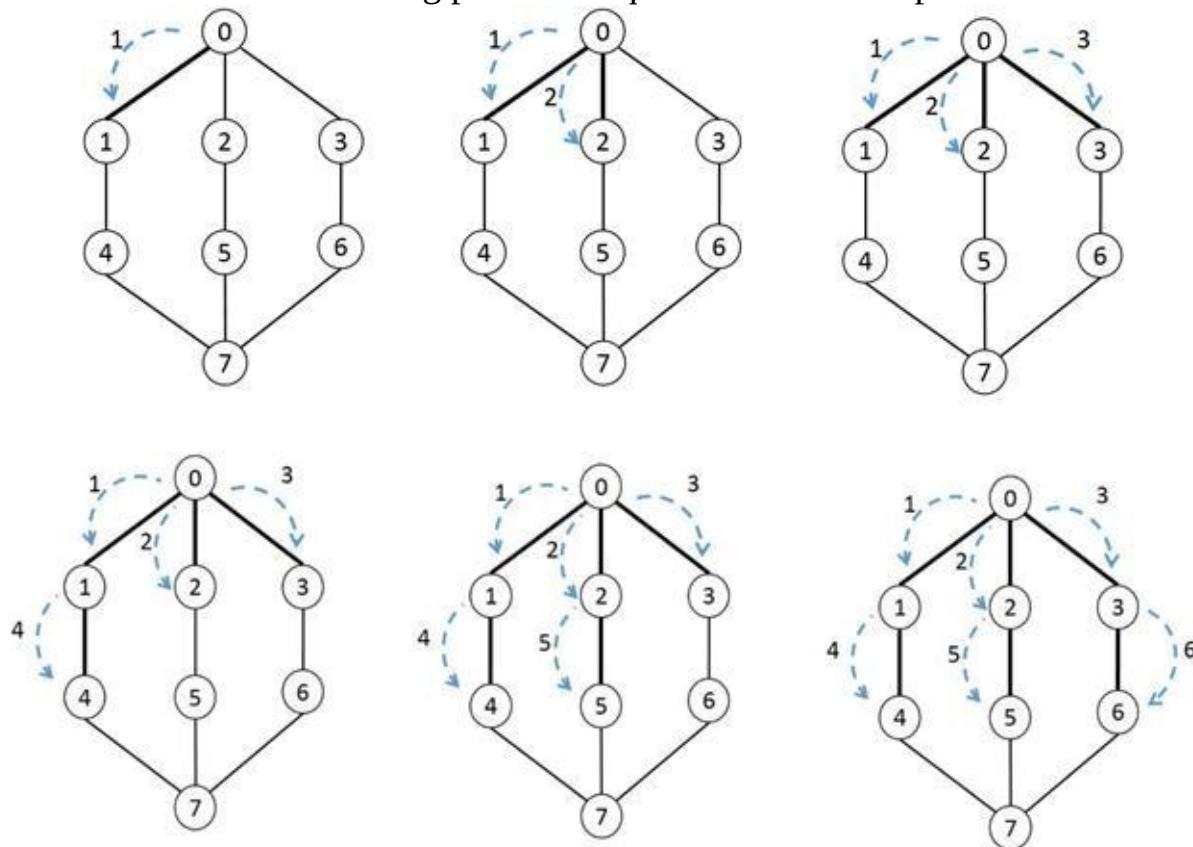


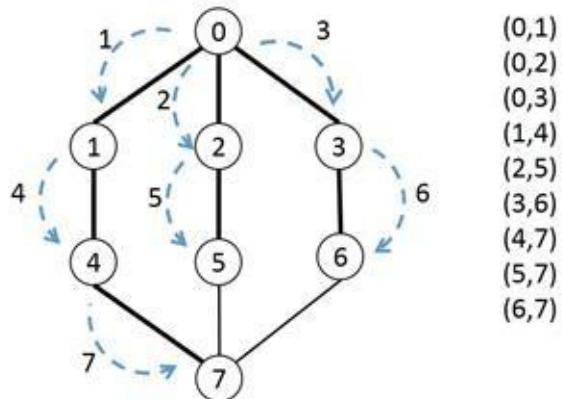


Depth First Traversal
0, 1, 4, 7, 5, 2, 6, 3

Breadth-First Search (BFS)

In BFS algorithm, a graph is traversed in layer-by-layer fashion. The graph is traversed closer to the starting point. The queue is used to implement BFS.

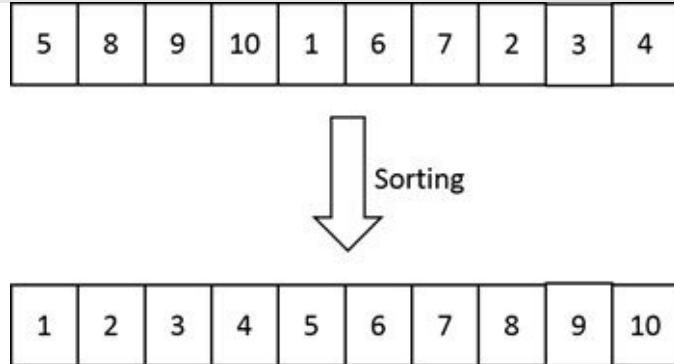




Breadth First Traversal

0, 1, 2, 3, 4, 5, 6, 7

Sorting Algorithms



Sorting is the process of placing elements from a collection into ascending or descending order.

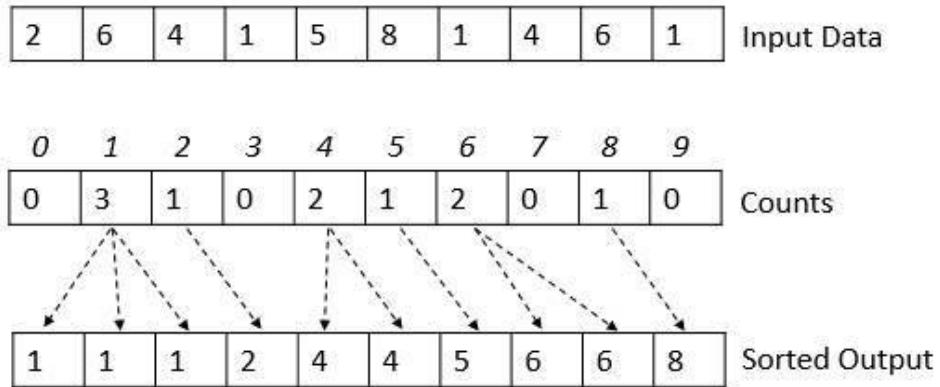
Sorting arranges data elements in order so that searching become easier.

There are good sorting functions available which does sorting in **O(nlogn)** time, so in this book when we need sorting we will use sort() function and will assume that the sorting is done in **O(nlogn)** time.

Counting Sort

Counting sort is the simplest and most efficient type of sorting. Counting sort has a strict requirement of a predefined range of data.

Sort how many people are there in which age group. We know that the age of people can vary between 1 and 130. We can directly store counts in an array of size 130.



If we know the range of input, then sorting can be done using counting in $O(n+k)$. Where n is the number of elements and k is the possible range, in above example it is 130.

End note

This chapter has provided a brief introduction of the various data structures, algorithms and their complexities. In the next chapters, we will look into all these data structure in details. If you know interface of a data structure, you can use it to solve other problems without knowing its internal implementation.

CHAPTER 4: SORTING

Introduction

Sorting is the process of arranging elements in ascending or descending order. For example, when we play cards, we sort cards according to their value so that we can find the required card easily.

When we go to some library, the books are arranged according to streams (Algorithm, Operating systems, Networking etc.). Sorting arranges data elements in order so that searching become easier. When books are arranged in proper indexing order, then it is easy to find a book we are looking for.

This chapter discusses algorithms for sorting an array of items. Understanding sorting algorithms are the first step towards understanding algorithm analysis. Many sorting algorithms are developed and analyzed.

Sorting algorithms like Bubble-Sort, Insertion-Sort and Selection-Sort are easy to implement and are suitable for the small input set. However, for large dataset they are slow. Sorting algorithms like Merge-Sort, Quick-Sort and Heap-Sort are some of the algorithms that are suitable for sorting large dataset. However, they are overkill if we want to sort a small dataset.

Some other algorithms are there to sort a huge data set that cannot be stored in memory completely, for which external sorting technique is developed.

Before we start a discussion of the various algorithms one by one. First, we should look at comparison function that is used to compare two values.

Less function will return true if value1 is less than value2 otherwise it will return false.

```
private boolean less(int value1, int value2) {  
    return value1 < value2;  
}
```

More function will return true if value1 is greater than value2 otherwise it will return false.

```
private boolean more(int value1, int value2) {  
    return value1 > value2;
```

}

The value in various sorting algorithms is compared using one of the above functions and it will be swapped depending upon the return value of these functions. If more() comparison function is used, then sorted output will be increasing in order and if less() is used then resulting output will be in descending order.

Type of Sorting

Internal Sorting: All the elements can be read into memory at the same time and sorting is performed in memory.

1. Selection-Sort
2. Insertion-Sort
3. Bubble-Sort
4. Quick-Sort
5. Merge-Sort

External Sorting: In this type of sorting, the dataset is so big that it is impossible to load the whole dataset into memory so sorting is done in chunks.

1. Merge-Sort

Bubble-Sort

Bubble-Sort is the slowest algorithm for sorting. It is easy to implement and used when data is small.

In Bubble-Sort, we compare each pair of adjacent values. We want to sort values in increasing order so if the second value is less than the first value then we swap these two values. Otherwise, we will go to the next pair. Thus, largest values bubble to the end of the array.

After the first pass, the largest value will be in the rightmost position. We will have N number of passes to get the array completely sorted.

First Pass

5	1	2	4	3	7	6	Swap
1	5	2	4	3	7	6	Swap
1	2	5	4	3	7	6	Swap
1	2	4	5	3	7	6	Swap
1	2	4	3	5	7	6	No Swap
1	2	4	3	5	7	6	Swap
1	2	4	3	5	6	7	

Example 4.1:

```
public class BubbleSort {  
  
    public static void sort(int[] arr) {  
        int size = arr.length;  
        int i, j, temp;  
        for (i = 0; i < (size - 1); i++) {  
            for (j = 0; j < size - i - 1; j++) {  
                if (more(arr[j], arr[j + 1])) {  
                    /* Swapping */  
                    temp = arr[j];  
                    arr[j] = arr[j + 1];  
                    arr[j + 1] = temp;  
                }  
            }  
        }  
    }  
}
```

```

        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
    }
}
}
}
}

public static void main(String[] args) {
int[] array = { 9, 1, 8, 2, 7, 3, 6, 4, 5 };
BubbleSort.sort(array);
for (int i = 0; i < array.length; i++) {
System.out.print(array[i] + " ");
}
}
}

}

```

Analysis:

- The outer loop represents the number of swaps that are done for comparison of data.
- The inner loop is actually used to do the comparison of data. At the end of each inner loop iteration, the largest value is moved to the end of the array. In the first iteration the largest value, in the second iteration the second largest and so on.
- more() function is used for comparison which means when the value of the first argument is greater than the value of the second argument then perform a swap. By this we are sorting in increasing order if we have, the less() function in place of more() then array will be sorted in decreasing order.

Complexity Analysis:

Each time the inner loop execute for $(n-1), (n-2), (n-3)\dots(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = n(n-1)/2$

Worst case performance	$O(n^2)$
Average case performance	$O(n^2)$
Space Complexity	$O(1)$ as we need only one temp variable
Stable Sorting	Yes

Modified (improved) Bubble-Sort

When there is, no more swap in one pass of the outer loop the array is already sorted. At this point, we should stop sorting. This sorting improvement in Bubble-Sort is extremely useful when we know that, except few elements rest of the array is already sorted.

Example 4.2:

```
public static void sort2(int[] arr) {  
    int size = arr.length;  
    int i, j, temp, swapped = 1;  
    for (i = 0; i < (size - 1) && swapped == 1; i++) {  
        swapped = 0;  
        for (j = 0; j < size - i - 1; j++) {  
            if (more(arr[j], arr[j + 1])) {  
                temp = arr[j];  
                arr[j] = arr[j + 1];  
                arr[j + 1] = temp;  
                swapped = 1;  
            }  
        }  
    }  
}
```

By applying this improvement, best-case performance of this algorithm is improved when an array is nearly sorted. In this case, we just need one single pass and the best-case complexity is **O(n)**

Complexity Analysis:

Worst case performance	$O(n^2)$
Average case performance	$O(n^2)$
Space Complexity	$O(1)$
Adaptive: When array is nearly sorted	$O(n)$
Stable Sorting	Yes

Insertion-Sort

Insertion-Sort Time Complexity is $O(n^2)$ which is same as Bubble-Sort but perform a bit better than it. It is the way we arrange our playing cards. We keep a sorted subarray. Each value is inserted into its proper position in the sorted subarray in the left of it.



5	6	2	4	7	3	1	Insert 5
5	6	2	4	7	3	1	Insert 6
2	5	6	4	7	3	1	Insert 2
2	4	5	6	7	3	1	Insert 4
2	4	5	6	7	3	1	Insert 7
2	3	4	5	6	7	1	Insert 3
1	2	3	4	5	6	7	Insert 1

Example 4.3:

```
public class InsertionSort {  
    private static boolean more(int value1, int value2) {  
        return value1 > value2;  
    }  
  
    public static void sort(int[] arr) {  
        int size = arr.length;  
        int temp, j;
```

```

for (int i = 1; i < size; i++) {
    temp = arr[i];
    for (j = i; j > 0 && more(arr[j - 1], temp); j--) {
        arr[j] = arr[j - 1];
    }
    arr[j] = temp;
}
}

public static void main(String[] args) {
    int[] array = { 9, 1, 8, 2, 7, 3, 6, 4, 5 };
    InsertionSort.sort(array);
    for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}
}

```

Analysis:

- The outer loop is used to pick the value we want to insert into the sorted array in left.
- The value we want to insert we have picked and saved in a temp variable.
- The inner loop is doing the comparison using the more() function. The values are shifted to the right until we find the proper position of the temp value for which we are doing this iteration.
- Finally, the value is placed into the proper position. In each iteration of the outer loop, the length of the sorted array increase by one. When we exit the outer loop, the whole array is sorted.

Complexity Analysis:

Worst case time complexity	$O(n^2)$
Best case time complexity	$O(n)$
Average case time complexity	$O(n^2)$
Space Complexity	$O(1)$

Stable sorting

Yes

Selection-Sort

Selection-Sort traverse the unsorted array and put the largest value at the end of it. This process is repeated $(n-1)$ number of times. This algorithm also have quadratic time complexity, but performs better than both bubble and Insertion-Sort as less number of swaps are required. The sorted array is created backward in Selection-Sort.

5	6	2	4	7	3	1		Swap
5	6	2	4	1	3	7		Swap
5	3	2	4	1	6	7		Swap
1	3	2	4	5	6	7		No Swap
1	3	2	4	5	6	7		Swap
1	2	3	4	5	6	7		No Swap
1	2	3	4	5	6	7		

Example 4.4:

```
public class SelectionSort {  
    public static void sort(int[] arr)// sorted array created from back.  
    {  
        int size = arr.length;  
        int i, j, max, temp;  
  
        for (i = 0; i < size - 1; i++) {  
            max = 0;  
            for (j = 1; j < size - 1 - i; j++) {  
                if (arr[j] > arr[max]) {  
                    max = j;  
                }  
            }  
            temp = arr[size - 1 - i];  
            arr[size - 1 - i] = arr[max];  
            arr[max] = temp;  
        }  
    }  
}
```

```

    }

public static void main(String[] args) {
    int[] array = { 9, 1, 8, 2, 7, 3, 6, 4, 5 };
    SelectionSort.sort(array); for (int i = 0; i < array.length; i++) {
        System.out.print(array[i] + " ");
    }
}
}

```

Analysis:

- The outer loop decides the number of times the inner loop will iterate. For an input of N elements, the inner loop will iterate N number of times.
- In each iteration of the inner loop, the largest value is calculated and is placed at the end of the array.
- This is the final replacement of the maximum value to the proper location. The sorted array is created backward.

Complexity Analysis:

Worst Case time complexity	$O(n^2)$
Best Case time complexity	$O(n^2)$
Average case time complexity	$O(n^2)$
Space Complexity	$O(1)$
Stable Sorting	No

The same algorithm can be implemented by creating the sorted array in the front of the array.

Example 4.5:

```

public static void sort2(int[] arr) // sorted array created from front
{
    int size = arr.length;
    int i, j, min, temp;

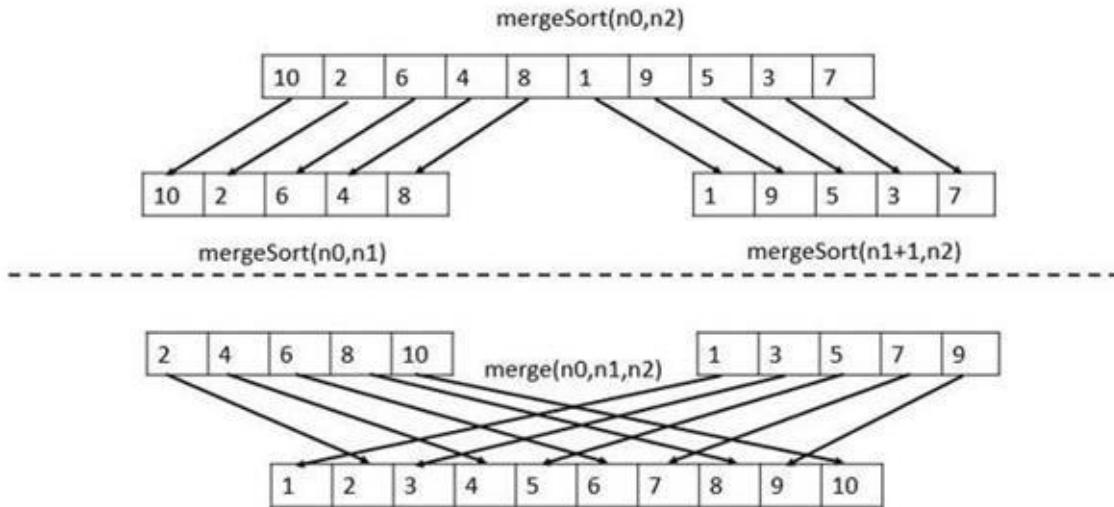
    for (i = 0; i < size - 1; i++) {
        min = i;

```

```
for (j = i + 1; j < size; j++) {  
    if (arr[j] < arr[min]) {  
        min = j;  
    }  
}  
  
temp = arr[i];  
arr[i] = arr[min];  
arr[min] = temp;  
}  
}
```

Merge-Sort

Merge sort divide the input into two half recursively. In each step, the data is divided into two half. The two parts are sorted separately and finally combine the result into final sorted output.



Example 4.6:

```
public class MergeSort {  
    private static void mergeSrt(int[] arr, int[] tempArray, int lowerIndex, int upperIndex) {  
        if (lowerIndex >= upperIndex) {  
            return;  
        }  
        int middleIndex = (lowerIndex + upperIndex) / 2;  
        mergeSrt(arr, tempArray, lowerIndex, middleIndex);  
        mergeSrt(arr, tempArray, middleIndex + 1, upperIndex);  
  
        //Merge Code  
        int lowerStart = lowerIndex;  
        int lowerStop = middleIndex;  
        int upperStart = middleIndex + 1;  
        int upperStop = upperIndex;  
        int count = lowerIndex;  
        while (lowerStart <= lowerStop && upperStart <= upperStop) {  
            if (arr[lowerStart] < arr[upperStart]) {  
                tempArray[count] = arr[lowerStart];  
                lowerStart++;  
            } else {  
                tempArray[count] = arr[upperStart];  
                upperStart++;  
            }  
            count++;  
        }  
        for (int i = lowerIndex; i <= upperIndex; i++) {  
            arr[i] = tempArray[i];  
        }  
    }  
}
```

```
tempArray[count++] = arr[lowerStart++];
} else {
tempArray[count++] = arr[upperStart++];
}
}
while (lowerStart <= lowerStop) {
tempArray[count++] = arr[lowerStart++];
}
while (upperStart <= upperStop) {
tempArray[count++] = arr[upperStart++];
}
for (int i = lowerIndex; i <= upperIndex; i++) {
arr[i] = tempArray[i];
}
}

public static void sort(int[] arr) {
int size = arr.length;
int[] tempArray = new int[size];
mergeSort(arr, tempArray, 0, size - 1);
}

public static void main(String[] args) {
int[] array = { 3, 4, 2, 1, 6, 5, 7, 8, 1, 1 };
MergeSort.sort(array); for (int i = 0; i < array.length; i++) {
System.out.print(array[i] + " ");
}
}
}
```

Analysis:

- Time Complexity of Merge-Sort is **O(nlogn)** in all 3 cases (best, average and worst) as Merge-Sort always divides the array into two halves and takes linear time to merge two halves.
 - It requires the equal amount of additional space as the unsorted array. Hence, it is not at all recommended for searching large unsorted arrays.
 - It is the best Sorting technique for sorting Linked Lists.

Complexity Analysis:

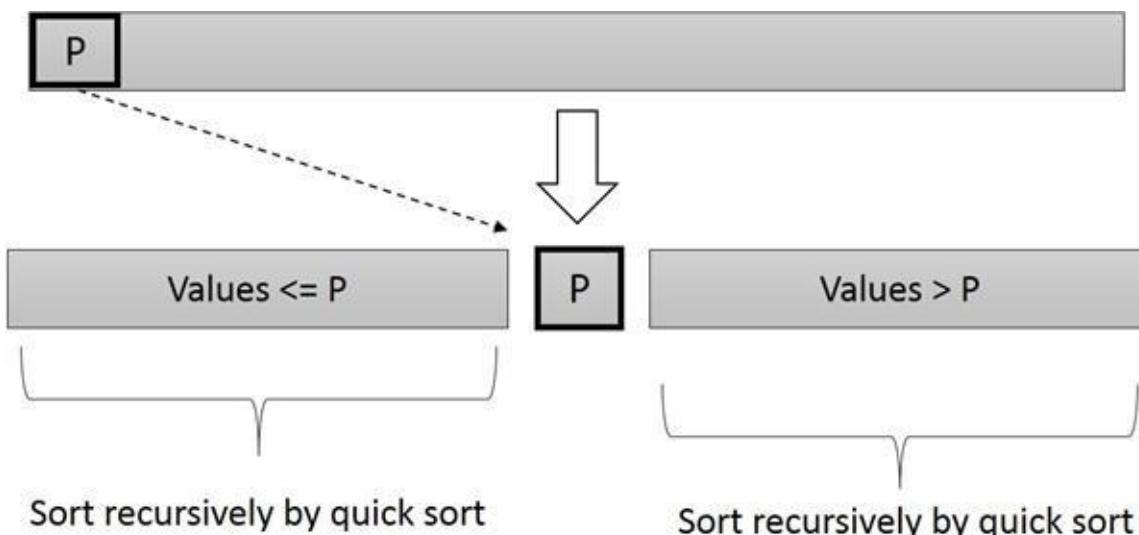
Worst Case time complexity	$O(n \log n)$
Best Case time complexity	$O(n \log n)$
Average case time complexity	$O(n \log n)$
Space Complexity	$O(n)$
Stable Sorting	Yes

Quick-Sort

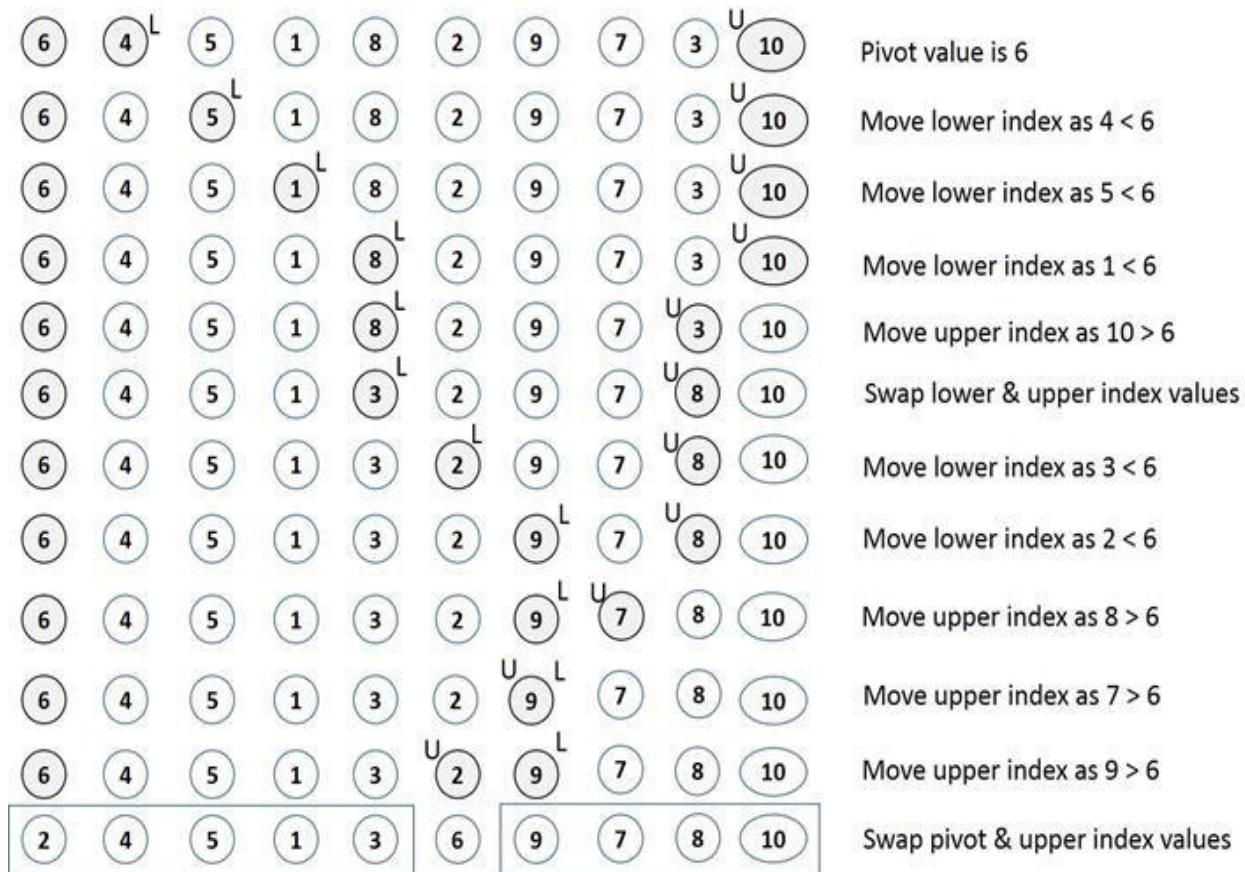
Quick sort algorithm:

- Quick sort is recursive algorithm. In each step, we select a pivot (let us say first element of array).
- Partition the array into two parts, such that all the elements of the array which are smaller than the pivot should be moved to the left side of array and all the elements that are greater than pivot are at moved the right side of the array.
 - Select pivot as first element in the array.
 - Use two index variable lower and upper.
 - Set lower index as second element in the array and upper index as last element of the array.
 - Increase lower index till value at lower index is less than pivot.
 - Then decrement upper index till the value at upper index is greater than pivot.
 - Then swap the value at lower and upper index.
 - Repeat the above 3 steps till upper index is greater than lower index.
 - In the end, swap value at pivot and upper index.
- Then we sort the left and right sub-array separately.
- When the algorithm returns the whole array is sorted.

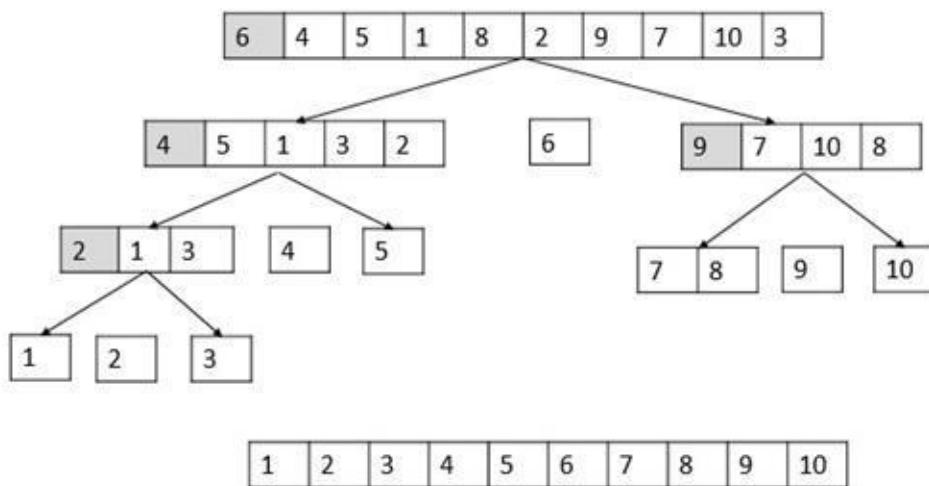
Below diagram demonstrating the partition step in quick sort.



Below diagram demonstrating each step in partition step of quicksort.



Quick sort demonstrated using diagram.



Example 4.7:

```
import java.util.Arrays;
```

```
public class QuickSort {  
    private static void quickSort(int arr[], int lower, int upper) {  
        if (upper <= lower)  
            return;  
        int pivot = arr[lower];  
        int start = lower;  
        int stop = upper;  
  
        while (lower < upper) {  
            while (arr[lower] <= pivot && lower < upper) {  
                lower++;  
            }  
            while (arr[upper] > pivot && lower <= upper) {  
                upper--;  
            }  
            if (lower < upper) {  
                swap(arr, upper, lower);  
            }  
        }  
        swap(arr, upper, start); // upper is the pivot position  
        quickSort(arr, start, upper - 1); // pivot -1 is the upper for left sub array.  
        quickSort(arr, upper + 1, stop); // pivot + 1 is the lower for right sub array  
    }  
  
    public static void sort(int arr[]) {  
        int size = arr.length;  
        quickSort(arr, 0, size - 1);  
    }  
  
    private static void swap(int arr[], int first, int second) {  
        int temp = arr[first];  
        arr[first] = arr[second];  
        arr[second] = temp;  
    }  
  
    public static void main(String[] args) {
```

```

int[] array = { 3, 4, 2, 1, 6, 5, 7, 8, 1, 1 };
QuickSort.sort(array);
for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
}
}

```

- The space required by Quick-Sort is very less, only $O(n \log n)$ additional space is required.
- Quicksort is not a stable sorting technique. It can reorder elements with identical keys.

Complexity Analysis:

Worst Case time complexity	$O(n^2)$
Best Case time complexity	$O(n \log n)$
Average case time complexity	$O(n \log n)$
Space Complexity	$O(n \log n)$
Stable Sorting	No

Quick Select

Quick select algorithm is used to find the element, which will be at the Kth position when the array will be sorted without actually sorting the whole array. Quick select is very similar to Quick-Sort in place of sorting the whole array we just ignore the one-half of the array at each step of Quick-Sort and just focus on the region of array on which the kth element lies.

Example 4.8:

```
public class QuickSelect {  
    public static void quickSelect(int arr[], int lower, int upper, int k) {  
        if (upper <= lower)  
            return;  
        int pivot = arr[lower];  
        int start = lower;  
        int stop = upper;  
        while (lower < upper) {  
            while (arr[lower] <= pivot && lower < upper) {  
                lower++;  
            }  
            while (arr[upper] > pivot && lower <= upper) {  
                upper--;  
            }  
            if (lower < upper) {  
                swap(arr, upper, lower);  
            }  
        }  
  
        swap(arr, upper, start); // upper is the pivot position  
        if (k < upper)  
            quickSelect(arr, start, upper - 1, k); // pivot -1 is the upper for left sub array.  
        if (k > upper)  
            quickSelect(arr, upper + 1, stop, k); // pivot + 1 is the lower for right sub  
array.  
    }  
  
    public static void swap(int arr[], int first, int second) {
```

```

int temp = arr[first];
arr[first] = arr[second];
arr[second] = temp;
}

public static int get(int arr[], int k) {
quickSelect(arr, 0, arr.length - 1, k);
return arr[4];
}

public static void main(String[] args) {
int[] array = { 3, 4, 2, 1, 6, 5, 7, 8, 10, 9 };
System.out.print("value at index 5 is : " + QuickSelect.get(array, 5));
}
}

```

Complexity Analysis:

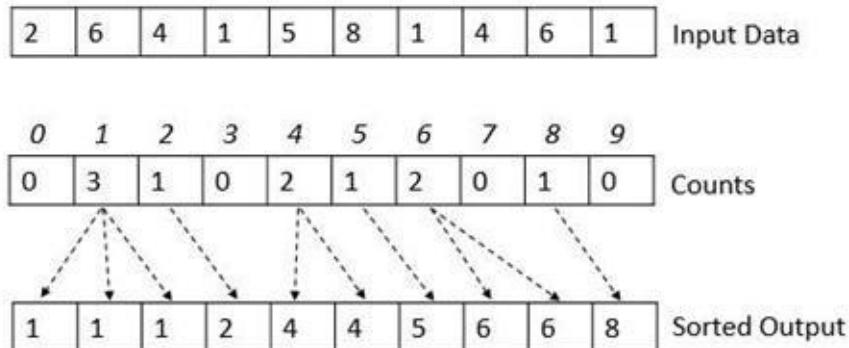
Worst Case time complexity	$O(n^2)$
Best Case time complexity	$O(\log n)$
Average case time complexity	$O(\log n)$
Space Complexity	$O(n \log n)$

Bucket Sort

Bucket sort is the simplest and most efficient type of sorting. Bucket sort has a strict requirement of a predefined range of data.

Like, sort how many people are in which age group. We know that the age of people can vary between 0 and 130. We can directly store counts in an array of size 130.

If we know the range of input, then sorting can be done using counting in $O(n+k)$. Where n is the number of elements and k is the possible range, in above example it is 130.



Example 4.9:

```
public class BucketSort {
```

```
    public static void sort(int[] array, int lowerRange, int upperRange) {
        int i, j;
        int size = array.length;
        int range = upperRange - lowerRange;
        int[] count = new int[range];

        for (i = 0; i < size; i++) {
            count[array[i] - lowerRange]++;
        }

        j = 0;
        for (i = 0; i < range; i++) {
            for (; count[i] > 0; (count[i])--) {
```

```
array[j++] = i + lowerRange;  
}  
}  
}  
  
public static void main(String[] args) {  
    int[] array = { 23, 24, 22, 21, 26, 25, 27, 28, 21, 21 };  
    BucketSort.sort(array, 20, 30);  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
}
```

Analysis:

- We have created a count array to store counts.
 - Count array elements are initialized to zero.
 - Index corresponding to input array is incremented.
 - Finally, the information stored in count array is saved in the array.

Complexity Analysis:

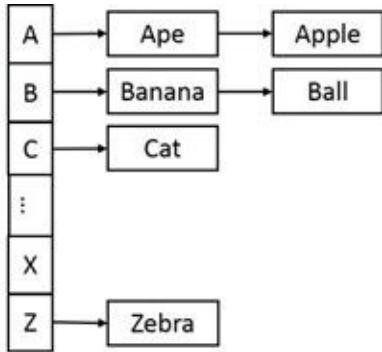
Data structure	Array
Worst case performance	$O(n+k)$
Average case performance	$O(n+k)$
Worst case Space Complexity	$O(k)$

k - Number of distinct elements.

n - Total number of elements in array.

Generalized Bucket Sort

There are cases when the elements falling into a bucket are not unique but are in the same range. When we want to sort an index of a name, we can use the pointer bucket to store names.



The buckets are already sorted and the elements inside each bucket can be kept sorted by using an Insertion-Sort algorithm. We are leaving this generalized bucket sort implementation to the reader of this book. The similar data structure will be defined in the coming chapter of Hash-Table using separate chaining.

Heap-Sort

Heap-Sort we will study in the Heap chapter.

Complexity Analysis:

Data structure	Array
Worst case performance	$O(n \log n)$
Average case performance	$O(n \log n)$
Worst case Space Complexity	$O(1)$

Tree Sorting

In-order traversal of the binary search tree can also be seen as a sorting algorithm. We will see this in binary search tree section of tree chapter.

Complexity Analysis:

Worst Case time complexity	$O(n^2)$
Best Case time complexity	$O(n \log n)$
Average case time complexity	$O(n \log n)$
Space Complexity	$O(n)$
Stable Sorting	Yes

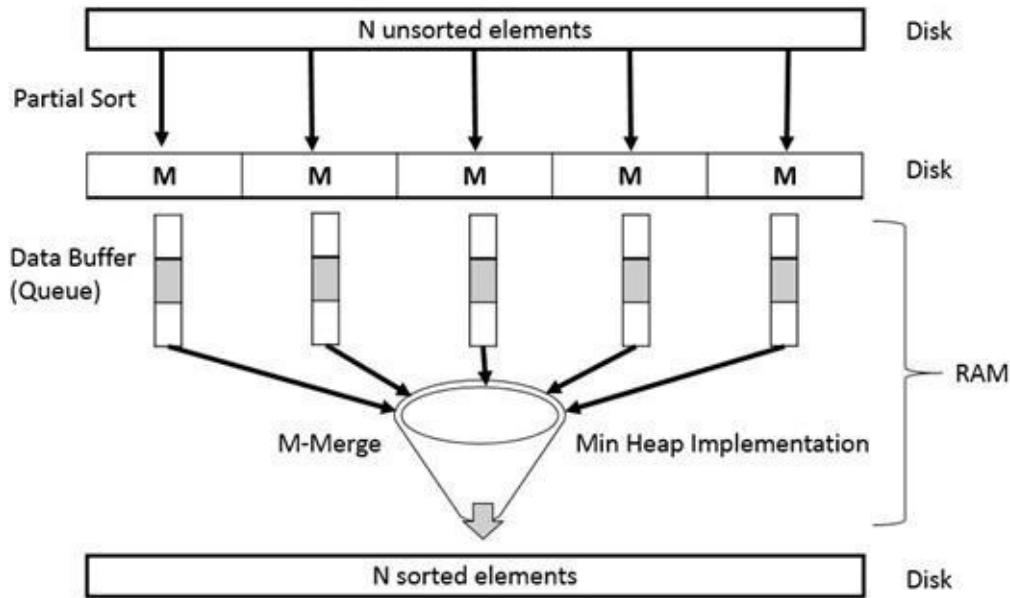
External Sort (External Merge-Sort)

When data needs to be sorted is huge and it is not possible to load it completely in memory (RAM), for such a dataset we use external sorting. Such data is sorted using external Merge-Sort algorithm. First data is picked in chunks and it is sorted in memory. Then this sorted data is written back to disk. Whole data are sorted in chunks using Merge-Sort. Now we need to combine these sorted chunks into final sorted data.

Then we create queues for the data, which will read from the sorted chunks. Each chunk will have its own queue. We will pop from this queue and these queues are responsible for reading from the sorted chunks. Let us suppose we have K different chunks of sorted data each of length M.

The third step is using a Min-Heap, which will take input data from each of this queue. It will take one element from each queue. The minimum value is taken from the Heap and added to the final output. Then queue from which this minimum value element is added to the heap will again be popped and one more element from this queue is added to the Heap. Finally, when the data is exhausted from some queue that queue is removed from the input list. Finally, we will get a sorted data coming out from the heap.

We can optimize this process further by adding an output buffer, which will store data coming out of Heap and will do a limited number of the write operation in the final Disk space.



Note: No one will be asking to implement external sorting in an interview, but it is good to know about it.

Stable Sorting

A sorting algorithm is said to be stable if two elements with equal key value appear in same order in sorted output as they are in unsorted input. Stable sorting algorithms guarantees not to reorder elements with identical keys.

Comparisons of the various sorting algorithms.

Below is comparison of various sorting algorithms:

Sort	Average Time	Best Time	Worst Time	Space	Stable
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Modified Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	Yes
Heap Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(1)$	No
Merge Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n)$	Yes
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$	$O(n) \text{ worst case}$ $O(\log(n)) \text{ average case}$	No
Bucket Sort	$O(n k)$	$O(n k)$	$O(n k)$	$O(n k)$	Yes

Selection of Best Sorting Algorithm

No sorting algorithm is perfect. Each of them has their own pros and cons. Let us read one by one: **Quick-Sort**: When you do not need a stable sort and average case performance matters more than worst-case performance. When data is random, we prefer the Quick-Sort. Average case time complexity of Quick-Sort is **$O(n\log n)$** and worst-case time complexity is **$O(n^2)$** . Space Complexity of Quick-Sort is **$O(\log n)$** auxiliary storage, which is stack space used in recursion.

Merge-Sort: When you need a stable sort and Time Complexity of **$O(n\log n)$** , Merge-Sort is used. In general, Merge-Sort is slower than Quick-Sort because of lot of copy happens in the merge phase. There are two uses of Merge-Sort when we want to merge two sorted linked lists and Merge-Sort is used in external sorting.

Heap-Sort: When you do not need a stable sort and you care more about worst-case performance than average case performance. It has guaranteed to be **$O(n\log n)$** , and uses **$O(1)$** auxiliary space, means you will not unpredictably run out of memory on very large inputs.

Insertion-Sort: When we need a stable sort, When N is guaranteed to be small, including as the base case of a Quick-Sort or Merge-Sort. Worst-case time complexity is **$O(n^2)$** . It has a very small constant factor multiplied to calculate actual time taken. Therefore, for smaller input size it performs better than Merge-Sort or Quick-Sort. It is also useful when the data is already pre-sorted. In this case, its running time is **$O(N)$** .

Bubble-Sort: Where we know the data is nearly sorted. Say only two elements are out of place. Then in one pass, Bubble Sort will make the data sorted and in the second pass, it will see everything is sorted and then exit. Only takes 2 passes of the array.

Selection-Sort: Best, Worst & Average Case running time all are **$O(n^2)$** . It is only useful when you want to do something quick. They can be used when you are just doing some prototyping.

Counting-Sort: When you are sorting data within a limited range.

Radix-Sort: When $\log(N)$ is significantly larger than K, where K is the number of radix digits.

Bucket-Sort: When your input is more or less uniformly distributed.

Problems based on sorting

Partition 0 and 1

Problem: Given an array containing 0s and 1s. Write an algorithms to sort array so that 0s come first followed by 1s. Also find the minimum number of swaps required to sort the array.

First solution: Start from both end, left will store start index and right will store end index. Traverse left forward till we have 0s value in the array. Then traverse right backward till we have 1s in the end. Then swap the two and follow the same process till left is less than right.

Example 4.10:

```
public static int Partition01(int[] arr, int size) {  
    int left = 0;  
    int right = size - 1;  
    int count = 0;  
    while (left < right) {  
        while (arr[left] == 0)  
            left += 1;  
  
        while (arr[right] == 1)  
            right -= 1;  
  
        if (left < right) {  
            swap(arr, left, right);  
            count += 1;  
        }  
    }  
    return count;  
}  
  
// Testing code  
public static void main(String[] args) {  
    int arr[] = { 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1 };
```

```
    Partition01(arr, arr.length);
    printArray(arr, arr.length);
}
```

Time complexity looks like quadratic (loop inside loop) but it is linear or $O(n)$. As at each iteration of inner loop, either left is increasing or right is decreasing. Once left is equal or greater than right the loops stop. Therefore, in total the inner loops combined runs for N number of time.

Partition 0, 1 and 2

Problem: Given an array containing 0s, 1s and 2s. Write an algorithms to sort array so that 0s come first followed by 1s and then 2s in the end.

First solution: You can use a counter for 0s, 1s and 2s. Then replace the values in the array. This will take two pass. What if we want to do this in single pass?

Second solution: The basic approach is to use three index. First left, second right and third to traverse the array. Index left starts form 0, Index right starts from $N-1$. We traverse the array whenever we find a 0 we swap it with the value at start and increment start. And whenever we finds a 2 we swaps this value with right and decrement right. When traversal is complete and we reach the right then the array is sorted.

Example 4.11:

```
public static void Partition012(int[] arr, int size) {
    int left = 0;
    int right = size - 1;
    int i = 0;
    while (i <= right) {
        if (arr[i] == 0) {
            swap(arr, i, left);
            i += 1;
            left += 1;
        } else if (arr[i] == 2) {
            swap(arr, i, right);
            right -= 1;
        }
    }
}
```

```

    } else {
        i += 1;
    }
}
}

// Testing code
public static void main(String[] args) {
    int arr[] = { 0, 1, 1, 0, 1, 2, 1, 2, 0, 0, 0, 1 };
    Partition012(arr, arr.length);
    printArray(arr, arr.length);
}

```

Time complexity is linear or O(n).

Range Partition

Problem: Given an array of integer and a range. Write an algorithms to partition array so that values smaller than range come to left, then values under the range followed with values greater than the range.

First solution: The basic approach is to use three index. First left, second right and third to traverse the array. Index left starts form 0, Index right starts from N-1. We traverse the array whenever we find a value lower than range we swap it with the value at start and increment start. And whenever we finds a value greater than range we swaps this value with right and decrement right. When traversal is complete we have the array partitioned about range.

Example 4.12:

```

public static void RangePartition(int[] arr, int size, int lower, int higher) {
    int start = 0;
    int end = size - 1;
    int i = 0;
    while (i <= end) {
        if (arr[i] < lower) {
            swap(arr, i, start);
            i += 1;
        }
    }
}

```

```

start += 1;
} else if (arr[i] > higher) {
    swap(arr, i, end);
    end -= 1;
} else {
    i += 1;
}
}

// Testing code
public static void main(String[] args) {
    int arr[] = { 1, 21, 2, 20, 3, 19, 4, 18, 5, 17, 6, 16, 7, 15, 8, 14, 9, 13, 10, 12, 11
};
    RangePartition(arr, arr.length, 9, 12);
    printArray(arr, arr.length);
}

```

Time complexity is linear or $O(n)$.

Minimum swaps

Problem: Minimum swaps required to bring all elements less than given value together at the start of array.

First solution: Use quick sort kind of technique by taking two index one from start and another from end and try to use the given value as key. Count the number of swaps that is answer.

Example 10.13:

```

int minSwaps(int arr[], int size, int val)
{
    int swapCount = 0;
    int first = 0;
    int second = size - 1;
    int temp;
    while (first < second)

```

```

{
if (arr[first] <= val)
first += 1;
else if (arr[second] > val)
second -= 1;
else
{
temp = arr[first];
arr[first] = arr[second];
arr[second] = temp;
swapCount += 1;
}
}
return swapCount;
}

```

Time complexity is linear or O(n).

Absolute Sort

Problem: Sort array according to the absolute difference from the given value.

Solution: Any sorting algorithms can be used to solve this problem we are using BubbleSort. The only change is compare function more. Which will take another field ref.

Example 4.14:

```

public static void AbsBubbleSort(int[] arr, int size, int ref) {
    for (int i = 0; i < (size - 1); i++) {
        for (int j = 0; j < (size - i - 1); j++) {
            if (AbsMore(arr[j], arr[j + 1], ref)) {
                swap(arr, j, j + 1);
            }
        }
    }
}

```

```

public static boolean AbsMore(int value1, int value2, int ref) {
    return (Math.abs(value1 - ref) > Math.abs(value2 - ref));
}

// Testing code
public static void main(String[] args) {
    int array[] = { 9, 1, 8, 2, 7, 3, 6, 4, 5 };
    int ref = 5;
    AbsBubbleSort(array, array.length, ref);
    printArray(array, array.length);
}

```

Output:

[5 6 4 7 3 8 2 9 1]

Time complexity is linear or $O(n)$.

Equation Sort

Problem: Sort array according to the equation $A.X^2$.

Solution: Any sorting algorithms can be used to solve this problem also. The only change is compare function more. Which will take another field A.

Example 4.15:

```

public static boolean EqMore(int value1, int value2, int A) {
    value1 = A * value1 * value1;
    value2 = A * value2 * value2;
    return value1 > value2;
}

```

Sort by Order

Problem: Given two array, sort first array according to the order defined in second array.

Solution: First the input array is traversed and frequency of values is calculated

using HashTable. The order array is traversed and those values which are in original array / hashtable are displayed and removed from hashtable. Then the rest of the values of hashtable is printed to screen.

Example 4.16:

```
public static void SortByOrder(int[] arr, int size, int arr2[], int size2) {  
    HashMap<Integer, Integer> ht = new HashMap<Integer, Integer>();  
    int value;  
    for (int i = 0; i < size; i++) {  
        if (ht.containsKey(arr[i])) {  
            value = ht.get(arr[i]);  
            ht.put(arr[i], value + 1);  
        } else {  
            ht.put(arr[i], 1);  
        }  
  
        for (int j = 0; j < size2; j++) {  
            if (ht.containsKey(arr2[j])) {  
                value = ht.get(arr2[j]);  
                for (int k = 0; k < value; k++) {  
                    System.out.print(arr2[j]);  
                }  
                ht.remove(arr2[j]);  
            }  
        }  
  
        for (int i = 0; i < size; i++) {  
            if (ht.containsKey(arr[i])) {  
                value = ht.get(arr[i]);  
                for (int k = 0; k < value; k++) {  
                    System.out.print(arr[i]);  
                }  
                ht.remove(arr[i]);  
            }  
        }  
    }  
}
```

```

// Testing code
public static void main(String[] args) {
    int arr[] = { 2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8 };
    int arr2[] = { 2, 1, 8, 3 };
    SortByOrder(arr, arr.length, arr2, arr2.length);
}

```

Time complexity is $O(n)$.

Separate even and odd numbers in List

Problem: Given an array of even and odd numbers, write a program to separate even numbers from the odd numbers.

First solution: allocate a separate list, then scan through the given list, and fill even numbers from the start and odd numbers from the end.

Second solution: Algorithm is as follows.

1. Initialize the two variable left and right. Variable left=0 and right= size-1.
2. Keep increasing the left index until the element at that index is even.
3. Keep decreasing the right index until the element at that index is odd.
4. Swap the number at left and right index.
5. Repeat steps 2 to 4 until left is less than right.

Example 4.17:

```

public static void seperateEvenAndOdd(int data[], int size) {
    int left = 0, right = size - 1;
    while (left < right) {
        if (data[left] % 2 == 0)
            left++;
        else if (data[right] % 2 == 1)
            right--;
        else {
            swap(data, left, right);
            left++;
            right--;
        }
    }
}

```

```
    }  
    }  
}
```

Time complexity is linear or $O(n)$.

Array Reduction

Problem: Element left after reductions. Given an array of positive elements. You need to perform reduction operation. In each reduction operation smallest positive element value is picked and all the elements are subtracted by that value. You need to print the number of elements left after each reduction process.

Input: [5, 1, 1, 1, 2, 3, 5]

Output:

4 corresponds to [4, 1, 2, 4]

3 corresponds to [3, 1, 3]

2 corresponds to [2, 2]

0 corresponds to [0]

Example 4.18:

```
public static void ArrayReduction(int[] arr, int size) {  
    Arrays.sort(arr);  
    int count = 1;  
    int reduction = arr[0];  
  
    for (int i = 0; i < size; i++) {  
        if (arr[i] - reduction > 0) {  
            System.out.println(size - i);  
            reduction = arr[i];  
            count += 1;  
        }  
    }  
    System.out.println("Total number of reductions " + count);  
}
```

```

// Testing code
public static void main(String[] args) {
    int arr[] = { 5, 1, 1, 1, 2, 3, 5 };
    ArrayReduction(arr, arr.length);
}

```

Time complexity: $O(n \log n)$. $O(n \log n)$ required for sorting.

Problem: If it is asked to find the total number of reduction operations that are needed. Then this can be done in linear time.

Hint: The total number of reductions is equal to the total number of distinct elements.

Merge Array

Problem: Given two sorted arrays. Sort the elements of these arrays so that first half of sorted elements will lie in first array and second half lies in second array. Extra space allowed is $O(1)$.

Solution: The first array will contain smaller part of the sorted output. We traverse the first array. Always compare the value of first array with the first element of the second array. If first array value is small than first element of second array we iterate further. If the first array value is greater than first element of second array. Then copy value of first element of second array into first array. And insert value of first array into second array in sorted order. Second array is always kept sorted so we need to compare only its first element.

Time complexity will $O(M*N)$ where M is length of first array and N is length of second array.

Example 4.19:

```

public static void merge(int[] arr1, int size1, int[] arr2, int size2) {
    int index = 0;
    while (index < size1) {
        if (arr1[index] <= arr2[0]) {

```

```

index += 1;
} else {
// always first element of arr2 is compared.
arr1[index] ^= arr2[0] ^= arr1[index] ^= arr2[0];
index += 1;
// After swap arr2 may be unsorted.
// Insertion of the element in proper sorted position.
for (int i = 0; i < (size2 - 1); i++) {
if (arr2[i] < arr2[i + 1])
break;
arr2[i] ^= arr2[i + 1] ^= arr2[i] ^= arr2[i + 1];
}
}
}

// Testing code.
public static void main(String[] args) {
int arr1[] = { 1, 5, 9, 10, 15, 20 };
int arr2[] = { 2, 3, 8, 13 };
merge(arr1, arr1.length, arr2, arr2.length);
printArray(arr1, arr1.length);
printArray(arr2, arr2.length);
}

```

Check Reverse

Problem: Given an array of integers, find if reversing a sub-array makes the array sorted.

Solution: In this algorithm start and stop are the boundary of reversed sub-array whose reversal makes the whole array sorted.

Example 4.20:

```

public static boolean checkReverse(int[] arr, int size) {
int start = -1;
int stop = -1;

```

```

for (int i = 0; i < (size - 1); i++) {
    if (arr[i] > arr[i + 1]) {
        start = i;
        break;
    }
}
if (start == -1)
    return true;

for (int i = start; i < (size - 1); i++) {
    if (arr[i] < arr[i + 1]) {
        stop = i;
        break;
    }
}

if (stop == -1)
    return true;

// increasing property
// after reversal the sub array should fit in the array.
if (arr[start - 1] > arr[stop] || arr[stop + 1] < arr[start])
    return false;

for (int i = stop + 1; i < size - 1; i++) {
    if (arr[i] > arr[i + 1]) {
        return false;
    }
}
return true;
}

```

Time complexity is linear or $O(n)$.

Union Intersection Sorted

Problem: Given two unsorted arrays, find union and intersection of these two arrays.

Solution: Sort both the arrays. Then traverse both the array, when we have common element we add it to intersection list and union list, when we have uncommon elements then we add it only union list.

Example 4.21:

```
public static void UnionIntersectionSorted(int arr1[], int size1, int arr2[], int size2) {
    int first = 0, second = 0;
    int[] unionArr = new int[size1 + size2];
    int[] interArr = new int[min(size1, size2)];
    int uIndex = 0;
    int iIndex = 0;

    while (first < size1 && second < size2) {
        if (arr1[first] == arr2[second]) {
            unionArr[uIndex++] = arr1[first];
            interArr[iIndex++] = arr1[first];
            first += 1;
            second += 1;
        } else if (arr1[first] < arr2[second]) {
            unionArr[uIndex++] = arr1[first];
            first += 1;
        } else {
            unionArr[uIndex++] = arr2[second];
            second += 1;
        }
    }

    while (first < size1) {
        unionArr[uIndex++] = arr1[first];
        first += 1;
    }

    while (second < size2) {
        unionArr[uIndex++] = arr2[second];
        second += 1;
    }
}
```

```
    }
    printArray(unionArr, uIndex);
    printArray(interArr, iIndex);
}

public static void UnionIntersectionUnsorted(int arr1[], int size1, int arr2[], int
size2) {
    Arrays.sort(arr1);
    Arrays.sort(arr2);
    UnionIntersectionSorted(arr1, size1, arr2, size2);
}

public static void main(String[] args) {
    int arr1[] = { 1, 11, 2, 3, 14, 5, 6, 8, 9 };
    int arr2[] = { 2, 4, 5, 12, 7, 8, 13, 10 };
    UnionIntersectionUnsorted(arr1, arr1.length, arr2, arr2.length);
}
```

Time complexity is $O(n \log n)$. $O(n \log n)$ required for sorting.

Exercise

1. In given text file, print the words with their frequency. Now print the kth word in term of frequency.

Hint:-

- a) First solution may be you can use the sorting and return the kth element.
- b) Second solution: You can use the kth element quick select algorithm.
- c) Third solution: You can use Hashtable or Trie to keep track of the frequency. Use Heap to get the Kth element.

2. In given K input streams of number in sorted order. You need to make a single output stream, which contains all the elements of the K streams in sorted order. The input streams support ReadNumber() operation and output stream support WriteNumber() operation.

Hint:-

- a) Read the first number from all the K input streams and add them to a Priority Queue. (Nodes should keep track of the input stream; data added to the PQ is value & stream id.)
- b) Dequeue one element at a time from PQ, Put this element value to the output stream, Read the input stream number and from the same input stream add another element to PQ.
- c) If the stream is empty, just continue
- d) Repeat until PQ is empty.

3. In given K sorted Lists of fixed length M. Also, given a final output list of length $M \times K$. Give an efficient algorithm to merge all the arrays into the final list, without using any extra space.

Hint: you can use the end of the final list to make PQ.

4. How will you sort 1 PB numbers? 1 PB = 1000 TB.

5. What will be the complexity of the above solution?

6. Any other improvement can be done on question 3 solution if the number of

CPU cores is eight.

7. In given integer list that support three functions findMin, findMax, findMedian. Sort the array.
8. In given pile of patient files of High, mid and low priority. Sort these files such that higher priority comes first, then mid and last low priority.
Hint: Bucket sort.
9. Write pros and cons of Heap-Sort, Merge-Sort and Quick-Sort.
10. In given rotated-sorted list of N integers. (The array was sorted then it was rotated some arbitrary number of times.) If all the elements in the array were unique, find the index of some value.

Hint: Modified binary search

11. In the problem 9, what if there are repetitions allowed and you need to find the index of the first occurrence of the element in the rotated-sorted list.
12. Merge two sorted Lists into a single sorted list.
Hint: Use merge method of Merge-Sort.
13. Given an array contain 0's and 1's, sort the array such that all the 0's come before 1's.
14. Given an array of English characters, sort the array in linear time.
15. Write a method to sort an array of strings so that all the anagrams are next to each other.

Hint:-

- a) Loop through the array.
- b) For each word, sort the characters and add it to the hash map with keys as sorted word and value as the original word. At the end of the loop, you will get all anagrams as the value to a key (which is sorted by its

constituent chars).

- c) Iterate over the hashmap, print all values of a key together and then move to the next key.

Space Complexity: $O(n)$, Time Complexity: $O(n)$

16. Given an array, sort elements in the order of their frequency.

Hint: First, the frequency of various elements of array is calculated by adding it to HashTable. Then sorting of the new data structures with value and key is done. The sorting function first give preference to frequency then value.

CHAPTER 5: SEARCHING

Introduction

Searching is the process of finding a particular item in a collection of items. The item may be a keyword in a file, a record in a database, a node in a tree or a value in an array etc.

Why Searching?

Imagine you are in a library with millions of books. You want to get a specific book with specific title. How will you find it? You will search the book in the section of library, which contains the books whose name starts with the initial letter of the desired book. Then you continue matching with a whole book title until you find your book. (By doing this small heuristic method you have reduced the search space by a factor of 26, consider we have an equal number of books whose title begin with particular char.)

Similarly, computer stores lots of information and to retrieve this information efficiently, we need very efficient searching algorithms. To make searching efficient, we keep the data in some proper order. If you keep the data organized in proper order, it is easy to search required value or key. For example, keeping the data in sorted order is one of the way to organize data.

Different Searching Algorithms

- Linear Search – Unsorted Input
- Linear Search – Sorted Input
- Binary Search (Sorted Input)
- String Search: Tries, Suffix Trees, Ternary Search.
- Hashing and Symbol Tables

Linear Search – Unsorted Input

When elements of an array are not ordered or sorted and we want to search for a particular value, we need to scan the full list until we find the desired value. This kind of algorithm is known as unordered linear search. The major problem with this algorithm is less performance or high Time Complexity in worst case.

Example 5.1

```
public static boolean linearSearchUnsorted(int[] arr, int size, int value) {  
    int i = 0;  
    for (i = 0; i < size; i++) {  
        if (value == arr[i]) {  
            return true;  
        }  
    }  
    return false;  
}
```

Time Complexity: $O(n)$. As we need to traverse the complete list in worst case. Worst case is when your desired element is at the last position of the array. Here, ‘n’ is the size of the array.

Space Complexity: $O(1)$. No extra memory is used to allocate the array.

Linear Search – Sorted

If elements of the array are sorted either in increasing order or in decreasing order, searching for a desired element will be much more efficient than unordered linear search. In many cases, we do not need to traverse the complete list. If the array is sorted in increasing order. We can traverse it from beginning and when we encounter a value greater than the key, we stop searching further and declare that the key is not present in the array. This is how this algorithm saves the time and improves the performance.

Example 5.2

```
public static boolean linearSearchSorted(int[] arr, int size, int value) {  
    int i = 0;  
    for (i = 0; i < size; i++) {  
        if (value == arr[i]) {  
            return true;  
        } else if (value < arr[i]) {  
            return false;  
        }  
    }  
    return false; }
```

Time Complexity: $O(n)$. As we need to traverse the complete list in worst case. Worst case is when your desired element is at the last position of the sorted list. However, in the average case this algorithm is more efficient even though the growth rate is same as unsorted.

Space Complexity: $O(1)$. No extra memory is used to allocate the array.

Binary Search

How do we search a word in a dictionary? In general, we go to some approximate page (mostly middle) and start searching from that word. If we see the word that we are searching is in the same page then we are done with the search. Else, if we see that alphabetically the word we are searching for is in the first half then we reject the second half and vice versa. We apply the same procedure repeatedly until we find the desired keyword.

Binary Search also works in the same way. When we want to search some key value in a sorted list. We go to the middle point from the sorted list and start comparing with the desired value. If the desired value is equal to the middle value then we are done. If the value is greater than the middle value then we reject the first half. If the value is less than the middle value then we reject the second half. At each comparison, we are reducing our search space by half.

Note: Binary search requires the array to be sorted otherwise binary search cannot be applied.

Example 5.3

```
// Binary Search Algorithm - Iterative Way
public static boolean Binarysearch(int[] arr, int size, int value) {
    int low = 0;
    int high = size - 1;
    int mid;

    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == value) {
            return true;
        } else if (arr[mid] < value) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return false;
```

```
}
```

Time Complexity: $O(\log n)$. We always take half input and throw out the other half. So the recurrence relation for binary search is $T(n) = T(n/2) + c$. Using master theorem (divide and conquer), we get $T(n) = O(\log n)$

Space Complexity: $O(1)$

Example 5.4: Binary search implementation using recursion.

```
public static boolean BinarySearchRec(int[] arr, int size, int value) {  
    int low = 0;  
    int high = size - 1;  
    return BinarySearchRecUtil(arr, low, high, value);  
}  
  
// Binary Search Algorithm - Recursive Way  
public static boolean BinarySearchRecUtil(int[] arr, int low, int high, int value)  
{  
    if (low > high) {  
        return false;  
    }  
    int mid = (low + high) / 2;  
    if (arr[mid] == value) {  
        return true;  
    } else if (arr[mid] < value) {  
        return BinarySearchRecUtil(arr, mid + 1, high, value);  
    } else {  
        return BinarySearchRecUtil(arr, low, mid - 1, value);  
    }  
}
```

Time Complexity: $O(\log n)$. Space Complexity: $O(\log n)$ for system stack in recursion

String Searching Algorithms

Refer String chapter.

Hashing and Symbol Tables

Refer Hash-Table chapter.

How sorting is useful in Selection Algorithm?

Selection problems can be converted into sorting problems. Once the array is sorted, it is easy to find the minimum / maximum (or desired element) from the sorted list. The method ‘Sorting and then Selecting’ is inefficient for selecting a single element, but it is efficient when many selections need to be made from the array. It is because only one initial expensive sort is needed, followed by many cheap selection operations.

For example, if we want to get the maximum element from an array. After sorting the array, we can simply return the last element from the array. What if we want to get second maximum. Now, we do not have to sort the array again and we can return the second last element from the sorted list. Similarly, we can return the k th maximum element by just one scan of the sorted list.

So, with the above discussion, sorting is used to improve the performance. In general this method requires **$O(n \log n)$** (for sorting) time.

Problems in Searching

First Repeated element in the array

Problem: Given an unsorted list of n elements, find the first element, which is repeated.

First solution: Exhaustive search or Brute force; for each element in list find if there is some other element with the same value. This is done using two loops, first loop to select the element and second loop to find its duplicate entry.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 5.5

```
public static int FirstRepeated(int[] arr, int size) {  
    int i, j;  
    for (i = 0; i < size; i++) {  
        for (j = i + 1; j < size; j++) {  
            if (arr[i] == arr[j]) {  
                return arr[i];  
            }  
        }  
    }  
    return 0;  
}  
  
public static void main(String[] args) {  
    int[] first = { 34, 56, 77, 1, 5, 6, 6, 6, 7, 8, 10, 34, 20, 30 };  
    System.out.println(FirstRepeated(first, first.length));  
}
```

Second solution: Hash-Table; using Hash-Table, we can keep track number of times a particular element came in the array. First scan just populate the Hashtable. In the second, scan just look the occurrence of the elements in the Hashtable. If occurrence is more for some element, then we have our solution

and the first repeated element.

Hash-Table insertion and finding take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity is also $O(n)$ for maintaining hash.

Print Duplicates in List

Problem: Given an array of n numbers, print the duplicate elements in the array.

First solution: Exhaustive search or Brute force; for each element in list, find if there is some other element with the same value. This is done using two loop, first loop to select the element and second loop to find its duplicate entry.

Example 5.6

```
public static void printRepeating(int[] arr, int size) {  
    int i, j;  
    System.out.print(" \nRepeating elements are ");  
    for (i = 0; i < size; i++) {  
        for (j = i + 1; j < size; j++) {  
            if (arr[i] == arr[j]) {  
                System.out.print(" " + arr[i]);  
            }  
        }  
    }  
}
```

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Second solution: Sorting; Sort all the elements in the array and after this in a single scan, we can find the duplicates.

Example 5.7

```
public static void printRepeating2(int[] arr, int size) {  
    int i;  
    Arrays.sort(arr);  
    System.out.print(" Repeating elements are ");
```

```

for (i = 1; i < size; i++) {
    if (arr[i] == arr[i - 1]) {
        System.out.print(" " + arr[i]);
    }
}
}

```

Sorting algorithms take $O(n \log n)$ time and single scan take $O(n)$ time.
Time Complexity is $O(n \log n)$ and Space Complexity is $O(1)$

Third solution: Hash-Table, using Hash-Table, we can keep track of the elements we have already seen and we can find the duplicates in just one scan.

Example 5.8

```

public static void printRepeating3(int[] arr, int size) {
    HashSet<Integer> hs = new HashSet<Integer>();
    int i;
    System.out.print(" Repeating elements are ");
    for (i = 0; i < size; i++) {
        if (hs.contains(arr[i])) {
            System.out.print(" " + arr[i]);
        } else {
            hs.add(arr[i]);
        }
    }
}

```

Hash-Table insert and find take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity is also $O(n)$

Forth solution: Counting, this solution is only possible if we know the range of the input. If we know that, the elements in the array are in the range 0 to $n-1$. We can reserve an array of length n call it counter and when we see an element, we can increase its corresponding count. In just one single scan, we know the duplicates. If we know the range of the elements, then this is the fastest way to find the duplicates.

Example 5.9

```
public static void printRepeating4(int[] arr, int size, int range) {  
    int[] count = new int[range];  
    int i;  
    for (i = 0; i < size; i++) {  
        count[i] = 0;  
    }  
    System.out.print(" Repeating elements are ");  
    for (i = 0; i < size; i++) {  
        if (count[arr[i]] == 1) {  
            System.out.print(" " + arr[i]);  
        } else {  
            count[arr[i]]++;  
        }  
    }  
}
```

Counting solution just uses an array so inserting and finding take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity for creating count list is also $O(n)$

Remove duplicates in an integer list

Problem: Remove duplicate in a given integer list.

First solution: Sorting, Steps are as follows:

1. Sort the array.
2. Take two references. A subarray will be created with all unique elements starting from 0 to the first reference (The first reference points to the last index of the subarray). The second reference iterates through the array from 1 to the end. Unique numbers will be copied from the second reference location to first reference location and the same elements are ignored.

Time Complexity calculation:

Time to sort the array = **$O(n \log n)$** . Time to remove duplicates = **$O(n)$** .

Overall Time Complexity = **O(nlogn)**.

No additional space is required so Space Complexity is **O(1)**.

Example 5.10

```
public static int[] removeDuplicates(int[] array, int size) {  
    int j = 0;  
    int i;  
    Arrays.sort(array);  
    for (i = 1; i < size; i++) {  
        if (array[i] != array[j]) {  
            j++;  
            array[j] = array[i];  
        }  
    }  
    int[] ret = Arrays.copyOf(array, j + 1);  
    return ret;  
}
```

Second solution: Use a Hash Table to keep track the elements already visited. Create a output array and add only unique elements to it and also add that value to hash table.

Overall Time Complexity is **O(n)**.

Space Complexity is **O(n)** for hash table.

Other solutions of the previous problems can also be used to solve this problem.

Find the missing number in an array

Problem: In given list of n-1 elements, which are in the range of 1 to n. There are no duplicates in the array. One of the integer is missing. Find the missing element.

First solution: Exhaustive search or Brute force, for each value in the range 1 to n, find if there is some element in list which have the same value. This is done using two loops, first loop to select value in the range 1 to n and the second loop to find if this element is in the array or not.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 5.11

```
public static int findMissingNumber(int[] arr, int size) {  
    int i, j, found = 0;  
    for (i = 1; i <= size; i++) {  
        found = 0;  
        for (j = 0; j < size; j++) {  
            if (arr[j] == i) {  
                found = 1;  
                break;  
            }  
        }  
        if (found == 0) {  
            return i;  
        }  
    }  
    return Integer.MAX_VALUE;  
}
```

Second solution: Sorting; Sort all the elements in the array and after this in a single scan, we can find the duplicates.

Sorting algorithms takes $O(n \log n)$ time and single scan takes $O(n)$ time.
Time Complexity of an algorithm is $O(n \log n)$ and Space Complexity is $O(1)$

Third solution: Hash-Table, using Hash-Table; we can keep track of the elements we have already seen and we can find the missing element in just one scan.

Hash-Table insertion and finding take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity is also $O(n)$

Forth solution: Counting, we know the range of the input so counting will work. As we know that, the elements in the array are in the range 0 to n . We can reserve an array of length n and when we see an element, we can increase its count. In just one single scan, we know the missing element.

Counting solution just uses an array so insertion and finding take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity for creating count list is also $O(n)$

Fifth solution: You are allowed to modify the given input list. Modify the given input list in such a way that in the next scan you can find the missing element.

When you scan through the array. When at index “index”, the value stored in the array will be $\text{arr}[index]$ so add the number “ $n + 1$ ” to $\text{arr}[\text{arr}[index]]$. Always read the value from the array using a reminder operator “%”. When you scan the array for the first time and modify all the values, then in one single scan you can see if there is some value in the array which is smaller than “ $n+1$ ” that index is the missing number.

In this solution, the array is scanned two times, Time Complexity of this algorithm is $O(n)$. Space Complexity is $O(1)$

Sixth solution: Summation formula to find the sum of n numbers from 1 to n . Subtract the values stored in the array and you will have your missing number. Time Complexity of this algorithm is $O(n)$. Space Complexity is $O(1)$

Seventh solution: XOR approach to find the sum of n numbers from 1 to n . XOR the values stored in the array and you will have your missing number. Time Complexity of this algorithm is $O(n)$. Space Complexity is $O(1)$

Example 5.12:

```
public static int findMissingNumber2(int[] arr, int size, int range) {  
    int i;  
    int xorSum = 0;  
    // get the XOR of all the numbers from 1 to range  
    for (i = 1; i <= range; i++) {  
        xorSum ^= i;  
    }  
    // loop through the array and get the XOR of elements  
    for (i = 0; i < size; i++) {  
        xorSum ^= arr[i];  
    }  
    return xorSum;  
}
```

```
    }
    return xorSum;
}
```

Eighth solution: Using set, first all the values in list to set then look for value from 1 to upperRange in the set if we do not find some value then return that value.

Example 5.13:

```
public static int findMissingNumber3(int[] arr, int size, int upperRange) {
    HashSet<Integer> st = new HashSet<Integer>();

    int i = 0;
    while (i < size) {
        st.add(arr[i]);
        i += 1;
    }
    i = 1;
    while (i <= upperRange) {
        if (st.contains(i) == false)
            return i;
        i += 1;
    }
    System.out.println("NoNumberMissing");
    return -1;
}
```

Note: Same problem can be asked in many forms (sometime you have to perform xor of the range):

1. There are numbers in the range of 1-n out of which all appears single time but there is one that appear two times.
2. All the elements in the range 1-n are appearing 16 times and one element appears 17 times. Find the element that appears 17 times.

Missing Values

Problem: Given an array, find the maximum and minimum value in the array

and also find the values in range minimum and maximum that are absent in the array.

First solution: Brute force approach, traverse the array find minimum and maximum value. Then find values from minimum to maximum in the array. Time complexity of this solution will be $O(n^2)$

Second solution: Sorting approach, we can sort the given array. Then traverse the whole array and print the missing values. Time complexity of this solution will be $O(n\log n)$ for sorting and traversal will take $O(n)$ so the overall Time complexity is $O(n\log n)$.

Example 5.14:

```
public static void MissingValues(int[] arr, int size) {  
    Arrays.sort(arr);  
    int value = arr[0];  
    int i = 0;  
    while (i < size) {  
        if (value == arr[i]) {  
            value += 1;  
            i += 1;  
        } else {  
            System.out.println(value);  
            value += 1;  
        }  
    }  
}
```

Third solution: Hashtable approach, we can traverse the array and insert its elements in a hashtable. Also in this single traversal we can find smallest and largest value in the array. Now find if the values between minimum and maximum are present in hashtable. If such values are not present then print those values.

Time complexity of this algorithm is $O(n)$. Space complexity is $O(n)$ for hashtable.

Example 5.15:

```
public static void MissingValues2(int[] arr, int size) {  
    HashSet<Integer> ht = new HashSet<Integer>();  
    int minVal = 999999;  
    int maxVal = -999999;  
  
    for (int i = 0; i < size; i++) {  
        ht.add(arr[i]);  
        if (minVal > arr[i])  
            minVal = arr[i];  
        if (maxVal < arr[i])  
            maxVal = arr[i];  
    }  
    for (int i = minVal; i < maxVal + 1; i++) {  
        if (!ht.contains(i)) {  
            System.out.println(i);  
        }  
    }  
}
```

Odd Count Element

Problem: Given an array in which all the elements appear even number of times except one, which appear odd number of times. Find the element which appear odd number of times.

First solution: XOR approach, XOR all the elements of the array the elements which appear even number of times will cancel themselves. Finally, we will get the number we are searching.

Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Second solution: Use Hashtable to keep track of frequency. Then traverse the Hashtable to find the odd number of times appearing element.

Time Complexity of this algorithm is O(n). Space Complexity is O(n)

Example 5.16

```
public static void OddCount(int[] arr, int size) {
```

```

HashMap<Integer, Integer> ctr = new HashMap<Integer, Integer>();
int count = 0;

for (int i = 0; i < size; i++) {
    if (ctr.containsKey(arr[i]))
        ctr.put(arr[i], ctr.get(arr[i]) + 1);
    else
        ctr.put(arr[i], 1);
}
for (int i = 0; i < size; i++) {
    if (ctr.containsKey(arr[i]) && (ctr.get(arr[i]) % 2 == 1)) {
        System.out.println(arr[i]);
        count++;
        ctr.remove(arr[i]);
    }
}
System.out.println("Odd count is :: " + count);
}

```

Odd Count Elements

Problem: Given an array in which all the elements appear even number of times except two, which appear odd number of times. Find the elements which appear odd number of times in O(n) time complexity and O(1) space complexity.

Solution:

- Since space complexity required is O(1) so we cannot use Hashtable.
- We know that when we xor all the elements of array then the even number of appearing elements will cancel themselves. So we have sum of the two values we are searching for.
- If we can divide the array elements in two groups such that these two values go in different groups and then xor the values in these groups then we can get these values separately.
- As shown in the algorithm below right most set bit is used to separate these two elements.

Example 5.17:

```

public static void OddCount2(int[] arr, int size) {
    int xorSum = 0;
    int first = 0;
    int second = 0;
    int setBit;

    /*
     * xor of all elements in arr[] even occurrence will cancel each other.
     * sum will contain sum of two odd elements.
     */

    for (int i = 0; i < size; i++)
        xorSum = xorSum ^ arr[i];

    /* Rightmost set bit.*/
    setBit = xorSum & ~(xorSum - 1);

    /*
     * Dividing elements in two group: Elements having setBit bit as 1. Elements
     * having setBit bit as 0. Even elements cancelled themselves if group and we
     * get our numbers.
     */

    for (int i = 0; i < size; i++) {
        if ((arr[i] & setBit) != 0)
            first ^= arr[i];
        else
            second ^= arr[i];
    }
    System.out.println(first + second);
}

```

Sum Distinct

Problem: Given an array of size N, the elements in the array may be repeated. You need to find sum of distinct elements of the array. If there is some value repeated continuously then they should be added once.

Example 5.18:

```
public static void SumDistinct(int[] arr, int size) {  
    int sum = 0;  
    Arrays.sort(arr);  
    for (int i = 0; i < (size - 1); i++) {  
        if (arr[i] != arr[i + 1])  
            sum += arr[i];  
    }  
    sum += arr[size - 1];  
    System.out.println(sum);  
}
```

Analysis: Sort the input array. Duplicate values will come adjacent. Create a sum variable and add only those values to it which are not equal to its next value.

Time Complexity of this algorithm is $O(n)$. Space Complexity is $O(1)$

Two elements whose sum is closest to zero

Problem: In given List of integers, both +ve and -ve. You need to find the two elements such that their sum is closest to zero.

First solution: Exhaustive search or Brute force; for each element in the array find the other element whose value when added will give minimum absolute value. This is done using two loops, first loop to select the element and second loop to find the element that should be added to it so that the absolute of the sum will be minimum or close to zero.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 5.19:

```
public static void minAbsSumPair(int[] arr, int size) {  
    int l, r, minSum, sum, minFirst, minSecond;  
    // Array should have at least two elements  
    if (size < 2) {
```

```

        System.out.println("Invalid Input");
        return;
    }
    // Initialization of values
    minFirst = 0;
    minSecond = 1;
    minSum = Math.abs(arr[0] + arr[1]);
    for (l = 0; l < size - 1; l++) {
        for (r = l + 1; r < size; r++) {
            sum = Math.abs(arr[l] + arr[r]);
            if (sum < minSum) {
                minSum = sum;
                minFirst = l;
                minSecond = r;
            }
        }
    }
    System.out.println(" Minimum sum elements are : " + arr[minFirst] + " , " +
arr[minSecond]);
}

```

Second solution: Sorting

Steps are as follows:

1. Sort all the elements in the array.
2. Take two variable `firstIndex = 0` and `secondIndex = size - 1`
3. Compute `sum = arr[firstIndex]+arr[secondIndex]`
4. If the sum is equal to 0 then we have the solution
5. If the sum is less than 0 then we will increase first
6. If the sum is greater than 0 then we will decrease the second
7. We repeat the above process 3 to 6, until we get the desired pair or we get `first >= second`

Example 5.20:

```

public static void minAbsSumPair2(int[] arr, int size) {
    int l, r, minSum, sum, minFirst, minSecond;
    // Array should have at least two elements
    if (size < 2) {

```

```

System.out.println("Invalid Input");
return;
}
Arrays.sort(arr);

// Initialization of values
minFirst = 0;
minSecond = size - 1;
minSum = Math.abs(arr[minFirst] + arr[minSecond]);
for (l = 0, r = size - 1; l < r;) {
    sum = (arr[l] + arr[r]);
    if (Math.abs(sum) < minSum) {
        minSum = Math.abs(sum);
        minFirst = l;
        minSecond = r;
    }
    if (sum < 0) {
        l++;
    } else if (sum > 0) {
        r--;
    } else {
        break;
    }
}
System.out.println(" Minimum sum pair : " + arr[minFirst] + " , " +
arr[minSecond]);
}

```

Time Complexity is O(nlogn) and Space Complexity is O(1)

Find Pair in an array

Problem: Given an array of n numbers, find two elements such that their sum is equal to “value”

First solution: Exhaustive search or Brute force, for each element in list find if there is some other element, which sums up to the desired value. This is done

using two loops, first loop is to select the element and second loop is to find another element.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 5.21

```
public static boolean FindPair(int[] arr, int size, int value) {  
    int i, j;  
    for (i = 0; i < size; i++) {  
        for (j = i + 1; j < size; j++) {  
            if ((arr[i] + arr[j]) == value) {  
                System.out.println("Pair is: " + arr[i] + "," + arr[j]);  
                return true;  
            }  
        }  
    }  
    return false;  
}
```

Second solution: Sorting, Steps are as follows:

1. Sort all the elements in the array.
2. Take two variable first and second. Variable first= 0 and second = size -1
3. Compute sum = arr[first]+arr[second]
4. If the sum is equal to the desired value, then we have the solution
5. If the sum is less than the desired value, then we will increase the first
6. If the sum is greater than the desired value, then we will decrease the second
7. We repeat the above process until we get the desired pair or we get first \geq second

Sorting algorithms takes $O(n.\log n)$ time and single scan takes $O(n)$ time.

Time Complexity is $O(n.\log n)$ and Space Complexity is $O(1)$

Example 5.22:

```
public static boolean FindPair2(int[] arr, int size, int value) {  
    int first = 0, second = size - 1;  
    int curr;
```

```

Arrays.sort(arr);
while (first < second) {
    curr = arr[first] + arr[second];
    if (curr == value) {
        System.out.println("Pair is " + arr[first] + "," + arr[second]);
        return true;
    } else if (curr < value) {
        first++;
    } else {
        second--;
    }
}
return false;
}

```

Third solution: Hash-Table, using Hash-Table; we can keep track of the elements we have already seen and we can find the pair in just one scan.

1. For each element, insert the value in Hashtable. Let's say current value is arr[index]
2. If value - arr[index] is in the Hashtable then we have the desired pair.
3. Else, proceed to the next entry in the array.

Hash-Table insertion and finding take constant time O(1) so the total time complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Example 5.23:

```

public static boolean FindPair3(int[] arr, int size, int value) {
    HashSet<Integer> hs = new HashSet<Integer>();
    int i;
    for (i = 0; i < size; i++) {
        if (hs.contains(value - arr[i])) {
            System.out.println("The pair is : " + arr[i] + " , " + (value - arr[i]));
            return true;
        }
        hs.add(arr[i]);
    }
    return false;
}

```

}

Forth solution: Counting approach, this approach is only possible if we know the range of the input. If we know that, the elements in the array are in the range 0 to n. We can reserve an array of length n and when we see an element, we can increase its count. In place of the Hashtable in the above approach, we will use this list and will find out the pair.

Counting approach just uses an array so insertion and finding take constant time O(1) so the total time complexity of the algorithm is O(n) time. Space Complexity for creating count list is also O(n)

Find the Pair in two Lists

Problem: Given two list X and Y. Find a pair of elements (x_i, y_i) such that $x_i \in X$ and $y_i \in Y$ where $x_i + y_i = \text{value}$.

First solution: Exhaustive search or Brute force; loop through element x_i of X and see if you can find $(\text{value} - x_i)$ in Y. This is done using two loops, first loop is to select an element from X and second loop is to find its corresponding value in Y.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Second solution: Sorting; Sort all the elements in the second list Y. For each element of X you can see if that element is there in Y by using binary search.

Sorting algorithms take $O(m \cdot \log m)$ and searching will take $O(n \cdot \log m)$ time. Time Complexity is $O(n \cdot \log m)$ or $O(m \cdot \log m)$ and Space Complexity is $O(1)$

Third solution: Sorting, Steps are as follows:

1. Sort the elements of both X and Y in increasing order.
2. Take the sum of the smallest element of X and the largest element of Y.
3. If the sum is equal to the value, we got our pair.
4. If the sum is smaller than value, take next element of X
5. If the sum is greater than value, take the previous element of Y

Sorting algorithms take $O(n \log n) + O(m \log m)$ for sorting and searching will take $O(n+m)$ time.

Time Complexity is $O(n \log n)$ and Space Complexity is $O(1)$

Forth solution: Hash-Table, Steps are as follows:

1. Scan through all the elements in the array Y and insert them into Hashtable.
2. Now scan through all the elements of list X, let us suppose the current element is x_i see if you can find $(value - x_i)$ in the Hashtable.
3. If you find the value, you got your pair.
4. If not, then go to the next value in the array X.

Hash-Table insertion and finding take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity is also $O(n)$

Fifth solution: Counting; this approach is only possible if we know the range of the input. Same as Hashtable implementation just use a simple list in place of Hashtable and you are done.

Counting approach just uses an array so insertion and finding take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity for creating count list is also $O(n)$

Find Difference Pair

Problem: In an array of positive integers, find a pair whose absolute value of difference is equal to a given value.

First solution: Brute force, find all the possible pair and find if absolute value of their difference is equal to the given value. Time complexity will $O(n^2)$

Example 5.24:

```
public static boolean FindDifference(int arr[], int size, int value) {  
    for (int i = 0; i < size; i++) {  
        for (int j = i + 1; j < size; j++) {  
            if (Math.abs(arr[i] - arr[j]) == value) {  
                System.out.println("Pair is:: " + arr[i] + " & " + arr[j]);  
                return true;  
            }  
        }  
    }  
}
```

```

    }
}
return false;
}

```

Second solution: Using sorting, the performance can be improved by sorting the array. We take two index from start of the array at index 0, call them first and second. When difference of values is less than the desired value then we increase the second index or if the value if difference is grater then the desired value we increase the first index. Time complexity is $O(n\log n)$ which is for sorting operation.

Example 5.25:

```

public static boolean FindDifference2(int arr[], int size, int value) {
    int first = 0;
    int second = 0;
    int diff;
    Arrays.sort(arr);
    while (first < size && second < size) {
        diff = Math.abs(arr[first] - arr[second]);
        if (diff == value) {
            System.out.println("Pair is::" + arr[first] + " & " + arr[second]);
            return true;
        } else if (diff > value)
            first += 1;
        else
            second += 1;
    }
    return false;
}

```

Find Min Diff

Problem: Given an array of integers, find the element pair with minimum difference.

First solution: you can always pick two elements, find difference between them,

and find the elements with minimum difference using two loops and comparing each pair. Time complexity is $O(n^2)$

Second solution: This performance can be improved by sorting the array. Since we need minimum sum so the pair which we are searching is adjacent to each other. Time complexity is $O(n \log n)$

Example 5.26

```
public static int findMinDiff(int[] arr, int size) {  
    Arrays.sort(arr);  
    int diff = 9999999;  
    for (int i = 0; i < (size - 1); i++) {  
        if ((arr[i + 1] - arr[i]) < diff)  
            diff = arr[i + 1] - arr[i];  
    }  
    return diff;  
}
```

Minimum Difference Pair

Problem: Given two array, find minimum difference pair such that it should take one element from each array.

First solution: Brute force solution, you can always pick two elements, one from the first array and another from second array. Then find their difference to find minimum difference. Time complexity is $O(nm)$

Second solution: This performance can be improved by sorting the arrays. Since we need minimum difference pair. We will pick one element from first array and find its difference from one element from second array. If the difference is negative then we will increase index of first array and if the difference is positive then we will increase index of second array. Time Complexity is $O(n \log n + m \log m)$ for sorting and $O(n+m)$ for comparison.
Total time complexity: $O(n \log n + m \log m)$

Example 5.27

```
public static int MinDiffPair(int arr1[], int size1, int arr2[], int size2) {
```

```

int minDiff = 9999999;
int first = 0;
int second = 0;
int out1 = 0, out2 = 0, diff;
Arrays.sort(arr1);
Arrays.sort(arr2);
while (first < size1 && second < size2) {
    diff = Math.abs(arr1[first] - arr2[second]);
    if (minDiff > diff) {
        minDiff = diff;
        out1 = arr1[first];
        out2 = arr2[second];
    }
    if (arr1[first] < arr2[second])
        first += 1;
    else
        second += 1;
}
System.out.println("The pair is :: " + out1 + out2);
System.out.println("Minimum difference is :: " + minDiff);
return minDiff;
}

```

Min Absolute Sum

Problem: Given an array of integers with both +ve and -ve values. Find the two elements in the array such that their sum is minimum (closest to zero).

First solution: Brute force solution, you can find all the possible pairs using two loops. Then find their sum and find it's absolute to find minimum absolute sum. Time complexity is $O(n^2)$

Example 5.28

```

public static void minAbsSumPair(int[] arr, int size) {
    int l, r, minSum, sum, minFirst, minSecond;
    // Array should have at least two elements
    if (size < 2) {

```

```

        System.out.println("Invalid Input");
        return;
    }
    // Initialization of values
    minFirst = 0;
    minSecond = 1;
    minSum = Math.abs(arr[0] + arr[1]);
    for (l = 0; l < size - 1; l++) {
        for (r = l + 1; r < size; r++) {
            sum = Math.abs(arr[l] + arr[r]);
            if (sum < minSum) {
                minSum = sum;
                minFirst = l;
                minSecond = r;
            }
        }
    }
    System.out.println(" Minimum sum elements are : " + arr[minFirst] + " , " +
arr[minSecond]);
}

```

Second solution: This performance can be improved by sorting the arrays. Since we need minimum sum pair. We will pick one element from start of array and another from end of array. Add this pair and calculate sum, also find absolute value of sum. If absolute value is smaller than the minimum absolute value then make this value as new minimum absolute value. If the sum is negative then we will increase index of start array and if the difference is positive then we decrease index from the end of array.

Time complexity is $O(n \log n)$ for sorting and $O(n)$ for comparison.

Total time complexity: $O(n \log n)$

Example 5.29

```

public static void minAbsSumPair2(int[] arr, int size) {
    int l, r, minSum, sum, minFirst, minSecond;
    // Array should have at least two elements
    if (size < 2) {

```

```

System.out.println("Invalid Input");
return;
}
Arrays.sort(arr);

// Initialization of values
minFirst = 0;
minSecond = size - 1;
minSum = Math.abs(arr[minFirst] + arr[minSecond]);
for (l = 0, r = size - 1; l < r;) {
    sum = (arr[l] + arr[r]);
    if (Math.abs(sum) < minSum) {
        minSum = Math.abs(sum);
        minFirst = l;
        minSecond = r;
    }
    if (sum < 0) {
        l++;
    } else if (sum > 0) {
        r--;
    } else {
        break;
    }
}
System.out.println(" Minimum sum pair : " + arr[minFirst] + " , " +
arr[minSecond]);
}

```

Closest Pair

Problem: Given an array of positive integers and a number. You need to find a pair in array whose sum is closest to given number.

First solution: Brute force, for each pair find their sum and get its absolute difference from the given value. This can be done using two loops. Time complexity is $O(n^2)$

Example 5.30

```
public static void ClosestPair(int arr[], int size, int value) {  
    int diff = 999999;  
    int first = -1;  
    int second = -1;  
    int curr;  
    for (int i = 0; i < size; i++) {  
        for (int j = i + 1; j < size; j++) {  
            curr = Math.abs(value - (arr[i] + arr[j]));  
            if (curr < diff) {  
                diff = curr;  
                first = arr[i];  
                second = arr[j];  
            }  
        }  
    }  
    System.out.println("closest pair is ::" + first + second);  
}
```

Second solution: Sorting, performance of above solution can be improved by sorting the array. Since we need a pair whose sum is closest to the given value. We start two index one from start and other from end. Add these two index values call it current. If the current sum is greater than the given value then decrease the end index and if the current sum is smaller than the value then we increase the start index.

Time complexity is $O(n\log n)$ for sorting and $O(n)$ for searching pair. So overall Time complexity is $O(n\log n)$

Example 5.31

```
public static void ClosestPair2(int arr[], int size, int value) {  
    int first = 0, second = 0;  
    int start = 0;  
    int stop = size - 1;  
    int diff, curr;  
    Arrays.sort(arr);  
    diff = 9999999;
```

```

{
while (start < stop) {
curr = (value - (arr[start] + arr[stop]));
if (Math.abs(curr) < diff) {
diff = Math.abs(curr);
first = arr[start];
second = arr[stop];
}
if (curr == 0) {
break;
} else if (curr > 0) {
start += 1;
} else {
stop -= 1;
}
}
System.out.println("closest pair is :: " + first + second);
}

```

Sum of Pair Equal to Rest Array

Problem: Given an array find if there is a pair whose sum is equal to the sum of rest of the elements of the array.

Solution: Sort the array. Sum all the elements of the array call this value total. Find a pair in the sorted array whose sum is total/2. Total time complexity is O(nlogn).

Example 5.32

```

public static boolean SumPairRestArray(int[] arr, int size) {
int total, low, high, curr, value;
Arrays.sort(arr);
total = 0;
for (int i = 0; i < size; i++)
total += arr[i];
value = total / 2;

```

```

low = 0;
high = size - 1;

while (low < high) {
    curr = arr[low] + arr[high];
    if (curr == value) {
        System.out.println("Pair is :: " + arr[low] + arr[high]);
        return true;
    } else if (curr < value)
        low += 1;
    else
        high -= 1;
}
return false;
}

```

Zero Sum Triplets

Problem: Given an array of integers, you need to find a triplet whose sum 0.

First solution: Brute force, for each triplet find their sum. This can be done using three loops. Time complexity is $O(n^3)$

Example 5.33:

```

public static void ZeroSumTriplets(int[] arr, int size) {

    for (int i = 0; i < (size - 2); i++) {
        for (int j = i + 1; j < (size - 1); j++) {
            for (int k = j + 1; k < size; k++) {
                if (arr[i] + arr[j] + arr[k] == 0)
                    System.out.println("Triplet :: " + arr[i] + arr[j] + arr[k]);
            }
        }
    }
}

```

Second solution: We can increase performance by sorting. Sort the given array.

Select element from the array in a loop and then find other two values such that their sum is negative of the first value.

Time complexity is $O(n^2)$

Example 5.34:

```
public static void ZeroSumTriplets2(int[] arr, int size) {  
    int start, stop, i;  
    Arrays.sort(arr);  
    for (i = 0; i < (size - 2); i++) {  
        start = i + 1;  
        stop = size - 1;  
  
        while (start < stop) {  
            if (arr[i] + arr[start] + arr[stop] == 0) {  
                System.out.println("Triplet :: " + arr[i] + arr[start] + arr[stop]);  
                start += 1;  
                stop -= 1;  
            } else if (arr[i] + arr[start] + arr[stop] > 0)  
                stop -= 1;  
            else  
                start += 1;  
        }  
    }  
}
```

Find Triplet

Problem: Given an array of integers, you need to find a triplet whose sum equal to given value.

First solution: Brute force, for each triplet find their sum. This can be done using three loops. Time complexity is $O(n^3)$

Example 5.35:

```
public static void findTriplet(int arr[], int size, int value) {  
    for (int i = 0; i < (size - 2); i++)  
        for (int j = i + 1; j < (size - 1); j++)
```

```

for (int k = j + 1; k < size; k++) {
    if ((arr[i] + arr[j] + arr[k]) == value)
        System.out.println("Triplet :: " + arr[i] + arr[j] + arr[k]);
}
}

```

Second solution: We can increase performance by sorting. Sort the given array. Select first element from the array in a loop and then find other two values such that sum of all three is equal to the given value. Time complexity is $O(n^2)$

Example 5.36:

```

public static void findTriplet2(int arr[], int size, int value) {
    int start, stop;
    Arrays.sort(arr);
    for (int i = 0; i < size - 2; i++) {
        start = i + 1;
        stop = size - 1;
        while (start < stop) {
            if (arr[i] + arr[start] + arr[stop] == value) {
                System.out.println("Triplet ::" + arr[i] + arr[start] + arr[stop]);
                start += 1;
                stop -= 1;
            } else if (arr[i] + arr[start] + arr[stop] > value)
                stop -= 1;
            else
                start += 1;
        }
    }
}

```

A+B = C Triplet

Problem: Given an array of integers, you need to find a triplet such that sum of two elements of triplet is equal to the third value. We need to find triplet (A, B, C) such that $A+B=C$.

First solution: Brute force, for each triplet find if constraint $A+B=C$ satisfies.

This can be done using three loops. Time complexity is $O(n^3)$

Second solution: We can increase performance by sorting. Sort the given array in decreasing order. Select first element from the array in a loop and then find other two values such that sum of these two is equal to the first value.

Time complexity is $O(n^2)$

Example 5.37:

```
public static void ABCTriplet(int[] arr, int size) {  
    int start, stop;  
    Arrays.sort(arr);  
  
    for (int i = 0; i < (size - 2); i++) {  
        start = i + 1;  
        stop = size - 1;  
  
        while (start < stop) {  
            if (arr[i] == arr[start] + arr[stop]) {  
                System.out.println("Triplet ::" + arr[i] + ", " + arr[start] + ", " + arr[stop]);  
                start += 1;  
                stop -= 1;  
            } else if (arr[i] > arr[start] + arr[stop])  
                stop -= 1;  
            else  
                start += 1;  
        }  
    }  
}
```

Smaller than triplets Count

Problem: Given an array of integers, you need to find a triplet such that sum of elements of triplet is less than the value. We need to find triplets (A, B, C) such that $A+B+C < \text{value}$.

First solution: Brute force, for each triplet find if constraint $A+B+C < \text{value}$ satisfies. This can be done using three loops. Time complexity is $O(n^3)$

Second solution: We can increase performance by sorting. Sort the given array. Select first element from the array in a loop and then find other two values such that sum of all three is less than the given value. Time complexity is $O(n^2)$

Example 5.38:

```
public static void SmallerThenTripletCount(int arr[], int size, int value) {  
    int start, stop;  
    int count = 0;  
    Arrays.sort(arr);  
  
    for (int i = 0; i < (size - 2); i++) {  
        start = i + 1;  
        stop = size - 1;  
        while (start < stop) {  
            if (arr[i] + arr[start] + arr[stop] >= value)  
                stop -= 1;  
            else {  
                count += stop - start;  
                start += 1;  
            }  
        }  
    }  
    System.out.println(count);  
}
```

Arithmetic progression triplet

Problem: Given a sorted array find all Arithmetic progression triplet possible.

Example 5.39:

```
public static void APTriplets(int[] arr, int size) {  
    int i, j, k;  
    for (i = 1; i < size - 1; i++) {  
        j = i - 1;  
        k = i + 1;  
        while (j >= 0 && k < size) {
```

```
if (arr[j] + arr[k] == 2 * arr[i]) {  
    System.out.println("Triplet ::" + arr[j] + arr[i] + arr[k]);  
    k += 1;  
    j -= 1;  
} else if (arr[j] + arr[k] < 2 * arr[i])  
    k += 1;  
else  
    j -= 1;  
}  
}  
}
```

Geometric Progression Triplet

Problem: Given a sorted array find all geometric progression triplet possible.

Example 5.40:

```
public static void GPTriplets(int[] arr, int size) {  
    int i, j, k;  
    for (i = 1; i < size - 1; i++) {  
        j = i - 1;  
        k = i + 1;  
        while (j >= 0 && k < size) {  
            if (arr[j] * arr[k] == arr[i] * arr[i]) {  
                System.out.println("Triplet is :: " + arr[j] + arr[i] + arr[k]);  
                k += 1;  
            } else if (arr[j] + arr[k] < 2 * arr[i])  
                k += 1;  
            else  
                j -= 1;  
        }  
    }  
}
```

Number of Triangles

Problem: Given an array of positive integers representing edges of triangles. Find the number of triangles that can be formed from these elements representing sides of triangles. For a triangle sum of two edges is always greater than third edge.

Input: [1, 2, 3, 4, 5]

Output: 3, Corresponds to (2, 3, 4) (2, 4, 5) (3, 4, 5)

First solution: Brute force solution, by picking all the triplets and then checking if the triangle property holds. Time Complexity is $O(n^3)$

Example 5.41:

```
public static int numberOfTriangles(int[] arr, int size) {  
    int i, j, k, count = 0;  
    for (i = 0; i < (size - 2); i++) {  
        for (j = i + 1; j < (size - 1); j++) {  
            for (k = j + 1; k < size; k++) {  
                if (arr[i] + arr[j] > arr[k])  
                    count += 1;  
            }  
        }  
    }  
    return count;  
}
```

Second solution: This solution takes advantage of one simple property. In a sorted array, If sum of $arr[i]$ & $arr[j]$ is greater than $arr[k]$ then sum of $arr[i]$ & $arr[j+1]$ is also greater than $arr[k]$. This improvement make time complexity of this algorithm as $O(n^2)$

Example 5.42:

```
public static int numberOfTriangles2(int[] arr, int size) {  
    int i, j, k, count = 0;  
    Arrays.sort(arr);  
  
    for (i = 0; i < (size - 2); i++) {  
        k = i + 2;  
        for (j = i + 1; j < (size - 1); j++) {
```

```

/*
 * if sum of arr[i] & arr[j] is greater arr[k] then sum of arr[i] & arr[j + 1] is
also greater than arr[k] this improvement make algo * O(n2)
*/
while (k < size && arr[i] + arr[j] > arr[k])
k += 1;

count += k - j - 1;
}
}
return count;
}

```

Find max, appearing element in an array

Problem: In given list of n numbers, find the element, which appears maximum number of times.

First solution: Exhaustive search or Brute force; for each element in array, find how many times this particular value appears in array. Keep track of the maxCount and when some element count is greater than maxCount then update the maxCount. This is done using two loops, first loop to select the element and second loop to count the occurrence of that element.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 5.43:

```

public static int getMax(int[] arr, int size) {
    int i, j;
    int max = arr[0], count = 1, maxCount = 1;
    for (i = 0; i < size; i++) {
        count = 1;
        for (j = i + 1; j < size; j++) {
            if (arr[i] == arr[j]) {
                count++;
            }
        }
    }
}
```

```

if (count > maxCount) {
    max = arr[i];
    maxCount = count;
}
}
return max;
}

```

Second solution: Sorting; Sort all the elements in the array. In a single scan, we can find the counts. Sorting algorithms takes $O(n \log n)$ time and single scan takes $O(n)$ time.

Time Complexity is $O(n \log n)$ and Space Complexity is $O(1)$

Example 5.44:

```

public static int getMax2(int[] arr, int size) {
    int max = arr[0], maxCount = 1;
    int curr = arr[0], currCount = 1;
    int i;
    Arrays.sort(arr); // Sort(arr, size);
    for (i = 1; i < size; i++) {
        if (arr[i] == arr[i - 1]) {
            currCount++;
        } else {
            currCount = 1;
            curr = arr[i];
        }
        if (currCount > maxCount) {
            maxCount = currCount;
            max = curr;
        }
    }
    return max;
}

```

Third solution: Counting, This approach is possible only if we know the range of the input. If we know that, the elements in the array are in the range 0 to $n-1$.

We can traverse an array of length n and when we see an element, we can increase its count. In just one single scan, we know the duplicates. If we know the range of the elements, then this is the fastest way to find the max count.

Counting approach just uses list so to increase count take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity for creating count list is also $O(n)$

Example 5.45:

```
public static int getMax3(int[] arr, int size, int range) {  
    int max = arr[0], maxCount = 1;  
    int[] count = new int[range];  
    int i;  
    for (i = 0; i < size; i++) {  
        count[arr[i]]++;  
        if (count[arr[i]] > maxCount) {  
            maxCount = count[arr[i]];  
            max = arr[i];  
        }  
    }  
    return max;  
}
```

Majority element in an array

Problem: In given an array of n elements. Find the majority element, which appears more than $n/2$ times. Return 0 in case there is no majority element.

First solution: Exhaustive search or Brute force, for each element in array find how many times its value appears in array. Keep track of the maxCount and when some element count is greater than maxCount then update the maxCount . This is done using two loops, first loop to select the element and second loop to count the occurrence of that element. If the final maxCount is greater than $n/2$ we have a majority otherwise we do not have any majority.

Time Complexity is $O(n^2) + O(1) = O(n^2)$ and Space Complexity is $O(1)$

Example 5.46:

```
public static int getMajority(int[] arr, int size) {  
    int i, j;  
    int max = 0, count = 0, maxCount = 0;  
    for (i = 0; i < size; i++) {  
        for (j = i + 1; j < size; j++) {  
            if (arr[i] == arr[j]) {  
                count++;  
            }  
        }  
        if (count > maxCount) {  
            max = arr[i];  
            maxCount = count;  
        }  
    }  
    if (maxCount > size / 2) {  
        return max;  
    } else {  
        return 0;  
    }  
}
```

Second solution: Sorting, Sort all the elements in the array. If there is a majority then the middle element at the index $n/2$ must be the majority number. So, just single scan can be used to find its count and see if the majority is there or not.

Sorting algorithms take $O(n.\log n)$ time and single scan take $O(n)$ time.
Time Complexity is $O(n.\log n)$ and Space Complexity is $O(1)$

Example 5.47:

```
public static int getMajority2(int[] arr, int size) {  
    int majIndex = size / 2, count = 1;  
    int i;  
    int candidate;  
    Arrays.sort(arr); // Sort(arr,size);  
    candidate = arr[majIndex];  
    count = 0;
```

```

for (i = 0; i < size; i++) {
    if (arr[i] == candidate) {
        count++;
    }
}
if (count > size / 2) {
    return arr[majIndex];
} else {
    return Integer.MIN_VALUE;
}
}

```

Third solution: This is a cancellation approach (Moore's Voting Algorithm), if all the elements stand against the majority and each element is cancelled with one element of majority, if there is majority then majority prevails.

- Set the first element of the array as majority candidate and initialize the count to be 1.
- Start scanning the array.
 - o If we get some element whose value is same as a majority candidate, then we increase the count.
 - o If we get an element whose value is different from the majority candidate, then we decrement the count.
 - o If count become 0, that means we have a new majority candidate. Make the current candidate as majority candidate and reset count to 1.
 - o At the end, we will have the only probable majority candidate.
- Now scan through the array once again to see if that candidate we found above have appeared more than $n/2$ times.

Counting approach just scans throw list two times. Time Complexity is $O(n)$ time. Space Complexity for creating count list is also $O(1)$

Example 5.48:

```

public static int getMajority3(int[] arr, int size) {
    int majIndex = 0, count = 1;
    int i;
    int candidate;
    for (i = 1; i < size; i++) {

```

```

if (arr[majIndex] == arr[i]) {
    count++;
} else {
    count--;
}
if (count == 0) {
    majIndex = i;
    count = 1;
}
candidate = arr[majIndex];
count = 0;
for (i = 0; i < size; i++) {
    if (arr[i] == candidate) {
        count++;
    }
}
if (count > size / 2) {
    return arr[majIndex];
} else {
    return 0;
}
}

```

Is Majority

Problem: Given a sorted array find if there is a majority and find the majority element.

First solution: Using brute force traversal of the array we can find the number of occurrence of the middle element in the array. If number of occurrences is not greater than or equal to ceil of count/2 then there is no majority.

Time complexity is $O(N)$

Second solution: Since the array is sorted our first thought should be binary search. We can find the probable majority candidate at size/2 location. Then we need to find its first occurrence in the sorted array say it index i. Then once we

have index i then if majority exists then element at index $i+n/2$ also has value same as majority candidate.

Time complexity is $O(\log N)$

Example 5.49:

```
/* Using binary search method.*/
public static int FirstIndex(int arr[], int size, int low, int high, int value)
{
    int mid=0;
    if (high >= low)
        mid = (low + high) / 2;

    /*
    Find first occurrence of value, either it should be the first
    element of the array or the value before it is smaller than it.
    */
    if ((mid == 0 || arr[mid - 1] < value) && (arr[mid] == value))
        return mid;
    else if (arr[mid] < value)
        return FirstIndex(arr, size, mid + 1, high, value);
    else
        return FirstIndex(arr, size, low, mid - 1, value);
}

public static boolean isMajority(int arr[], int size)
{
    int i;
    int majority = arr[size / 2];
    i = FirstIndex(arr, size, 0, size - 1, majority);
    /*
    we are using majority element from array so
    we will get some valid index always.
    */
    if (((i + size / 2) <= (size - 1)) && arr[i + size / 2] == majority)
        return true;
    else
        return false;
```

}

Find 2nd largest number in an array with minimum comparisons

Problem: Suppose you are given an unsorted list of n distinct elements. How will you identify the second largest element with minimum number of comparisons?

First solution: Find the largest element in the array. Then replace the last element with the largest element. Then search the second largest element in the remaining $n-1$ elements.

The total number of comparisons is: $(n-1) + (n-2)$

Second solution: Sort the array and then give the $(n-1)$ element. This approach is still more inefficient.

Third solution: Using priority queue / Heap; in this approach, we will look into heap chapter. Use buildHeap() function to build heap from the array. This is done in n comparisons. $\text{Arr}[0]$ is the largest number, and the greater among $\text{arr}[1]$ and $\text{arr}[2]$ is the second largest.

The total number of comparisons are: $(n-1) + 1 = n$

Find a median of an array

Problem: In an unsorted list of numbers of size n , if all the elements of the array are sorted then find the element, which lie at the index $n/2$.

First solution: Sort the array and return the element in the middle.

Sorting algorithms takes $O(n \cdot \log n)$.

Time Complexity is $O(n \cdot \log n)$ and Space Complexity is $O(1)$

Example 5.50:

```
public static int getMedian(int[] arr, int size) {  
    Arrays.sort(arr);  
    return arr[size / 2];  
}
```

Second solution: Use QuickSelect algorithm. This algorithm we will look in the next chapter.

The Average case time complexity of this algorithm will be $O(n)$

Find maxima in a bitonic list

Problem: A bitonic list comprises of an increasing sequence of integers immediately followed by a decreasing sequence of integers.

First solution: Sequential search, traverse through the array and find a point at which the next value is less than the current value. This is the maxima and return this value.

Second solution: Binary search method, since the elements are sorted in some order, we should go for algorithm similar to binary search. The steps are as follows:

1. Take two variable for storing start and end index. Variable start=0 and end=size-1
2. Find the middle element of the array.
3. See if the middle element is the maxima. If yes, return the middle element.
4. Alternatively, if the middle element is in increasing part, then we need to look for in mid+1 and end.
5. Alternatively, if the middle element is in the decreasing part, then we need to look in the start and mid-1.
6. Repeat step 2 to 5 until we get the maxima.

Example 5.51:

```
public static int SearchBotonicArrayMax(int[] arr, int size) {  
    int start = 0, end = size - 1;  
    int mid = (start + end) / 2;  
    int maximaFound = 0;  
    if (size < 3) {  
        System.out.println("error");  
        return 0;  
    }
```

```

while (start <= end) {
    mid = (start + end) / 2;
    if (arr[mid - 1] < arr[mid] && arr[mid + 1] < arr[mid])// maxima
    {
        maximaFound = 1;
        break;
    } else if (arr[mid - 1] < arr[mid] && arr[mid] < arr[mid + 1])// increasing
    {
        start = mid + 1;
    } else if (arr[mid - 1] > arr[mid] && arr[mid] > arr[mid + 1])// decreasing
    {
        end = mid - 1;
    } else {
        break;
    }
}
if (maximaFound == 0) {
    System.out.println("error");
    return 0;
}
return arr[mid];
}

```

Note: - This algorithm works with strictly increasing then decreasing bitonic array. If there are repetitive values then this algorithm will fail.

Search element in a bitonic list

Problem: A bitonic list comprises of an increasing sequence of integers immediately followed by a decreasing sequence of integers. Find an element in a bitonic list.

Solution: To search an element in a bitonic list:

1. Find the index or maximum element in the array. By finding the end of increasing part of the array, using binary search.
2. Once we have the maximum element, search the given value in increasing part of the array using binary search.

3. If the value is not found in increasing part, search the same value in decreasing part of the array using binary search.

Example 5.52:

```
public static int SearchBitonicArray(int[] arr, int size, int key) {  
    int max = FindMaxBitonicArray(arr, size);  
    int k = BinarySearch(arr, 0, max, key, true);  
    if (k != -1) {  
        return k;  
    } else {  
        return BinarySearch(arr, max + 1, size - 1, key, false);  
    }  
}  
  
public static int FindMaxBitonicArray(int[] arr, int size) {  
    int start = 0, end = size - 1, mid;  
    if (size < 3) {  
        System.out.println("error");  
        return -1;  
    }  
    while (start <= end) {  
        mid = (start + end) / 2;  
        if (arr[mid - 1] < arr[mid] && arr[mid + 1] < arr[mid])// maximum  
        {  
            return mid;  
        } else if (arr[mid - 1] < arr[mid] && arr[mid] < arr[mid + 1])// increasing  
        {  
            start = mid + 1;  
        } else if (arr[mid - 1] > arr[mid] && arr[mid] > arr[mid + 1])// increasing  
        {  
            end = mid - 1;  
        } else {  
            break;  
        }  
    }  
    System.out.println("error");  
    return -1;
```

```
}
```

Occurrence counts in sorted List

Problem: Given a sorted list arr[] find the number of occurrences of a number.

First solution: Brute force, Traverse the array and in linear time we will get the occurrence count of the number. This is done using one loop.

Time Complexity is O(n) and Space Complexity is O(1)

Example 5.53:

```
public static int findKeyCount(int[] arr, int size, int key) {  
    int i, count = 0;  
    for (i = 0; i < size; i++) {  
        if (arr[i] == key) {  
            count++;  
        }  
    }  
    return count;  
}
```

Second solution: Since we have sorted list, we should think about some binary search.

1. First, we should find the first occurrence of the key.
2. Then we should find the last occurrence of the key.
3. Take the difference of these two values and you will have the solution.

Example 5.54:

```
public static int findKeyCount2(int[] arr, int size, int key) {  
    int firstIndex, lastIndex;  
    firstIndex = findFirstIndex(arr, 0, size - 1, key);  
    lastIndex = findLastIndex(arr, 0, size - 1, key);  
    return (lastIndex - firstIndex + 1);  
}
```

```
public static int findFirstIndex(int[] arr, int start, int end, int key) {
```

```

int mid;
if (end < start) {
    return -1;
}
mid = (start + end) / 2;
if (key == arr[mid] && (mid == start || arr[mid - 1] != key)) {
    return mid;
}
if (key <= arr[mid])// <= is us the number.t in sorted array.
{
    return findFirstIndex(arr, start, mid - 1, key);
} else {
    return findFirstIndex(arr, mid + 1, end, key);
}
}

```

```

public static int findLastIndex(int[] arr, int start, int end, int key) {
    int mid;
    if (end < start) {
        return -1;
    }
    mid = (start + end) / 2;
    if (key == arr[mid] && (mid == end || arr[mid + 1] != key)) {
        return mid;
    }
    if (key < arr[mid])// <
    {
        return findLastIndex(arr, start, mid - 1, key);
    } else {
        return findLastIndex(arr, mid + 1, end, key);
    }
}

```

Stock purchase-sell

Problem: In given list, in which nth element is the price of the stock on nth day. You are asked to buy once and sell once, on what date you will be buying and at

what date you will be selling to get maximum profit.

Or

In given list of numbers, you need to maximize the difference between two numbers, such that you can subtract the number, which appears before form the number that appear after it.

First solution: Brute force; for each element in the array, find other element whose difference is maximum or for which profit is maximum. This is done using two loops, first loop to select buy date index and the second loop to find its selling date entry.

Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Second solution: Another clever solution is to keep track of the smallest value seen so far from the start. At each point, we can find the difference and keep track of the maximum profit. This is a linear solution.

Time Complexity is $O(n)$ time. Space Complexity for creating count list is also $O(1)$

Example 5.55:

```
public static int maxProfit(int stocks[], int size) {  
    int buy = 0, sell = 0;  
    int curMin = 0;  
    int currProfit = 0;  
    int maxProfit = 0;  
    int i;  
    for (i = 0; i < size; i++) {  
        if (stocks[i] < stocks[curMin]) {  
            curMin = i;  
        }  
        currProfit = stocks[i] - stocks[curMin];  
        if (currProfit > maxProfit) {  
            buy = curMin;  
            sell = i;  
            maxProfit = currProfit;  
        }  
    }  
    System.out.println("Purchase day is- " + buy + " at price " + stocks[buy]);
```

```
System.out.println("Sell day is- " + sell + " at price " + stocks[sell]);  
return maxProfit;  
}
```

Find median of two sorted Lists.

Problem: Given two sorted lists. Find the median of the arrays if they are combined to form a bigger list.

First solution: We need to Keep track of the index of both the array, say the index are i and j. keep increasing the index of the array which ever have a smaller value. Use a counter to keep track of the elements that we have already traced. Once count is equal to the half of the combined length of two lists, we have our median.

Time Complexity is O(n) and Space Complexity is O(1)

Example 5.56:

```
public static int findMedian(int[] arrFirst, int sizeFirst, int[] arrSecond, int  
sizeSecond) {  
    int medianIndex = ((sizeFirst + sizeSecond) + (sizeFirst + sizeSecond) % 2) /  
2;// cealing function.  
    int i = 0, j = 0;  
    int count = 0;  
    while (count < medianIndex - 1) {  
        if (i < sizeFirst - 1 && arrFirst[i] < arrSecond[j]) {  
            i++;  
        } else {  
            j++;  
        }  
        count++;  
    }  
    if (arrFirst[i] < arrSecond[j]) {  
        return arrFirst[i];  
    } else {  
        return arrSecond[j];  
    }  
}
```

```
}
```

Search 01 List

Problem: In given list of 0's and 1's in which all the 0's come before 1's. Write an algorithm to find the index of the first 1.

Or

You are given an array which contains either 0 or 1, and they are in sorted order
Ex. a [] = {0, 0, 0, 1, 1, 1}, How will you count no of 1's and 0's?

First solution: Linear Search, we can always find the index of first 1 in the array using traversal.

Time Complexity is O(n) and Space Complexity is O(1)

Second solution: Binary Search, since the array is sorted using binary search to find the desired index.

Time Complexity is O(logn) and Space Complexity is O(1)

Example 5.57:

```
public static int BinarySearch01(int[] arr, int size) {
    if (size == 1 && arr[0] == 1) {
        return 0;
    }
    return BinarySearch01Util(arr, 0, size - 1);
}
```

```
public static int BinarySearch01Util(int[] arr, int start, int end) {
    int mid;
    if (end < start) {
        return -1;
    }
    mid = (start + end) / 2;
    if (1 == arr[mid] && 0 == arr[mid - 1]) {
        return mid;
    }
    if (0 == arr[mid]) {
        return BinarySearch01Util(arr, mid + 1, end);
```

```

} else {
    return BinarySearch01Util(arr, start, mid - 1);
}
}

```

Find Max in Rotated array

Problem: Given a sorted list S of N integer. S is rotated an unknown number of times. Find maximum value element in the array.

First solution: Linear Search, we can always find the index of first 1 in the array using traversal.

Time Complexity is O(n) and Space Complexity is O(1)

Second solution: Since the array is sorted, we can use modified binary search to find the element.

Time Complexity is O(logn) and Space Complexity is O(1)

Example 5.58:

```

public static int RotationMaxUtil(int arr[], int start, int end) {
    int mid;
    if (end <= start) {
        return arr[start];
    }
    mid = (start + end) / 2;
    if (arr[mid] > arr[mid + 1])
        return arr[mid];

    if (arr[start] <= arr[mid]) /* increasing part. */
        return RotationMaxUtil(arr, mid + 1, end);
    else
        return RotationMaxUtil(arr, start, mid - 1);
}

public static int RotationMax(int[] arr, int size) {
    return RotationMaxUtil(arr, 0, size - 1);
}

```

Find index of max value in a Rotated array

Problem: Given a sorted list S of N integer. S is rotated an unknown number of times. Find index of largest element in the array.

First solution: Linear Search, we can always find the index of first 1 in the array using traversal.

Time Complexity is O(n) and Space Complexity is O(1)

Second solution: Since the array is sorted, we can use modified binary search to find the element.

Time Complexity is O(logn) and Space Complexity is O(1)

Example 5.59:

```
public static int FindRotationMaxUtil(int arr[], int start, int end) {  
    /* single element case. */  
    int mid;  
    if (end <= start)  
        return start;  
  
    mid = (start + end) / 2;  
    if (arr[mid] > arr[mid + 1])  
        return mid;  
  
    if (arr[start] <= arr[mid]) /* increasing part. */  
        return FindRotationMaxUtil(arr, mid + 1, end);  
    else  
        return FindRotationMaxUtil(arr, start, mid - 1);  
}  
  
public static int FindRotationMax(int[] arr, int size) {  
    return FindRotationMaxUtil(arr, 0, size - 1);  
}
```

Count Rotation

Problem: Given a rotated array find the count of rotation.

Example 5.60:

```
public static int CountRotation(int[] arr, int size) {  
    int maxIndex = FindRotationMaxUtil(arr, 0, size - 1);  
    return (maxIndex + 1) % size;  
}
```

Search in sorted rotated List

Problem: Given a sorted list S of N integer. S is rotated an unknown number of times. Find an element in the array.

First solution: Linear Search, we can always find the index of first 1 in the array using traversal.

Time Complexity is O(n) and Space Complexity is O(1)

Second solution: Since the array is sorted, we can use modified binary search to find the element.

Time Complexity is O(logn) and Space Complexity is O(1)

Example 5.61:

```
public static int BinarySearchRotateArrayUtil(int[] arr, int start, int end, int key) {  
    int mid;  
    if (end < start) {  
        return -1;  
    }  
    mid = (start + end) / 2;  
    if (key == arr[mid]) {  
        return mid;  
    }  
    if (arr[mid] > arr[start]) {  
        if (arr[start] <= key && key < arr[mid]) {  
            return BinarySearchRotateArrayUtil(arr, start, mid - 1, key);  
        } else {  
            return BinarySearchRotateArrayUtil(arr, mid + 1, end, key);  
        }  
    } else {  
        return BinarySearchRotateArrayUtil(arr, mid + 1, end, key);  
    }  
}
```

```

} else {
if (arr[mid] < key && key <= arr[end]) {
return BinarySearchRotateArrayUtil(arr, mid + 1, end, key);
} else {
return BinarySearchRotateArrayUtil(arr, start, mid - 1, key);
}
}
}

public static int BinarySearchRotateArray(int[] arr, int size, int key) {
    return BinarySearchRotateArrayUtil(arr, 0, size - 1, key);
}

```

Minimum Absolute Difference Adjacent Circular Array

Problem: Given an array of integers, find minimum absolute difference of adjacent element consider circular array.

Example 5.62:

```

public static int minAbsDiffAdjCircular(int[] arr, int size) {
    int diff = 9999999;
    if (size < 2)
        return -1;
    for (int i = 0; i < size; i++)
        diff = Math.min(diff, Math.abs(arr[i] - arr[(i + 1) % size]));
    return diff;
}

```

Transform List

Problem: How would you swap elements of an array like [a1 a2 a3 a4 b1 b2 b3 b4] to convert it into [a1 b1 a2 b2 a3 b3 a4 b4]?

Approach:

- First swap elements in the middle pair
- Next swap elements in the middle two pairs

- Next swap elements in the middle three pairs
- Iterate n-1 steps.

Example 5.63:

```
public static void swapch(char[] arr, int first, int second) {
    char temp = arr[first];
    arr[first] = arr[second];
    arr[second] = temp;
}

public static void transformArrayAB1(char[] arr, int size) {
    int N = size / 2, i, j;
    for (i = 1; i < N; i++) {
        for (j = 0; j < i; j++) {
            swapch(arr, N - i + 2 * j, N - i + 2 * j + 1);
        }
    }
}
```

Check if two Lists are permutation of each other

Problem: In given two integer Lists; You have to check whether they are permutation of each other.

First solution: Sorting; Sort all the elements of both the arrays and Compare each element of both the arrays from beginning to end. If there is no mismatch, return true. Otherwise, false.

Sorting algorithms takes $O(n \log n)$ time and comparison takes $O(n)$ time.
Time Complexity is $O(n \log n)$ and Space Complexity is $O(1)$

Example 5.64:

```
public static boolean checkPermutation(char[] array1, int size1, char[] array2,
int size2) {
    if (size1 != size2) {
        return false;
    }
    Arrays.sort(array1);
```

```

Arrays.sort(array2);
for (int i = 0; i < size1; i++) {
    if (array1[i] != array2[i]) {
        return false;
    }
}
return true;
}

```

Second solution: Hash-Table (Assumption: No duplicates).

1. Create a Hash-Table for all the elements of the first list.
2. Traverse the other list from beginning to the end and search for each element in the Hash-Table.
3. If all the elements are found in, the Hash-Table return true, otherwise return false.

Hash-Table insert and find take constant time $O(1)$ so the total time complexity of the algorithm is $O(n)$ time. Space Complexity is also $O(n)$

Time Complexity = $O(n)$ (For creation of Hash-Table and look-up),

Space Complexity = $O(n)$ (For creation of Hash-Table).

Searching for an element in a 2-d sorted list

Problem: In given 2 dimensional list. Each row and column are sorted in ascending order. How would you find an element in it?

Solution: The algorithm works as:

1. Start with element at last column and first row
2. If the element is the value we are looking for, return true.
3. If the element is greater than the value we are looking for, go to the element at previous column but same row.
4. If the element is less than the value we are looking for, go to the element at next row but same column.
5. Return false, if the element is not found after reaching the element of the last row of the first column. Condition ($\text{row} < r \&\& \text{column} \geq 0$) is false.

Example 5.65:

```
public static boolean FindElementIn2DArray(int[] arr[], int r, int c, int value) {
```

```

int row = 0;
int column = c - 1;
while (row < r && column >= 0) {
    if (arr[row][column] == value) {
        return true;
    } else if (arr[row][column] > value) {
        column--;
    } else {
        row++;
    }
}
return false;
}

```

Running time = **O(N)**.

Arithmetic progression

Problem: Given an array of N integers, you need to find if array elements can form an Arithmetic progression.

First solution: Arithmetic relation can be found only by sorting the elements. Then traversing that AP exists. Time complexity will O(nlogn) for sorting

Example 5.66:

```

public static boolean isAP(int[] arr, int size) {
    int diff;
    if (size <= 1)
        return true;

    Arrays.sort(arr);
    diff = arr[1] - arr[0];
    for (int i = 2; i < size; i++) {
        if (arr[i] - arr[i - 1] != diff)
            return false;
    }
    return true;
}

```

```
}
```

Second solution: We can find the first and second lowest elements of the array in single traversal. By this, we can find first value of AP and increment value of AP. We traverse values in input array and add them to hashtable. Finally, we have first element of AP and difference so we can test that all the elements are in HashTable.

Time complexity is $O(n)$ and space complexity is $O(n)$ for hashtable.

Example 5.67:

```
public static boolean isAP2(int[] arr, int size) {
    int first = 9999999;
    int second = 9999999;
    int diff, value;
    HashSet<Integer> hs = new HashSet<Integer>();
    for (int i = 0; i < size; i++) {
        if (arr[i] < first) {
            second = first;
            first = arr[i];
        } else if (arr[i] < second)
            second = arr[i];
    }
    diff = second - first;

    for (int i = 0; i < size; i++) {
        if (hs.contains(arr[i]))
            return false;
        hs.add(arr[i]);
    }
    for (int i = 0; i < size; i++) {
        value = first + i * diff;
        if (!hs.contains(value))
            return false;
    }
    return true;
}
```

Third solution: We can solve this problem in $O(n)$ time and $O(1)$ space. We find the first and second lowest elements of the array in single traversal. Now we will traverse the array again and will put each element into its proper position by swapping. Index of each element will be $(\text{value} - \text{first}) / \text{diff}$. We will keep the count also so that we can find duplicate values too.

Example 5.68:

```
public static boolean isAP3(int[] arr, int size) {  
    int first = 9999999;  
    int second = 9999999;  
    int[] count = new int[size];  
    int diff, index = -1;  
    for (int i = 0; i < size; i++) {  
        if (arr[i] < first) {  
            second = first;  
            first = arr[i];  
        } else if (arr[i] < second)  
            second = arr[i];  
    }  
    diff = second - first;  
  
    for (int i = 0; i < size; i++)  
        index = (arr[i] - first) / diff;  
    if (index > size - 1 || count[index] != 0)  
        return false;  
    count[index] = 1;  
  
    for (int i = 0; i < size; i++)  
        if (count[i] != 1)  
            return false;  
    return true;  
}
```

Balance Point

Problem: Given an array, you need to find balance point or balance index. An

index is balanced index if the element in the left of it and elements in the right of it have same sum.

Solution: Create two set first and second. Sum all the element from index 1 to size-1 in second. One by one add elements to first set and remove them from second set.

Time complexity is $O(n)$

Example 5.69:

```
public static int findBalancedPoint(int[] arr, int size) {  
    int first = 0;  
    int second = 0;  
    for (int i = 1; i < size; i++)  
        second += arr[i];  
  
    for (int i = 0; i < size; i++) {  
        if (first == second) {  
            System.out.println(i);  
            return i;  
        }  
        if (i < size - 1)  
            first += arr[i];  
            second -= arr[i + 1];  
    }  
    return -1;  
}
```

Find Floor and Ceil

Problem: Given a sorted array you need to find ceil or floor of an input value. A ceil is the value in array which is just greater than the given input value. A floor is a value in array which is just smaller than the given input value.

Solution: Since the array is sorted the floor and ceil can be found by binary search like algorithm. Time complexity will $O(\log n)$

Example 5.70:

```
public static int findFloor(int arr[], int size, int value) {  
    int start = 0;  
    int stop = size - 1;  
    int mid;  
    while (start <= stop) {  
        mid = (start + stop) / 2;  
        /*  
         * search value is equal to arr[mid] value.. search value is greater than mid  
         * index value and less than mid+1 index value. value is greater than  
         * arr[size-1] then floor is arr[size-1]  
         */  
        if (arr[mid] == value || (arr[mid] < value && (mid == size - 1 || arr[mid + 1] >  
            value)))  
            return mid;  
        else if (arr[mid] < value)  
            start = mid + 1;  
        else  
            stop = mid - 1;  
    }  
    return -1;  
}
```

Example 5.71:

```
public static int findCeil(int arr[], int size, int value) {  
    int start = 0;  
    int stop = size - 1;  
    int mid;  
  
    while (start <= stop) {  
        mid = (start + stop) / 2;  
        /*  
         * search value is equal to arr[mid] value.. search value is less than  
         * mid index value and greater than mid-1 index value. value is less  
         * than arr[0] then ceil is arr[0]  
         */  
        if (arr[mid] == value || (arr[mid] > value && (mid == 0 || arr[mid - 1] <
```

```

value)))
return mid;
else if (arr[mid] < value)
start = mid + 1;
else
stop = mid - 1;
}
return -1;
}

```

Closest Number

Problem: Given a sorted array and a number. You need to find the element in array which is closest to the given number.

Solution: Since the array is sorted we can perform binary search.

Example 5.72:

```

public static int ClosestNumber(int arr[], int size, int num) {
int start = 0;
int stop = size - 1;
int output = -1;
int minDist = 9999;
int mid;

while (start <= stop) {
mid = (start + stop) / 2;
if (minDist > Math.abs(arr[mid] - num)) {
minDist = Math.abs(arr[mid] - num);
output = arr[mid];
}
if (arr[mid] == num)
break;
else if (arr[mid] > num)
stop = mid - 1;
else
start = mid + 1;
}
return output;
}

```

```
    }
    return output;
}
```

Duplicate K Distance

Problem: Given an array of integers, you need to find if duplicate values exist in the range of K units.

In array [1, 2, 3, 1, 4, 5] and range 3 the answer will be true. As 1 repeats in the range 3.

However, in the same array if the range is 2 then the answer is false.

Solution: We keep mapping of values and their index in a Hashtable. We traverse the array for each element we first look if this value is already present in the Hashtable. If it is not present then we add element as key and its index as value in the hashtable. If the element is present in the hashtable then we can find if the repeated value is in the range by subtracting current index with the index of the previous repetition. We keep on updating the index corresponds to the value.

Time complexity is O(n) and space complexity is O(n)

Example 5.73:

```
public static boolean DuplicateKDistance(int arr[], int size, int k) {
    HashMap<Integer, Integer> hm = new HashMap<Integer, Integer>();

    for (int i = 0; i < size; i++) {
        if (hm.containsKey(arr[i]) && i - hm.get(arr[i]) <= k) {
            System.out.println("Value:" + arr[i] + " Index: " + hm.get(arr[i]) + " & " + i);
            return true;
        } else
            hm.put(arr[i], i);
    }
    return false;
}
```

Frequency Counts

Problem: Given an array of size N, which contain integers from 1 to N. Elements can appear any number of times. Print frequency of all elements in the array also print the missing elements frequency as 0

Input: [1, 2, 2, 2, 1]

Output:

1: 2

2: 3

3: 0

4: 0

5: 0

Frist solution: Use Hashtable to keep track of value and its frequency. Traverse in the range and keep on printing the values.

Second solution: Using sorting we can sort the array first, then we can traverse the array and print the element and its frequency.

Third solution: Using auxiliary array, now since we have data in the range 1 to N so an extra auxiliary array can be used to keep track of frequency. While traversing the input array, we found a value V then its corresponding index in the auxiliary array will be (V-1), since array indexing starts form 0.

Fourth solution: Now solve this problem in linear time, without using any extra space. We have to look carefully that this problem provides us with extra information regarding the range of values from 1 to N.

We traverse the input array, we found a value V then its corresponding index values in the array is (V - 1) if there is some valid value in the range (1 to N) in that index then we copy that value to our traversal index and then mark the value at index (V-1) as -1. If the value at index (V-1) is not in range then we decrease it by 1. Now we can print absolute value to the output.

Example 5.74:

```
public static void frequencyCounts(int[] arr, int size) {
```

```

int index;
for (int i = 0; i < size; i++) {
    while (arr[i] > 0) {
        index = arr[i] - 1;
        if (arr[index] > 0) {
            arr[i] = arr[index];
            arr[index] = -1;
        } else {
            arr[index] -= 1;
            arr[i] = 0;
        }
    }
}
for (int i = 0; i < size; i++)
    System.out.println((i + 1) + Math.abs(arr[i]));
}

```

K Largest Elements

Problem: Given an array of integers of size N, you need to print k largest elements in the array in order in which they appear in the array.

First solution: Another solution is to sort the array and again find the kth largest element. Scan the array and print all the elements which have value greater than or equal to the kth largest element.

Example 5.75:

```

public static int KLargestElements(int arrIn[], int size, int k) {
    int[] arr = new int[size];
    for (int i = 0; i < size; i++)
        arr[i] = arrIn[i];
    Arrays.sort(arr);
    for (int i = 0; i < size; i++) {
        if (arrIn[i] >= arr[size - k]) {
            System.out.println(arrIn[i]);
            return arrIn[i];
        }
    }
}

```

```
    }
    return -1;
}
```

Second solution:

First copy the array into another array. Then using quick select find the N-Kth elements in the array.

Then scan the original array and print all the elements which have value Greater than or equal to k.

$O(N)$ for k and $O(N)$ for scan.

Example 5.76:

```
public static void QuickSelectUtil(int arr[], int lower, int upper, int k) {
    if (upper <= lower)
        return;

    int pivot = arr[lower];

    int start = lower;
    int stop = upper;

    while (lower < upper) {
        while (arr[lower] <= pivot) {
            lower++;
        }
        while (arr[upper] > pivot) {
            upper--;
        }
        if (lower < upper) {
            swap(arr, upper, lower);
        }
    }

    swap(arr, upper, start); // upper is the pivot position
    if (k < upper)
        QuickSelectUtil(arr, start, upper - 1, k); // pivot -1 is the upper for left sub
array.
```

```

if (k > upper)
    QuickSelectUtil(arr, upper + 1, stop, k); // pivot + 1 is the lower for right sub
array.
}

public static int KLargestElements2(int arrIn[], int size, int k) {
    int[] arr = new int[size];
    for (int i = 0; i < size; i++)
        arr[i] = arrIn[i];

    QuickSelectUtil(arr, 0, size - 1, size - k);
    for (int i = 0; i < size; i++) {
        if (arrIn[i] >= arr[size - k]) {
            System.out.println(arrIn[i]);
            return arrIn[i];
        }
    }
    return -1;
}

```

Note: Quick Select algorithms, closely related to quick sort can be studied in shorting chapter.

Note: There is a catch in the above solutions it may happen that the number of values printed may me more than K so make sure they are exactly k.

Fix Point

Problem: Given a sorted array of integers, you need to find the fix point. Fix point is an index of array in which index and value is same.

First solution: Brute force approach, traverse the array to find fix point. Time complexity is $O(n)$

Example 5.77:

```

/* linear search method */
public static int FixPoint(int[] arr, int size) {

```

```

for (int i = 0; i < size; i++) {
    if (arr[i] == i)
        return i;
    } /* fix point not found so return invalid index */
    return -1;
}

```

Second solution: Since the array is sorted, we should think about binary sort algorithm.

Example 5.78:

```

/* Binary search method */
public static int FixPoint2(int[] arr, int size) {
    int low = 0;
    int high = size - 1;
    int mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == mid)
            return mid;
        else if (arr[mid] < mid)
            low = mid + 1;
        else
            high = mid - 1;
    }
    /* fix point not found so return invalid index */
    return -1;
}

```

Sub Array Sums

Problem: Given an array of positive integers, you need to find if there is some range in array such that if we add all the elements in that range then it became equal to given value.

Example 5.79:

```

public static int subArraySums(int arr[], int size, int value) {

```

```

int first = 0;
int second = 0;
int sum = arr[first];
while (second < size && first < size) {
if (sum == value)
System.out.println(first + second);

if (sum < value) {
second += 1;
if (second < size)
sum += arr[second];
} else {
sum -= arr[first];
first += 1;
}
}
return sum;
}

```

Time complexity: O(n)

Maximum contiguous subarray sum

Problem: Given an array of positive and negative integers, find maximum contiguous subarray in A.

Example 5.80: Kadane's Algorithm to find maximum contiguous subarray sum.

```

public static int MaxConSub(int[] arr, int size) {
    int currMax = 0;
    int maximum = 0;

    for (int i = 0; i < size; i++) {
        currMax = Math.max(arr[i], currMax + arr[i]);
        if (currMax < 0)
            currMax = 0;
        if (maximum < currMax)
            maximum = currMax;
    }
}

```

```
}

System.out.println(maximum);
return maximum;
}
```

Maximum contiguous subarray sum (A-B)

Problem: Given an array A of integers and array B of integers, find maximum contiguous subarray in A
Such that it does not contains elements in B.

First Solution: Kadane's Algorithm is modified by using a HashTable to solve this problem.

Example 5.81:

```
public static int MaxConSubArr(int[] A, int sizeA, int[] B, int sizeB) {
    int currMax = 0;
    int maximum = 0;
    HashSet<Integer> hs = new HashSet<Integer>();

    for (int i = 0; i < sizeB; i++)
        hs.add(B[i]);

    for (int i = 0; i < sizeA; i++)
        if (hs.contains(A[i]))
            currMax = 0;
        else
            currMax = Math.max(A[i], currMax + A[i]);
        if (currMax < 0)
            currMax = 0;
        if (maximum < currMax)
            maximum = currMax;
    System.out.println(maximum);
    return maximum;
}
```

Time complexity: $O(M + N)$.

Second Solution: Sort array B. In place of binary search is used to find if some element is present in B.

Time complexity: $O(M\log M + N\log M)$ For sorting and then for search each element.

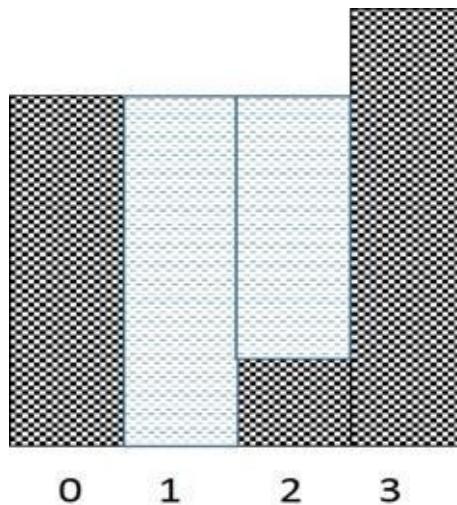
Example 5.82:

```
public static int MaxConSubArr2(int A[], int sizeA, int B[], int sizeB) {  
    Arrays.sort(B);  
    int currMax = 0;  
    int maximum = 0;  
  
    for (int i = 0; i < sizeA; i++) {  
        if (Binarysearch(B, sizeB, A[i]))  
            currMax = 0;  
        else {  
            currMax = Math.max(A[i], currMax + A[i]);  
            if (currMax < 0)  
                currMax = 0;  
            if (maximum < currMax)  
                maximum = currMax;  
        }  
    }  
    System.out.println(maximum);  
    return maximum;  
}
```

Rain Water

Problem: Given an array of N non-negative integers. Each element of array represents a bar of histogram. Considering that each bar is one unit wide. You need to find how much water can be accommodate in the structure.

For example: [4, 0, 1, 5] will contain 7 units of water.



Solution: This problem can be solved very easily if we are able to find how much water a particular bar of unit width can contain.

Water contained at i^{th} index = minimum (maximum length in left, maximum length in right) – length i^{th} bar.

Example 5.83:

```
public static int RainWater(int[] arr, int size) {
    int water = 0;
    int[] leftHigh = new int[size];
    int[] rightHigh = new int[size];

    int max = arr[0];
    leftHigh[0] = arr[0];
    for (int i = 1; i < size; i++) {
        if (max < arr[i])
            max = arr[i];
        leftHigh[i] = max;
    }
    max = arr[size - 1];
    rightHigh[size - 1] = arr[size - 1];
    for (int i = (size - 2); i >= 0; i--) {
        if (max < arr[i])
            max = arr[i];
        rightHigh[i] = max;
    }
}
```

```

    }

    for (int i = 0; i < size; i++)
        water += Math.min(leftHigh[i], rightHigh[i]) - arr[i];
    System.out.println("Water : " + water);
    return water;
}

```

Time complexity: O(N)

Further optimization on the above problem in a single iteration the water is calculated along with leftMax and rightMax calculation.

Example 5.84:

```

public static int RainWater2(int[] arr, int size) {
    int water = 0;
    int leftMax = 0, rightMax = 0;
    int left = 0;
    int right = size - 1;

    while (left <= right) {
        if (arr[left] < arr[right]) {
            if (arr[left] > leftMax)
                leftMax = arr[left];
            else
                water += leftMax - arr[left];
            left += 1;
        } else {
            if (arr[right] > rightMax)
                rightMax = arr[right];
            else
                water += rightMax - arr[right];
            right -= 1;
        }
    }
    System.out.println("Water : " + water);
    return water;
}

```

}

Exercise

1. In given list of n elements, we need to find the first repeated element. Which of the following methods will work for us. If a method works, then implement it.
 - Brute force exhaustive search.
 - Use Hash-Table to keep an index of the elements and use the second scan to find the element.
 - Sorting the elements.
 - If we know the range of the element then we can use counting technique.

Hint: When order in which elements appear in input is important, we cannot use sorting.

2. In given list of n elements, write an algorithm to find three elements in an array whose sum is a given value.

Hint: Try to do this problem using a brute force approach. Then try to apply the sorting approach along with a brute force approach. Time Complexity is $O(n^2)$

3. In given list of -ve and +ve numbers, write a program to separate -ve numbers from the +ve numbers.

4. In given list of 1's and 0's, write a program to separate 0's from 1's.

Hint: QuickSelect, counting

5. In given list of 0's, 1's and 2's, write a program to separate 0's, 1's and 2's.

6. In given list whose elements is monotonically increasing with both negative and positive numbers. Write an algorithm to find the point at which list becomes positive.

7. In a sorted list, find a number. If found then return the index if not found then insert into the array.

8. Find max in sorted rotated list.

9. Find min in the sorted rotated list.
10. Find kth Smallest Element in the Union of Two Sorted Lists

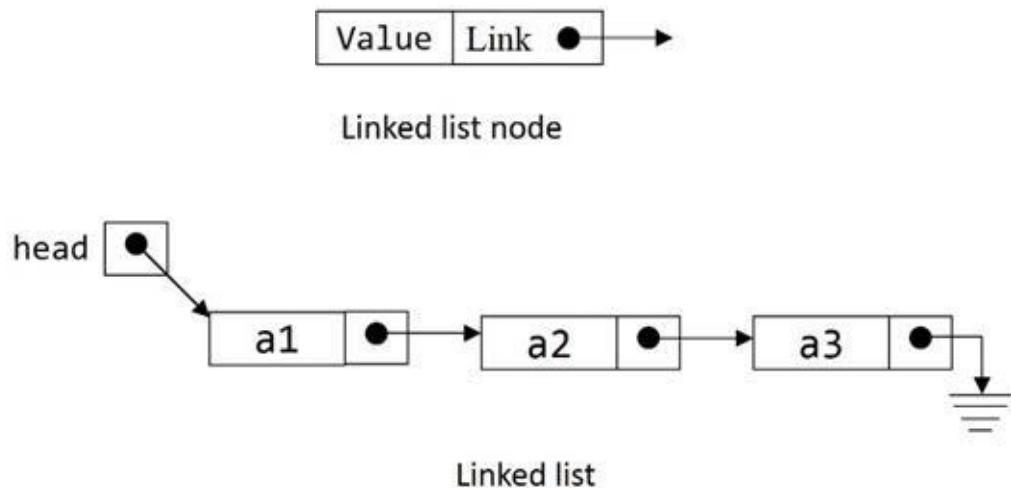
CHAPTER 6: LINKED LIST

Introduction

Let us suppose we have an array that contains following five elements 1, 2, 4, 5, 6. We want to insert a new element with value “3” in between “2” and “4”. In the array, we cannot do it so easily. We need to create another array that is long enough to store the current values and one more space for “3”. Then we need to copy these elements in the new space. This copy operation is inefficient. To remove this inefficiency in addition of new data linked list is used.

Linked List

The linked list is a list of items, called nodes. Nodes have two parts, value part and link part. Value part is used to stores the data. The value part of the node can be a basic data-type like an integer or it can be some other data-type like a structure. The link part is a pointer, which is used to store addresses of the next element in the list.



The various parts of linked list:

1. **Head:** Head is a pointer that holds the address of the first node in the linked list.

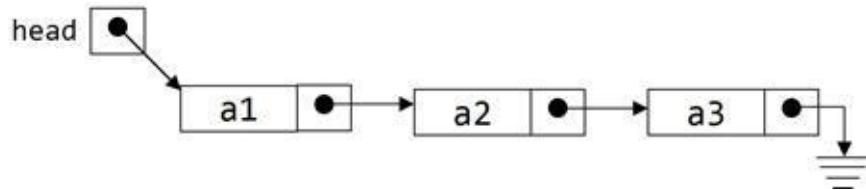
2. **Nodes:** Items in the linked list are called nodes.
3. **Value:** The data that is stored in each node of the linked list.
4. **Link:** Link part of the node is used to store the reference of other node.
 - a. We will use “next” and “prev” to store address of next or previous nodes.

Types of Linked list

There are different types of linked lists. The main difference among them is how their nodes connected to each other. We will discuss singly linked list, doubly linked list, circular linked list and doubly circular linked list.

Singly Linked List

Each node (Except the last node) has a reference to the next node in the linked list. The link portion of node contains the address of the next node. The link portion of the last node contains the value null.



Let us look at the Node. The Value part of node is of type integer, but it can be some other data-type. The link part of node is named as Pointer in the below diagram.



Example 6.1: Singly Linked List node struct

```
public class LinkedList {  
    private static class Node {  
        private int value;  
        private Node next;  
  
        public Node(int v, Node n) {  
            value = v;  
            next = n;  
        }  
    }  
  
    private Node head;  
    private int size = 0;  
}
```

For a singly linked, we should always test these three test cases:

- Zero element / Empty linked list.
- One element / Single node case.

- General case.

One node and zero node case are used to test boundary cases. It is always mandatory to take care of these cases before submitting code.

The various basic operations that we can perform on linked lists, many of these operations require list traversal:

- Insert an element in the list, this operation is used to create a linked list.
- Print various elements of the list.
- Search an element in the list.
- Delete an element from the list.
- Reverse a linked list.

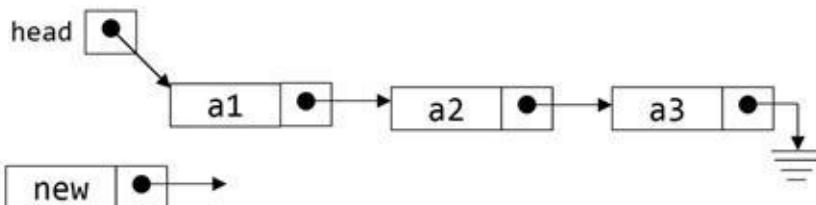
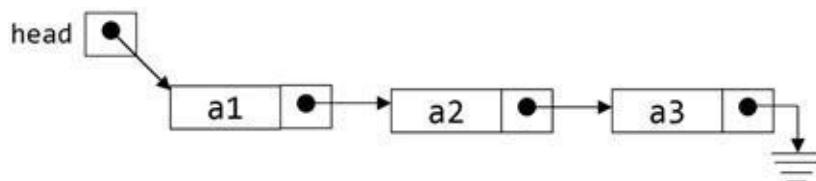
Insert element in linked list

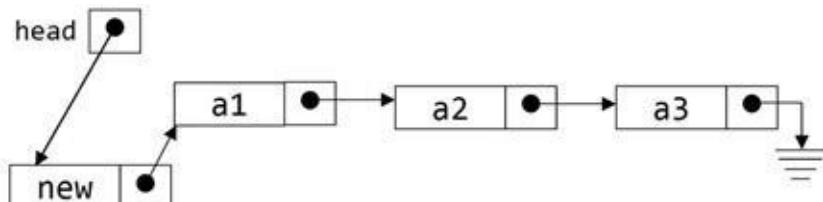
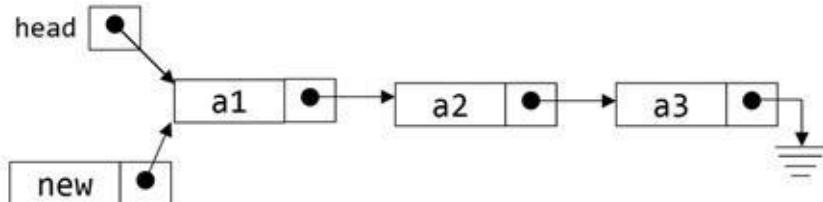
An element can be inserted into a linked list in various orders. Some of the example cases are mentioned below:

1. Insertion of an element at the start of linked list
2. Insertion of an element at the end of linked list
3. Insertion of an element at the N^{th} position in linked list
4. Insert element in sorted order in linked list

Insert element at the Head

Problem: Insert an element at the start of linked list.





Example 6.2:

```
public void addHead(int value) {
    head = new Node(value, head);
    size++;
}
```

Analysis:

- We need to create a new node with the value passed to the function as argument.
- The next pointer of the new node will point to the head of the linked list or null in case when list is empty.
- The newly created node will become head of the linked list.

Time Complexity: O(1).

Insertion of an element at the end

Problem: Insertion of an element at the end of linked list

Example 6.3:

```
public void addTail(int value) {
    Node newNode = new Node(value, null);
    Node curr = head;

    if (head == null) {
```

```

head = newNode;
}

while (curr.next != null) {
    curr = curr.next;
}
curr.next = newNode;
}

```

Analysis:

- New node is created and the value is stored inside it and store null value to its next pointer.
- If the list is empty then this new node will become head of linked list.
- If list is not empty then we have to traverse to the end of the list.
- Finally, new node is added to the end of the list.

Time Complexity: O(n).

Note: This operation is un-efficient as each time you want to insert an element you have to traverse to the end of the list. Therefore, the complexity of creation of the list is $O(n^2)$. So how to make it efficient we have to keep track of the last element by keeping a tail pointer. Therefore, if it is required to insert element at the end of linked list, then we will keep track of the tail reference also.

Traversing Linked List

Problem: Print various elements of a linked list

Example 6.4:

```

public void print() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.value + " ");
        temp = temp.next;
    }
}

```

Analysis:

- We will traverse the list and print the value stored in nodes. List is traversed by making head pointing to its next.

Time Complexity: O(n).

Complete code for list creation and printing the list.

Example 6.5: Test code for linked list creation, adding value at head and printing elements.

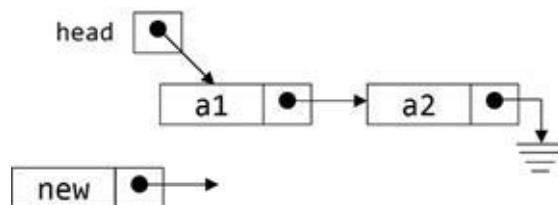
```
public static void main(String[] args) {  
    LinkedList ll = new LinkedList();  
    ll.addHead(1);  
    ll.addHead(2);  
    ll.addHead(3);  
    ll.print();  
}
```

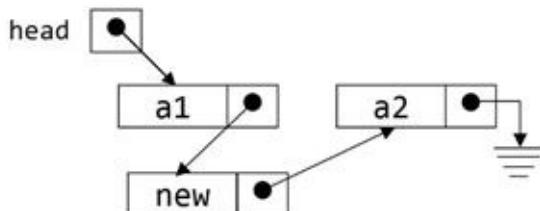
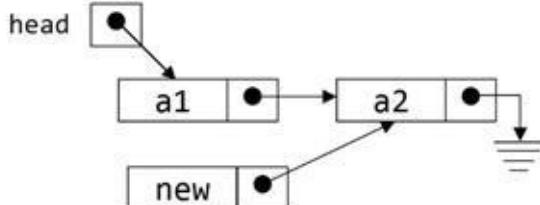
Analysis:

- New instance of linked list is created. Various elements are added to list by calling InsertNode() method.
- Finally all the content of list is printed to screen by calling PrintList() method.

Sorted Insert

Problem: Insert an element in sorted order in linked list given head pointer





Example 6.6:

```
public void sortedInsert(int value) {
    Node newNode = new Node(value, null);
    Node curr = head;

    if (curr == null || curr.value > value) {
        newNode.next = head;
        head = newNode;
        return;
    }
    while (curr.next != null && curr.next.value < value) {
        curr = curr.next;
    }

    newNode.next = curr.next;
    curr.next = newNode;
}
```

Analysis:

- A new empty node of the linked list is created. And initialized by storing an argument value into its value. Next of the node will point to null.
- If the list is empty or if the value stored in the first node is greater than the new created node value. Then this new created node will be added to the start of the list. And head need to be modified.
- In all other cases, we iterate through the list to find the proper position where

- the node can be inserted to keep the list sorted. .
- Finally, the node will be added to the list.

Time Complexity: O(n).

Search Element in a Linked-List

Problem: Search element in linked list. Given a head pointer and value. Returns 1 if value found in list else returns 0.

Example 6.7:

```
public boolean searchList(int data) {  
    Node temp = head;  
    while (temp != null) {  
        if (temp.value == data)  
            return true;  
        temp = temp.next;  
    }  
    return false;  
}
```

Analysis:

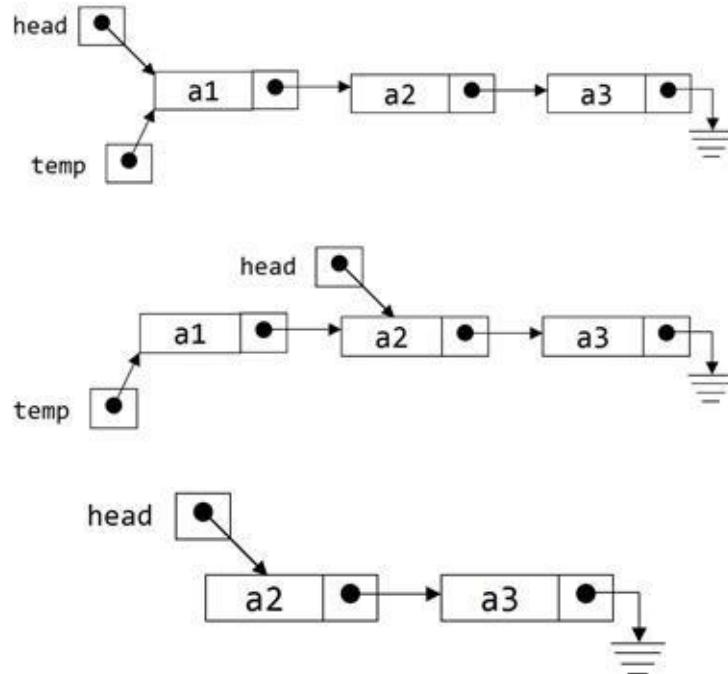
- Using a loop we will iterate through the list.
- Value of each element of list is compared with the given value. If value is found, then the function will return true.
- If the value is not found, then false will be returned from the function in the end.

Time Complexity: O(n).

Note: Search in a single linked list can only be done in one direction. Since all elements in the list have reference to the next item in the list. Therefore, traversal of linked list is linear in nature.

Delete First element in a linked list.

Problem: Delete element at the head of the linked list.



Example 6.8:

```
public int removeHead() throws IllegalStateException {
    if (isEmpty())
        throw new IllegalStateException("EmptyListException");
    int value = head.value;
    head = head.next;
    size--;
    return value;
}
```

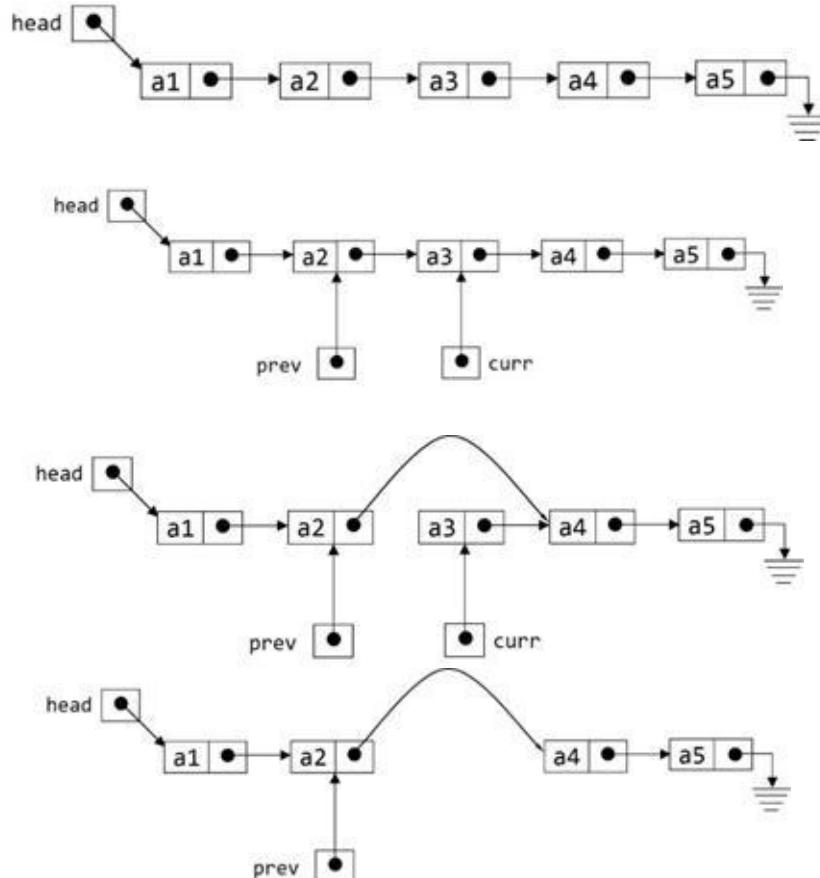
Analysis:

- First, we need to check if the list is already empty. If empty then return.
- If list is not empty then the head of the list will store the reference of next node of the current head.

Time Complexity: O(1).

Delete node from the linked list given its value.

Problem: Delete the first node whose value is equal to the given value.



Example 6.9:

```

public boolean deleteNode(int delValue) {
    Node temp = head;

    if (isEmpty())
        return false;

    if (delValue == head.value) {
        head = head.next;
        size--;
        return true;
    }

    while (temp.next != null) {
        if (temp.next.value == delValue) {
            temp.next = temp.next.next;
            size--;
        }
    }
}

```

```

        return true;
    }
    temp = temp.next;
}
return false;
}

```

Analysis:

- There are two cases, first case when the node that need to be deleted is first node and second case when the node that need to be deleted is not the first node in the list. When the node that need to be deleted is first node then head of the linked list will be modified. In other cases, it will not modify.
- First, we check if the first node is the node with the value we are searching for then the head of the list will point to the next reference of the current head.
- In other case, we will traverse the link list using a while loop and try to find the node that need to be deleted. If the node is found then, we will point its previous node next point to the node next to the node we want to remove.

Time Complexity: O(n).

Delete all the occurrence of particular value in linked list.

Problem: Delete all the nodes whose value is equal to the given value.

Example 6.10:

```

public void deleteNodes(int delValue) {
    Node currNode = head;
    Node nextNode;

    while (currNode != null && currNode.value == delValue)/* first node */
    {
        head = currNode.next;
        currNode = head;
    }

    while (currNode != null) {
        nextNode = currNode.next;

```

```
if (nextNode != null && nextNode.value == delValue) {  
    currNode.next = nextNode.next;  
} else {  
    currNode = nextNode;  
}  
}  
}
```

Analysis:

- In the first while loop we will remove all the nodes that are at the front of the list, which have valued equal to the value we want to delete. In this, we need to update head of the list.
- In the second while loop, we will remove all the nodes that are not at the begening of the list and have value equal to the value we want to delete. Remember that we are not returning we traverse till the end of the list.

Time Complexity: O(n).

Delete a single linked list

Problem: Given a pointer of head of linked list, delete all the elements of a list.

Example 6.11:

```
public void deleteList() {  
    head = null;  
    size = 0;  
}
```

Analysis: Mark head of list as null.

Time Complexity: O(n).

Reverse a linked list.

Problem: Reverse a singly linked List iteratively using three Pointers

Example 6.12:

```
public void reverse() {  
    Node curr = head;
```

```

Node prev = null;
Node next = null;
while (curr != null) {
    next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
}
head = prev;
}

```

Analysis: The list is iterated. Make next variable point to the next of current node. Make next of current node point to previous node. Make prev as current node and curr as next node.

Time Complexity: O(n).

Recursively Reverse a singly linked List

Problem: Reverse a singly linked list using Recursion.

Example 6.13:

```

public Node reverseRecurseUtil(Node currentNode, Node nextNode) {
    Node ret;
    if (currentNode == null)
        return null;
    if (currentNode.next == null) {
        currentNode.next = nextNode;
        return currentNode;
    }
    ret = reverseRecurseUtil(currentNode.next, currentNode);
    currentNode.next = nextNode;
    return ret;
}

```

```
public void reverseRecurse() {  
    head = reverseRecurseUtil(head, null);  
}
```

Analysis:

- ReverseRecurse() function will call a reverseRecurseUtil() function to reverse the list and the pointer returned by the reverseRecurseUtil will be the head of the reversed list.
- The current node will point to the nextNode that is previous node of the old list.

Time Complexity: O(n).

Note: A linked list can be reversed using two solutions the First solution is by using three pointers. The Second solution is using recursion both are linear solution, but three-pointer solution is more efficient.

Remove duplicates from the linked list

Problem: Remove duplicate values from the linked list. The linked list is sorted and it contains some duplicate values, you need to remove those duplicate values. (You can create the required linked list using SortedInsert() function)

Example 6.14:

```
public void removeDuplicate() {  
    Node curr = head;  
    while (curr != null) {  
        if (curr.next != null && curr.value == curr.next.value) {  
            curr.next = curr.next.next;  
        } else {  
            curr = curr.next;  
        }  
    }  
}
```

Analysis: Traverse the list, if there is a node whose value is equal to it's next node's value than current node next will point to the next of next node. Whole

list is processed and the repeated values are removed from the list.

Time Complexity: O(n).

Copy List Reversed

Problem: Copy the content of linked list in another linked list in reverse order. If the original linked list contains elements in order 1,2,3,4, the new list should contain the elements in order 4,3,2,1.

Example 6.15:

```
public LinkedList copyListReversed() {  
    Node tempNode = null;  
    Node tempNode2 = null;  
    Node curr = head;  
    while (curr != null) {  
        tempNode2 = new Node(curr.value, tempNode);  
        curr = curr.next;  
        tempNode = tempNode2;  
    }  
    LinkedList ll2 = new LinkedList();  
    ll2.head = tempNode;  
    return ll2;  
}
```

Analysis: Traverse the list and add the node value to the head of new list. Since the list is traversed in the forward direction and each node is added to the head of the new list the new list is formed is reverse of the original list.

Time Complexity: O(n).

Copy the content of given linked list into another linked list

Problem: Copy the content of given linked list into another linked list. If the original linked list contains elements in order 1,2,3,4, the new list should contain the elements in order 1,2,3,4.

Example 6.16:

```
public LinkedList copyList() {  
    Node headNode = null;  
    Node tailNode = null;  
    Node tempNode = null;  
    Node curr = head;  
  
    if (curr == null)  
        return null;  
  
    headNode = new Node(curr.value, null);  
    tailNode = headNode;  
    curr = curr.next;  
  
    while (curr != null) {  
        tempNode = new Node(curr.value, null);  
        tailNode.next = tempNode;  
        tailNode = tempNode;  
        curr = curr.next;  
    }  
    LinkedList ll2 = new LinkedList();  
    ll2.head = headNode;  
    return ll2;  
}
```

Analysis: Traverse the list and add the node's value to new list, but this time always at the end of the list. Another pointer tailNode is used to keep track of the end of the list. Since the list is traversed in the forward direction and each node's value is added to the end of new list. Therefore, the formed list is same as the given list.

Time Complexity: $O(n)$.

Compare List

Problem: Compare the values of two linked lists given their head pointers.

Example 6.17: Recursive Solution

```
public boolean compareList(LinkedList ll) {  
    return compareList(head, ll.head);  
}  
  
public boolean compareList(Node head1, Node head2) {  
    if (head1 == null && head2 == null)  
        return true;  
    else if ((head1 == null) || (head2 == null) || (head1.value != head2.value))  
        return false;  
    else  
        return compareList(head1.next, head2.next);  
}
```

Analysis:

- List is compared recursively. Moreover, if we reach the end of the list and both the lists are null. Then both the lists are equal and so return true.
- List is compared recursively. If either one of the list is empty or the value of corresponding nodes is unequal, then the lists are not equal and compareList() function will return false.
- Recursively calls compare list function for the next node of the current nodes.

Example 6.18: Iterative Solution

```
public boolean compareList2(LinkedList ll2) {  
    Node head1 = head;  
    Node head2 = ll2.head;  
  
    while (head1 == null && head2 == null) {  
        if (head1.value != head2.value)  
            return false;  
        head1 = head1.next;  
        head2 = head2.next;  
    }  
  
    if (head1 == null && head2 == null)  
        return true;  
    return false;
```

```
}
```

Analysis:

- Traverse both the list in a loop and if any point if the values of both the nodes of the list is not equal then return false that they are not equal lists.
- If at the end if both the lists are completely traversed then return true else return false if any one of them have some untraversed elements.

Time Complexity: Both solutions have same $O(n)$ time complexity.

Find Length

Problem: Find the length of given linked list.

Example 6.19:

```
public int findLength() {  
    Node curr = head;  
    int count = 0;  
    while (curr != null) {  
        count++;  
        curr = curr.next;  
    }  
    return count;  
}
```

Analysis: Length of linked list is found by traversing the list until we reach the end of list.

Time Complexity: $O(n)$.

Nth Node from Beginning

Problem: Find Nth node from beginning

Example 6.20:

```
public int nthNodeFromBeginning(int index) {  
    if (index > size() || index < 1)
```

```

return Integer.MAX_VALUE;
int count = 0;
Node curr = head;
while (curr != null && count < index - 1) {
    count++;
    curr = curr.next;
}
return curr.value;
}

```

Analysis: Nth node can be found by traversing the list $N-1$ number of time and then return the node. If list does not have N elements then the method return null.

Time Complexity: $O(K)$ if we are searching kth node from start.

Nth Node from End

Problem: Find Nth node from end

Example 6.21:

```

public int nthNodeFromEnd(int index) {
    int size = findLength();
    int startIndex;
    if (size != 0 && size < index) {
        return Integer.MAX_VALUE;
    }
    startIndex = size - index + 1;
    return nthNodeFromBegining(startIndex);
}

```

Analysis: First, find the length of list, then nth node from end will be $(length - nth + 1)$ node from the beginning.

Time Complexity: $O(n)$.

Example 6.22:

```

public int nthNodeFromEnd2(int index) {
    int count = 1;
    Node forward = head;
    Node curr = head;
    while (forward != null && count <= index ) {
        count++;
        forward = forward.next;
    }

    if (forward == null)
        return Integer.MAX_VALUE;

    while (forward != null) {
        forward = forward.next;
        curr = curr.next;
    }
    return curr.value;
}

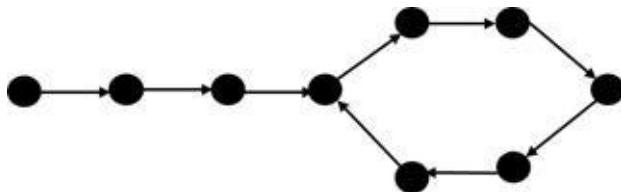
```

Analysis: Second solution is to use two pointers one is N steps / nodes ahead of the other when forward pointer reach the end of the list then the backward pointer will point to the desired node.

Time Complexity: O(n).

Loop Detect

Problem: Find if there is a loop in a linked list. If there is a loop, then return true and if there is no loop found then return false.

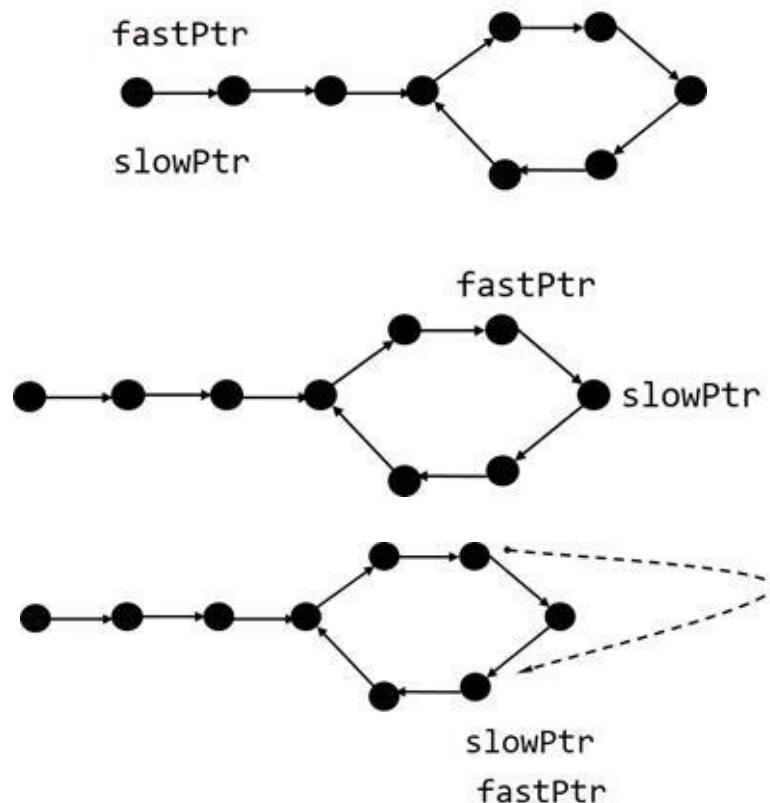


There are many ways to find if there is a loop in a linked list:

First solution: User some map or hash-table

- a) Traverse through the list.
 - b) If the current node is, not there in the Hash-Table then insert it into the Hash-Table.
 - c) If the current node is already in the Hashtable then we have a loop.

Second solution: Slow pointer and fast pointer approach (SPFP), we will use two pointers, one will move 2 steps at a time and another will move 1 step at time. If there is, a loop then both will meet at a point.



Example 6.23:

```
public boolean loopDetect() {  
    Node slowPtr;  
    Node fastPtr;  
    slowPtr = fastPtr = head;  
  
    while (fastPtr.next != null && fastPtr.next.next != null) {  
        slowPtr = slowPtr.next;  
        fastPtr = fastPtr.next.next;  
    }  
    if (slowPtr == fastPtr) {  
        return true;  
    }  
    return false;  
}
```

```

if (slowPtr == fastPtr) {
    System.out.println("loop found");
    return true;
}
}
System.out.println("loop not found");
return false;
}

```

Analysis:

- The list is traversed with two pointers, one is slow pointer and another is fast pointer. Slow pointer always moves one-step. Fast pointer always moves two steps. If there is no loop, then control will come out of while loop. So return false.
- If there is a loop, then there comes a point in a loop where the fast pointer will come and try to pass slow pointer and they will meet at a point. When this point arrives, we come to know that there is a loop in the list. So return true.

Time Complexity: O(n).

Third solution: Reverse list loop detect approach, if there is a loop in a linked list, then reverse list function will give head of the original list as the head of the new list.

Example 6.24: Find if there is a loop in a linked list. Use reverse list approach.

```

public boolean reverseListLoopDetect() {
    Node tempHead = head;
    reverse();
    if (tempHead == head) {
        reverse();
        System.out.println("loop found");
        return true;
    } else {
        reverse();
        System.out.println("loop not found");
        return false;
    }
}

```

```
}
```

Analysis:

- Store pointer of the head of list in a temp variable.
- Reverse the list
- Compare the reversed list head pointer to the current list head pointer.
- If the head of reversed list and the original list are same then reverse the list back and return true.
- If the head of the reversed list and the original list are not same, then reverse the list back and return false. Which means there is no loop.

Time Complexity: O(n).

Note: Both SPFP and Reverse List approaches are linear in nature, but still in SPFP approach, we do not require to modify the linked list so it is preferred.

Loop Type Detect

Problem: Find if there is a loop in a linked list. If there is no loop, then return 0, if there is loop return 1, if the list is circular then 2. Use slow pointer fast pointer approach.

Example 6.25:

```
public int loopTypeDetect() {  
    Node slowPtr;  
    Node fastPtr;  
    slowPtr = fastPtr = head;  
  
    while (fastPtr.next != null && fastPtr.next.next != null) {  
        if (head == fastPtr.next || head == fastPtr.next.next) {  
            System.out.println("circular list loop found");  
            return 2;  
        }  
        slowPtr = slowPtr.next;  
        fastPtr = fastPtr.next.next;  
        if (slowPtr == fastPtr) {  
            System.out.println("loop found");  
            return 1;  
        }  
    }  
}
```

```
    }
}
System.out.println("loop not found");
return 0;
}
```

Analysis: This program is same as the loop detect program only if it is a circular list than the fast pointer reaches the slow pointer at the head of the list this means that there is a loop at the beginning of the list.

Time Complexity: O(n).

Remove Loop

Problem: Given there is a loop in linked list remove the loop.

Example 6.26:

```
public Node loopPointDetect() {
    Node slowPtr;
    Node fastPtr;
    slowPtr = fastPtr = head;

    while (fastPtr.next != null && fastPtr.next.next != null) {
        slowPtr = slowPtr.next;
        fastPtr = fastPtr.next.next;
        if (slowPtr == fastPtr) {
            return slowPtr;
        }
    }
    return null;
}

public void removeLoop() {
    Node loopPoint = loopPointDetect();
    if (loopPoint == null)
        return;

    Node firstPtr = head;
```

```

if (loopPoint == head) {
    while (firstPtr.next != head)
        firstPtr = firstPtr.next;
    firstPtr.next = null;
    return;
}

Node secondPtr = loopPoint;
while (firstPtr.next != secondPtr.next) {
    firstPtr = firstPtr.next;
    secondPtr = secondPtr.next;
}
secondPtr.next = null;
}

```

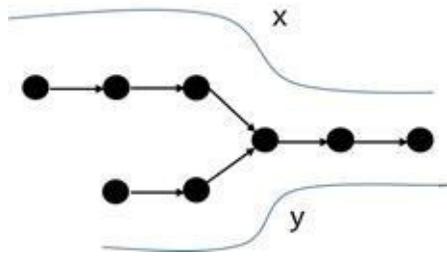
Analysis:

- Loop through the list by two pointer, one fast pointer and one slow pointer. Fast pointer jumps two nodes at a time and slow pointer jump one node at a time. The point where these two pointer intersect is a point in the loop.
- If that intersection point is head of the list, this is a circular list case and you need to again traverse through the list and make the node before head point to null.
- In the other case, you need to use two pointer variables one starts from head and another starts form the intersection-point. They both will meet at the point of loop. (You can mathematically prove it ;))

Time Complexity: $O(n)$, all operations are linear in nature.

Find Intersection

Problem: In given two-linked list that meet at some point, find that intersection point.



Example 6.27:

```

public Node findIntersection(LinkedList lst2) {
    Node head2 = lst2.head;
    int l1 = 0;
    int l2 = 0;
    Node tempHead = this.head;
    Node tempHead2 = head2;
    while (tempHead != null) {
        l1++;
        tempHead = tempHead.next;
    }
    while (tempHead2 != null) {
        l2++;
        tempHead2 = tempHead2.next;
    }

    int diff;
    if (l1 < l2) {
        Node temp = head;
        head = head2;
        head2 = temp;
        diff = l2 - l1;
    } else {
        diff = l1 - l2;
    }

    for (; diff > 0; diff--) {
        head = head.next;
    }
    while (head != head2) {
        head = head.next;
    }
}

```

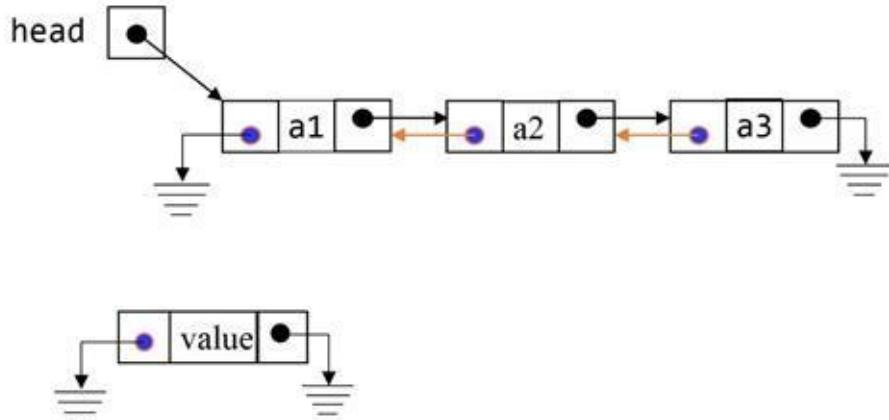
```
    head2 = head2.next;
}
return head;
}
```

Analysis: Find length of both the lists. Find the difference of length of both the lists. Increment the longer list by diff steps, and then increment both the lists and get the intersection point.

Time Complexity: $O(n)$.

Doubly Linked List

In a Doubly Linked list, there are two pointers in each node. These pointers are called prev and next. The prev pointer of the node will point to the node before it and the next pointer will point to the node next to the given node.



Let us look at the Node. The value part of the node is of type integer, but it can be of some other data-type. The two link pointers are prev and next.

Example 6.28: Doubly Linked List node class.

```
public class DoublyLinkedList {  
    private Node head;  
    private Node tail;  
    private int size = 0;  
  
    private static class Node {  
        private int value;  
        private Node next;  
        private Node prev;  
  
        public Node(int v, Node nxt, Node prv) {  
            value = v;  
            next = nxt;  
            prev = prv;  
        }  
  
        public Node(int v) {  
    }
```

```
    value = v;
    next = null;
    prev = null;
}
}

/* Other methods */
}
```

Basic operations of Linked List

Basic operation of a linked list requires traversing a linked list. The various operations that we can perform on linked lists, many of these operations require list traversal:

1. Insert an element in the list, this operation is used to create a linked list.
2. Print various elements of the list.
3. Search an element in the list.
4. Delete an element from the list.
5. Reverse a linked list.

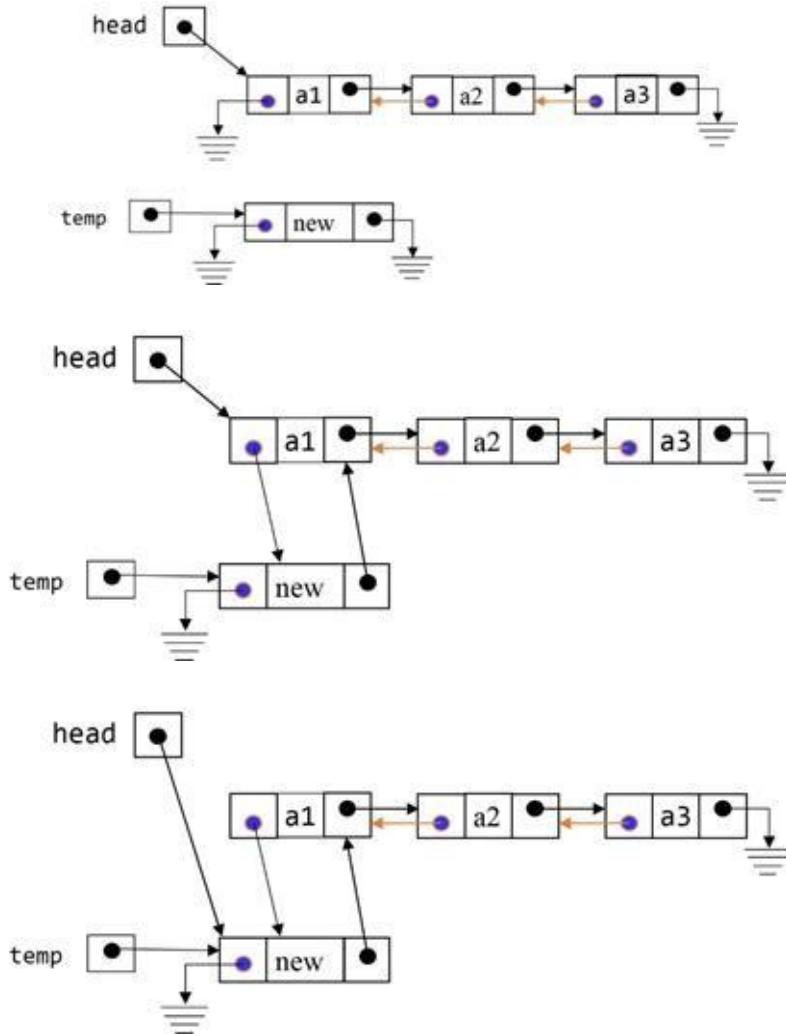
Search in a doubly linked list can only be done in one direction. Since all elements in the list has reference to the next item in the list. Therefore, traversal of linked list is linear in nature.

In doubly linked list linked list below are few cases that we need to keep in mind while coding:

- Zero element case (head will be modified)
- One element case (head can be modified)
- First element (head can be modified)
- General case

Note: Any program that is likely to change head pointer is to be passed as a double reference, which is pointing to head pointer.

Insert at Head



Problem: Insert node at the start of the linked list.

Example 6.29:

```
public void addHead(int value) {
    Node newNode = new Node(value, null, null);
    if (size == 0) {
        tail = head = newNode;
    } else {
        head.prev = newNode;
        newNode.next = head;
        head = newNode;
    }
    size++;
}
```

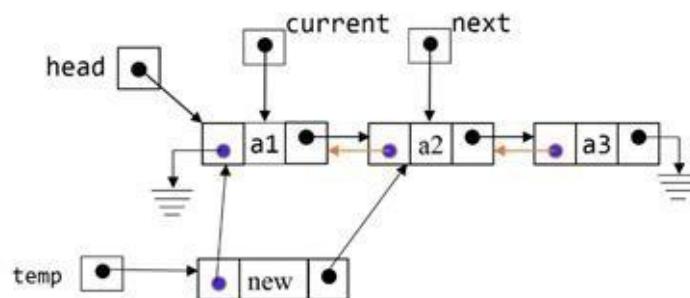
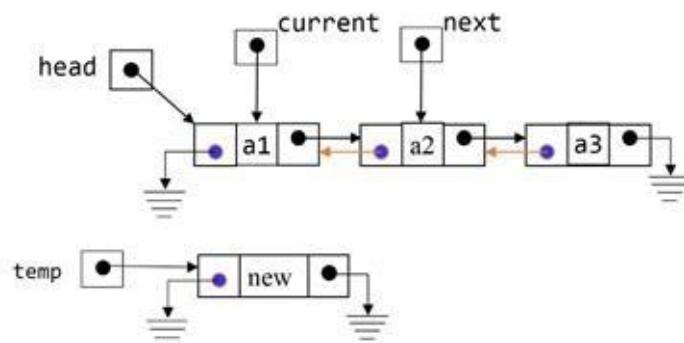
Analysis: Insert in double linked list is similar as insert in a singly linked list.

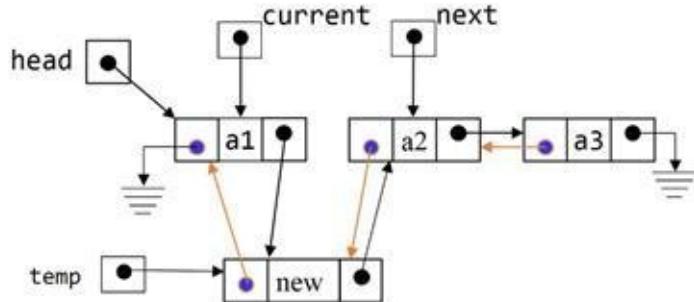
- Create a node assign null to prev pointer of the node.
- If the list is empty then tail and head will point to the new node.
- If the list is not empty then prev of head will point to newNode and next of newNode will point to head. Then head will be modified to point to newNode.

Time Complexity: O(1).

Sorted Insert

Problem: Insert elements in linked list in sorted order.





Example 6.30:

```

public void sortedInsert(int value) {
    Node temp = new Node(value);

    Node curr = head;
    if (curr == null)// first element
    {
        head = temp;
        tail = temp;
    }

    if (head.value <= value)// at the begining
    {
        temp.next = head;
        head.prev = temp;
        head = temp;
    }

    while (curr.next != null && curr.next.value > value)// traversal
    {
        curr = curr.next;
    }

    if (curr.next == null)// at the end
    {
        tail = temp;
        temp.prev = curr;
        curr.next = temp;
    }
}

```

```

} else { // all other
    temp.next = curr.next;
    temp.prev = curr;
    curr.next = temp;
    temp.next.prev = temp;
}
}

```

Analysis:

- We need to consider only element case first. In this case, both head and tail will modify.
- Then we need to consider the case when head will be modified when new node is added to the beginning of the list.
- Then we need to consider general cases
- Finally, we need to consider the case when tail will be modified.

Time Complexity: O(n).

Remove Head

Problem: Remove head node of the linked list.

Example 6.31:

```

public int removeHead() {
    if (isEmpty())
        throw new IllegalStateException("EmptyListException");
    int value = head.value;
    head = head.next;

    if (head == null)
        tail = null;
    else
        head.prev = null;

    size--;
    return value;
}

```

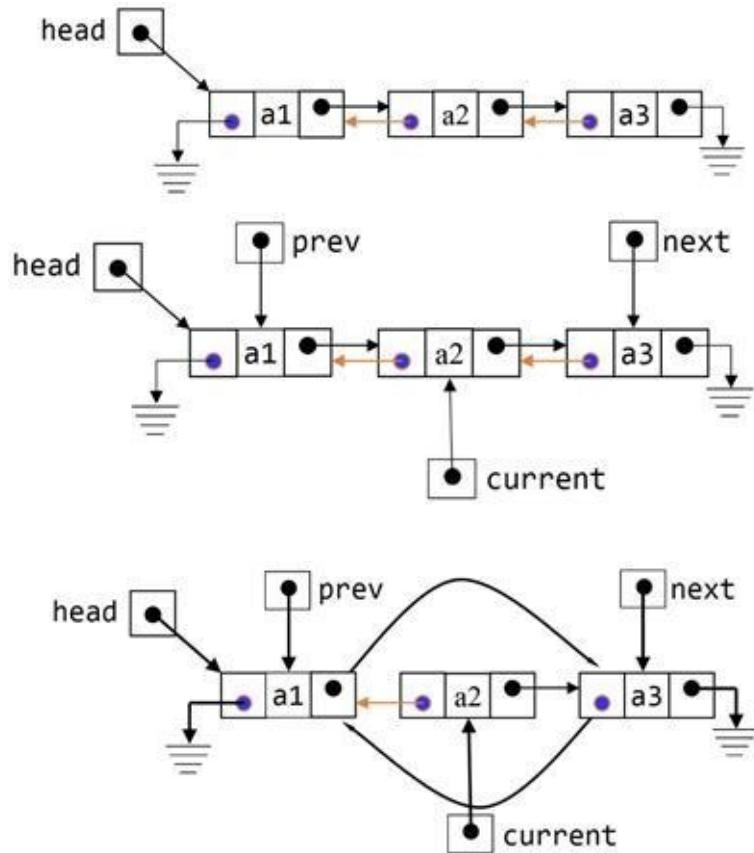
Analysis:

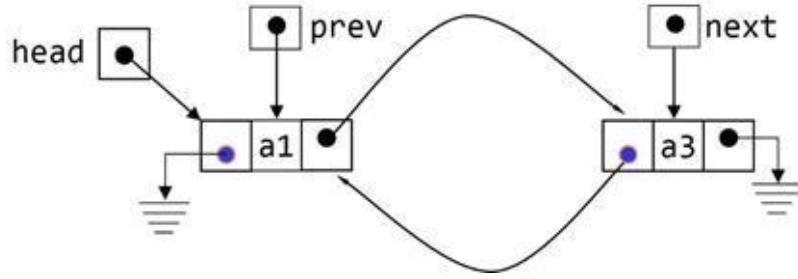
- If the list is empty then we do not do anything.
- Head will point to next of head. If now head is not null then its prev will point to null.
- Update the list so that its head will point to second node.

Time Complexity: O(1).

Delete a node

Problem: Delete node with given value in a linked list





Example 6.32:

```

public boolean removeNode(int key) {
    Node curr = head;
    if (curr == null) // empty list
        return false;

    if (curr.value == key) // head is the node with value key.
    {
        head = head.next;
        size--;
        if (head != null)
            head.prev = null;
        else
            tail = null; // only one element in list.
        return true;
    }

    while (curr.next != null) {
        if (curr.next.value == key) {
            curr.next = curr.next.next;
            if (curr.next == null) // last element case.
                tail = curr;
            else
                curr.next = curr;
            size--;
            return true;
        }
        curr = curr.next;
    }
    return false;
}

```

```
}
```

Analysis: Traverse the list find the node that needs to be removed. Then remove it and adjust next pointer of the node before it and prev pointer of the node next to it.

Time Complexity: $O(n)$.

Remove Duplicate

Problem: Consider the list as sorted remove the repeated value nodes of the list.

Example 6.33:

```
public void removeDuplicate() {  
    Node curr = head;  
    Node deleteMe;  
    while (curr != null) {  
        if ((curr.next != null) && curr.value == curr.next.value) {  
            deleteMe = curr.next;  
            curr.next = deleteMe.next;  
            curr.next.prev = curr;  
            if (deleteMe == tail) {  
                tail = curr;  
            }  
            } else {  
                curr = curr.next;  
            }  
    }  
}
```

Analysis:

- Head can never modify.
- Find the node that have value same as the previous node. Remove this node by pointer adjustment for it.

Time Complexity: $O(n)$.

Reverse a doubly linked List iteratively

Problem: Reverse elements of doubly linked list iteratively.

Example 6.34:

```
public void reverseList() {  
    Node curr = head;  
    Node tempNode;  
    while (curr != null) {  
        tempNode = curr.next;  
        curr.next = curr.prev;  
        curr.prev = tempNode;  
  
        if (curr.prev == null) {  
            tail = head;  
            head = curr;  
            return;  
        }  
        curr = curr.prev;  
    }  
    return;  
}
```

Analysis: Traverse the list. Swap the next and prev. then traverse to the direction curr.prev, which is next before swap. If you reach the end of the list then set head and tail.

Time Complexity: O(n).

Copy List Reversed

Problem: Copy the content of the list into another list in reverse order.

Example 6.35:

```
public DoublyLinkedList copyListReversed() {  
    DoublyLinkedList dll = new DoublyLinkedList();  
    Node curr = head;
```

```

while (curr != null) {
    dll.addHead(curr.value);
    curr = curr.next;
}
return dll;
}

```

Analysis:

- Traverse through the list and copy the value of the nodes into another list by calling addHead() method.
- Since the new nodes are added to the head of the list, the new list formed have nodes order reverse there by making reverse list.

Time Complexity: $O(n)$.

Copy List

Problem: Copy the content of the list into another in same order.

Example 6.36:

```

public DoublyLinkedList copyList() {
    DoublyLinkedList dll = new DoublyLinkedList();
    Node curr = head;

    while (curr != null) {
        dll.addTail(curr.value);
        curr = curr.next;
    }
    return dll;
}

```

Analysis:

- Traverse through the list and copy the value of the nodes into another list by calling addTail() method.
- Since the new nodes are added to the tail of the list, the new list formed have nodes order same as the original list.

Time Complexity: $O(n)$.

Search list

Problem: Search a particular value in a linked list.

Solution: Traverse the list same as done in singly linked list.

Free List

Problem: Free all elements of linked list

Solution: Traverse the list and remove nodes is the same way as done in singly linked list.

Print list

Problem: Print all the elements of linked list.

Solution: Traverse the list and print the value of each node in doubly linked list is same as done in singly linked list.

Find Length

Problem: Find number of elements in linked list.

Solution: Traverse the list and find its length in doubly linked list is same as done in singly linked list.

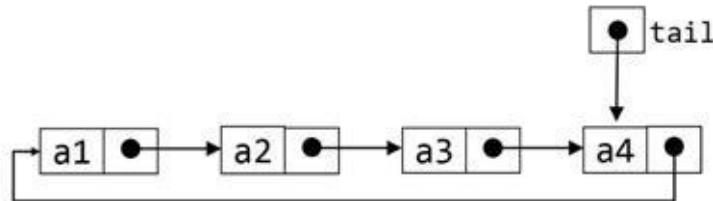
Compare Lists

Problem: Compare two linked lists.

Solution: Traverse both the lists and compare in doubly linked list is same as done in singly linked list.

Circular Linked List

This type is similar to the singly linked list except that the last element points to the first node of the list. The link portion of the last node contains the address of the first node.



Note: In circular linked list, we can insert at the end and remove nodes at the head in constant time. These two operations efficiency makes it appropriate for use as a queue.

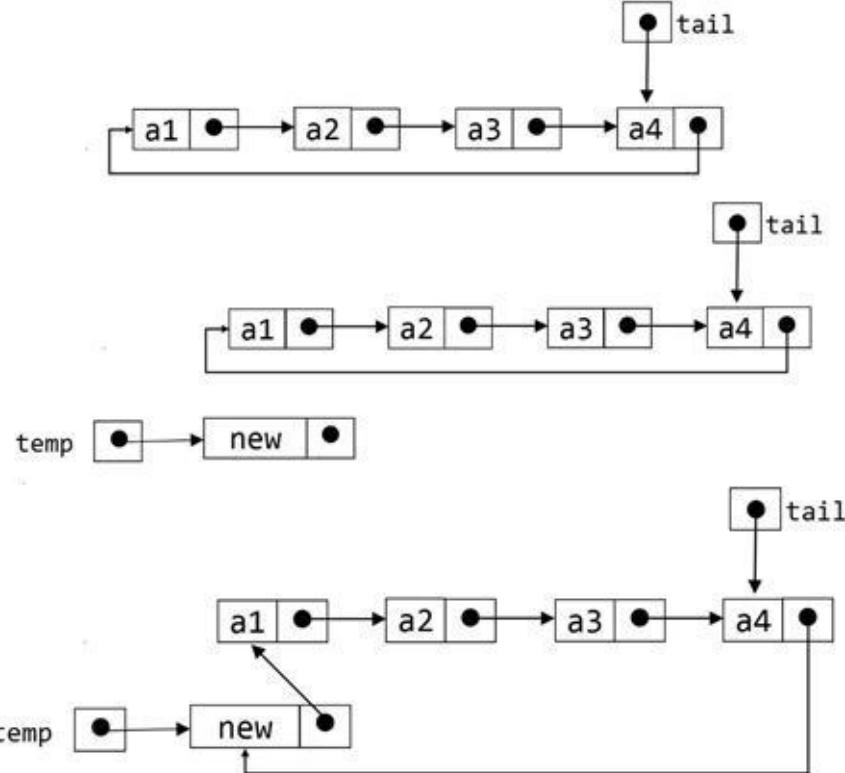
Example 6.37: Circular Linked List node.

```
public class CircularLinkedList {  
    private Node tail;  
    private int size = 0;  
  
    private static class Node {  
        private int value;  
        private Node next;  
  
        public Node(int v, Node n) {  
            value = v;  
            next = n;  
        }  
  
        public Node(int v) {  
            value = v;  
            next = null;  
        }  
    }  
}
```

Analysis: In the circular linked list, node is just like normal singly linked list. The last element node next always point to first node of the list.

Insert element in front

Problem: Add element to the head of circular linked list.



Example 6.38:

```
public void addHead(int value) {
    Node temp = new Node(value, null);
    if (isEmpty()) {
        tail = temp;
        temp.next = temp;
    } else {
        temp.next = tail.next;
        tail.next = temp;
    }
    size++;
}
```

```
public static void main(String[] args) {
```

```

CircularLinkedList ll = new CircularLinkedList();
ll.addHead(1);
ll.addHead(2);
ll.addHead(3);
ll.addHead(1);
ll.addHead(2);
ll.addHead(3);
ll.print();
}

```

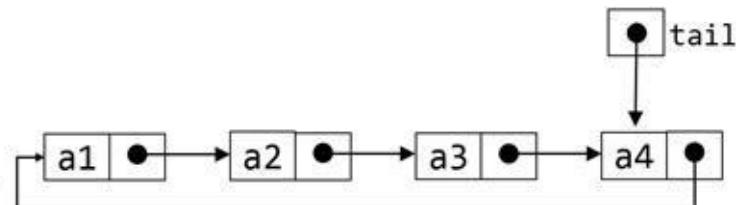
Analysis:

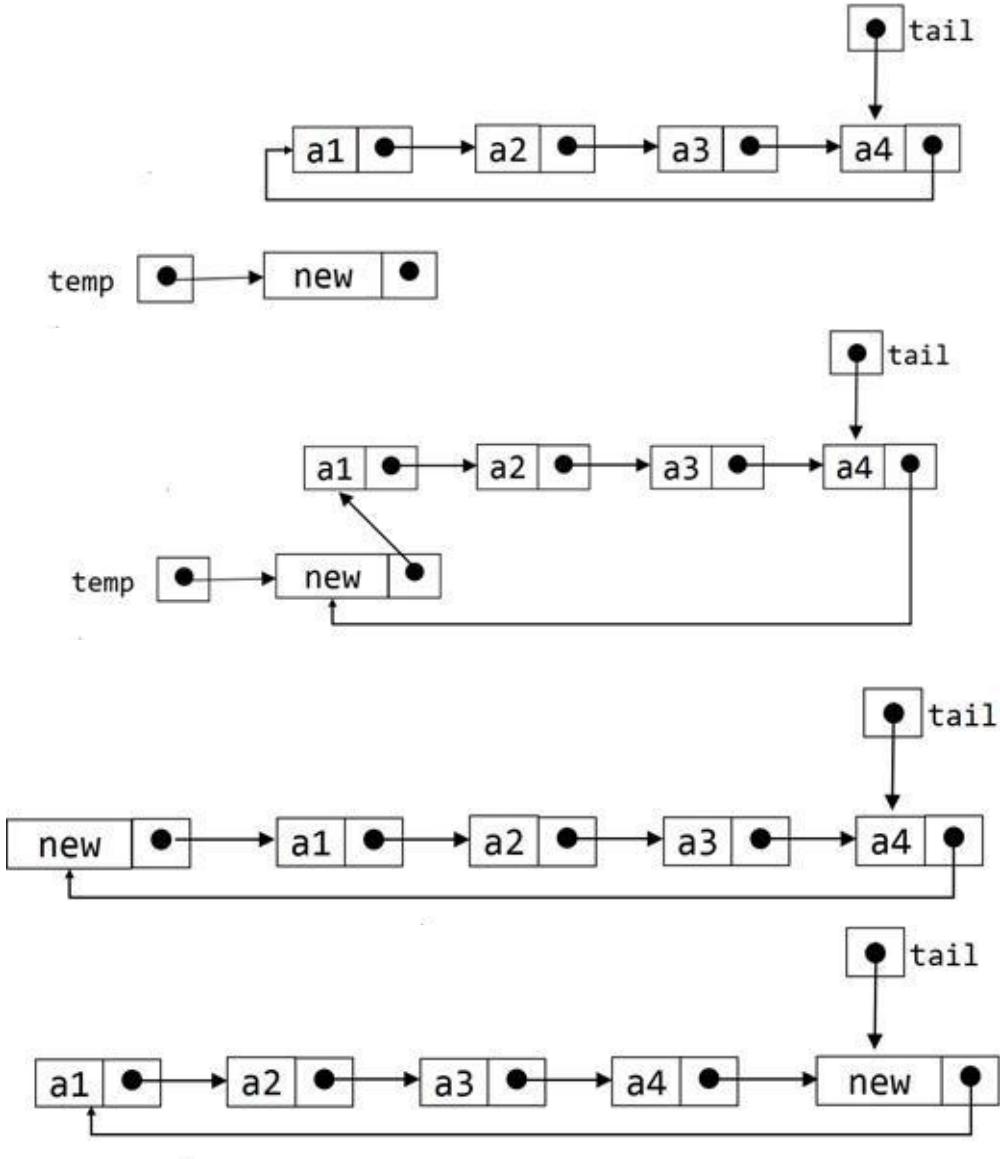
- First, we create node with given value and its next pointing to null.
- If the list is empty then tail of the list will point to it. In addition, the next of node will point to itself
- If the list is not empty then the next of the new node will be next of the tail. In addition, tail next will start pointing to the new node.
- Thus, the new node is added to the head of the list.
- The demo program creates an instance of CircularLinkedList. Then add some value to it and finally print the content of the list.

Time Complexity: O(1).

Insert element at the end

Problem: Insert element at the end of circular linked list.





Example 6.39:

```
public void addTail(int value) {
    Node temp = new Node(value, null);
    if (isEmpty()) {
        tail = temp;
        temp.next = temp;
    } else {
        temp.next = tail.next;
        tail.next = temp;
        tail = temp;
    }
}
```

```
    size++;
}
```

Analysis: Adding node at the end is same as adding at the beginning. We just need to modify tail pointer in place of the head pointer.

Time Complexity: O(1).

Search element in the list

Problem: Search particular value in a circular linked list.

Example 6.40:

```
public boolean searchList(int data) {
    Node temp = tail;
    for (int i = 0; i < size; i++) {
        if (temp.value == data)
            return true;
        temp = temp.next;
    }
    return false;
}
```

Analysis: Iterate through the list to find if particular value is there or not.

Time Complexity: O(n).

Print the content of list

Problem: Print all the elements of a circular linked list.

Example 6.41:

```
public void print() {
    if (isEmpty())
        return;
    Node temp = tail.next;
```

```

while (temp != tail) {
    System.out.print(temp.value + " ");
    temp = temp.next;
}
System.out.print(temp.value);
}

```

Analysis: In circular list, end of list is not there so we cannot check with null. In place of null, tail is used to check end of the list.

Time Complexity: O(n).

Remove element in the front

Problem: Remove first element of the linked list.

Example 6.42:

```

public int removeHead() throws IllegalStateException {
    if (isEmpty()) {
        throw new IllegalStateException("EmptyListException");
    }
    int value = tail.next.value;
    if (tail == tail.next)
        tail = null;
    else
        tail.next = tail.next.next;
    size--;
    return value;
}

```

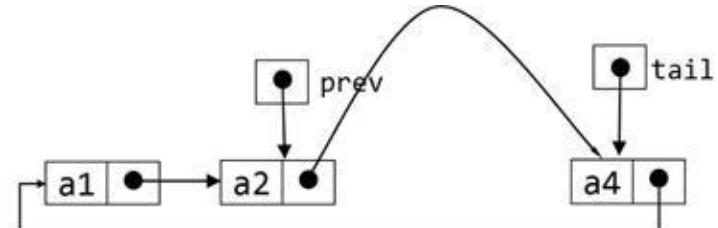
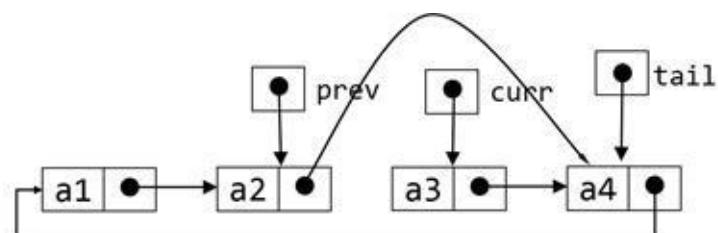
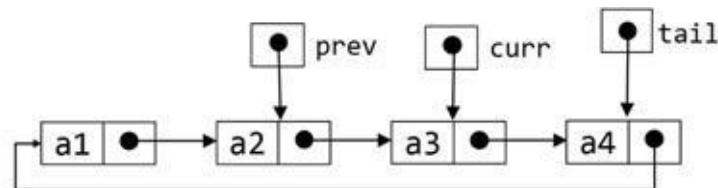
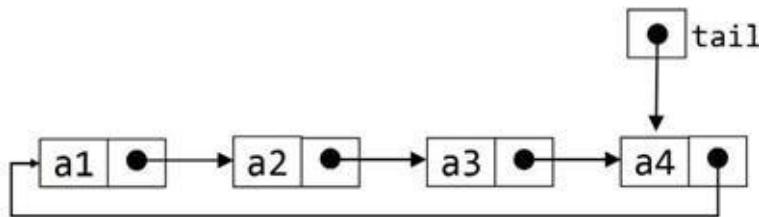
Analysis:

- If the list is empty then return.
- If next node of tail is same as tail then it is a single node case. Mark tail as null.
- If it is not a single node case then point tail node next to second node from the head.

Time Complexity: O(1).

Remove a node given its value

Problem: Delete a node with given value in a circular linked list



Example 6.43:

```
public boolean removeNode(int key) {  
    if (isEmpty()) {  
        return false;  
    }  
    Node prev = tail;
```

```

Node curr = tail.next;
Node head = tail.next;

if (curr.value == key)// head and single node case.
{
if (curr == curr.next)// single node case
tail = null;
else // head case
tail.next = tail.next.next;
return true;
}

prev = curr;
curr = curr.next;

while (curr != head) {
if (curr.value == key) {
if (curr == tail)
tail = prev;
prev.next = curr.next;
return true;
}
prev = curr;
curr = curr.next;
}
return false;
}

```

Analysis: Find the node that needs to be removed. Only difference is that while traversing the list end of list is tracked by the tail pointer in place of null. If the node that need to be removed is other than tail node then tail node pointer is not modified. If the node that need to be removed is tail node then tail node pointer of the list will be modified.

Time Complexity: O(n).

Delete List

Problem: Delete a circular linked list.

Example 6.44:

```
public void deleteList() {  
    tail = null;  
    size = 0;  
}
```

Analysis: Tail node will point to null. The whole list memory is collected by garbage collection.

Time Complexity: O(1).

Copy circular linked-list reversed

Problem: Copy a circular linked list in reversed order.

Example 6.45:

```
public CircularLinkedList copyListReversed() {  
    CircularLinkedList cl = new CircularLinkedList();  
    Node curr = tail.next;  
    Node head = curr;  
  
    if (curr != null) {  
        cl.addHead(curr.value);  
        curr = curr.next;  
    }  
    while (curr != head) {  
        cl.addHead(curr.value);  
        curr = curr.next;  
    }  
    return cl;  
}
```

Time Complexity: O(n).

Copy circular linked list

Problem: Copy a circular linked list.

Example 6.46:

```
public CircularLinkedList copyList() {  
    CircularLinkedList cl = new CircularLinkedList();  
    Node curr = tail.next;  
    Node head = curr;  
  
    if (curr != null) {  
        cl.addTail(curr.value);  
        curr = curr.next;  
    }  
    while (curr != head) {  
        cl.addTail(curr.value);  
        curr = curr.next;  
    }  
    return cl;  
}
```

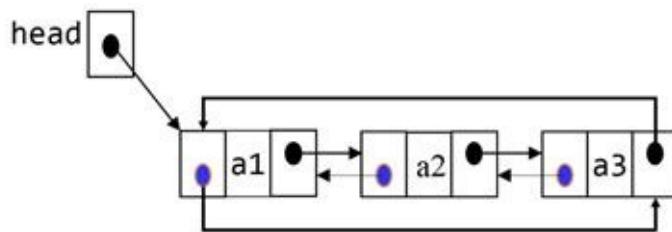
Time Complexity: O(n).

Doubly Circular list

In a Doubly Circular Linked list, there are two pointers in each node. These pointers are called prev and next. The prev pointer of the node will point to the node before it and the next pointer will point to the node next to the given node. The previous node of first element points to the last node of the list and the next pointer of last node points to the first node of the list.

In doubly linked list, we have following cases:

1. Zero element case.
2. Only element.
3. General case
4. Avoid using recursion solutions it makes life harder



Example 6.47: Node of Doubly Circular Linked List.

```
public class DoublyCircularLinkedList {  
    private Node head = null;  
    private Node tail = null;  
    private int size = 0;  
  
    private static class Node {  
        private int value;  
        private Node next;  
        private Node prev;  
  
        public Node(int v, Node nxt, Node prv) {  
            value = v;  
            next = nxt;  
            prev = prv;  
        }  
    }  
}
```

```
public Node(int v) {  
    value = v;  
    next = this;  
    prev = this;  
}  
}  
/* Other methods */  
}
```

Analysis: The node of doubly circular linked list is same as node of doubly linked list. Only difference is that the previous pointer of first node points to last node of list and next pointer of last node points to first node of the list.

Insert Node at head

Problem: Insert value at the front of the list.

Example 6.48:

```
public void addHead(int value) {  
    Node newNode = new Node(value, null, null);  
    if (size == 0) {  
        tail = head = newNode;  
        newNode.next = newNode;  
        newNode.prev = newNode;  
    } else {  
        newNode.next = head;  
        newNode.prev = head.prev;  
        head.prev = newNode;  
        newNode.prev.next = newNode;  
        head = newNode;  
    }  
    size++;  
}
```

Analysis:

- A new node is created and if the list is empty then head and tail will point to it.

The newly created newNode's next and prev also point to newNode.

- If the list is not empty then the pointers are adjusted and a new node is added to the front of the list. Only head needs to be changed in this case.
- Size of the list is increased by one.

Time Complexity: O(1).

Insert Node at tail

Problem: Insert value at the end of a linked list.

Example 6.49:

```
public void addTail(int value) {  
    Node newNode = new Node(value, null, null);  
    if (size == 0) {  
        head = tail = newNode;  
        newNode.next = newNode;  
        newNode.prev = newNode;  
    } else {  
        newNode.next = tail.next;  
        newNode.prev = tail;  
        tail.next = newNode;  
        newNode.next.prev = newNode;  
        tail = newNode;  
    }  
    size++;  
}
```

Analysis:

- A new node is created and if the list is empty then head and tail will point to it. The newly created newNode's next and prev also point to newNode.
- If the list is not empty then the pointers are adjusted and a new node is added to the end of the list. Only tail needs to be changed in this case.
- Size of the list is increased by one.

Time Complexity: O(1).

Remove head node

Problem: Remove node at the head of a doubly circular linked list

Example 6.50:

```
public int removeHead() {  
    if (size == 0)  
        throw new IllegalStateException("EmptyListException");  
    int value = head.value;  
    size--;  
    if (size == 0) {  
        head = null;  
        tail = null;  
        return value;  
    }  
  
    Node next = head.next;  
    next.prev = tail;  
    tail.next = next;  
    head = next;  
    return value;  
}
```

Analysis: Remove node in a doubly circular linked list is just same as remove node in a circular linked list. Just few extra pointer need to be adjusted.

Time Complexity: O(1).

Remove tail node

Problem: Remove tail node of a linked list.

Example 6.51:

```
public int removeTail() {  
    if (size == 0)  
        throw new IllegalStateException("EmptyListException");  
    int value = tail.value;
```

```

size--;
if (size == 0) {
head = null;
tail = null;
return value;
}

Node prev = tail.prev;
prev.next = head;
head.prev = prev;
tail = prev;
return value;
}

```

Analysis: In case of empty list return. In case of single node case head and tail will store null. In other cases next of second last node will point to head, previous of head will point to second last node.

Time Complexity: O(1).

Delete list

Problem: Free a linked list.

Solution: Delete complete list is same as given in circular linked list.

Search value

Problem: Search a value in doubly circular linked list.

Solution: Algorithm for searching a value in doubly circular linked list is same as circular linked list.

Print List

Problem: Print all the elements of linked list.

Solution: Algorithm to print content of doubly circular list is same as given in circular linked list.

Exercise

1. Insert an element at k^{th} position from the start of linked list. Return true if success and if list is not long enough, then return -1.

Hint: Take a pointer of head and then advance it by K steps forward, and inserts the node.

2. Insert an element at k^{th} position from the end of linked list. Return true if success and if list is not long enough, then return -1.

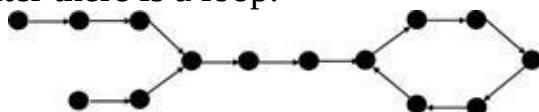
Hint: Take a pointer of head and then advance it by K steps forward, then take another pointer and then advance both simultaneously, so that when the first pointer reaches the end of a linked list then second pointer is at the point where you need to insert the node.

3. Consider there is a loop in a linked list, Write a program to remove loop if there is a loop in this linked list.

4. In the above SearchList program return, the count of how many instances of same value are found else if value not found then return 0. For example, if the value passed is “4”. The elements in the list are 1,2,4,3 & 4. The program should return 2.

Hint: In place of return true in the above program increment a counter and then return its value.

5. Given two linked list head-pointer and they meet at some point and need to find the point of intersection. However, in place of the end of both the linked list to be a null pointer there is a loop.



6. If linked list having a loop is given. Count the number of nodes in the linked list

7. We were supposed to write the complete code for the addition of polynomials using Linked Lists. This takes time if you do not have it by heart, so revise it well.

8. In given two linked lists. We have to find whether the data in one is reverse that of data in another. No extra space should be used and traverse the linked lists only once.

9. Find the middle element in a singly linked list. Tell the complexity of your solution.

Hint:-

First solution: Find the length of linked list. Then find the middle element and return it.

Second solution: Use two pointer one will move fast and another will move slow, make sure you handle border case properly. (Even length and odd length linked list cases.)

10. Print list in reverse order.

Hint: Use recursion.

11. In a huge linked list, you are given a pointer to some middle node. Write a program to remove this node.

Hint: Copy the values of next node to current node. Then remove next node.

12. Given a special list, whose node have extra pointer random which point to some other node in linked list. Create another list that is copy of the given list. Also, make sure that random pointer is also allocated to respective node in new list.

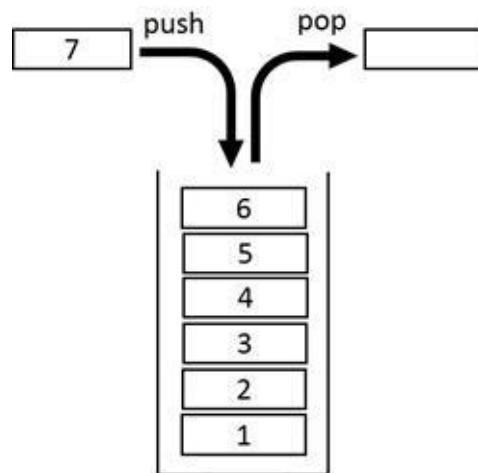
Hint: Traverse the list and go on adding nodes to the list, which have same value but have random pointer points to null. Now traverse the modified list again. Let us call the node, which was already present in the list as old node and the node, which is added as new node. Find the random pointer node pointed of old node and assign its next node to the random pointer of new node. Follow this procedure for all the nodes. Finally separate the old and new nodes. The list formed by new nodes is the desired list.

CHAPTER 7: STACK

Introduction

A stack is a basic data structure that organizes items in last-in-first-out (LIFO) manner. Last element inserted in a stack will be the first to be removed from it. The real-life analogy of the stack is "stack of plates". Imagine a stack of plates in a dining area everybody takes a plate at the top of the stack, thereby uncovering the next plate for the next person.

Stack allow to only access the top element. The elements that are at the bottom of the stack are the one that is going to stay in the stack for the longest time.



Computer science also has the common example of a stack. Function call stack is a good example of a stack. Function main() calls function foo() and then foo() calls bar(). These function calls are implemented using stack. First, bar() exists, then foo() and then finally main().

As we navigate from web page to web page, the URL of web pages are kept in a stack, with the current page URL at the top. If we click back button, then each URL entry is popped one by one.

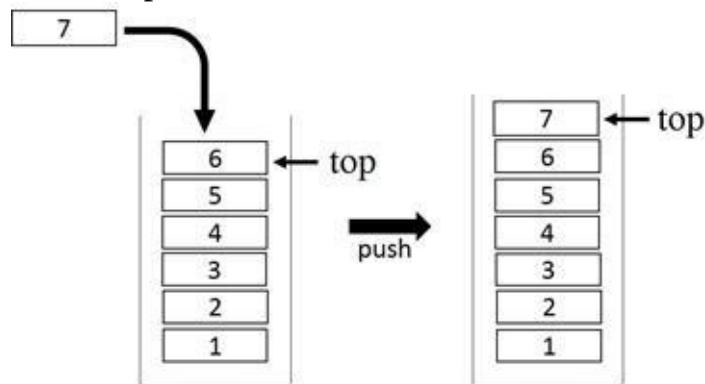
The Stack Abstract Data Type

Stack abstract data type is defined a class, which follows LIFO or last-in-first-out for the elements, added to it.

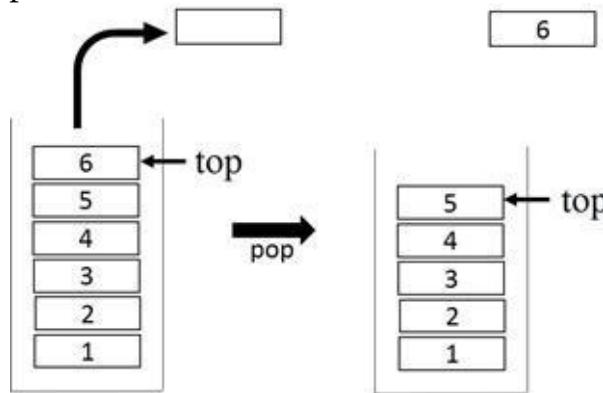
The stack should support the following operations:

1. Push(): Which adds a single element at the top of the stack
2. Pop(): Which removes a single element from the top of a stack.
3. Top(): Reads the value of the top element of the stack (does not remove it)
4. isEmpty(): Returns 1 if stack is empty
5. Size(): Returns the number of elements in a stack.

Push : Add value to the top of a stack



Pop : Remove the top element of the stack and return it to the caller function.



The stack can be implemented using an array or a linked list.

1. When stack is implemented using arrays, its capacity is fixed.
2. In case of a linked list, there is no such limit on the number of elements it can contain.

When a stack is implemented, using an array top of the stack is managed using an index variable called top.

When a stack is implemented using a linked list, push() and pop() is implemented using insert at the head of the linked list and remove from the head of the linked list.

Stack using array

Problem: Implement a stack using a fixed length array.

Example 7.1: Stack implementation using array

```
public class Stack {  
    private int capacity = 1000;  
    private int[] data;  
    private int top = -1;  
    public Stack() {  
        data = new int[capacity];  
    }  
  
    public Stack(int size) {  
        data = new int[size];  
        capacity = size;  
    }  
    /* Other Methods */  
}
```

isEmpty() function returns true if stack is empty or false in all other cases.

```
public boolean isEmpty() {  
    return (top == -1);  
}
```

size() function returns the number of elements in the stack.

```
public int size() {  
    return (top + 1);  
}
```

The print function will print the elements of the array.

```
public void print() {  
    for (int i = top; i > -1; i--) {  
        System.out.print(" " + data[i]);  
    }  
}
```

push() function append value to the data.

```
public void push(int value) throws IllegalStateException {  
    if (size() == data.length) {  
        throw new IllegalStateException("StackOverflowException"); }  
    top++;  
    data[top] = value;  
}
```

In the **pop()** function, first it will check that the stack is not empty. Then it will pop the value from data array and return it.

```
public int pop() {  
    if (isEmpty()) {  
        throw new IllegalStateException("StackEmptyException"); }  
    int topVal = data[top];  
    top--;  
    return topVal;  
}
```

top() function returns the value of stored in the top element of stack (does not remove it)

```
public int top() throws IllegalStateException {  
    if (isEmpty()) {  
        throw new IllegalStateException("StackEmptyException"); }  
    return data[top];  
}
```

Test code for stack.

```
public static void main(String[] args) {  
    Stack s = new Stack(); s.push(1); s.push(2); s.push(3); s.print();  
    System.out.println(s.pop()); System.out.println(s.pop()); s.print(); }
```

Analysis:

- Stack object is declared and initialized.
- push() and pop() functions to are used to add and remove values to the stack.

Stack using array with memory management

Problem: Create a stack using an array, but the memory of the array need to be managed. When the stack is full its capacity is doubled. Also when the number of elements fall below capacity/2, the capacity of stack is halved. Also we do not want to let the capacity of the stack below the initially allocated size. We can define min length when stack is created.

Example 7.2:

```
public class Stack2 {  
    private int[] data;  
    private int top = -1;  
    private int minCapacity;  
    private int capacity;  
  
    public Stack2() {  
        this(1000);  
    }  
  
    public Stack2(int size) {  
        data = new int[size];  
        capacity = minCapacity = size;  
    }  
    /* Other methods */  
}
```

Analysis:

- Stack class contains four fields. “top” which is index of topmost element, a pointer “data” that will point to dynamically allocated memory. “size” which is used to store the size of stack and “capacity” which is the capacity of stack.

Push and pop function are implemented next. All the other functions like, isEmpty(), size() and top() will remain same.

Push function growing capacity implementation

Implementing push() function of stack such that when the stack is filled, then double the capacity of the stack.

Example 7.3:

```
public void push(int value) throws IllegalStateException {  
    if (size() == capacity) {  
        System.out.println("size doubled");  
        int[] newData = new int[capacity * 2];  
        System.arraycopy(data, 0, newData, 0, capacity);  
        data = newData;  
        capacity = capacity * 2;  
    }  
    top++;  
    data[top] = value;  
}
```

Analysis:

- We are doubling the max capacity variable of the stack.
- Size of the memory is reallocated to a bigger value in this line. The realloc function is used to increase or decrease the size of memory allocated. All the data on the stack is copied into the new location.
- Finally the StackPush() function is called recursively to push the value into the increased capacity stack.

Pop function reducing capacity implementation

Implementing pop() function such that when count of elements in stack fall below capacity/2 the capacity of stack is reduced.

Example 7.4:

```
public int pop() {  
    if (isEmpty()) {  
        throw new IllegalStateException("StackEmptyException"); }  
  
    int topVal = data[top];  
    top--;  
    if (size() == capacity / 2 && capacity > minCapacity) {
```

```
System.out.println("size halved");
capacity = capacity / 2;
int[] newData = new int[capacity];
System.arraycopy(data, 0, newData, 0, capacity);
data = newData;
}
return topVal;
}
```

Analysis: The size of the stack is checked if it goes below half of the capacity and is greater than the minimum size. The capacity of the stack is divided by 2 and memory is reallocated to point to half size memory.

Stack using linked list

Problem: Stack can be implemented using a linked list. Add elements at the head of list and remove from the head of the list.

Example 7.5: Implement stack using a linked list.

```
public class StackLL {  
    private Node head = null;  
    private int size = 0;  
  
    private static class Node {  
        private int value;  
        private Node next;  
  
        public Node(int v, Node n) {  
            value = v;  
            next = n;  
        }  
    }  
  
    public int size() {  
        return size;  
    }  
  
    public boolean isEmpty() {  
        return size == 0;  
    }  
  
    public int peek() throws IllegalStateException {  
        if (isEmpty()) {  
            throw new IllegalStateException("StackEmptyException");  
        }  
        return head.value;  
    }  
  
    public void push(int value) {  
        head = new Node(value, head);  
    }  
}
```

```

size++;
}

public int pop() throws IllegalStateException {
if (isEmpty()) {
throw new IllegalStateException("StackEmptyException");
}

int value = head.value;
head = head.next;
size--;
return value;
}

public void print() {
Node temp = head;
while (temp != null) {
System.out.print(temp.value + " ");
temp = temp.next;
}
}
}

public static void main(String[] args) {
StackLL s = new StackLL();
s.push(1); s.push(2); s.push(3); s.print(); System.out.println(s.pop());
System.out.println(s.pop()); s.print();
}
}

```

Analysis:

- Stack implemented using a linked list is simply insertion and deletion at the head of a singly linked list.
- In StackPush() function, memory is created for one node. Then the value is stored into that node. Finally, the node is inserted at the beginning of the list.
- In StackPop() function, the head of the linked list starts pointing to the second node. And the value of first node is returned.

System stack and Method Calls

Methods calls are implemented using stack called system stack. When a method is called, the current execution is stopped and the control goes to the called method. After the called method exits / returns, the execution resumes from the point at which the execution was stopped.

To get the exact point at which execution should be resumed, the address of the next instruction is stored in the system stack. When the method call completes, the address at the top of the stack is taken out.

Example 7.6: Function call flow

```
public void function2() {  
    System.out.println("fun2 line 1");  
}  
  
public void function1() {  
    System.out.println("fun1 line 1");  
    function2();  
    System.out.println("fun1 line 2");  
}  
  
public void main() {  
    System.out.println("main line 1");  
    function1();  
    System.out.println("main line 2");  
}
```

Output:

```
main line 1  
fun1 line 1  
fun2 line 1  
fun1 line 2  
main line 2
```

Analysis:

- This program starts with main() method.

- The first statement of main() will be executed. This will print “main line 1” as output.
- Then statement function1() is called. Before control goes to function1() then next instruction that is address of next line is stored in the system stack.
- Control goes to function1() method.
- The first statement inside function1() is executed, this will print “fun1 line 1” to output.
- function2() is called from function1(). Before control goes to function2() address of the next instruction that is address of next line is added to the system stack.
- Control goes to function2() method.
- “fun2 line 1” is printed to screen.
- When function2() exits, control come back to function1(). Then the program reads the next instruction from the stack, and the next line is executed and print “fun1 line 2” to screen.
- When fun1 exits, control comes back to the main method. Then program reads the next instruction from the stack and executes it and finally “main line 2” is printed to screen.

Points to remember:

1. Methods are implemented using a stack.
2. When a method is called the address of the next instruction is pushed into the stack.
3. When a method is finished the address of the execution is taken out of the stack.

Problems in Stack

Sorted Insert

Problem: Given a stack whose elements are sorted, write a function which will insert elements in sorted order. With highest at the top and lowest at the bottom.

Solution: Pop elements from stack till the top of stack is greater than the current value. Then add current value and then add the popped elements to the stack again. Time complexity $O(n)$

Example 7.7:

```
public static void sortedInsert(Stack<Integer> stk, int element) {  
    int temp;  
    if (stk.isEmpty() || element > stk.peek())  
        stk.push(element);  
    else {  
        temp = stk.pop();  
        sortedInsert(stk, element);  
        stk.push(temp);  
    }  
}
```

Sort Stack

Problem: Given a stack, sort elements such that largest value is at the top.

First solution: Recursively call sort stack function. Then call sorted insert function inside recursion to make the stack sorted. Time complexity is $O(n^2)$

Example 7.8:

```
public static void sortStack(Stack<Integer> stk) {  
    int temp;  
    if (stk.isEmpty() == false) {  
        temp = stk.pop();  
        sortStack(stk);  
        sortedInsert(stk, temp);  
    }  
}
```

```

sortStack(stk);
stk.push(temp);
}
}

```

Second solution: Same problem can be solved iteratively by using another stack stk2. We use a stack stk2 which will store elements always in sorted order. Each time new element is popped from the original input stack and get added to the new stack. Time Complexity is $O(n^2)$

Example 7.9:

```

public static void sortStack2(Stack<Integer> stk) {
    int temp;
    Stack<Integer> stk2 = new Stack<Integer>();
    while (stk.isEmpty() == false) {
        temp = stk.pop();
        while ((stk.isEmpty() == false) && (stk2.peek() < temp))
            stk.push(stk2.pop()); stk2.push(temp);
    }
    while (stk2.isEmpty() == false)
        stk.push(stk2.pop());
}

```

Bottom Insert

Problem: Given stack write a function to insert element at the bottom of the stack.

Solution: Pop elements from stack recursively till it is empty. Once empty then insert the input value in the stack .Then again put the popped values back to the stack. Time complexity is $O(n)$

Example 7.10:

```

public static void bottomInsert(Stack<Integer> stk, int element) {
    int temp;
    if (stk.isEmpty())
        stk.push(element);
    else {

```

```

temp = stk.pop();
bottomInsert(stk, element);
stk.push(temp);
}
}

```

Reverse Stack

Problem: Given a stack, reverse its elements.

First solution: Using recursion, in a recursion pop elements from stack and use bottomInsert function to add them at the bottom. Finally the whole stack will be reversed. Time complexity is $O(n^2)$

Example 7.11:

```

public static <T> void reverseStack(Stack<T> stk) {
    if (stk.isEmpty()) {
        return;
    } else {

        T value = stk.pop();
        reverseStack(stk);
        insertAtBottom(stk, value);
    }
}

```

Second solution: Using queue, pop all the elements of the stack insert them into queue then remove elements from queue and insert them into stack and stack elements will be reversed. Time complexity is $O(n)$

Example 7.12:

```

public static void reverseStack2(Stack<Integer> stk) {
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();
    while (stk.isEmpty() == false)
        que.add(stk.pop());
    while (que.isEmpty() == false)
        stk.push(que.remove());
}

```

Reverse K Element in a Stack

Problem: Reverse k elements at the top of the stack.

Solution: Create a queue, pop k elements from stack and add them to queue. Then remove from queue and push them back to the stack. Top k elements will be reversed. Time complexity is O(n)

Example 7.13:

```
public static void reverseKElementInStack(Stack<Integer> stk, int k) {  
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();  
    int i = 0;  
    while (stk.isEmpty() == false && i < k) {  
        que.add(stk.pop()); i++;  
    }  
    while (que.isEmpty() == false)  
        stk.push(que.remove()); }
```

Reverse Queue

Problem: Reverse elements of a queue.

Solution: Remove all the elements of the queue and push them to the stack till queue is empty. Then pop elements from stack and add them to the queue. Finally all the elements of queue are reversed. Time complexity is O(n)

Example 7.14:

```
public static void reverseQueue(ArrayDeque<Integer> que) {  
    Stack<Integer> stk = new Stack<Integer>();  
    while (que.isEmpty() == false)  
        stk.push(que.remove());  
    while (stk.isEmpty() == false)  
        que.add(stk.pop()); }
```

Reverse K Element in a Queue

Problem: Reverse first K elements in a queue.

Solution: Use a stack, remove first K elements of queue and push them into stack. Then pop the element from stack and add them to the queue. The K elements are reversed but they are added to the end of queue. So we need to bring those element at the start of queue by removing the elements at the front and adding them at the back of queue. Time complexity is O(n)

Example 7.15:

```
public static void reverseKElementInQueue(ArrayDeque<Integer> que, int k)
{
    Stack<Integer> stk = new Stack<Integer>();
    int i = 0, diff, temp;
    while (que.isEmpty() == false && i < k) {
        stk.push(que.remove()); i++;
    }

    while (stk.isEmpty() == false) {
        que.add(stk.pop());
    }

    diff = que.size() - k;
    while (diff > 0) {
        temp = que.remove();
        que.add(temp);
        diff -= 1;
    }
}
```

Two stacks using single list

Problem: How to implement two stacks using one single list.

Solution: Dequeue is used. First stack is used by adding to the left and removing elements from left. Second stack is used by adding to the right and removing element from right. Push and pop of both the stack has time complexity of O(1).

Example 7.16:

```
public class TwoStack {  
    private final int MAX_SIZE = 50;  
    int top1;  
    int top2;  
    int[] data;  
  
    public TwoStack() {  
        top1 = -1;  
        top2 = MAX_SIZE;  
        data = new int[MAX_SIZE];  
    }  
  
    public void StackPush1(int value) {  
        if (top1 < top2 - 1) {  
            data[++top1] = value;  
        } else {  
            System.out.print("Stack is Full!");  
        }  
    }  
  
    public void StackPush2(int value) {  
        if (top1 < top2 - 1) {  
            data[--top2] = value;  
        } else {  
            System.out.print("Stack is Full!");  
        }  
    }  
  
    public int StackPop1() {  
        if (top1 >= 0) {  
            int value = data[top1--];  
            return value;  
        } else {  
            System.out.print("Stack Empty!");  
        }  
        return -999;  
    }  
}
```

```

public int StackPop2() {
    if (top2 < MAX_SIZE) {
        int value = data[top2++];
        return value;
    } else {
        System.out.print("Stack Empty!");
    }
    return -999;
}

public static void main(String[] args) {
    TwoStack st = new TwoStack();
    for (int i = 0; i < 10; i++) {
        st.StackPush1(i);
    }
    for (int j = 0; j < 10; j++) {
        st.StackPush2(j + 10);
    }
    for (int i = 0; i < 10; i++) {
        System.out.println("stack one pop value: " + st.StackPop1());
        System.out.println("stack two pop value: " + st.StackPop2());
    }
}
}

```

Analysis: Same list is used to implement two stack. First stack is filled from the beginning of the array and second stack is filled from the end of the array. Overflow and underflow conditions need to be taken care of carefully.

Min stack

Problem: Design a stack in which we can get minimum value in stack should also work in **O(1)** Time Complexity.

Hint: Keep two stack one will be general stack, which will just keep the elements. The second will keep the min value.

1. Push: Push an element to the top of stack1. Compare the new value with the value at the top of the stack2. If the new value is smaller, then push the new value into stack2. Or push the value at the top of the stack2 to itself once more.
2. Pop: Pop an element from top of stack1 and return. Pop an element from top of stack2 too.
3. Min: Reads from the top of the stack2 this value will be the min.

Stack using a queue

Problem: How to implement a stack using a queue. Analyze the running time of the stack operations.

See queue chapter for its solution.

Balanced Parenthesis

Problem: Write a program to check balanced symbols (such as {}, (), []). The closing symbol should be matched with the most recently seen opening symbol. e.g. {{}} is legal, {{()({})}} is legal, but {{() and {()}} are not legal

Example 7.17:

```
public static boolean isBalancedParenthesis(String expn) {
    Stack<Character> stk = new Stack<Character>();
    for (char ch : expn.toCharArray()) {
        switch (ch) {
            case '{':
            case '[':
            case '(':
                stk.push(ch);
                break;
            case '}':
            case ']':
            case ')':
                if (stk.pop() != '{') {
                    return false;
                }
                break;
            case ')':
                if (stk.pop() != '(') {
                    return false;
                }
                break;
        }
    }
    return stk.isEmpty();
}
```

```

if (stk.pop() != '[') {
    return false;
}
break;
case ')':
if (stk.pop() != '(') {
    return false;
}
break;
}
return stk.isEmpty();
}

public static void main(String[] args) {
    String expn = "{}[]";
    boolean value = isBalancedParenthesis(expn); System.out.println("Given
Expn:" + expn);
    System.out.println("Result after isParenthesisMatched:" + value);
}

```

Analysis:

- Traverse the input string when we get an opening parenthesis we push it into stack. Moreover, when we get a closing parenthesis then we pop a parenthesis from the stack and compare if it is the corresponding closing parenthesis.
- We return false if there is a mismatch of parenthesis.
- If at the end of the whole staring traversal, we reach to the end of the string and the stack is empty then we have balanced parenthesis.

Max Depth Parenthesis

Problem: Given a balanced parenthesis expression you need to find maximum depth of parenthesis.

For example “()” maximum depth is 1 and for “((())())” maximum depth is 3.

First solution: Create a stack, when we get opening parenthesis then we insert it to the stack and increase depth counter. When we get a closing parenthesis then

we pop opening parenthesis from stack and decrease the depth counter. We keep track of depth to find maximum depth.

Example 7.18:

```
public static int maxDepthParenthesis(String expn, int size) {  
    Stack<Character> stk = new Stack<Character>();  
    int maxDepth = 0;  
    int depth = 0;  
    char ch;  
  
    for (int i = 0; i < size; i++) {  
        ch = expn.charAt(i);  
  
        if (ch == '(') {  
            stk.push(ch);  
            depth += 1;  
        } else if (ch == ')') {  
            stk.pop();  
            depth -= 1;  
        }  
        if (depth > maxDepth)  
            maxDepth = depth;  
    }  
    return maxDepth;  
}  
  
public static void main(String[] args) {  
    String expn = "((((A)))((((BBB())))))OOOO";  
    int size = expn.length();  
    int value = maxDepthParenthesis(expn, size);  
    System.out.println("Max depth parenthesis is " + value);  
}
```

Second solution: We do not need to find if the expression is balanced or not. It is given that it is balanced so we don't need a stack as shown in previous solution.

Example 7.19:

```
public static int maxDepthParenthesis2(String expn, int size) {  
    int maxDepth = 0;  
    int depth = 0;  
    char ch;  
    for (int i = 0; i < size; i++) {  
        ch = expn.charAt(i);  
        if (ch == '(')  
            depth += 1;  
        else if (ch == ')')  
            depth -= 1;  
  
        if (depth > maxDepth)  
            maxDepth = depth;  
    }  
    return maxDepth;  
}
```

Longest Continuous Balanced Parenthesis

Problem: Given a string of opening and closing parenthesis, you need to find a sub-string which have balanced parenthesis.

Example 7.20:

```
public static int longestContBalParen(String string, int size) {  
    Stack<Integer> stk = new Stack<Integer>();  
    stk.push(-1);  
    int length = 0;  
  
    for (int i = 0; i < size; i++) {  
  
        if (string.charAt(i) == '(')  
            stk.push(i);  
        else // string[i] == ')'  
        {  
            stk.pop();  
            if (stk.size() != 0)
```

```

length = Math.max(length, i - stk.peek());
else
    stk.push(i);
}
}
return length;
}

public static void main(String[] args) {
    String expn = "())(((())())()";
    int size = expn.length();
    int value = longestContBalParen(expn, size);
    System.out.println("longestContBalParen " + value);
}

```

Reverse Parenthesis

Problem: How many reversal will be needed to make an unbalanced expression to balanced expression.

Input: ")())(("

Output: 3

Input: ")((("

Output: 3

Solution:

- First the parenthesis which are already balanced don't need to be flipped so they are first removed. What all the parenthesis which are balanced is removed then we will have the parenthesis of the form "...))((((...".
- So let's call the total number of open parenthesis is OpenCount and total number of closed parenthesis is CloseCount.
- When OpenCount is even CloseCount is also even and their half element reversal will make the expression balanced.
- When OpenCount is odd and also CloseCount, then once you had reversed OpenCount/2 and CloseCount/2. You will be left with ")" which need 2 more reversal, so the formula is derived from this.

$$\text{Total number of reversal} = \text{math.ceil}(\text{OpenCount} / 2.0) +$$

`math.ceil(CloseCount/2.0)`

Example 7.21:

```
public static int reverseParenthesis(String expn, int size) {  
    Stack<Character> stk = new Stack<Character>();  
    int openCount = 0;  
    int closeCount = 0;  
    char ch;  
  
    if (size % 2 == 1) {  
        System.out.println("Invalid odd length " + size);  
        return -1;  
    }  
    for (int i = 0; i < size; i++) {  
        ch = expn.charAt(i);  
        if (ch == '(')  
            stk.push(ch);  
        else if (ch == ')')  
            if (stk.size() != 0 && stk.peek() == '(')  
                stk.pop();  
            else  
                stk.push(')');  
        }  
        while (stk.size() != 0) {  
            if (stk.pop() == '(')  
                openCount += 1;  
            else  
                closeCount += 1;  
        }  
        int reversal = (int) Math.ceil(openCount / 2.0) + (int) Math.ceil(closeCount /  
2.0);  
        return reversal;  
    }  
  
    public static void main(String[] args) {  
        String expn = "())(((())())());";  
        String expn2 = ")())((";
```

```

int size = expn2.length();
int value = reverseParenthesis(expn2, size);
System.out.println("Given expn : " + expn2);
System.out.println("reverse Parenthesis is : " + value);
}

```

Find Duplicate Parenthesis

Problem: Given an expression, you need to find duplicate or redundant parenthesis in it. Redundant parenthesis are those parenthesis pair which does not change the outcome of expression.

Solution: A parenthesis pair is redundant if it contain 0 or 1 element between them. The algorithm works by adding all the element except ')' to the stack. When we get a ')' at that point we find its corresponding pair and count all the elements between this pair. If the number of elements is 0 or 1 then we have a redundant parenthesis.

Example 7.22:

```

public static boolean findDuplicateParenthesis(String expn, int size) {
    Stack<Character> stk = new Stack<Character>();
    char ch;
    int count;

    for (int i = 0; i < size; i++) {
        ch = expn.charAt(i);
        if (ch == ')') {
            count = 0;
            while (stk.size() != 0 && stk.peek() != '(') {
                stk.pop();
                count += 1;
            }
            if (count <= 1)
                return true;
        } else
            stk.push(ch);
    }
}

```

```

        return false;
    }

public static void main(String[] args) {
    String expn = "(((a+b))+c)";
    System.out.println("Given expn : " + expn);
    int size = expn.length();
    boolean value = findDuplicateParenthesis(expn, size);
    System.out.println("Duplicate Found : " + value);
}

```

Print Parenthesis Number

Problem: Given an expression, Number each parenthesis pair such that for each pair the opening and closing parenthesis have same number.

Example:

Input: '(((a+(b))+(c+d)))'

Output: [1, 2, 3, 4, 4, 3, 5, 5, 2, 1]

Example 7.23:

```

public static void printParenthesisNumber(String expn, int size) {
    char ch;
    Stack<Integer> stk = new Stack<Integer>();
    String output = new String();
    int count = 1;
    for (int i = 0; i < size; i++) {
        ch = expn.charAt(i);
        if (ch == '(') {
            stk.push(count);
            output += count;
            count += 1;
        } else if (ch == ')')
            output += stk.pop();
    }
    System.out.println("Parenthesis Count ");
    System.out.println(output);
}

```

```
public static void main(String[] args) {  
    String expn1 = "(((a+(b))+(c+d)))";  
    int size = expn1.length();  
    System.out.println("Given expn " + expn1);  
    printParenthesisNumber(expn1, size);  
}
```

Infix, Prefix and Postfix Expressions

When we have an algebraic expression like A + B then we know that the variable A is being added to variable B. This type of expression is called infix expression because the operator “+” is there between operands A and operand B.

Now consider another infix expression A + B * C. In the expression there is a problem that in which order + and * work. Are A and B are added first and then the result is multiplied. Alternatively, B and C are multiplied first and then the result is added to A. This makes the expression ambiguous. To deal with this ambiguity we define the precedence rule or use parentheses to remove ambiguity.

So if we want to multiply B and C first and then add the result to A. Then the same expression can be written unambiguously using parentheses as A + (B * C). On the other hand, if we want to add A and B first and then the sum will be multiplied by C we will write it as (A + B) * C. Therefore, in the infix expression to make the expression unambiguous, we need parenthesis.

Infix expression: In this notation, we place operator in the middle of the operands.

< Operand > < Operator > < Operand >

Prefix expressions: In this notation, we place operator at the beginning of the operands.

< Operator > < Operand > < Operand >

Postfix expression: In this notation, we place operator at the end of the operands.

< Operand > < Operand > < Operator >

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + (B * C)	+ A * B C	A B C * +
(A + B) * C	* + ABC	A B + C *

Now comes the most obvious question why we need so unnatural Prefix or Postfix expressions when we already have infix expressions which words just fine for us. The answer to this is that infix expressions are ambiguous and they need parenthesis to make them unambiguous. While postfix and prefix notations do not need any parenthesis.

Infix-to-Postfix Conversion

Problem: Function to convert infix expression to postfix expression.

Example 7.24:

```
public static int precedence(char x) {  
    if (x == '(') {  
        return (0);  
    }  
    if (x == '+' || x == '-') {  
        return (1);  
    }  
    if (x == '*' || x == '/' || x == '%')  
        return (2);  
    if (x == '^') {  
        return (3);  
    }  
    return (4);  
}
```

```
public static String infixToPostfix(String expn) {  
    String output = "";
```

```

char[] out = infixToPostfix(expn.toCharArray());

for (char ch : out) {
    output = output + ch;
}
return output;
}

public static char[] infixToPostfix(char[] expn) {
    Stack<Character> stk = new Stack<Character>();

    String output = "";
    char out;

    for (char ch : expn) {
        if (ch <= '9' && ch >= '0') {
            output = output + ch;
        } else {
            switch (ch) {
                case '+':
                case '-':
                case '*':
                case '/':
                case '%':
                case '^':
                    while (stk.isEmpty() == false && precedence(ch) <=
precedence(stk.peek())) {
                        out = stk.pop();
                        output = output + " " + out;
                    }
                    stk.push(ch);
                    output = output + " ";
                    break;
                case '(':
                    stk.push(ch);
                    break;
                case ')':

```

```

        while (stk.isEmpty() == false && (out = stk.pop()) != '(') {
            output = output + " " + out + " ";
        }
        break;
    }
}
}

while (stk.isEmpty() == false) {
    out = stk.pop();
    output = output + out + " ";
}
return output.toCharArray();
}

public static void main(String[] args) {
    String expn = "10+((3))*5/(16-4)";
    String value = infixToPostfix(expn); System.out.println("Infix Expn: " +
expn);
    System.out.println("Postfix Expn: " + value);
}

```

Analysis:

- Print operands in the same order as they arrive.
- If the stack is empty or contains a left parenthesis “(” on top, we should push the incoming operator in the stack.
- If the incoming symbol is a left parenthesis “(”, push left parenthesis in the stack.
- If the incoming symbol is a right parenthesis “)”, pop from the stack and print the operators until you see a left parenthesis “(”). Discard the pair of parentheses.
- If the precedence of incoming symbol is higher than the precedence of operator at the top of the stack, then push it to the stack.
- If the incoming symbol has, an equal precedence compared to the top of the stack, use association. If the association is left to right, then pop and print the symbol at the top of the stack and then push the incoming operator. If the association is right to left, then push the incoming operator.

- If the precedence of incoming symbol is lower than the precedence of operator on the top of the stack, then pop and print the top operator. Then compare the incoming operator against the new operator at the top of the stack.
- At the end of the expression, pop and print all operators on the stack.

Infix-to-Prefix Conversion

Problem: Function to convert infix expression to postfix expression.

Example 7.25:

```
public static String infixToPrefix(String expn) {
    char[] arr = expn.toCharArray();
    reverseString(arr);
    replaceParanthesis(arr);
    arr = infixToPostfix(arr);
    reverseString(arr);
    expn = new String(arr);
    return expn;
}
```

```
public static void replaceParanthesis(char[] a) {
    int lower = 0;
    int upper = a.length - 1;
    while (lower <= upper) {
        if (a[lower] == '(') {
            a[lower] = ')';
        } else if (a[lower] == ')') {
            a[lower] = '(';
        }
        lower++;
    }
}
```

```
public static void reverseString(char[] expn) {
    int lower = 0;
    int upper = expn.length - 1;
    char tempChar;
```

```

        while (lower < upper) {
            tempChar = expn[lower];
            expn[lower] = expn[upper];
            expn[upper] = tempChar;
            lower++;
            upper--;
        }
    }

public static void main(String[] args) {
    String expn = "10+((3))*5/(16-4)";
    String value = infixToPrefix(expn);
    System.out.println("Infix Expn: " + expn);
    System.out.println("Prefix Expn: " + value);
}

```

Analysis:

1. Reverse the given infix expression.
2. Replace '(' with ')' and ')' with '(' in the reversed expression.
3. Now, apply infix to postfix subroutine already discussed.
4. Reverse the generated postfix expression and this will give required prefix expression.

Postfix Evaluate

Problem: Write a function to evaluate a postfix expression. Such as: expression “1 2 + 3 4 + *” gives output 21

Example 7.26:

```

public static int postfixEvaluate(String expn) {
    Stack<Integer> stk = new Stack<Integer>();
    Scanner tokens = new Scanner(expn);

    while (tokens.hasNext()) {
        if (tokens.hasNextInt()) {
            stk.push(tokens.nextInt()); } else {
            int num1 = stk.pop();

```

```

int num2 = stk.pop();
char op = tokens.next().charAt(0);
switch (op) {
    case '+':
        stk.push(num1 + num2);
        break;
    case '-':
        stk.push(num1 - num2);
        break;
    case '*':
        stk.push(num1 * num2);
        break;
    case '/':
        stk.push(num1 / num2);
        break;
}
tokens.close();
return stk.pop();
}

```

```

public static void main(String[] args) {
    String expn = "6 5 2 3 + 8 * + 3 + *";
    int value = postfixEvaluate(expn);
    System.out.println("Given Postfix Expn: " + expn);
    System.out.println("Result after Evaluation: " + value);
}

```

Analysis:

- 1) Create a stack to store values or operands.
- 2) Scan through the given expression and do following for each element:
 - a) If the element is a number, then push it into the stack.
 - b) If the element is an operator, then pop values from the stack. Evaluate the operator over the values and push the result into the stack.
- 4) When the expression is scanned completely, the number in the stack is the result.

Stack Based Rejection Method

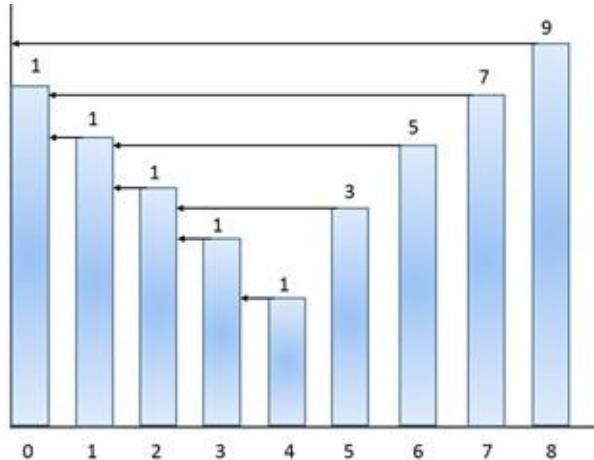
Stack Based Rejection technique is used when processing data follow specific property of rejection. That value at some index can be used to reject some other values that are processed before it. The values that are rejected are those values, which are unimportant for the rest of the processing. Below examples of “Stock Span Problem”, “Get Max Rectangular Area in a Histogram” “Stock Analyst Problem” etc are few of its example.

Stock Span Problem

Problem: In given list of daily stock price in an array A[i]. Find the span of the stocks for each day. A span of stock is the maximum number of days for which the price of stock was lower than that day.

Or

Given a histogram, find the number of consecutive bars in the left side of the each current bar that have values less than the current bar.



First solution: Brute force, Two loops are used. Outer loop to select the bar / elements in histogram. Inner loop is used to find count of consecutive bars smaller to current bar.

Example 7.27:

```
public static int[] StockSpanRange(int[] arr) {  
    int[] SR = new int[arr.length];  
    SR[0] = 1;
```

```

for (int i = 1; i < arr.length; i++) {
    SR[i] = 1;
    for (int j = i - 1; (j >= 0) && (arr[i] >= arr[j]); j--) {
        SR[i]++;
    }
}
return SR;
}

public static void main(String[] args) {
    int[] arr = { 6, 5, 4, 3, 2, 4, 5, 7, 9 };
    int size = arr.length;
    int[] value = StockSpanRange(arr);
    System.out.print("StockSpanRange : ");
    for (int val : value)
        System.out.print(" " + val);
}

```

Time complexity: $O(n^2)$, length of each bar is compared with the bars before it.

Second solution: The above algorithms for each bar we are comparing length of bars before it. If somehow we can get the index of the bar that is before current bar and is having length greater than the current bar. Let us suppose the index of this hypothetical bar is M and index of current bar is N then the stock span range will be $(N - M)$. By using a stack such index can be obtained by doing some bookkeeping.

Let us call stock input array as “arr”. Create a stack to store index value and add value 0 to it. Then traverse the values in stock input array. Let call current index we are analyzing as “curr”, the index at the top of stack is “top”. For each value of the stock input arr[curr] is compared with arr[top]. If value arr[curr] is less than or equal to arr[top] then add curr to the stack. If value of arr[curr] is greater than arr[top] then start popping values from stack till $arr[curr] > arr[top]$. Then add curr to the stack. Store stock range to $SR[curr] = curr - top$. Repeat this process for all the index of input array and SR array will be populated.

Example 7.28:

```

public static int[] StockSpanRange2(int[] arr) {
    Stack<Integer> stk = new Stack<Integer>();

    int[] SR = new int[arr.length];
    stk.push(0);
    SR[0] = 1;
    for (int i = 1; i < arr.length; i++) {
        while (!stk.isEmpty() && arr[stk.peek()] <= arr[i]) {
            stk.pop();
        }
        SR[i] = (stk.isEmpty()) ? (i + 1) : (i - stk.peek());
        stk.push(i);
    }
    return SR;
}

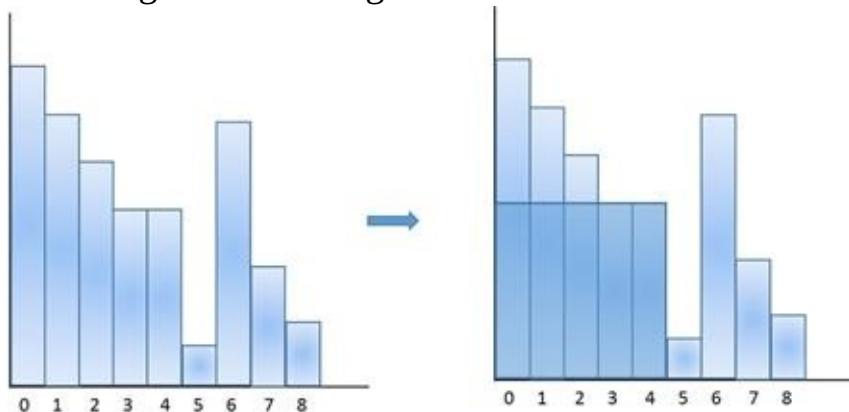
```

For each value in input array, index is added to stack only once and that index can be taken out of stack only once. Each comparison operation leads to either adding a value to stack or taking a value out of stack. Therefore, if there are n elements in array then at most there will be $2n$ comparison so this algorithm is linear.

Time complexity is $O(n)$, space complexity will also be $O(n)$ for stack.

Get Max Rectangular Area in a Histogram

Problem: In given histogram of rectangle bars of each one unit wide. Find the maximum area rectangle in the histogram.



First solution: Brute force approach. We use two loops to find the desired greatest area. First loop to find the right boundary of a rectangle and then another inner loop to find the left boundary of the rectangle. This approach have time complexity of $O(n^2)$

Example 7.29:

```
public static int GetMaxArea(int[] arr) {  
    int size = arr.length;  
    int maxArea = -1;  
    int currArea;  
    int minHeight = 0;  
    for (int i = 1; i < size; i++) {  
        minHeight = arr[i];  
        for (int j = i - 1; j >= 0; j--) {  
            if (minHeight > arr[j]) {  
                minHeight = arr[j];  
            }  
            currArea = minHeight * (i - j + 1);  
            if (maxArea < currArea) {  
                maxArea = currArea;  
            }  
        }  
    }  
    return maxArea;  
}  
  
public static void main(String[] args) {  
    int[] arr = { 7, 6, 5, 4, 4, 1, 6, 3, 1 };  
    int size = arr.length;  
    int value = GetMaxArea(arr);  
    System.out.println("GetMaxArea :: " + value);  
}
```

Second solution: Using Stack

- Create a stack that contain index in increasing order.
- When a current index value is smaller than the values at the top of the stack,

current index is added to the stack.

- When a current index value is smaller than the values at the top of the stack, stack is popped till the top value is less than the current index value.
- The value that is popped out from stack will contribute to create rectangle. The height of rectangle is arr[top] and width will be “i” if stack is empty and is i-stk[-1]-1

Time complexity is O(n)

Example 7.30:

```
public static int GetMaxArea2(int[] arr) {  
    int size = arr.length;  
    Stack<Integer> stk = new Stack<Integer>();  
    int maxArea = 0;  
    int top;  
    int topArea;  
    int i = 0;  
    while (i < size) {  
        while ((i < size) && (stk.isEmpty() || arr[stk.peek()] <= arr[i])) {  
            stk.push(i);  
            i++;  
        }  
        while (!stk.isEmpty() && (i == size || arr[stk.peek()] > arr[i])) {  
            top = stk.peek();  
            stk.pop();  
            topArea = arr[top] * (stk.isEmpty() ? i : i - stk.peek() - 1);  
            if (maxArea < topArea) {  
                maxArea = topArea;  
            }  
        }  
    }  
    return maxArea;  
}
```

Stock Analyst Problem

Problem: A stock analyst had approached you to write a program to find stock

hike points in some stocks. A stock hike point is the values of stock that is greater than all its previous values. You are provided stock price in an infinite stream with most recent value first (in term of time the data is provided in reverse order.).

For example in stock values are [20, 19, 10, 21, 40, 35, 39, 50, 45, 42], stock hike points values are 20, 21, 40 and 50 but the data is provided in reverse order.

Analysis:

- Data is provided in order of old value first then new values. Then this can be done in linear time just by using a single variable to store maximum value so far. However, in our problem data is provided in reverse order.
- When we are processing some value at that time we do not know if it will become a hike point.
- We will use a stack to store the data. We read input from stream; if value at top of stack is smaller than input value then we pop value from stack until the value at the top of stack is smaller than the input value. Then we add the input value to stack.
- If the value at the top of stack is greater than the input value then we add the input value to the stack. At the end, the content of stack is stock hike points.
- Time complexity of this solution is $O(n)$

Next Larger Element

Problem: Function to print next larger element of each element of array.

First Solution: Brute force approach, for each index we can traverse the array indexes right to it to find a larger value. Time complexity is $O(n^2)$

Example 7.31:

```
public static void nextLargerElement(int[] arr, int size) {  
    int[] output = new int[size];  
    int outIndex = 0;  
    int next;  
  
    for (int i = 0; i < size; i++) {  
        next = -1;
```

```

for (int j = i + 1; j < size; j++) {
    if (arr[i] < arr[j]) {
        next = arr[j];
        break;
    }
}
output[outIndex++] = next;
}
for (int val : output)
    System.out.print(val + " ");
}

```

Second Solution: let us suppose we are processing elements of array. The elements which are less than ith element is does not have largest element on the right of ith index. So we can say the elements which are left of ith element and which are less the ith element are independent of the elements at the index higher than ith. So stack based reduction method applies.

- Create an empty stack. This stack will contain index of elements, which have value in decreasing order.
- We will traverse through input array.
- If the top of stack is smaller than the current element, then pop from the stack and mark its next larger element as current index. We will pop from stack and repeat this process until the value of index at the top of the stack is grater then the current value.
- When we processed the input array then for those indexes for which we have a next largest element is populated.
- The indexes which are present in the stack do not have any next largest element to we should mark those indexes in the output array as -1.

Example 7.32:

```

public static void nextLargerElement2(int[] arr, int size) {
    Stack<Integer> stk = new Stack<Integer>();
    // output = [-1] * size;
    int[] output = new int[size];
    int index = 0;
    int curr;

```

```

for (int i = 0; i < size; i++) {
    curr = arr[i];
    // stack always have values in decreasing order.
    while (stk.isEmpty() == false && arr[stk.peek()] <= curr) {
        index = stk.pop();
        output[index] = curr;
    }
    stk.push(i);
}
// index which dont have any next Larger.
while (stk.isEmpty() == false) {
    index = stk.pop();
    output[index] = -1;
}
for (int val : output)
    System.out.print(val + " ");
}

```

Next Smaller Element

Problem: Function to print next smallest element of each element of array.

First Solution: Brute force approach, for each index we can traverse the array indexes right to it to find a smaller value. Time complexity is $O(n^2)$

Second Solution: let us suppose we are processing elements of array. The elements that are greater than ith element does not have smaller element on the right of ith index. Therefore, we can say the elements which are left of ith element and which are grater the ith element are independent of the elements at the index higher than ith. So stack based reduction method applies.

Example 7.33:

```

public static void nextSmallerElement(int[] arr, int size) {
    Stack<Integer> stk = new Stack<Integer>();
    int[] output = new int[size];
    int curr, index;
}

```

```

for (int i = 0; i < size; i++) {
    curr = arr[i];
    // stack always have values in increasing order.
    while (stk.isEmpty() == false && arr[stk.peek()] > curr) {
        index = stk.pop();
        output[index] = curr;
    }
    stk.push(i);
}
// index which dont have any next Smaller.
while (stk.isEmpty() == false) {
    index = stk.pop();
    output[index] = -1;
}
for (int val : output)
    System.out.print(val + " ");
}

```

Next Larger Element Circular

Problem: Function to print next larger element of each element of circular array.

Input: [6, 3, 9, 8, 10, 2, 1, 15, 7]

Output: [9, 9, 10, 10, 15, 15, 15, -1, 9]

First Solution: Brute force approach, let us suppose the number of elements in array is n. For each index, we will traverse (n-1) number of nodes (in circular manner) to find a larger value. Time complexity is $O(n^2)$

Second Solution: The solution of this problem is same as Next Largest Element problem. The only difference is that $(2n - 1)$ nodes are traversed. As for each node, the farthest node that have its effect is at max $(n-1)$ distance apart. Therefore, total $(n + n - 1)$ nodes are traversed. Time Complexity of stack based reduction process is $O(n)$

Example 7.34:

```

public static void nextLargerElementCircular(int[] arr, int size) {
    Stack<Integer> stk = new Stack<Integer>();

```

```

int curr, index;
int[] output = new int[size];
for (int i = 0; i < (2 * size - 1); i++) {
    curr = arr[i % size];
    // stack always have values in decreasing order.
    while (stk.isEmpty() == false && arr[stk.peek()] <= curr) {
        index = stk.pop();
        output[index] = curr;
    }
    stk.push(i % size);
}
// index which dont have any next Larger.
while (stk.isEmpty() == false) {
    index = stk.pop();
    output[index] = -1;
}
for (int val : output)
    System.out.print(val + " ");
}

public static void main(String[] args) {
    int arr[] = { 6, 3, 9, 8, 10, 2, 1, 15, 7 };
    int size = arr.length;
    nextLargerElementCircular(arr, size);
}

```

Depth-First Search with a Stack

In a depth-first search, we traverse down a path until we get a dead end; then we backtrack by popping a stack to get an alternative path.

- Create a stack
- Create a start point
- Push the start point onto the stack
- While (value searching is not found and the stack is not empty)
 - o Pop the stack
 - o Find all possible points after the one which we just tried
 - o Push these points onto the stack

Grid Based Problems (Using Recursion.)

In grid based problems when we are given some condition under which some traversal in 2D array is performed. For example in a problem like In a chessboard, minimum number of steps required to move a knight from source position to destination position. Given fruits arranged in grid that is represented by 2D array some fruits are rotten and we need to find how many days are needed for all the fruits rot.

In these problems, we need BFS or DFS traversal. We create a 2D array is used to keep track if some particular cell is processed or not. At each step we find all the possible states that can be reached from any given state. We move form one state to another by using recursion.

Rotten Fruit

Problem: Given that, fruits are arranged in a 2 dimensional grid. Among which few fruits are rotten. In one day, a rotten fruit rot adjacent fruits that comes to its contact. You need to find the maximum number of days in which the fruits of whole grid will rot.

First solution: Using DFS / Recursion, each fruit can rot at max 4 other fruits that are adjacent to it.

Example 7.35:

```
public static void RottenFruitUtil(int[][] arr, int maxCol, int maxRow, int currCol, int currRow, int[][] traversed, int day) { // Range check
    if (currCol < 0 || currCol >= maxCol || currRow < 0 || currRow >= maxRow)
        return;
    // Traversable and rot if not already rotten.
    if (traversed[currCol][currRow] <= day || arr[currCol][currRow] == 0)
        return;
    // Update rot time.
    traversed[currCol][currRow] = day;
    // each line corresponding to 4 direction.
```

```

RottenFruitUtil(arr, maxCol, maxRow, currCol - 1, currRow, traversed, day
+ 1);
RottenFruitUtil(arr, maxCol, maxRow, currCol + 1, currRow, traversed, day
+ 1);
RottenFruitUtil(arr, maxCol, maxRow, currCol, currRow + 1, traversed, day
+ 1);
RottenFruitUtil(arr, maxCol, maxRow, currCol, currRow - 1, traversed, day
+ 1);
}

public static int RottenFruit(int[][] arr, int maxCol, int maxRow) {
    int[][] traversed = new int[maxCol][maxRow];
    for (int i = 0; i < maxCol; i++) {
        for (int j = 0; j < maxRow; j++) {
            traversed[i][j] = Integer.MAX_VALUE;
        }
    }

    for (int i = 0; i < maxCol - 1; i++) {
        for (int j = 0; j < maxRow - 1; j++) {
            if (arr[i][j] == 2)
                RottenFruitUtil(arr, maxCol, maxRow, i, j, traversed, 0);
        }
    }

    int maxDay = 0;
    for (int i = 0; i < maxCol - 1; i++) {
        for (int j = 0; j < maxRow - 1; j++) {
            if (arr[i][j] == 1) {
                if (traversed[i][j] == Integer.MAX_VALUE)
                    return -1;
                if (maxDay < traversed[i][j])
                    maxDay = traversed[i][j];
            }
        }
    }
    return maxDay;
}

```

```

}

public static void main(String[] args) {
    int arr[][] = { { 1, 0, 1, 1, 0 }, { 2, 1, 0, 1, 0 }, { 0, 0, 0, 2, 1 }, { 0, 2, 0, 0, 1 },
    { 1, 1, 0, 0, 1 } };
    System.out.println(RottenFruit(arr, 5, 5));
}

```

Analysis: We create a traversal array to keep trace of

Steps of Knight

Problem: Given a chessboard and a knight start position. You need to find minimum number of steps required to move a knight from start position to final position.

Solution: Each knight can go to 8 other positions. Find if the positions are not already visited and do the DFS traversal of the chess board.

Example 7.36:

```

public static void StepsOfKnightUtil(int size, int currCol, int currRow, int[][] traversed, int dist) {
    // Range check
    if (currCol < 0 || currCol >= size || currRow < 0 || currRow >= size)
        return;

    // Traversable and rot if not already rotten.
    if (traversed[currCol][currRow] <= dist)
        return;

    // Update rot time.
    traversed[currCol][currRow] = dist;
    // each line corresponding to 4 direction.
    StepsOfKnightUtil(size, currCol - 2, currRow - 1, traversed, dist + 1);
    StepsOfKnightUtil(size, currCol - 2, currRow + 1, traversed, dist + 1);
    StepsOfKnightUtil(size, currCol + 2, currRow - 1, traversed, dist + 1);
    StepsOfKnightUtil(size, currCol + 2, currRow + 1, traversed, dist + 1);
}

```

```

StepsOfKnightUtil(size, currCol - 1, currRow - 2, traversed, dist + 1);
StepsOfKnightUtil(size, currCol + 1, currRow - 2, traversed, dist + 1);
StepsOfKnightUtil(size, currCol - 1, currRow + 2, traversed, dist + 1);
StepsOfKnightUtil(size, currCol + 1, currRow + 2, traversed, dist + 1);
}

public static int StepsOfKnight(int size, int srcX, int srcY, int dstX, int dstY) {
    int[][] traversed = new int[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            traversed[i][j] = Integer.MAX_VALUE;
        }
    }
}

StepsOfKnightUtil(size, srcX - 1, srcY - 1, traversed, 0);
int retval = traversed[dstX - 1][dstY - 1];
return retval;
}

public static void main(String[] args) {
    System.out.println(StepsOfKnight(20, 10, 10, 20, 20));
}

```

Distance nearest Fill

Problem: Given a matrix 2D array, with some cells filled by 1 and other 0. You need to create a 2D array that will contain the minimum distance of each 0 element to any one of the cell with value 1.

Solution: Grid based traversal using DFS is performed to find the nearest fill. We create a 2 dimensional traversed array to keep track of distance.

Example 7.37:

```

public static void DistNearestFillUtil(int[][] arr, int maxCol, int maxRow, int
currCol, int currRow,
    int[][] traversed, int dist) { // Range check
    if (currCol < 0 || currCol >= maxCol || currRow < 0 || currRow >= maxRow)

```

```

return;
// Traversable if their is a better distance.
if (traversed[currCol][currRow] <= dist)
return;
// Update distance.
traversed[currCol][currRow] = dist;
// each line corresponding to 4 direction.
DistNearestFillUtil(arr, maxCol, maxRow, currCol - 1, currRow, traversed,
dist + 1);
DistNearestFillUtil(arr, maxCol, maxRow, currCol + 1, currRow, traversed,
dist + 1);
DistNearestFillUtil(arr, maxCol, maxRow, currCol, currRow + 1, traversed,
dist + 1);
DistNearestFillUtil(arr, maxCol, maxRow, currCol, currRow - 1, traversed,
dist + 1);
}

```

```

public static void DistNearestFill(int[][] arr, int maxCol, int maxRow) {
int[][] traversed = new int[maxCol][maxRow];
for (int i = 0; i < maxCol; i++) {
for (int j = 0; j < maxRow; j++) {
traversed[i][j] = Integer.MAX_VALUE;
}
}
for (int i = 0; i < maxCol; i++) {
for (int j = 0; j < maxRow; j++) {
if (arr[i][j] == 1)
DistNearestFillUtil(arr, maxCol, maxRow, i, j, traversed, 0);
}
}
for (int i = 0; i < maxCol; i++) {
for (int j = 0; j < maxRow; j++) {
System.out.println("'" + traversed[i][j]);
}
System.out.println("\n");
}
}

```

```

}

public static void main(String[] args) {
    int arr[][] = { { 1, 0, 1, 1, 0 }, { 1, 1, 0, 1, 0 }, { 0, 0, 0, 0, 1 }, { 0, 0, 0, 0, 1 },
    { 0, 0, 0, 0, 1 } };
    DistNearestFill(arr, 5, 5);
}

```

Largest island

Problem: Given a map represented by 2D array. 1s are land and 0s are water. You need to find largest land mass. Largest landmass has largest number of 1s.

Or

Given a map represented by 2D array in which 1s are land and 0s are water. You need to find the largest path that contain largest number of 1s.

Solution: Largest island is found by doing DFS traversal of neighbouring nodes of a current node. Largest island of current node is sum of largest island of all the neighbouring nodes + 1 (for current node).

Example 7.38:

```

public static int findLargestIslandUtil(int[][] arr, int maxCol, int maxRow, int
currCol, int currRow, int value,
int[][] traversed) {
    if (currCol < 0 || currCol >= maxCol || currRow < 0 || currRow >= maxRow)
        return 0;
    if (traversed[currCol][currRow] == 1 || arr[currCol][currRow] != value)
        return 0;
    traversed[currCol][currRow] = 1;
    // each call corresponding to 8 direction.
    return 1 + findLargestIslandUtil(arr, maxCol, maxRow, currCol - 1, currRow
- 1, value, traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol-1, currRow, value,
traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol-1, currRow+1, value,
traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol, currRow-1, value,
traversed)

```

```

traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol, currRow+1, value,
traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol+1, currRow-1, value,
traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol+1, currRow, value,
traversed)
+findLargestIslandUtil(arr, maxCol, maxRow, currCol+1, currRow+1,
value,traversed);
}

public static int findLargestIsland(int[][] arr, int maxCol, int maxRow) {
    int maxVal = 0;
    int currVal = 0;
    int[][] traversed = new int[maxCol][maxRow];
    for (int i = 0; i < maxCol; i++) {
        for (int j = 0; j < maxRow; j++) {
            traversed[i][j] = Integer.MAX_VALUE;
        }
    }
    for (int i = 0; i < maxCol; i++) {
        for (int j = 0; j < maxRow; j++) {
            {
                currVal = findLargestIslandUtil(arr, maxCol, maxRow, i, j, arr[i][j],
traversed);
                if (currVal > maxVal)
                    maxVal = currVal;
            }
        }
    }
    return maxVal;
}

public static void main(String[] args) {
    int arr[][] = { { 1, 0, 1, 1, 0 }, { 1, 0, 0, 1, 0 }, { 0, 1, 1, 1, 1 }, { 0, 1, 0, 0, 0 },
{ 1, 1, 0, 0, 1 } };
    System.out.println("Largest Island : " + findLargestIsland(arr, 5, 5));
}

```

}

Number of islands

Problem: Given a map represented by 2D array. 1s are land and 0s are water. You need to find number of islands. An island is a land mass formed by group of connected 1s.

Hint: Solve this problem in same way you had solved the above largest island problem.

Snake and Ladder Problem

Problem: Given a snake and ladder board, which starts with 1 and end at 100. You need to find the minimum number of dice throws required to reach 100th cell from 1st cell. There are list of snakes coordinates provided and list of ladder coordinate provided as pair.

Hint: create a dice count array of length 100. Process each cell start from 1st to 100th. Each cell have 6 outcome. Which changes with the presence of snakes and leader. Use BFS or DFS and reach destination.

Palindrome string

Problem: Find if given string is a palindrome or not using a stack.

Definition of palindrome: A palindrome is a sequence of characters that is same backward or forward.

Eg. “AAABBCCCBBAAA”, “ABA” & “ABBA”

Hint: Push characters to the stack until the half-length of the string. Then pop these characters and then compare. Make sure you take care of the odd length and even length.

Find Celebrity Problem

Problem: In a party, there is possibility that a celebrity had visited it. A celebrity is a person who does not know anyone in the party and everyone in the party

knows celebrity. You want to find celebrity in the party. You are allowed to ask only one question $\text{DoYouKnow}(X, Y)$, which means to X you can ask only one question that do you know Y . X will answer the question as yes or no.

First solution: brute force approach, you can traverse the guest one by one and ask them if they know the other guest one by one. If you find a guest who do not know anyone then you have a probable candidate. If the guest is known to all other guest than he is a celebrity.

This is an inefficient solution with complexity of $O(N^2)$

Second solution: Use stack,

- We add the entire guest list index from 1 to N in a stack. We take two index values out of stack and store them in two variable first and second.
- If the guest at index first knows guest at index second, then first is not celebrity so copy second to first. Else if the guest at index first does not know guest second, then second is not celebrity.
- In both the cases pop another element from stack and mark it second.
- At each comparison, one value is rejected.
- At last first will contain the probable celebrity.
- Then we need to check that the celebrity candidate is known by all the other guests.

Example 7.39:

```
public static boolean isKnown(int relation[][], int a, int b) {  
    if (relation[a][b] == 1)  
        return true;  
    return false;  
}
```

```
public static int findCelebrity(int relation[][], int count) {  
    Stack<Integer> stk = new Stack<Integer>();  
    int first = 0, second = 0;  
    for (int i = 0; i < count; i++) {  
        stk.push(i);  
    }  
    first = stk.pop();  
    while (stk.size() != 0) {
```

```

second = stk.pop();
if (isKnown(relation, first, second))
    first = second;
}
for (int i = 0; i < count; i++) {
    if (first != i && isKnown(relation, first, i))
        return -1;
    if (first != i && isKnown(relation, i, first) == false)
        return -1;
}
return first;
}

public static void main(String[] args) {
    int[][] arr = { { 1, 0, 1, 1, 0 }, { 1, 0, 0, 1, 0 }, { 0, 0, 1, 1, 1 }, { 0, 0, 0, 0, 0 },
{ 1, 1, 0, 1, 1 } };

    System.out.println("Celebrity : " + findCelebrity(arr, 5));
}

```

Third solution: let us suppose we have guest list as [g1, g2, g3, g4 ...]. We take two index counter namely first and second. First is assigned 0 value and second is assigned 1 value.

We will ask guest at index first if he know guest at index second. If answer is yes then we make first = second and second = second + 1 (guest at first index is not a celebrity.)

If the answer is no then we make second = second + 1 (guest at index second is not a celebrity.)

In the end, we will have probable celebrity at the first index.

Then check that the guest at the first index is known by all the other guests.

Example 7.40:

```

public static int findCelebrity2(int relation[][], int count) {
    int first = 0;
    int second = 1;

```

```
for (int i = 0; i < (count - 1); i++) {  
    if (isKnown(relation, first, second))  
        first = second;  
    second = second + 1;  
}  
for (int i = 0; i < count; i++) {  
    if (first != i && isKnown(relation, first, i))  
        return -1;  
    if (first != i && isKnown(relation, i, first) == false)  
        return -1;  
}  
return first;  
}
```

Problem: In the above question, the party can also contain stranger who is not known by anyone and who don't know anyone. Find celebrity function should return 0 in this case. Do the needful modifications in the solution of above problem.

Uses of Stack

- Recursion can also be done using stack. (In place of the system stack)
- The function call is implemented using stack.
- When we want to reverse a sequence, we just push everything in stack and pop from it.
- Grammar checking, balance parenthesis, infix to postfix conversion, postfix evaluation of expression etc.

Exercise

1. Converting Decimal Numbers to Binary Numbers using stack data structure.

Hint: store reminders into the stack and then print the stack.

2. Convert an infix expression to prefix expression.

Hint: Reverse given expression, Apply infix to postfix, and then reverse the expression again.

Step 1. Reverse the infix expression.

$5^E + D^*) C^B + A ($

Step 2. Make Every '(' as ')' and every ')' as '('

$5^E + D^*(C^B + A)$

Step 3. Convert an expression to postfix form.

Step 4. Reverse the expression.

$+^* + A^B C D^E 5$

3. Write an HTML opening tag and closing tag-matching program.

Hint: parenthesis matching.

4. Write a function that will transform [Postfix to Infix Conversion](#)

5. Write a function that will transform [Prefix to Infix Conversion](#)

6. Write a palindrome matching function, which ignores characters other than English alphabet and digits. String "Madam, I'm Adam." should return true.

7. In the Growing-Reducing Stack implementation using list. Try to figure out a better algorithm which will work similar to `Vector<>` or `ArrayDeque<>`.

CHAPTER 8: QUEUE

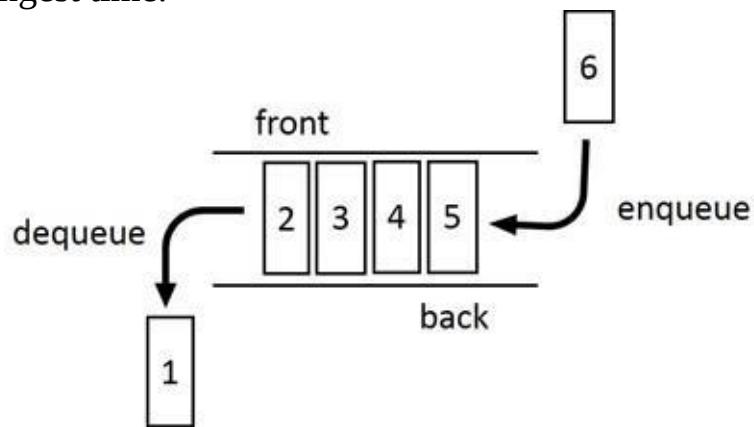
Introduction

A queue is a basic data structure that organizes items in first-in-first-out (FIFO) manner. First element, inserted into a queue, will be the first to be removed. It is also known as "first-come-first-served".

The real life analogy of queue is typical lines in which we all participate time to time.

- We wait in a line of railway reservation counter.
- We wait in the cafeteria line.
- We wait in a queue when we call to some customer-care.

The elements, which are at the front of the queue, are the one that stayed in the queue for the longest time.



Computer science also has many common examples of queues. We issue a print command from our office to a single printer per floor. The print task are lined up in a printer queue. The print command that is issued first will be printed before the next commands in line.

In addition to printing queues, operating system is also using different queues to control process scheduling. Processes are added to processing queue, which is used by an operating system for various scheduling algorithms.

Soon we will study about graphs and will come to know about breadth-first traversal of graph also uses queue.

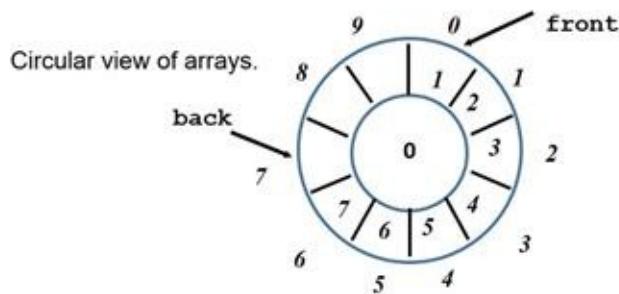
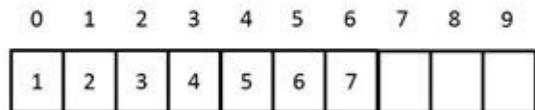
The Queue Abstract Data Type

Queue abstract data type is defined as a class whose instance follows FIFO or first-in-first-out for the elements, added to it.

Queue should support the following operations:

1. add(): Which adds a single element at the back of a queue
2. remove(): Which removes a single element from the front of a queue.
3. isEmpty(): Returns 1 if the queue is empty
4. size(): Returns the number of elements in a queue.

Queue Using Array



Example 8.1:

```
public class Queue {  
    private int size;  
    private int capacity = 100;  
    private int[] data;  
    int front = 0;  
    int back = 0;  
  
    public Queue() {  
        size = 0;  
        data = new int[100];  
    }  
  
    public boolean add(int value) {  
        if (size >= capacity) {  
            System.out.println("Queue is full.");  
            return false;  
        } else {  
            size++;  
            data[back] = value;  
            back = (++back) % (capacity - 1);  
        }  
        return true;  
    }  
}
```

```
}

public int remove() {
    int value;
    if (size <= 0) {
        System.out.println("Queue is empty.");
        return -999;
    } else {
        size--;
        value = data[front];
        front = (++front) % (capacity - 1);
    }
    return value;
}

boolean isEmpty() {
    return size == 0;
}

int size() {
    return size;
}
}
```

Analysis:

- Here queue is created from an list.
- Add() to insert one element at the back of the queue.
- Remove() to delete one element from the front of the queue.

Queue Using linked list

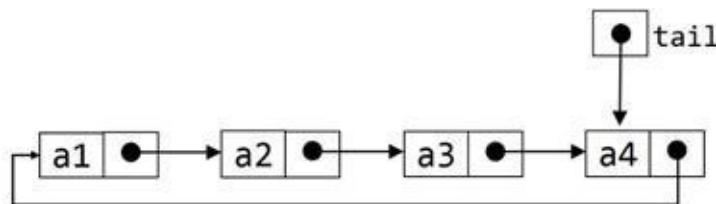
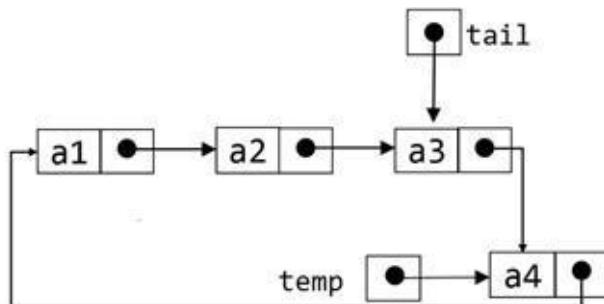
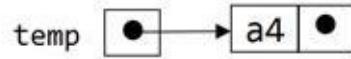
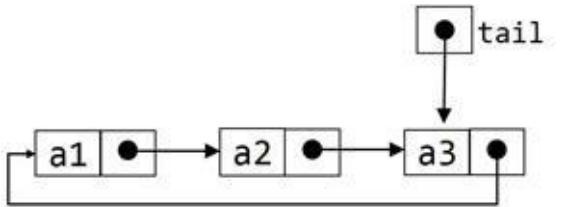
Queue is implemented using a circular linked list. The advantage of using circular linked list is that we can get head and tail node in constant time. When we want to insert we add at tail. When we need to delete we delete from the head of list.

Example 8.2:

```
public class QueueLL {  
    private Node tail = null;  
    private int size = 0;  
  
    private static class Node {  
        private int value;  
        private Node next;  
  
        public Node(int v, Node n) {  
            value = v;  
            next = n;  
        }  
    }  
  
    public int size() {  
        return size;  
    }  
  
    public boolean isEmpty() {  
        return size == 0;  
    }  
}
```

Add

Enqueue into a queue using circular linked list. Nodes are added to the end of the linked list. Below diagram indicates how a new node is added to the list. The tail is modified every time when a new value is added to the queue.



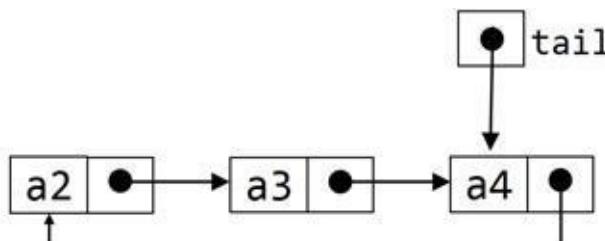
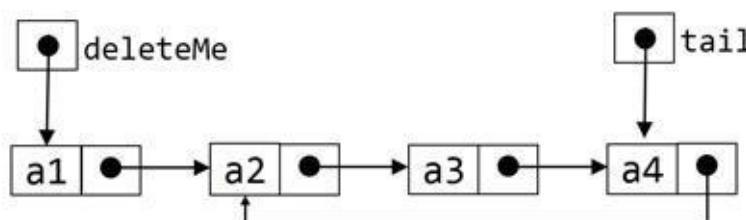
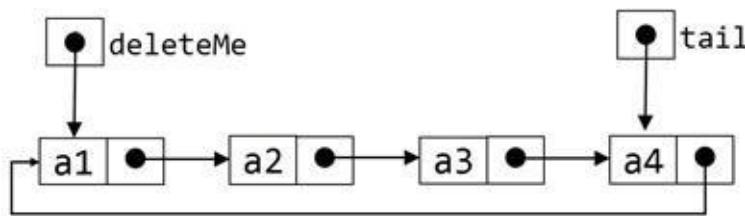
Example 8.3:

```
public void add(int value) {
    Node temp = new Node(value, null);
    if (tail == null){
        tail = temp;
        tail.next = tail;
    }
    else {
        temp.next = tail.next;
        tail.next = temp;
        tail = temp;
    }
    size++;
}
```

Analysis: add operation add one element at the end of the Queue (circular linked list).

Remove

Dequeue operation is done by deleting head node. Next node of the tail node is the head node we store head node location in deleteMe temporary pointer. Tail node next will point to the next of the deleteMe node. Finally, head node which is stored in deleteMe pointer is deleted.



Example 8.4:

```
public int remove() throws IllegalStateException {  
    if (isEmpty()) throw new IllegalStateException("StackEmptyException");  
    int value=0;  
    if (tail == tail.next){
```

```

        value = tail.value;
        tail = null;
    }
    else{
        value = tail.next.value;
        tail.next = tail.next.next;
    }
    size--;
    return value;
}

public int peek() throws IllegalStateException {
    if (isEmpty()) throw new IllegalStateException("StackEmptyException");
    int value;
    if (tail == tail.next)
        value = tail.value;
    else
        value = tail.next.value;

    return value;
}

public static void main(String[] args) {
    QueueLL q = new QueueLL(); q.add(1); q.add(2); q.add(3); for (int i = 0; i < 3; i++)
    System.out.println(q.remove());
}

```

Analysis: Remove operation removes first node from the start of the queue (circular linked list).

Problems in Queue

Queue using a stack

Problem: How to implement a queue using a stack. You can use multiple one stack.

Solution: We can use two stack to implement queue.

1. **Enqueue Operation:** new elements are added to the top of first stack.
2. **Dequeue Operation:** elements are popped from the second stack. When second stack is empty then all the elements of first stack are popped one by one and pushed into second stack.

Example 8.5:

```
import java.util.Stack;

public class QueueUsingStack {

    private Stack<Integer> stk1;
    private Stack<Integer> stk2;

    public QueueUsingStack() {
        stk1 = new Stack<Integer>();
        stk2 = new Stack<Integer>();
    }

    void add(int value) {
        stk1.push(value);
    }

    int remove() {
        int value;
        if (stk2.isEmpty() == false) {
            return stk2.pop();
        }

        while (stk1.isEmpty() == false) {
```

```

value = stk1.pop();
stk2.push(value); }
return stk2.pop();
}

public static void main(String[] args) {
QueueUsingStack que = new QueueUsingStack();
que.add(1);
que.add(11);
que.add(111); System.out.println(que.remove());
System.out.println(que.remove()); }
}

```

Analysis: All add() happens to stack 1. When remove() is called removal happens from stack 2. When the stack 2 is empty then stack 1 is popped and pushed into stack 2. This popping from stack 1 and pushing into stack 2 revert the order of retrieval there by making queue behavior out of two stacks.

Time complexity is $O(1)$ on an average, elements are added to first stack only once. They are taken to second stack only once and taken out using pop operation.

Stack using a Queue

Problem: Implement stack using a queue.

First Solution: use two queues

Push: add new elements to queue1.

Pop: while size of queue1 is bigger than 1. Push all items from queue 1 to queue 2 except the last item. Switch the name of queue 1 and queue 2. Then return the last item.

Push operation is **O(1)** and Pop operation is **O(n)**

Second Solution: This same can be done using just one queue.

Push: add the element to queue.

Pop: find the size of queue. If size is zero then return error. Else, if size is positive then remove size- 1 elements from the queue and again add to the same

queue. At last, remove the next element and return it.
Push operation is **O(1)** and Pop operation is **O(n)**

Third Solution: In the above solutions, the push is efficient and pop is inefficient can we make pop efficient **O(1)** and push inefficient **O(n)**

Push: add new elements to queue2. Then add all the elements of queue 1 to queue 2. Then switch names of queue1 and queue 2.

Pop: remove from queue1

Reverse a stack

Problem: Reverse a stack using a queue

First Solution:

- Pop all the elements of stack and add them into a queue.
- Then remove all the elements of the queue into stack
- We have the elements of the stack reversed.

Second Solution:

- Since dynamic list or [] list is used to implement stack in C, we can iterate from both the directions of the array and swap the elements.

Reverse a queue

Problem: Reverse a queue-using stack

Solution:

- Dequeue all the elements of the queue into stack (append to the C list [])
- Then pop all the elements of stack and add them into a queue. (pop the elements from the array)
- We have the elements of the queue reversed.

Breadth-First Search with a Queue

In breadth-first search, we explore all the nearest nodes first by finding all possible successors and add them to a queue.

- Create a queue
- Create a start point

- Enqueue the start point onto the queue
- while (value searching not found and the queue is not empty)
 - o Dequeue from the queue
 - o Find all possible paths from the dequeued point.
 - o Enqueue these paths in the queue

Josephus problem

Problem: There are n people standing in a queue waiting to be executed. The counting begins at the front of the queue. In each step, k number of people are removed and again added one by one from the queue. Then the next person is executed. The execution proceeds around the circle until only the last person remains, who will be freedom. Find that position where you want to stand and gain your freedom.

Solution:

- Just insert integer for 1 to k in a queue. (corresponds to k people)
- Define a Kpop() function such that it will remove and add the queue k-1 times and then remove one more time. (This man is dead.)
- Repeat second step until size of queue is 1.
- Print the value in the last element. This is the solution.

Circular tour

Problem: There are N number of petrol pumps in a circular path. Each petrol pump have some limited amount of petrol. You are given the amount of petrol each petrol pump has and the distance from next petrol pump. Find if there is a circular tour possible to visit all the petrol pumps.

First Solution: For each find all the possible paths starting from different petrol pump.

For each petrol pump we will add (net petrol available – net petrol needed to reach its next petrol pump.). While iterating the pumps if we are able to find a path with petrol value always positive then it is the path which we are searching. Otherwise find another path by selecting other staring point.

Time complexity is $O(n^2)$

Solution: There is inefficiency in the above problem. Net petrol calculations are done repeatedly for same node. If we had started from i^{th} pump and got a negative value of net petrol after visiting k^{th} petrol pump. Then we can find net petrol value starting from $i+1^{\text{th}}$ pump and ends to k^{th} pump in constant time by subtracting petrol value of i^{th} pump.

So when we are traversing the array of pumps we go on adding index of pumps to the array. When we get a positive value of net petrol then we go on adding next petrol pump and keep track of net petrol. When we get a negative net petrol value then we remove starting pumps id from queue and do needed calculation on the net petrol. Once a value is added to queue we don't need to do more calculations for it. So this solution is linear in nature. Time complexity is $O(N)$.

Example 8.6:

```
public static int CircularTour(int[][] arr, int n) {  
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();  
    int nextPump = 0, prevPump;  
    int count = 0;  
    int petrol = 0;  
  
    while (que.size() != n) {  
        while (petrol >= 0 && que.size() != n) {  
            que.add(nextPump); petrol += (arr[nextPump][0] - arr[nextPump][1]);  
            nextPump = (nextPump + 1) % n;  
        }  
        while (petrol < 0 && que.size() > 0) {  
            prevPump = que.remove();  
            petrol -= (arr[prevPump][0] - arr[prevPump][1]);  
        }  
        count += 1;  
        if (count == n)  
            return -1;  
    }  
    if (petrol >= 0)  
        return que.remove();  
    else  
        return -1;
```

```

}

public static void main(String[] args) {
    // Testing code
    int tour[][] = { { 8, 6 }, { 1, 4 }, { 7, 6 } };
    System.out.println(" Circular Tour : " + CircularTour(tour, 3));
}

```

Convert XY

Problem: Given two values X and Y. You need to convert X to Y by performing various steps. In a step, we can either multiply 2 to the value or subtract 1 from it.

Solution: Breadth first traversal (BFS) of the possible values that can be generated from source value is done. The various values that can be generated are added to the queue for BFS. In addition, the various vales are added to map to keep track of the values already processed. Each value is added to the queue only once so the time complexity of this algorithm is O(n).

Example 8.7:

```

public static int convertXY(int src, int dst) {
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();
    int arr[] = new int[100];
    int steps = 0;
    int index = 0;
    int value;

    que.add(src); while (que.size() != 0) {
        value = que.remove();
        arr[index++] = value;

        if (value == dst) {
            for (int i=0;i<index;i++)
                System.out.print(arr[i]); System.out.print("Steps countr :: " + steps);

        return steps;
    }
}

```

```

    }
    steps++;
    if (value < dst)
        que.add(value * 2);
    else
        que.add(value - 1);
    }
    return -1;
}

public static void main(String[] args) {
    convertXY(2, 7);
}

```

Maximum value in Sliding Windows

Problem: Given an array of integer, find maximum value in all the sliding windows of length k.

Input: 11, 2, 75, 92, 59, 90, 55 and k = 3

Output: 75, 92, 92, 92, 90

First Solution: Brute force approach, run loop for all the index of array. Inside a loop run another loop of length k, find the maximum of this inner loop and display it to the screen. This time complexity is O(nk).

Second Solution:

- We traverse the input array and add the index values to a queue.
- When the index added to the queue is out of range then we remove them from the queue.
- We are searching for maximum value so those values which are in the queue and which are less than the value of current index will be of no use so we will pop them from the queue.
- So the maximum value of the window is always present at the index que[0] so they will be displayed.

Note: Problems that involve sliding windows, are solved efficiently using a doubly ended queue.

Example 8.8:

```
public static void maxSlidingWindows(int arr[], int size, int k) {  
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();  
    for (int i = 0; i < size; i++) {  
        // Remove out of range elements  
        if (que.size() > 0 && que.peek() <= i - k)  
            que.remove(); // Remove smaller values at left.  
        while (que.size() > 0 && arr[que.peekLast()] <= arr[i])  
            que.removeLast();  
        que.add(i);  
        // Largest value in window of size k is at index que[0]  
        // It is displayed to the screen.  
        if (i >= (k - 1))  
            System.out.println(arr[que.peek()]); }  
}  
  
public static void main(String[] args) {  
    int arr[] = { 11, 2, 75, 92, 59, 90, 55 };  
    int k = 3;  
    maxSlidingWindows(arr, 7, 3);  
}
```

Output

```
75, 92, 92, 92, 90
```

Minimum of Maximum Values in Sliding Windows

Problem: Given an array of integer, find minimum of all the maximum values in the sliding windows of length k.

Input: 11, 2, 75, 92, 59, 90, 55 and k = 3

Output: 75

Solution: Same as above problem. You will keep track of the minimum of all the maximum values in sliding windows too.

Example 8.9:

```

public static int minOfMaxSlidingWindows(int arr[], int size, int k) {
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();
    int minVal = 999999;
    for (int i = 0; i < size; i++) {
        // Remove out of range elements
        if (que.size() > 0 && que.peek() <= i - k)
            que.remove(); // Remove smaller values at left.
        while (que.size() > 0 && arr[que.peekLast()] <= arr[i])
            que.remove(); que.add(i);
        // window of size k
        if (i >= (k - 1) && minVal > arr[que.peek()])
            minVal = arr[que.peek()];
    }
    System.out.println("Min of max is :: " + minVal);
    return minVal;
}

public static void main(String[] args) {
    int arr[] = { 11, 2, 75, 92, 59, 90, 55 };
    int k = 3;
    minOfMaxSlidingWindows(arr, 7, 3);
}

```

Output

75

Maximum of Minimum Values in Sliding Windows

Problem: Given an array of integer, find maximum of all the minimum values in the sliding windows of length k.

Input: 11, 2, 75, 92, 59, 90, 55 and k = 3

Output: 59, as minimum values in sliding windows are [2, 2, 59, 59, 55]

Solution: Same as above problem. You will keep track of the minimum of all the maximum values in sliding windows too.

Example 8.10:

```

public static void maxOfMinSlidingWindows(int arr[], int size, int k) {
    ArrayDeque<Integer> que = new ArrayDeque<Integer>();
    int maxVal = -999999;
    for (int i = 0; i < size; i++) {
        // Remove out of range elements
        if (que.size() > 0 && que.peek() <= i - k)
            que.remove(); // Remove smaller values at left.
        while (que.size() > 0 && arr[que.peekLast()] >= arr[i])
            que.remove(); que.add(i);
        // window of size k
        if (i >= (k - 1) && maxVal < arr[que.peek()])
            maxVal = arr[que.peek()];
    }
    System.out.println("Max of min is :: " + maxVal);
}

public static void main(String[] args) {
    int arr[] = { 11, 2, 75, 92, 59, 90, 55 };
    int k = 3;
    maxOfMinSlidingWindows(arr, 7, 3);
}

```

First Negative Sliding Windows

Problem: Given an array of integer, find first negative of all the values in the sliding windows of length k.

Input:

Arr = [13, -2, -6, 10, -14, 50, 14, 21]

k = 3

Output: [-2, -2, -6, -14, -14, NAN]

Solution: We will create a queue. Only those index are added to queue which have negative values. Pop those values which are out of range of window.

Example 8.11:

```

public static void firstNegSlidingWindows(int arr[], int size, int k) {

```

```
ArrayDeque<Integer> que = new ArrayDeque<Integer>();  
  
for (int i = 0; i < size; i++) {  
    // Remove out of range elements  
    if (que.size() > 0 && que.peek() <= i - k)  
        que.remove(); if (arr[i] < 0)  
            que.add(i);  
    // window of size k  
    if (i >= (k - 1)) {  
        if (que.size() > 0)  
            System.out.print(arr[que.peek()]); else  
            System.out.print("NAN"); }  
    }  
}  
  
public static void main(String[] args) {  
    int arr[] = { 3, -2, -6, 10, -14, 50, 14, 21 };  
    int k = 3;  
    firstNegSlidingWindows(arr, 8, 3);  
    // Output [-2, -2, -6, -14, -14, NAN]  
}
```

Exercise

1. Implement queue using dynamic memory allocation, such that the implementation should follow the following constraints.
 - a. The user should use memory allocation from the heap using new operator. In this, you need to take care of the max value in the queue.
 - b. Once you are done with the above exercise and you are able to test your queue. Then you can add some more complexity to your code. In add() function when the queue is full, in place of printing, "Queue is full" you should allocate more space using new operator.
 - c. Once you are done with the above exercise. Now in remove function once you are below half of the capacity of the queue, you need to decrease the size of the queue by half. You should add one more variable "min" to queue so that you can track what is the original value capacity passed at initialization() function. Moreover, the capacity of the queue will not go below the value passed in the initialization.

(If you are not able to solve the above exercise, and then have a look into stack chapter, where we have done similar problems for stack)

2. Implement the below function for the queue:
 - a. IsEmpty: This is left as an exercise for the user. Take a variable, which will take care of the size of a queue if the value of that variable is zero, isEmpty should return true. If the queue is not empty, then it should return false.
 - b. Size: Use the size variable to be used under size function call. Size() function should return the number of elements in the queue.
3. Implement stack using a queue. Write a program for this problem. You can use just one queue.
4. Write a program to Reverse a stack using queue
5. Write a program to Reverse a queue using stack
6. Write a CompStack() function which takes pointer to two stack as an argument and return true or false depending upon whether all the elements of

the stack are equal or not. You are given `isEqual(int, int)` which will compare and return true if both values are equal and 0 if they are different.

7. Given two values X and Y. You need to convert X to Y by performing various steps. In a step, we can either multiply 2 to the value or subtract 1 from it. You need to display the number of steps and the path.

To reach 3 from 1, the path will be [1, 2, 4, 3] and number of steps will be 3.

Hint: Create an array to keep track of the number of steps and another array to keep track of parent value.

CHAPTER 9: TREE

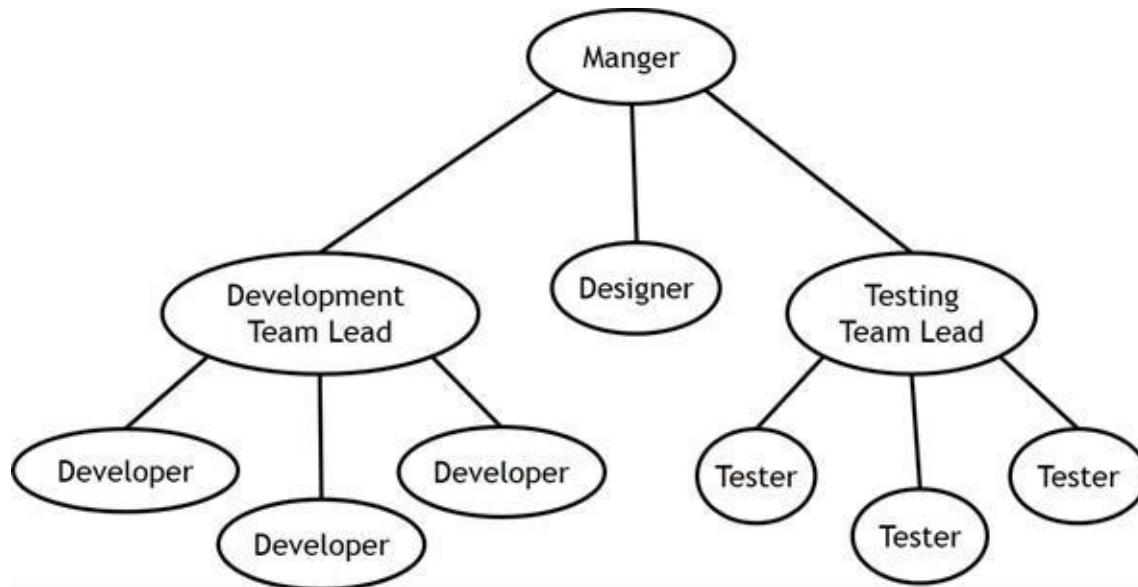
Introduction

We have already read about various linear data structures like an array, linked list, stack, queue etc.

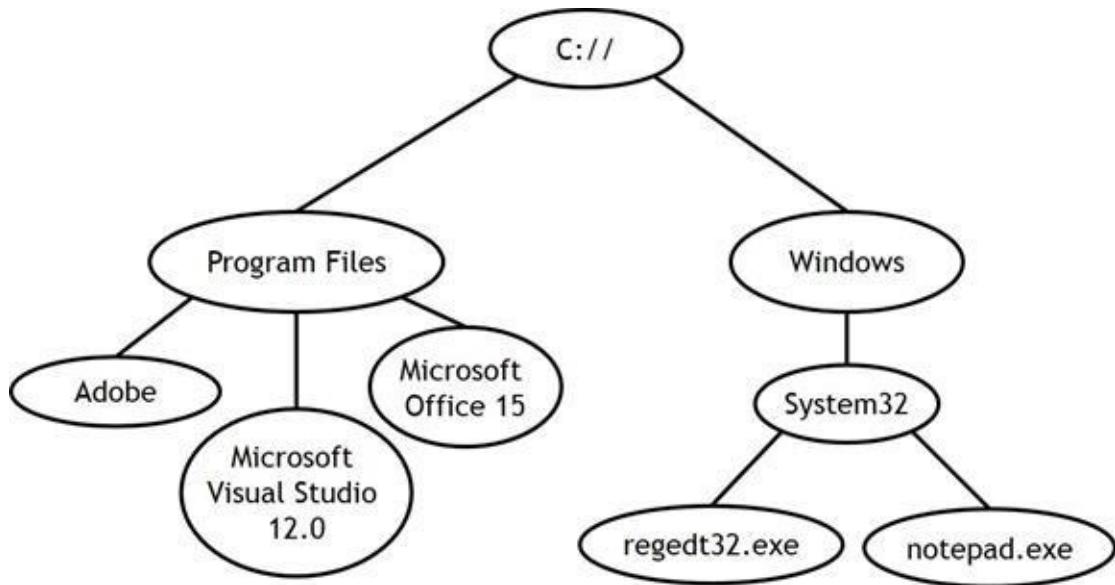
Both array and linked list have a drawback of linear time required for searching an element.

A tree is a nonlinear data structure, which is used to represent hierarchical relationships (parent-child relationship). Each node is connected by another node by directed edges.

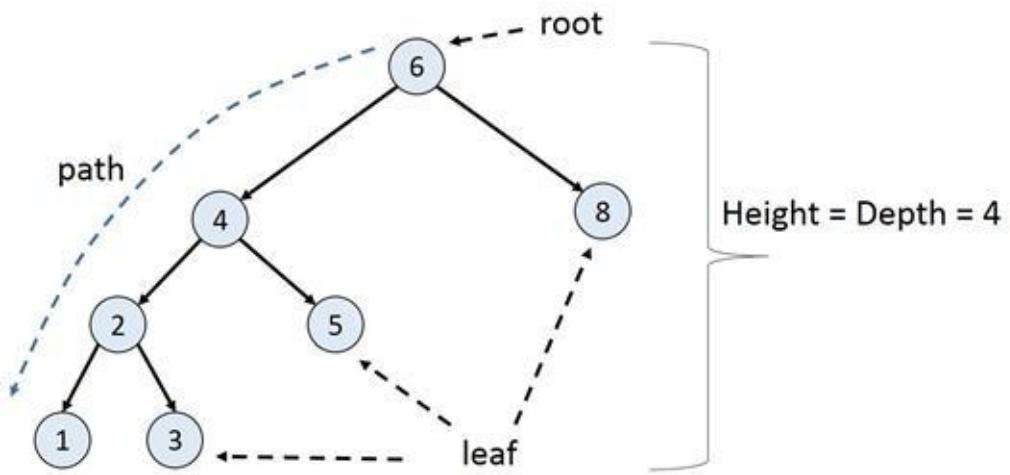
Example 1: Tree in team of some manager



Example 2: Tree in a file system



Terminology in tree



Root: The root of the tree is the only node without incoming edges. It is the top node of a tree.

Node: It is a fundamental element of a tree. Each node has data and two pointers that may point to null or its children

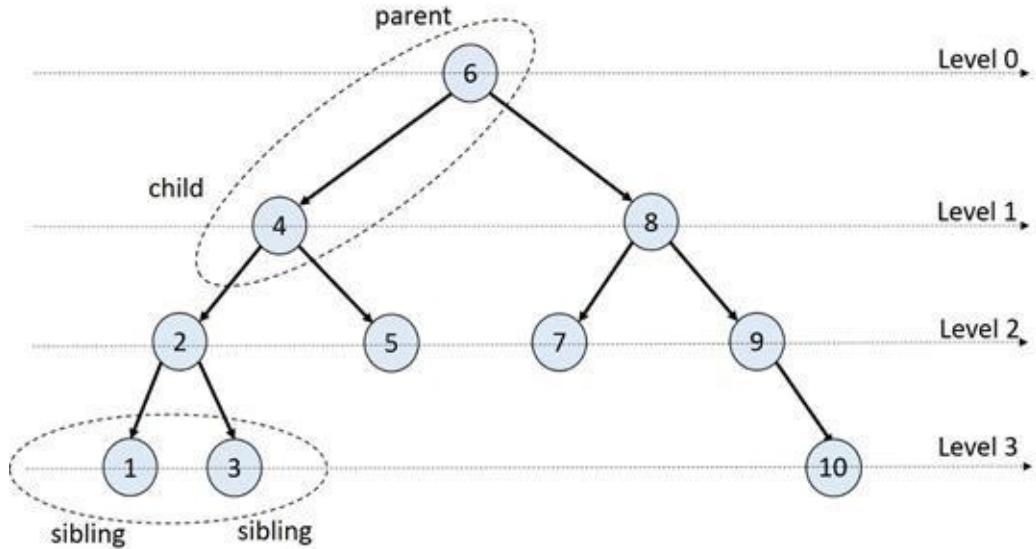
Edge: It is also a fundamental part of a tree, which is used to connect two nodes.

Path: A path is an ordered list of nodes that are connected by edges.

Leaf: A leaf node is a node that has no children.

Height of the tree: The height of a tree is the number of edges on the longest path between the root and a leaf.

The level of node: The level of a node is the number of edges on the path from the root node to that node.



Children: Nodes that have incoming edges from the same node is said to be the children of that node.

Parent: Node is a parent of all the child nodes that are linked by outgoing edges.

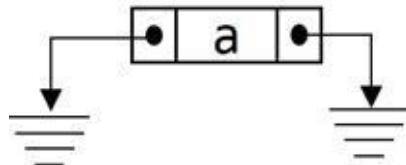
Sibling: Nodes in the tree that are children of the same parent are called siblings.

Ancestor: A node reachable through repeated moving from child to parent.

Binary Tree

A binary tree is a type tree in which each node has at most two children (0, 1 or 2), which are referred to as the left child and the right child.

Below is a node of the binary tree with "a" stored as data and whose left child (lChild) and whose right child (rchild) both are pointing towards null.



Example 9.1: Binary tree class.

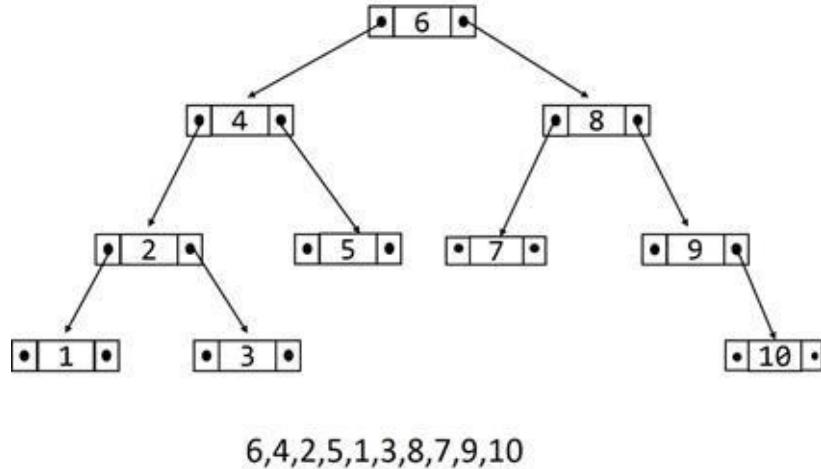
```
public class Tree {  
    private Node root;  
  
    private static class Node {  
        private int value;  
        private Node lChild;  
        private Node rChild;  
  
        public Node(int v, Node l, Node r) {  
            value = v;  
            lChild = l;  
            rChild = r;  
        }  
    }  
  
    public Node(int v) {  
        value = v;  
        lChild = null;  
        rChild = null;  
    }  
}  
  
public Tree() {  
    root = null;  
}
```

```

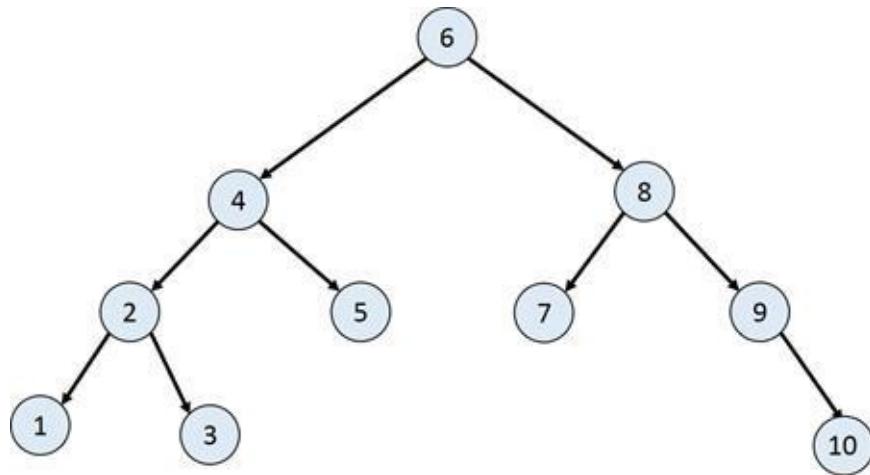
    }
/* Other methods */
}

```

Below is a binary tree whose nodes contains data from 1 to 10



In the rest of the book, binary tree will be represented as below:



Properties of Binary tree are:

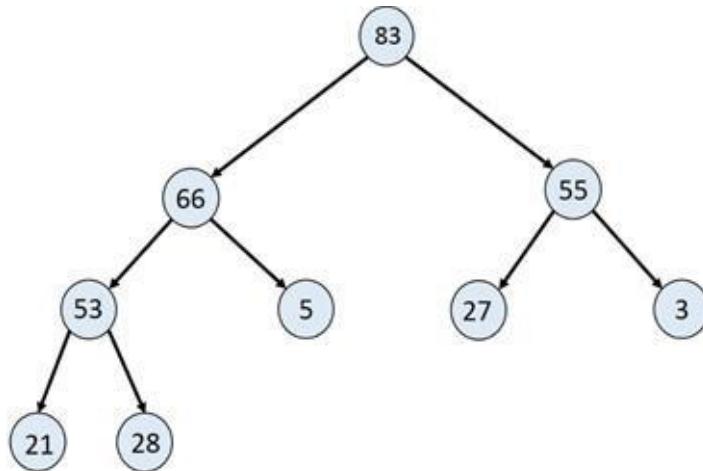
1. The maximum number of nodes on level i of a binary tree is 2^i , where $i \geq 1$
2. The maximum number of nodes in a binary tree of depth k is 2^{k+1} , where $k \geq 1$
3. There is exactly one path from the root to any nodes in a tree.

4. A tree with N nodes have exactly N-1 edges connecting these nodes.
5. The height of a complete binary tree of N nodes is $\log_2 N$.

Types of Binary trees

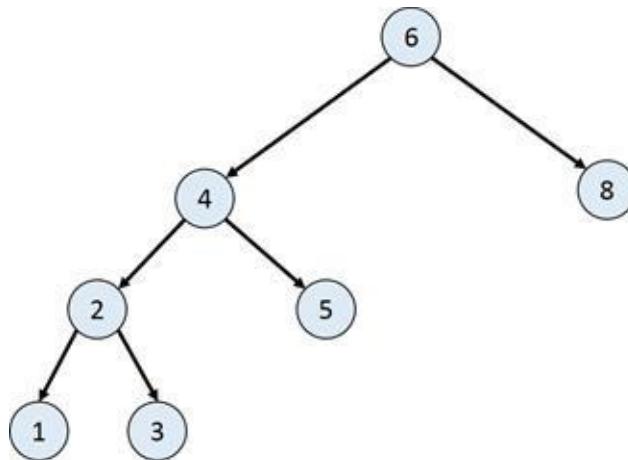
Complete binary tree

In a complete binary tree, every level except the last one is completely filled. All nodes in the left are filled first, then the right one. A binary heap is an example of a complete binary tree.



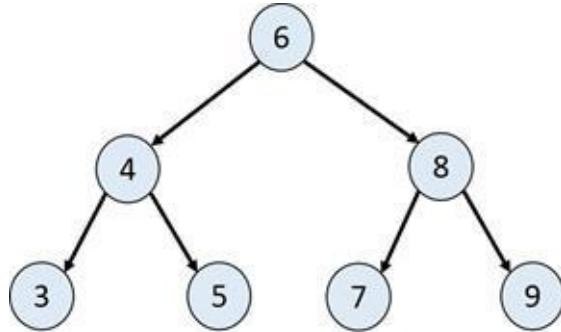
Full / Strictly binary tree

The full binary tree is a binary tree in which each node has exactly zero or two children.



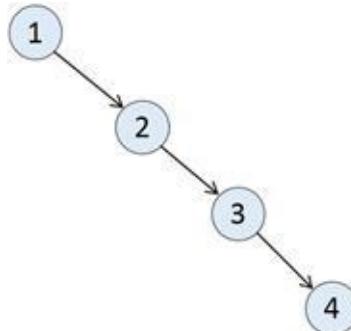
Perfect binary tree

The perfect binary tree is a type of full binary tree in which each non-leaf node has exactly two child nodes. All leaf nodes have identical path length and all possible node slots are occupied



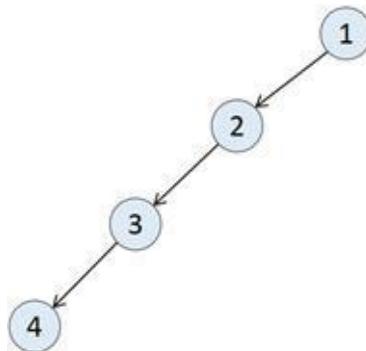
Right skewed binary tree

A binary tree in which either each node is has a right child or no child (leaf) is called as right skewed binary tree



Left skewed binary tree

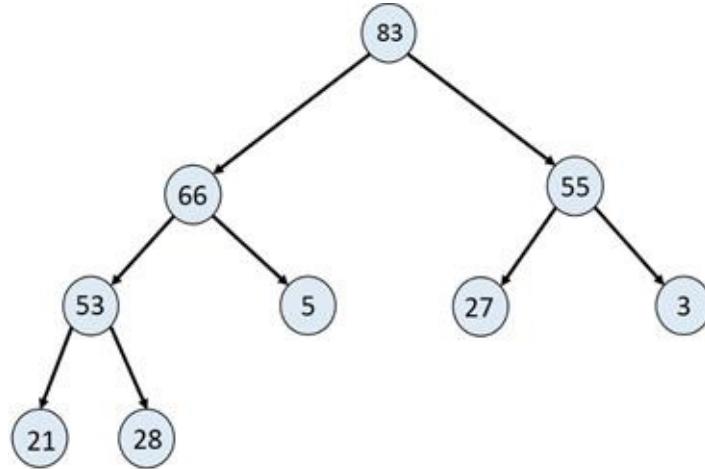
A binary tree in which either each node is has a left child or no child (leaf) is called as Left skewed binary tree



Height-balanced Binary Tree

A height-balanced binary tree is a binary tree such that the left & right subtrees for any given node differs in height by max one. AVL tree and RB tree are an example of height-balanced tree.

Note: Each complete binary tree is a height-balanced binary tree



Problems in Binary Tree

Create a Complete binary tree

Problem: Create a complete binary tree from values given as array.

Solution: Since there is no order defined in a binary tree, so nodes can be inserted in any order so it can be a skewed binary tree. But it is inefficient to do anything in a skewed binary tree so we will create a Complete binary tree. At each node, the middle value stored in the array is assigned to node. The left of array is passed to the left child of the node to create left sub-tree and the right portion of array is passed to right child of the node to create right sub-tree.

Example 9.2:

```
public void levelOrderBinaryTree(int[] arr) {  
    root = levelOrderBinaryTree(arr, 0);  
}  
  
public Node levelOrderBinaryTree(int[] arr, int start) {  
    int size = arr.length;  
    Node curr = new Node(arr[start]);  
    int left = 2 * start + 1;  
    int right = 2 * start + 2;  
  
    if (left < size)  
        curr.lChild = levelOrderBinaryTree(arr, left);  
    if (right < size)  
        curr.rChild = levelOrderBinaryTree(arr, right);  
  
    return curr;  
}  
  
public static void main(String[] args) {  
    Tree t = new Tree();  
    int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
    t.levelOrderBinaryTree(arr);
```

}

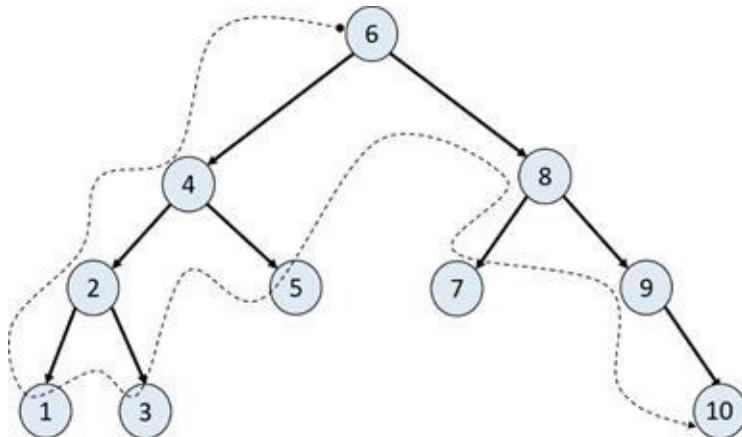
Complexity Analysis: This is an efficient algorithm for creating a complete binary tree.

Time Complexity: O(n), Space Complexity: O(n)

Pre-Order Traversal

Problem: Perform Pre-Order Traversal of binary tree.

Solution: In Pre-Order Traversal, parent is visited / traversed first, then left child subtree and then right child subtree. At each node, the value stored in it is printed and then followed by the values of left child subtree and right child subtree.



Example 9.3:

```
public void PrintPreOrder() {  
    PrintPreOrder(root);  
}  
  
private void PrintPreOrder(Node node)/* pre order */  
{  
    if (node != null) {  
        System.out.print(" " + node.value);  
        PrintPreOrder(node.lChild);  
        PrintPreOrder(node.rChild);  
    }  
}
```

```
}
```

Output:

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

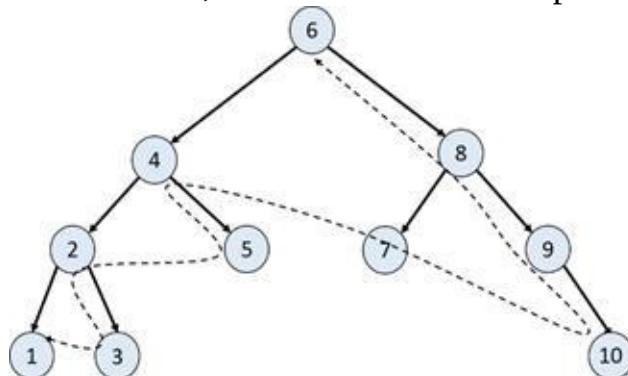
Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Note: If there is an algorithm, in which all nodes are traversed then complexity cannot be less than O(n). When there is a large portion of the tree, which is not traversed, then complexity reduces.

Post-Order Traversal

Problem: Perform Post-Order Traversal of binary tree.

Solution: In Post-Order Traversal, left child is visited / traversed first, then right child and then parent node. At each node, left child subtree is traversed then right child subtree and in the end, current node value is printed to the screen.



Example 9.4:

```
public void PrintPostOrder() {
    PrintPostOrder(root);
}

private void PrintPostOrder(Node node)/* post order */
{
    if (node != null) {
        PrintPostOrder(node.lChild);
        PrintPostOrder(node.rChild);
        System.out.print(" " + node.value);
    }
}
```

```
}
```

Output:

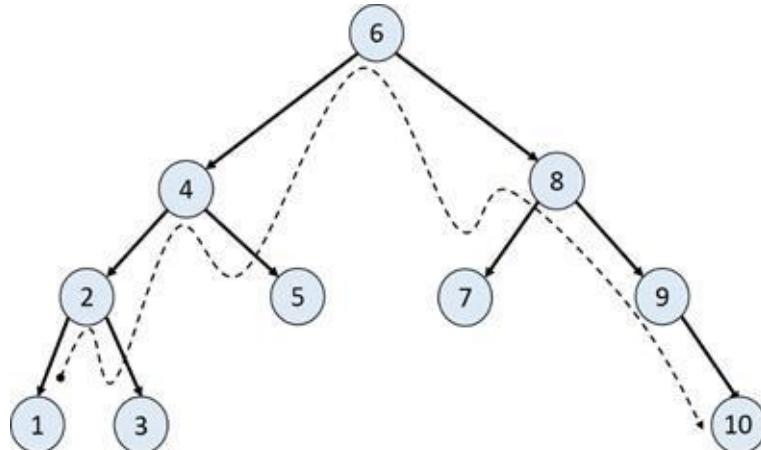
```
1 3 2 5 4 7 10 9 8 6
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

In-Order Traversal

Problem: Perform In-Order Traversal of binary tree.

Solution: In In-Order traversal, the nodes of left child subtree are traversed, then the value of node is printed to the screen and then the values of right child subtree are traversed.



Note: The output of In-Order traversal of BST is a sorted list in increasing order.

Example 9.5:

```
public void PrintInOrder() {  
    PrintInOrder(root);  
}
```

```
private void PrintInOrder(Node node)/* In order */  
{  
    if (node != null) {  
        PrintInOrder(node.lChild);  
        System.out.print(" " + node.value);  
        PrintInOrder(node.rChild);  
    }  
}
```

```
    }  
}
```

Output:

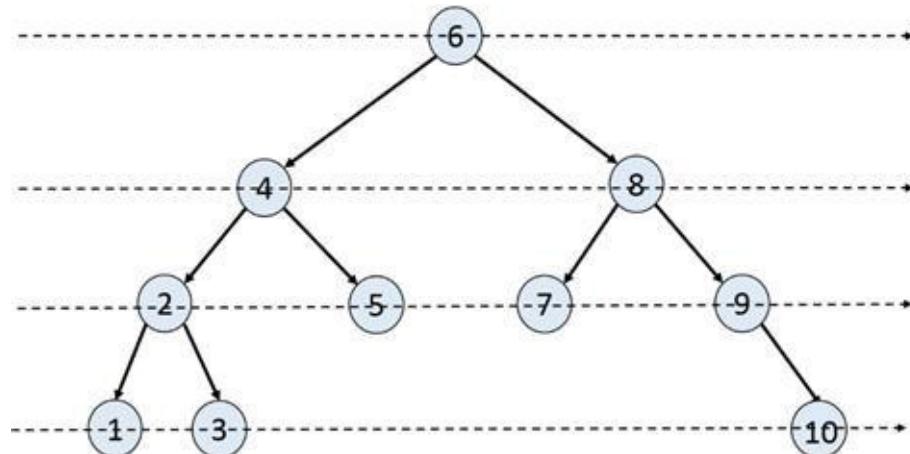
```
1 2 3 4 5 6 7 8 9 10
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Note: Pre-Order, Post-Order, and In-Order traversal are meant for all binary trees. They can be used to traverse any kind of a binary tree.

Level order traversal / Breadth First traversal

Problem: Write code to implement level order traversal of a tree. Such that nodes at depth k is printed before nodes at depth k+1.



Solution: Level order traversal or Breadth First traversal of a tree is done using a queue. At first, the root node pointer is added to a queue. The traversal of tree is done until the queue is empty. When we traverse the tree, we first remove an element from the queue, print the value stored in that node and then its left child and right child will be added to the queue.

Example 9.6:

```
public void PrintBredthFirst() {  
    ArrayDeque<Node> que = new ArrayDeque<Node>();  
    Node temp;
```

```

if (root != null)
que.add(root);

while (que.isEmpty() == false) {
temp = que.remove();
System.out.println(temp.value);
if (temp.lChild != null)
que.add(temp.lChild);
if (temp.rChild != null)
que.add(temp.rChild);
}
}

```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Print Depth First without using the recursion / system stack.

Problem: Perform depth first search of a binary tree without using recursion.

Solution: Depth first traversal of the tree is done using recursion by using system stack. The same can be done using stack. In the beginning, root node reference is added to the stack. The whole tree is traversed until the stack is empty. In each iteration, an element is popped from the stack, its value is printed to screen. Then right child and then left child of the node is added to stack.

Example 9.7:

```

public void PrintDepthFirst() {
ArrayDeque<Node> stk = new ArrayDeque<Node>();
Node temp;

if (root != null)
stk.push(root);

while (stk.isEmpty() == false) {
temp = stk.pop();
System.out.println(temp.value);
if (temp.lChild != null)
}
}

```

```

    stk.push(temp.lChild);
    if (temp.rChild != null)
        stk.push(temp.rChild);
    }
}

```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Print Level Order Line By Line

Problem: Perform level order traversal of binary tree, such that the levels are printed line by line.

First Solution: We will use two queue to perform level order traversal. Alternatively we will process queues and put the children of elements of queue into another queue so that when each level is processed we can print the output in different line.

Example 9.8:

```

void PrintLevelOrderLineByLine()
{
    ArrayDeque<Node> que1 = new ArrayDeque<Node>();
    ArrayDeque<Node> que2 = new ArrayDeque<Node>();
    Node temp = null;
    if (root != null)
        que1.add(root);
    while (que1.size() != 0 || que2.size() != 0)
    {
        while (que1.size() != 0)
        {
            temp = que1.remove();
            System.out.print(" " + temp.value);
            if (temp.lChild != null)
                que2.add(temp.lChild);
            if (temp.rChild != null)
                que2.add(temp.rChild);
        }
    }
}

```

```

System.out.println("");

while (que2.size() != 0)
{
    temp = (Node)que2.remove();
    System.out.print(" "+ temp.value);
    if (temp.lChild != null)
        que1.add(temp.lChild);
    if (temp.rChild != null)
        que1.add(temp.rChild);
}
System.out.println("");
}
}

```

Second solution: We can solve this problem using single queue. Let us suppose at any time we have all the nodes of kth level present in a queue. We can find the count of these elements. We can process elements from the queue count number of time and add their children to the queue. Then we can print a new line. At this point we have all the nodes at k+1th level. Start with adding root node as the first level to the queue and follow the steps mentioned above.

Example 9.9:

```

void PrintLevelOrderLineByLine2()
{
    ArrayDeque<Node> que = new ArrayDeque<Node>();
    Node temp = null;
    int count=0;

    if (root != null)
        que.add(root);
    while (que.size() != 0)
    {
        count = que.size();
        while (count > 0)
        {
            temp = que.remove();

```

```

        System.out.print(" " + temp.value);
        if (temp.lChild != null)
            que.add(temp.lChild);
        if (temp.rChild != null)
            que.add(temp.rChild);
        count -= 1;
    }
    System.out.println("");
}
}

```

Print Spiral Tree

Problem: Given a binary tree, print the nodes breadth first in spiral order.

Solution: Stacks are last in first out, so two stacks are used to process each level alternatively. The nodes are added and processed in such an order that nodes are printed in spiral order.

Example 9.10:

```

void PrintSpiralTree()
{
    Stack<Node> stk1 = new Stack<Node>();
    Stack<Node> stk2 = new Stack<Node>();

    Node temp;
    if (root != null)
        stk1.push(root);
    while (stk1.size() != 0 || stk2.size() != 0)
    {
        while (stk1.size() != 0)
        {
            temp = stk1.pop();
            System.out.print(" "+temp.value);
            if (temp.rChild != null)
                stk2.push(temp.rChild);
            if (temp.lChild != null)
                stk1.push(temp.lChild);
        }
    }
}

```

```

        stk2.push(temp.lChild);
    }
    while (stk2.size() != 0)
    {
        temp = stk2.pop();
        System.out.print(" "+temp.value);
        if (temp.lChild != null)
            stk1.push(temp.lChild);
        if (temp.rChild != null)
            stk1.push(temp.rChild);
    }
}
}

```

Nth Pre-Order

Problem: Given a binary tree, print the value of nodes that will be at nth index when tree is traversed in pre-order.

Solution: We want to print the node, which will be at the nth index when we print the tree in PreOrder traversal. Therefore, we keep a counter to keep track of the index. When the counter is equal to index, then we print the value and return the Nth preorder index node.

Example 9.11:

```

public void NthPreOrder(int index) {
    int[] counter = { 0 };
    NthPreOrder(root, index, counter);
}

private void NthPreOrder(Node node, int index, int[] counter)/* pre order */
{
    if (node != null) {
        counter[0]++;
        if (counter[0] == index) {
            System.out.print(node.value);
        }
    }
}

```

```
NthPreOrder(node.lChild, index, counter);
NthPreOrder(node.rChild, index, counter);
}
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth Post-Order

Problem: Given a binary tree, print the value of nodes that will be at nth index when tree is traversed in post-order.

Solution: We want to print the node that will be at the nth index when we print the tree in post order traversal. Therefore, we keep a counter to keep track of the index, but at this time, we will increment the counter after left child and right child traversal. When the counter is equal to index, then we print the value and return the nth post-order index node.

Example 9.12:

```
public void NthPostOrder(int index) {
    int[] counter = { 0 };
    NthPostOrder(root, index, counter);
}

private void NthPostOrder(Node node, int index, int[] counter)/* post order */
{
    if (node != null) {
        NthPostOrder(node.lChild, index, counter);
        NthPostOrder(node.rChild, index, counter);
        counter[0]++;
        if (counter[0] == index) {
            System.out.print(" "+node.value);
        }
    }
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth In Order

Problem: Given a binary tree, print the value of nodes that will be at nth index when tree is traversed in In-order.

Solution: We want to print the node that will be at the nth index when we print the tree in in-order traversal. Therefore, we keep a counter to keep track of the index, but at this time, we will increment the counter after left child traversal but before the right child traversal. When the counter is equal to index, then we print the value and return the nth in-order index node.

Example 9.13:

```
public void NthInOrder(int index) {  
    int[] counter = { 0 };  
    NthInOrder(root, index, counter);  
}  
  
private void NthInOrder(Node node, int index, int[] counter) {  
    if (node != null) {  
        NthInOrder(node.lChild, index, counter);  
        counter[0]++;  
        if (counter[0] == index) {  
            System.out.print(" "+node.value);  
        }  
        NthInOrder(node.rChild, index, counter);  
    }  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Print all the paths

Problem: Given a binary tree, print all the paths from the roots to the leaf.

Solution: Whenever we traverse a node, we add that node to the stack. When we reach a leaf, we print the whole list. When we return from a function, then we

remove the element that was added to the stack when we entered this function.

Example 9.14:

```
public void printAllPath()
{
    Stack<Integer> stk = new Stack<Integer>();
    printAllPathUtil(root, stk);
}

private void printAllPathUtil(Node curr, Stack<Integer> stk) {
    if (curr == null)
        return;

    stk.push(curr.value);

    if (curr.lChild == null && curr.rChild == null) {
        System.out.println(stk);
        stk.pop();
        return;
    }

    printAllPathUtil(curr.rChild, stk);
    printAllPathUtil(curr.lChild, stk);
    stk.pop();
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Number of Element

Problem: Find total number of nodes in binary tree.

Solution: Number of nodes at the right child and the number of nodes at the left child is added by one and we get the total number of nodes in any tree / sub-tree.

Example 9.15:

```
public int numNodes() {
```

```

    return numNodes(root);
}

public int numNodes(Node curr) {
    if (curr == null)
        return 0;
    else
        return (1 + numNodes(curr.rChild) + numNodes(curr.lChild));
}

```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Sum of All nodes in a BT

Problem: Given a binary tree, find sum of values of all the nodes of it.

Solution: We will find the sum of all the nodes recursively. sumAllBT() will return the sum of all the node of left and right subtree then we will add the value of current node and will return the final sum.

Example 9.16:

```

public int sumAllBT() {
    return sumAllBT(root);
}

private int sumAllBT(Node curr) {
    if (curr == null)
        return 0;

    return (curr.value + sumAllBT(curr.lChild) + sumAllBT(curr.rChild));
}

```

Number of Leaf nodes

Problem: Given a binary tree, find the number of leaf nodes in it.

Solution: If we add the number of leaf node in the right child with the number of

leaf nodes in the left child, we will get the total number of leaf node in any tree or subtree.

Example 9.17:

```
public int numLeafNodes() {  
    return numLeafNodes(root);  
}  
  
private int numLeafNodes(Node curr) {  
    if (curr == null)  
        return 0;  
    if (curr.lChild == null && curr.rChild == null)  
        return 1;  
    else  
        return (numLeafNodes(curr.rChild) + numLeafNodes(curr.lChild));  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Number of Full Nodes in a BT

Problem: Given a binary tree, find count of full nodes in it. A full node is one that have non-null left and right child.

Solution: A full node is a node that has both left and right child. We will recursively traverse the whole tree and will increase the count of full node as we find them.

Example 9.18:

```
public int numFullNodesBT() {  
    return numNodes(root);  
}  
  
public int numFullNodesBT(Node curr) {  
    int count;  
    if (curr == null)  
        return 0;
```

```

        count = numFullNodesBT(curr.rChild) + numFullNodesBT(curr.lChild);
        if (curr.rChild != null && curr.lChild != null)
            count++;

        return count;
    }
}

```

Search value in a Binary Tree

Problem: Search a particular value in binary tree.

Solution: To find if some value is there in a binary tree or not it is done using exhaustive search of the binary tree. First, the value of current node is compared with the value, which we are looking for. Then it is compared recursively inside the left child and right child.

Example 9.19:

```

public boolean searchBT(int value) {
    return searchBTUtil(root, value);
}

public boolean searchBTUtil(Node curr, int value) {
    boolean left, right;
    if (curr == null)
        return false;

    if (curr.value == value)
        return true;

    left = searchBTUtil(curr.lChild, value);
    if (left)
        return true;

    right = searchBTUtil(curr.rChild, value);
    if (right)
        return true;
}

```

```
    return false;  
}
```

Find Max in Binary Tree

Problem: Given a binary tree, find maximum value in it.

Solution: We recursively traverse the nodes of a binary tree. We will find the maximum value in the left and right subtree of any node then will compare the value with the value of the current node and finally return the largest of the three values.

Example 9.20:

```
public int findMaxBT() {  
    int ans = findMaxBT(root);  
    return ans;  
}  
  
private int findMaxBT(Node curr) {  
    int left, right;  
  
    if (curr == null)  
        return Integer.MIN_VALUE;  
  
    int max = curr.value;  
  
    left = findMaxBT(curr.lChild); right = findMaxBT(curr.rChild);  
    if (left > max)  
        max = left;  
    if (right > max)  
        max = right;  
  
    return max;  
}
```

Tree Depth

Problem: Given a binary tree, find its depth.

Solution: Depth of tree is calculated recursively by traversing left and right child of the root. At each level of traversal depth of both left & right child is calculated. The greater depth among the left and right child is added by one (which is the depth of the current node) and this value is returned.

Example 9.21:

```
public int TreeDepth() {  
    return TreeDepth(root);  
}  
  
private int TreeDepth(Node curr) {  
    if (curr == null)  
        return 0;  
    else {  
        int lDepth = TreeDepth(curr.lChild);  
        int rDepth = TreeDepth(curr.rChild);  
  
        if (lDepth > rDepth)  
            return lDepth + 1;  
        else  
            return rDepth + 1;  
    }  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Maximum Length Path in a BT/ Diameter of BT

Problem: Given a binary tree, find maximum length path in it.

Solution: To find the diameter of BT we need to find the depth of left child and right child then will add these two values and increment it by one so that we will get the maximum length path (diameter candidate) which contains the current node. Then we will find max length path in the left child sub-tree. We will also find the max length path in the right child sub-tree. Finally, we will compare the

three values and return the maximum value among them. This value will be the diameter of the Binary tree.

Example 9.22:

```
public int maxLengthPathBT() {  
    return maxLengthPathBT(root);  
}  
  
private int maxLengthPathBT(Node curr)// diameter  
{  
    int max;  
    int leftPath, rightPath;  
    int leftMax, rightMax;  
  
    if (curr == null)  
        return 0;  
  
    leftPath = TreeDepth(curr.lChild); rightPath = TreeDepth(curr.rChild);  
    max = leftPath + rightPath + 1;  
  
    leftMax = maxLengthPathBT(curr.lChild); rightMax =  
    maxLengthPathBT(curr.rChild);  
    if (leftMax > max)  
        max = leftMax;  
  
    if (rightMax > max)  
        max = rightMax;  
  
    return max;  
}
```

Identical

Problem: Find if two binary tree have identical value.

Solution: Two trees have identical values if at each level the value is equal.

Example 9.23:

```
public boolean isEqual(Tree T2) {  
    return isEqualUtil(root, T2.root);  
}  
  
private boolean isEqualUtil(Node node1, Node node2) {  
    if (node1 == null && node2 == null)  
        return true;  
    else if (node1 == null || node2 == null)  
        return false;  
    else  
        return (isEqualUtil(node1.lChild, node2.lChild) &&  
                isEqualUtil(node1.rChild, node2.rChild))  
            && (node1.value == node2.value));  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Copy Tree

Problem: Given a binary tree, copy its value in another binary tree.

Solution: Copy tree is done by copy nodes of the input tree at each level of the traversal of the tree. At each level of the traversal of nodes of the tree, a new node is created and the value of the input tree node is copied to it. The left child tree is copied recursively and then pointer to new subtree is returned which will be assigned to the left child pointer of the current new node. Similarly for the right child node too. Finally, the tree is copied.

Example 9.24:

```
public Tree CopyTree() {  
    Tree tree2 = new Tree();  
    tree2.root = CopyTree(root);  
    return tree2;  
}  
  
private Node CopyTree(Node curr) {
```

```

Node temp;
if (curr != null) {
    temp = new Node(curr.value);
    temp.lChild = CopyTree(curr.lChild);
    temp.rChild = CopyTree(curr.rChild);
    return temp;
} else
    return null;
}

```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Copy Mirror Tree

Problem: Given a binary tree, copy its value to create another tree, which is mirror image of the original tree.

Solution: Copy mirror image of the tree is done same as copy tree, but in place of left child pointing to the tree that is formed by left child traversal of input tree. This time left child points to the tree formed by right child traversal of the input tree. Similarly right child points to the tree formed by the traversal of the left child of the input tree.

Example 9.25:

```

public Tree CopyMirrorTree() {
    Tree tree2 = new Tree();
    tree2.root = CopyMirrorTree(root); return tree2;
}

private Node CopyMirrorTree(Node curr) {
    Node temp;
    if (curr != null) {
        temp = new Node(curr.value);
        temp.rChild = CopyMirrorTree(curr.lChild);
        temp.lChild = CopyMirrorTree(curr.rChild);
        return temp;
    } else
        return null;
}

```

```
    return null;  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Free Tree

Problem: Given a binary tree, free all its nodes.

Solution: Point root of tree to null. The memory of nodes of tree will be freed using garbage collection.

Example 9.26:

```
public void Free() {  
    root = null;  
}
```

Complexity Analysis: Time Complexity: O(1), Space Complexity: O(1)

Is Complete Tree

Problem: Given a binary tree find if it is a complete tree.

First Solution: We perform breadth first traversal of tree using a queue. If we get a node that does not have left child then it cannot have right child too. In general, if we have any node which does not have both the child then it is not possible for any other node in breadth first traversal to have any child.

Example 9.27:

```
boolean isCompleteTree()  
{  
    ArrayDeque<Node> que = new ArrayDeque<Node>();  
    Node temp = null;  
    int noChild = 0;  
    if (root != null)  
        que.add(root);  
    while (que.size() != 0)  
    {
```

```

temp = que.remove();
if (temp.lChild != null)
{
    if (noChild == 1)
        return false;
    que.add(temp.lChild);
}
else
    noChild = 1;

if (temp.rChild != null)
{
    if (noChild == 1)
        return false;
    que.add(temp.rChild);
}
else
    noChild = 1;
}
return true;
}

```

Second Solution: Just like a heap if we number child based on parent. Let parent location is index so left child location will be $(2 * \text{index} + 1)$ and right child location will be $(2 * \text{index} + 2)$.

Example 9.28:

```

boolean isCompleteTreeUtil(Node curr, int index, int count)
{
    if (curr == null)
        return true;
    if (index > count)
        return false;
    return isCompleteTreeUtil(curr.lChild, index * 2 + 1, count)
        && isCompleteTreeUtil(curr.rChild, index * 2 + 2, count);
}

```

```

boolean isCompleteTree2()
{
    int count = numNodes();
    return isCompleteTreeUtil(root, 0, count);
}

```

Is a Heap

Problem: Given a binary tree find if it represent a Min Heap.

To see if tree is a heap we need to check two conditions:

- 1) It is a complete tree.
- 2) Value of a parent node is smaller than or equal to its left and right child.

First Solution: First solution is to test if given tree is complete and second is to check if parent-child property is followed. If tree is a complete tree and all parent nodes in tree have value less than or equal to its children than tree represents a Min Heap.

isCompleteTree() function call takes linear time so is the isHeapUtil() function. So the total time complexity is $O(n)$, the whole tree is traversed three times. First to find total number of elements in tree, second to find if it is complete tree. Then once for testing heap property.

Example 9.29:

```

boolean isHeapUtil(Node curr, int parentValue)
{
    if (curr == null)
        return true;
    if (curr.value < parentValue)
        return false;
    return (isHeapUtil(curr.lChild, curr.value) && isHeapUtil(curr.rChild,
curr.value));
}

boolean isHeap()
{

```

```

int infi = -9999999;
return (isCompleteTree() && isHeapUtil(root, infi));
}

```

Second Solution: We can combine isCompleteTree and isHeapUtil function inside single function.

Example 9.30:

```

boolean isHeapUtil2(Node curr, int index, int count, int parentValue)
{
    if (curr == null)
        return true;
    if (index > count)
        return false;
    if (curr.value < parentValue)
        return false;
    return isHeapUtil2(curr.lChild, index * 2 + 1, count, curr.value) &&
isHeapUtil2(curr.rChild, index * 2 + 2, count, curr.value);
}

```

```

boolean isHeap2()
{
    int count = numNodes();
    int parentValue = -9999999;
    return isHeapUtil2(root, 0, count, parentValue);
}

```

Iterative Pre-order

Problem: Given a binary tree, without using recursion perform pre-order traversal of tree.

Solution: In place of using system stack in recursion, we can traverse the tree using stack data structure.

Example 9.31:

```

public void iterativePreOrder() {

```

```

Stack<Node> stk = new Stack<Node>();
Node curr;

if (root != null)
    stk.add(root);

while (stk.isEmpty() == false) {
    curr = stk.pop();
    System.out.print(curr.value + " ");

    if (curr.rChild != null)
        stk.push(curr.rChild);

    if (curr.lChild != null)
        stk.push(curr.lChild);
}

```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Iterative Post-order

Problem: Given a binary tree, without using recursion perform post-order traversal of tree.

Solution: In place of using system stack in recursion, we can traverse the tree using stack data structure.

Example 9.32:

```

public void iterativePostOrder() {
    Stack<Node> stk = new Stack<Node>();
    Stack<Integer> visited = new Stack<Integer>();
    Node curr;
    int vtd;

    if (root != null) {
        stk.add(root);
        visited.add(0);
    }
}
```

```

    }

    while (stk.isEmpty() == false) {
        curr = stk.pop();
        vtd = visited.pop();
        if (vtd == 1) {
            System.out.print(curr.value + " ");
        } else {
            stk.push(curr);
            visited.push(1);
        }
        if (curr.rChild != null) {
            stk.push(curr.rChild);
            visited.push(0);
        }
        if (curr.lChild != null) {
            stk.push(curr.lChild);
            visited.push(0);
        }
    }
}

```

Complexity Analysis: Time Complexity: $O(n)$, Space Complexity: $O(n)$

Iterative In-order

Problem: Given a binary tree, without using recursion perform In-order traversal of tree.

Solution: In place of using system stack in recursion, we can traverse the tree using stack data structure.

Example 9.33:

```

public void iterativeInOrder() {
    Stack<Node> stk = new Stack<Node>();
    Stack<Integer> visited = new Stack<Integer>();
    Node curr;
    int vtd;
}

```

```
if (root != null) {  
    stk.add(root);  
    visited.add(0);  
}  
  
while (stk.isEmpty() == false) {  
    curr = stk.pop();  
    vtd = visited.pop();  
    if (vtd == 1) {  
        System.out.print(curr.value + " ");  
    } else {  
        if (curr.rChild != null) {  
            stk.push(curr.rChild);  
            visited.push(0);  
        }  
        stk.push(curr);  
        visited.push(1);  
        if (curr.lChild != null) {  
            stk.push(curr.lChild);  
            visited.push(0);  
        }  
    }  
}
```

Complexity Analysis: Time Complexity: $O(n)$, Space Complexity: $O(n)$

Tree to List Recursively

Problem: Given a binary tree, create a doubly linked list from this such that the elements are in the order of in-order traversal of tree.

Solution: Tree to the array is done recursively. At each node, we have to assume that the tree to list function will do its job for the left child and right child. Then we will combine the result of the left child and right child traversal. We need a head and tail pointer of the left list and right list to combine them with the current node. In the process of integration, the current node will be added to the

tail of the left list and current node will be added to the head to the right list. Head of the left list will become the head of the newly formed list and tail of the right list will become the tail of the newly created list.

Example 9.34:

```
public Node treeToListRec() {  
    Node head = treeToListRec(root); Node temp = head;  
    return temp;  
}  
  
private Node treeToListRec(Node curr) {  
    Node Head = null, Tail = null;  
    if (curr == null)  
        return null;  
  
    if (curr.lChild == null && curr.rChild == null) {  
        curr.lChild = curr;  
        curr.rChild = curr;  
        return curr;  
    }  
  
    if (curr.lChild != null) {  
        Head = treeToListRec(curr.lChild); Tail = Head.lChild;  
  
        curr.lChild = Tail;  
        Tail.rChild = curr;  
    } else  
        Head = curr;  
  
    if (curr.rChild != null) {  
        Node tempHead = treeToListRec(curr.rChild);  
        Tail = tempHead.lChild;  
  
        curr.rChild = tempHead;  
        tempHead.lChild = curr;  
    } else  
        Tail = curr;
```

```
    Head.lChild = Tail;
    Tail.rChild = Head;
    return Head;
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

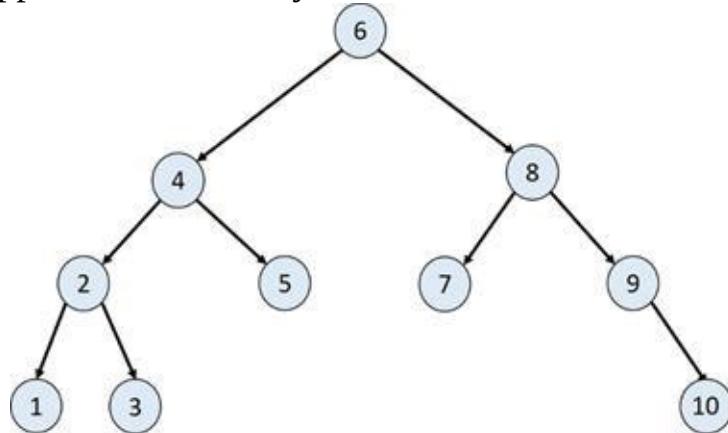
Binary Search Tree (BST)

A binary search tree (BST) is a binary tree on which nodes are ordered in the following way:

- The key in the left subtree is less than the key in its parent node.
- The key in the right subtree is greater than the key in its parent node.
- No duplicate key is allowed.

Note: there can be two separate key and value fields in the tree node. But for simplicity, we are considering value as the key. All problems in the binary search tree are solved using this supposition that the value in the node is key for the tree.

Note: Since binary search tree is a binary tree. So all the above algorithm of a binary tree are applicable to a binary search tree.



Problems in Binary Search Tree (BST)

All binary tree algorithms are valid for binary search tree too.

Create a binary search tree from sorted list

Problem: Create a binary tree from list of values in sorted order. Since the elements in the array are in sorted order and we want to create a binary search tree in which left subtree nodes are having values less than the current node and right subtree nodes have value greater than the value of the current node.

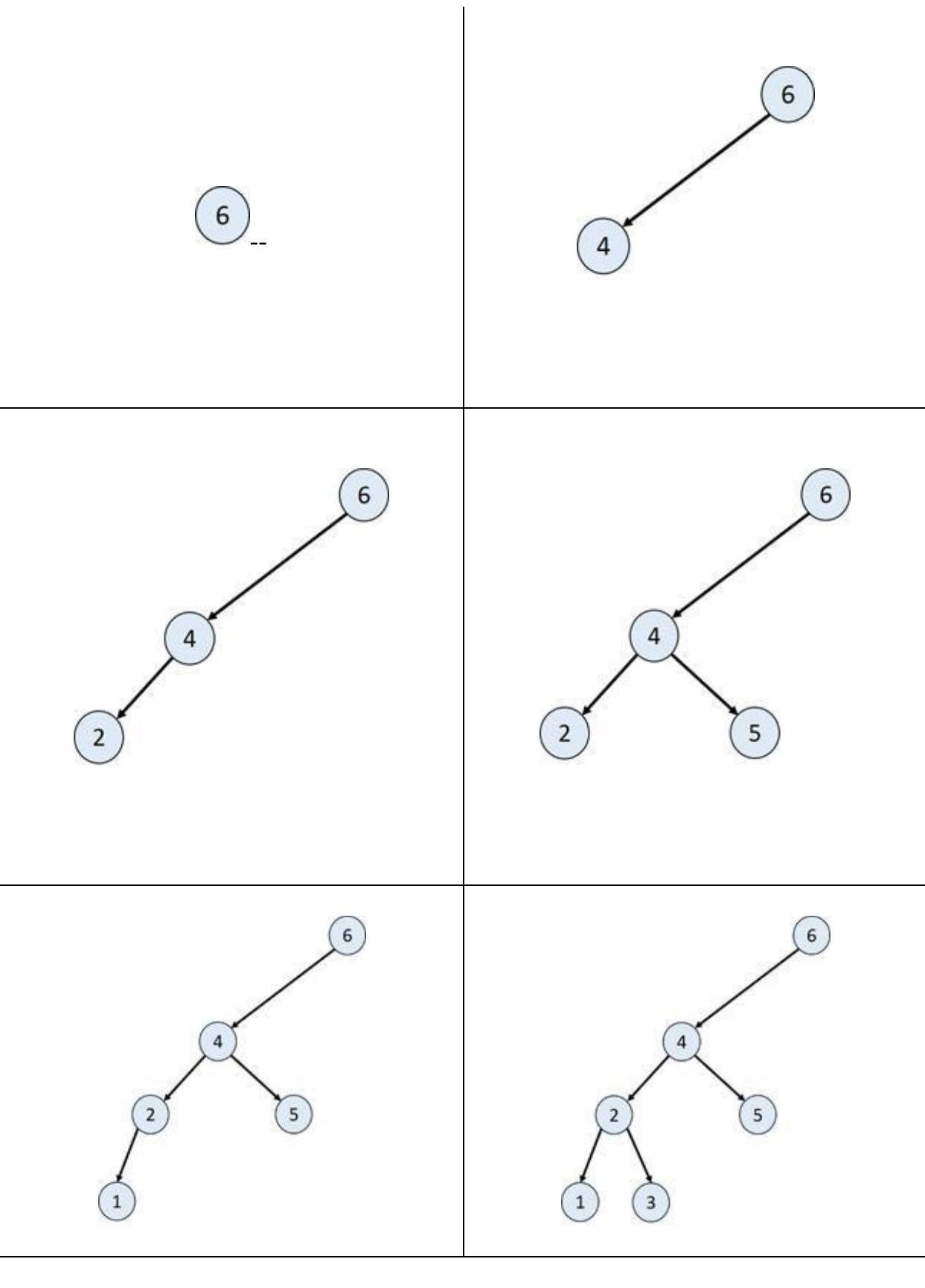
Solution: We have to find the middle node to create a current node and send the rest of the array to construct left and right subtree.

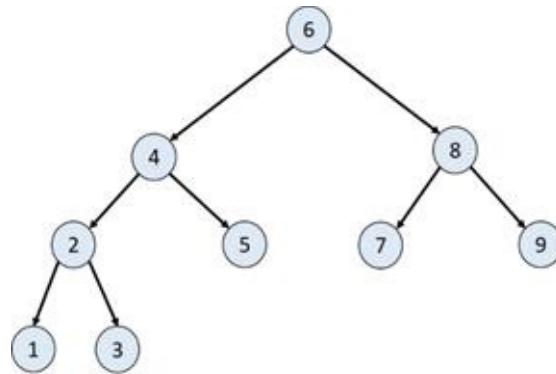
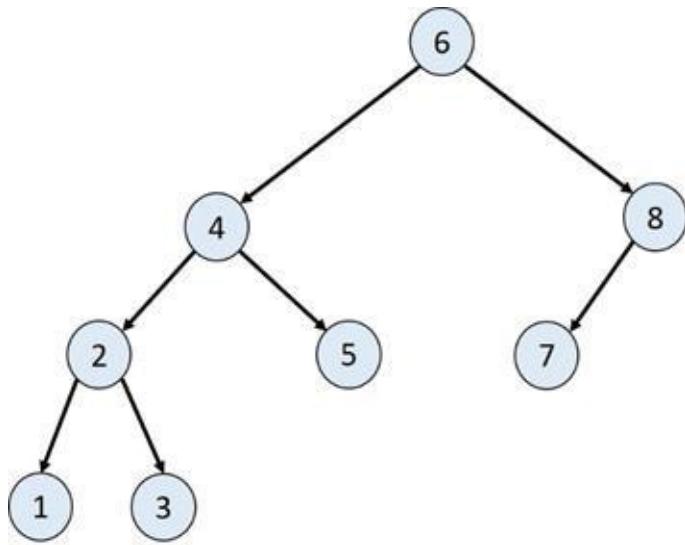
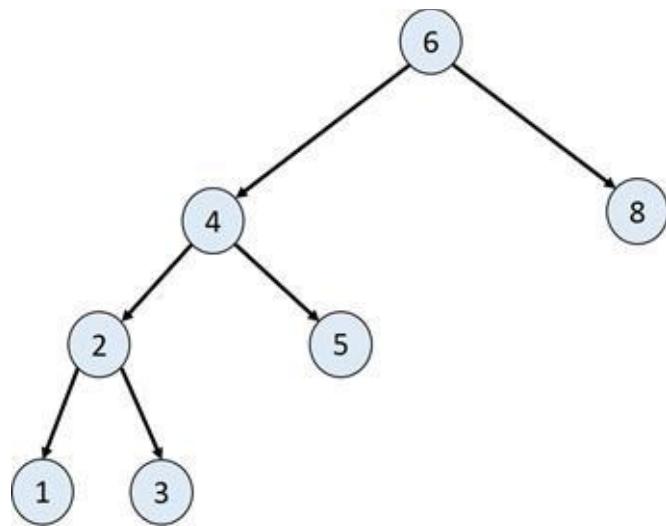
Example 9.35:

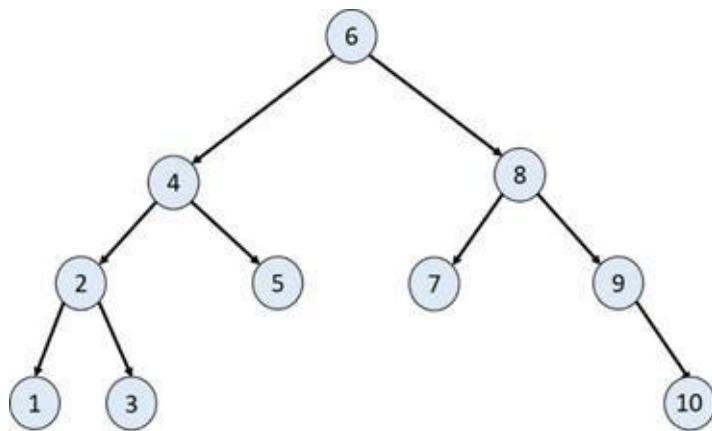
```
public void CreateBinaryTree(int[] arr) {  
    root = CreateBinaryTree(arr, 0, arr.length - 1);  
}  
  
private Node CreateBinaryTree(int[] arr, int start, int end) {  
    Node curr = null;  
    if (start > end)  
        return null;  
  
    int mid = (start + end) / 2;  
    curr = new Node(arr[mid]); curr.lChild = CreateBinaryTree(arr, start, mid - 1);  
    curr.rChild = CreateBinaryTree(arr, mid + 1, end);  
    return curr;  
}
```

Insertion

Below is a step by step tree after inserting nodes in the order. Nodes with key 6,4,2,5,1,3,8,7,9,10 are inserted in a tree.







Solution: Smaller values will be added to the left child sub-tree of a node and greater value will be added to the right child sub-tree of the current node.

Example 9.36:

```

public void InsertNode(int value) {
    root = InsertNode(root, value);
}

private Node InsertNode(Node node, int value) {
    if (node == null) {
        node = new Node(value, null, null);
    } else {
        if (node.value > value) {
            node.lChild = InsertNode(node.lChild, value);
        } else {
            node.rChild = InsertNode(node.rChild, value);
        }
    }
    return node;
}
  
```

Complexity Analysis: Time Complexity: $O(n)$, Space Complexity: $O(n)$

Find Node

Problem: Find the node with the value given.

Solution: The value greater than the current node value will be in the right child sub-tree and the value smaller than the current node is there in the left child sub-tree. We can find a value by traversing the left or right subtree iteratively.

Example 9.37:

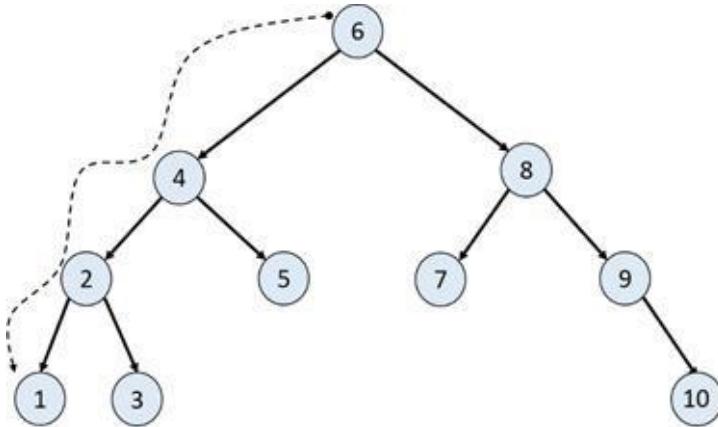
```
public boolean Find(int value) {  
    Node curr = root;  
  
    while (curr != null) {  
        if (curr.value == value) {  
            return true;  
        } else if (curr.value > value) {  
            curr = curr.lChild;  
        } else {  
            curr = curr.rChild;  
        }  
    }  
    return false;  
}
```

Complexity Analysis: Time Complexity: $O(n)$, Space Complexity: $O(1)$

Find Min

Find the node with the minimum value.

Solution: left most child of the tree will be the node with the minimum value.



Example 9.38:

```
public int FindMin() {
    Node node = root;
    if (node == null) {
        return Integer.MAX_VALUE;
    }

    while (node.lChild != null) {
        node = node.lChild;
    }
    return node.value;
}
```

Example 9.39:

```
public Node FindMinNode(Node curr) {
    Node node = curr;
    if (node == null) {
        return null;
    }

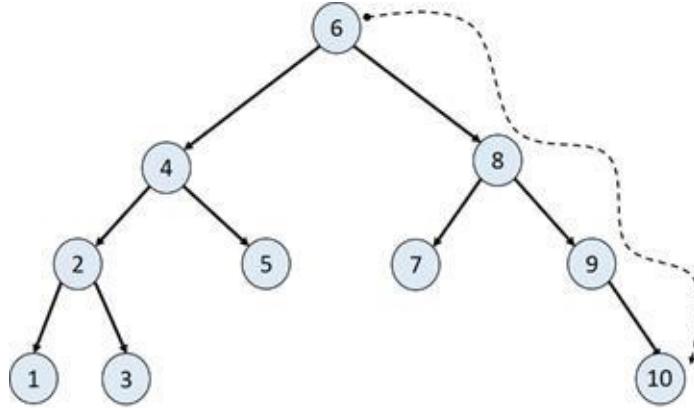
    while (node.lChild != null) {
        node = node.lChild;
    }
    return node;
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Find Max

Problem: Find the node in the tree with the maximum value.

Solution: Right most node of the tree will be the node with the maximum value.



Example 9.40:

```
public int FindMax() {  
    Node node = root;  
    if (node == null) {  
        return Integer.MIN_VALUE;  
    }  
  
    while (node.rChild != null) {  
        node = node.rChild;  
    }  
    return node.value;  
}
```

Example 9.41:

```
public Node FindMaxNode(Node curr) {  
    Node node = curr;  
    if (node == null) {  
        return null;  
    }  
  
    while (node.rChild != null) {  
        node = node.rChild;  
    }
```

```
    return node;  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Is tree a BST

Problem: Find is a given binary tree is a binary search tree.

First solution: At each node we check whether, max value of left subtree is smaller than the value of current node and min value of right subtree is greater than the current node or not.

Example 9.42:

```
public boolean isBST3(Node root) {  
    if (root == null)  
        return true;  
    if (root.lChild != null && FindMax(root.lChild).value > root.value)  
        return false;  
    if (root.rChild != null && FindMin(root.rChild).value <= root.value)  
        return false;  
    return (isBST3(root.lChild) && isBST3(root.rChild));  
}
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

The above solution is correct but it is not efficient, as same tree nodes are traversed many times.

Second solution: A better solution will be the one in which we will look into each node only once. This is done by narrowing the range. We will be use an isBSTUtil() function which takes the max and min range of the values of the nodes. The initial value of min and max should be minimum and maximum value of integer (for simplicity we are taking -999999 and 999999).

Example 9.43:

```
public boolean isBST() {  
    return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);  
}
```

```

public boolean isBST(Node curr, int min, int max) {
    if (curr == null)
        return true;

    if (curr.value < min || curr.value > max)
        return false;

    return isBST(curr.lChild, min, curr.value) && isBST(curr.rChild, curr.value,
max);
}

```

Complexity Analysis: Time Complexity: $O(n)$, Space Complexity: $O(n)$ for stack.

Third solution: Above method is correct and efficient but there is an easy method to do the same. We can do in-order traversal of nodes and see if we are getting a strictly increasing sequence

Example 9.44:

```

public boolean isBST2() {

    int[] count = new int[1];
    return isBST2(root, count);
}

private boolean isBST2(Node root, int[] count)/* in order traversal */
{
    boolean ret;
    if (root != null) {
        ret = isBST2(root.lChild, count);
        if (!ret)
            return false;

        if (count[0] > root.value)
            return false;
        count[0] = root.value;
    }
}

```

```

ret = isBST2(root.rChild, count);
if (!ret)
return false;
}
return true;
}

```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n) for stack

Delete Node

Problem: Remove the node x from the binary search tree, reorganize nodes of binary search tree to maintain its necessary properties.

There are three cases in delete node, let us call the node that need to be deleted as x.

Case 1: node x has no children. Just delete it (i.e. Change parent node so that it does not point to x)

Case 2: node x has one child. Splice out x by linking x's parent to x's child

Case 3: node x has two children. Splice out the x's successor and replace x with x's successor

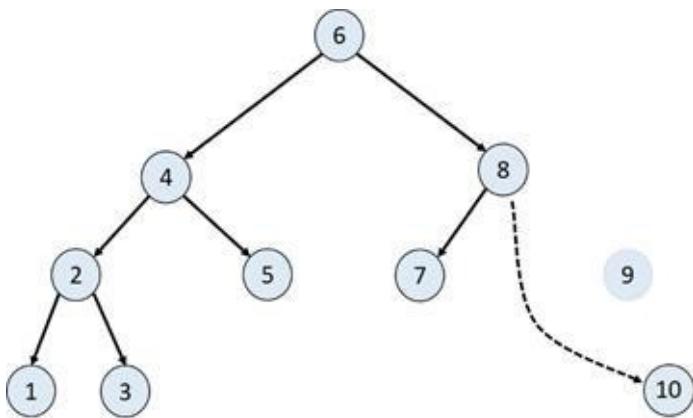
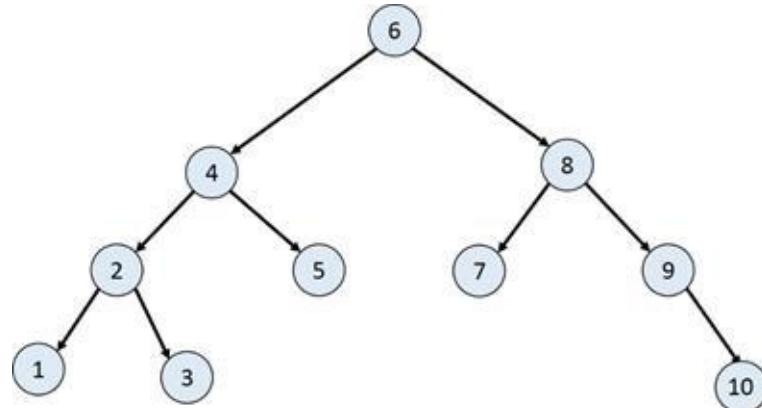
When the node to be deleted has no children

This is a trivial case, in which we directly remove the node by returning null.

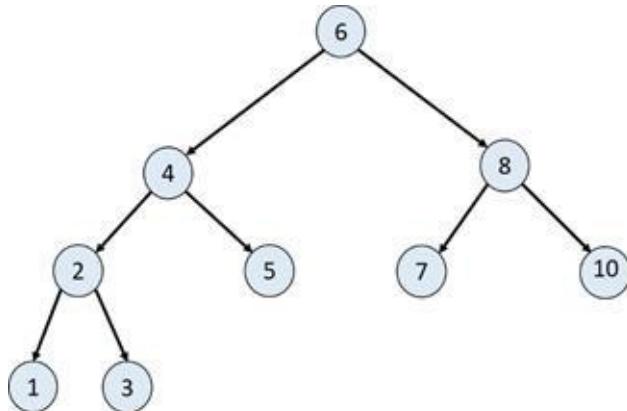
When the node to be deleted has only one child.

In this case, we return the child of the node.

	We want to remove node with value 9. The node has only one child.
--	---

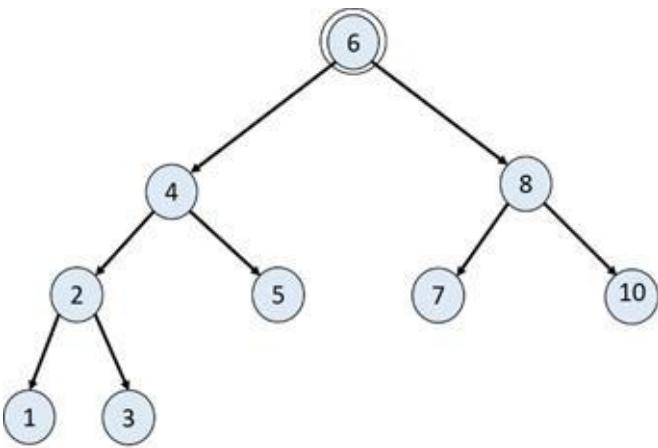


Right child of the parent of node with value 9 that is the node with value 8 will point to the child node of node with value 9. i.e node with value 10.

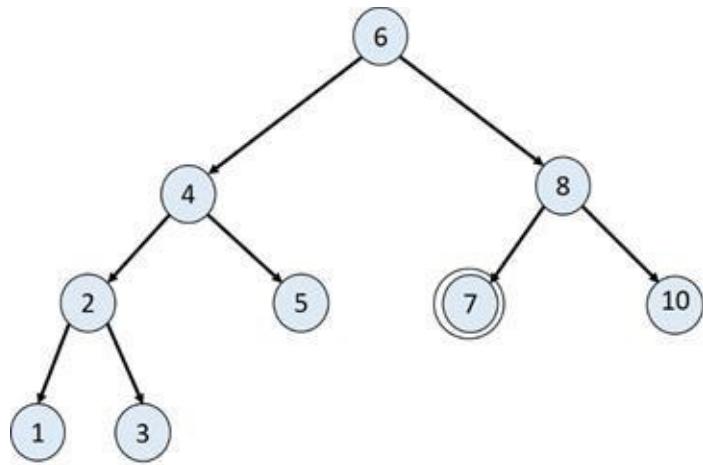


Finally, node with value 9 is removed from the tree.

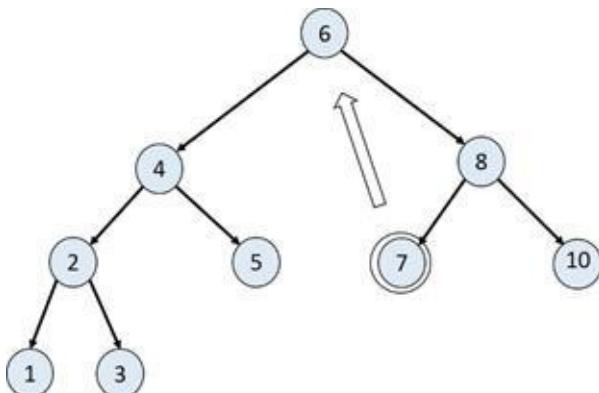
When the node to be deleted has two children.



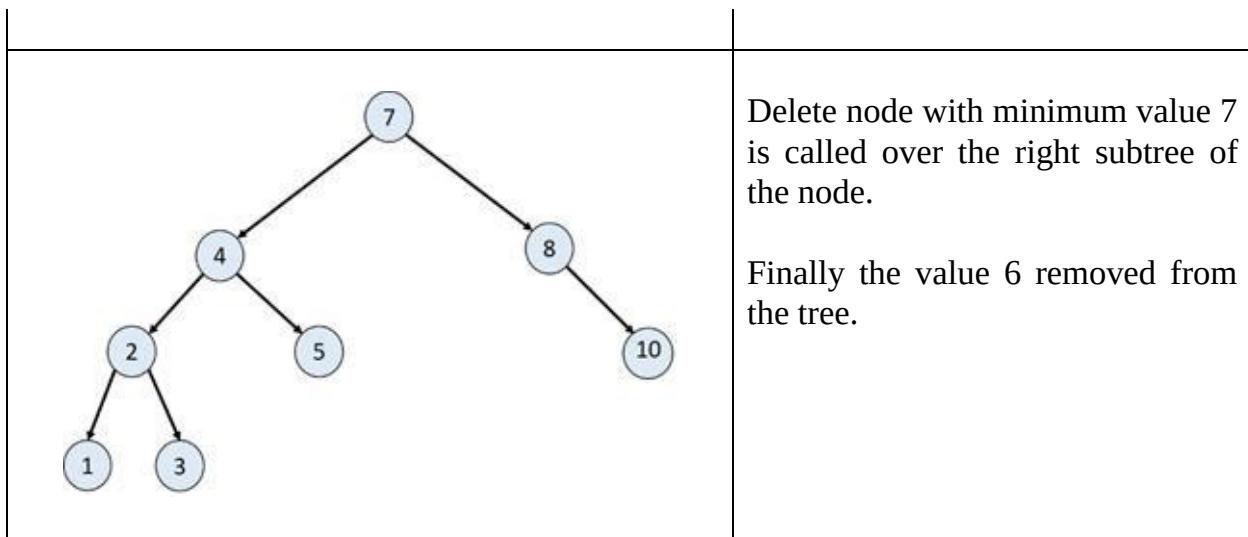
We want to delete node with value 6. Which have two children.



We have found minimum value node of the right subtree of node with value 6.



Minimum value is copied to the node with value 6.



Example 9.45:

```

public void DeleteNode(int value) {
    root = DeleteNode(root, value);
}

private Node DeleteNode(Node node, int value) {
    Node temp = null;

    if (node != null) {
        if (node.value == value) {
            if (node.lChild == null && node.rChild == null) {
                return null;
            } else {
                if (node.lChild == null) {
                    temp = node.rChild;
                    return temp;
                }

                if (node.rChild == null) {
                    temp = node.lChild;
                    return temp;
                }
            }
        }
    }

    Node minNode = FindMinNode(node.rChild);
    int minValue = minNode.value;
  
```

```

node.value = minValue;
node.rChild = DeleteNode(node.rChild, minValue);
}
} else {
if (node.value > value) {
node.lChild = DeleteNode(node.lChild, value);
} else {
node.rChild = DeleteNode(node.rChild, value);
}
}
return node;
}

```

Analysis: Time Complexity: O(n), Space Complexity: O(n)

Least Common Ancestor

Problem: In a tree T. The least common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants.

Example 9.46:

```

public int LcaBST(int first, int second) {
    return LcaBST(root, first, second);
}

private int LcaBST(Node curr, int first, int second) {
    if (curr == null) {
        return Integer.MAX_VALUE;
    }

    if (curr.value > first && curr.value > second) {
        return LcaBST(curr.lChild, first, second);
    }
    if (curr.value < first && curr.value < second) {
        return LcaBST(curr.rChild, first, second);
    }
}

```

```
    return curr.value;  
}
```

Trim the Tree nodes which are Outside Range

Problem: Given a binary search tree and a range as min, max. We need to delete all the nodes of the tree that are out of this range.

Solution: Traverse the tree and each node that is having value outside the range will delete itself. All the deletion will happen from inside out so we do not have to care about the children of a node as if they are out of range then they had already had deleted themselves.

Example 9.47:

```
public void trimOutsideRange(int min, int max) {  
    trimOutsideRange(root, min, max);  
}  
  
private Node trimOutsideRange(Node curr, int min, int max) {  
    if (curr == null)  
        return null;  
  
    curr.lChild = trimOutsideRange(curr.lChild, min, max);  
    curr.rChild = trimOutsideRange(curr.rChild, min, max);  
  
    if (curr.value < min) {  
        return curr.rChild;  
    }  
  
    if (curr.value > max) {  
        return curr.lChild;  
    }  
  
    return curr;  
}
```

Print Tree nodes which are in Range

Problem: Print only those nodes of the tree whose value is in the given range.

Solution: Just normal inorder traversal and at the time of printing we will check if the value is inside the given range.

Example 9.48:

```
public void printInRange(int min, int max) {  
    printInRange(root, min, max);  
}  
  
private void printInRange(Node root, int min, int max) {  
    if (root == null)  
        return;  
  
    printInRange(root.lChild, min, max);  
    if (root.value >= min && root.value <= max)  
        System.out.print(root.value + " ");  
    printInRange(root.rChild, min, max);  
}
```

Find Ceil and Floor value inside BST given key

Problem: In given tree and a value, we need to find the floor value in tree which is smaller than the given value and need to find the ceil value in tree which is bigger. For a given value our aim is to find ceil and floor value as close as possible.

Solution: Just use search in BST to find ceil and floor of a value in a BST. When we are searching for ceil, if we find any value greater than the given input we save this value as probable value. We narrow down our search for a value much closer to our input value. This algorithm taken $O(\log n)$ time if BST is balanced. Similarly, we can find floor too.

Example 9.49:

```
public int CeilBST(int val) {
```

```
Node curr = root;
int ceil = Integer.MIN_VALUE;

while (curr != null) {
    if (curr.value == val) {
        ceil = curr.value;
        break;
    } else if (curr.value > val) {
        ceil = curr.value;
        curr = curr.lChild;
    } else {
        curr = curr.rChild;
    }
}
return ceil;
```

Example 9.50:

```
public int FloorBST(int val) {
    Node curr = root;
    int floor = Integer.MAX_VALUE;

    while (curr != null) {
        if (curr.value == val) {
            floor = curr.value;
            break;
        } else if (curr.value > val) {
            curr = curr.lChild;
        } else {
            floor = curr.value;
            curr = curr.rChild;
        }
    }
    return floor;
}
```

Smaller element in right side

Problem: Given an array. Construct an output array which the number of elements on the right side which are smaller than elements of given array.

First solution: Brute force, use two loop for each value loop through all the elements in the right side of it and count the values which are smaller than it. Time complexity is $O(n^2)$

Second solution: Another efficient solution is to use balanced BST. Traverse from right to left put values in tree. The nodes of the tree will keep track the number of elements in its left child or nodes which have value less than it. So finding nodes which have value less than it is $O(\log n)$ this step will be repeated for all the nodes. Time complexity is $O(n \log n)$

Is a BST Array

Problem: Given an array of integers, you need to find if it represents a PreOrder traversal of binary search tree.

Solution: In PreOrder traversal <Root><Left child><Right child>. Once we have a value greater than root value then we are in Right child subtree. So all the nodes that will come will have value greater than root value.

Example 9.51:

```
boolean isBSTArray(int preorder[], int size)
{
    Stack<Integer> stk = new Stack<Integer>();
    int value;
    int root = -999999;
    for (int i = 0; i < size; i++)
    {
        value = preorder[i];
        // If value of the right child is less than root.
        if (value < root)
            return false;
    }
}
```

```
// First left child values will be popped
// Last popped value will be the root.
while (stk.size() > 0 && stk.peek() < value)
    root = stk.pop();
    // add current value to the stack.
    stk.push(value); }
return true;
}
```

Segment Tree

Segment tree is a binary tree that is used to make multiple range queries and range update in an array.

- Examples of problems for which Segment Tree can be used are:
1. Finding the sum of all the elements of an array in a given range of index
 2. Finding the maximum value of the array in a given range of index.
 3. Finding the minimum value of the array in a given range of index (also known as Range Minimum Query problem)

Properties of Segment Tree:

1. Segment tree is a binary tree.
2. Each node in a segment tree represents an interval in the array.
3. The root of tree represents the whole array.
4. Each leaf node represents a single element.

Note: Segment tree solves problems, which can be solved in linear time by just scanning and updating the elements of array. The only benefit we are getting from segment tree is that it does update and query operation in logarithmic time that is more efficient than the linear solution.

Let us consider a simple problem:

Given an array of N numbers. You need to perform the following operations:

1. Update any element in the array
2. Find the maximum in any given range (i, j)

First Solution:

Updating: Just update the element in the array, $a[i] = x$. Finding maximum in the range (i, j), by traversing through the elements of the array in that range.

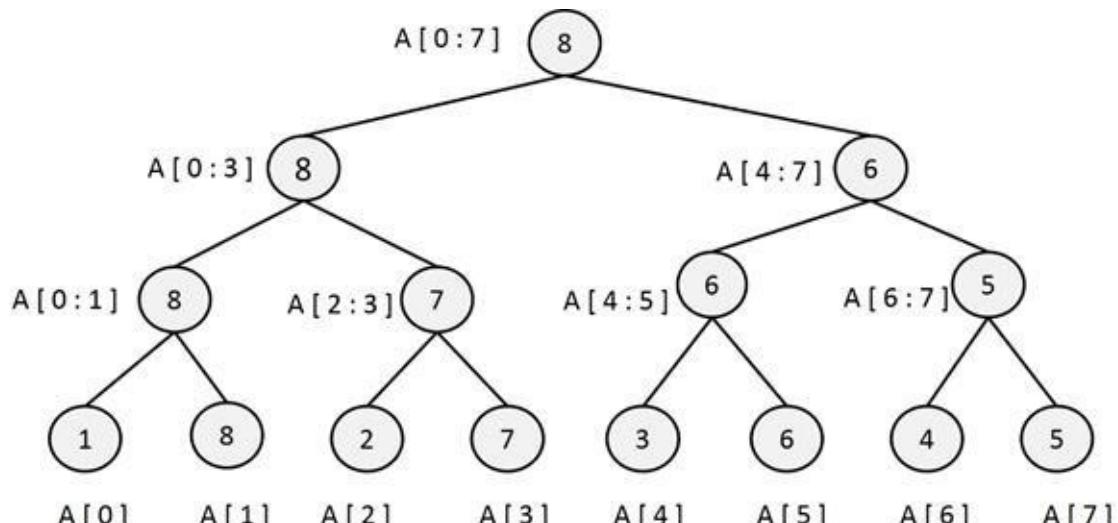
Time Complexity of Update is $O(1)$ and of Finding is $O(n)$

Second Solution: The above solution is good. However, can we improve performance of Finding?

The answer is yes. In fact, we can do both the operations in $O(\log n)$ where n is the size of the array. This we can do using a segment tree.

Let us suppose we are given an input array $A = \{1, 8, 2, 7, 3, 6, 4, 5\}$. Moreover,

the below diagram will represent the segment tree formed corresponding to the input array A.



Input Array: A = {1, 8, 2, 7, 3, 6, 4, 5}

AVL Trees

An AVL tree is a binary search tree (BST) with an additional property that the subtrees of every node differ in height by at most one. An AVL tree is a height balanced BST.

AVL tree is a balanced binary search tree. Adding or removing a node from AVL tree may make the AVL tree unbalanced. Such violations of AVL balance property is corrected by one or two simple steps called rotations. Let us assume that insertion of a new node converted a previously balanced AVL tree into an unbalanced tree. Since the tree is previously balanced and a single new node is added to it, the unbalance maximum difference in height will be 2.

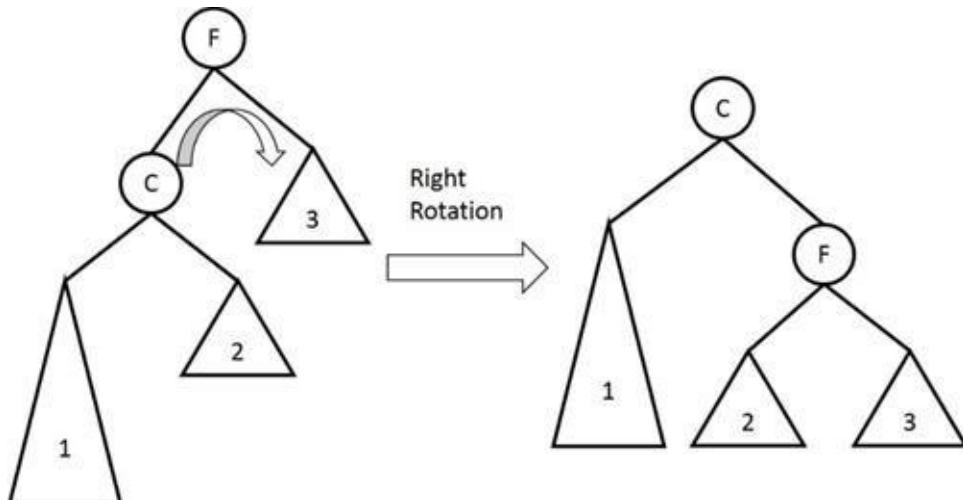
Therefore, in the bottom most unbalanced node there are only four cases:

Case 1: The new node is left child of the left child of the current node.

Case 2: The new node is right child of the left child of the current node.

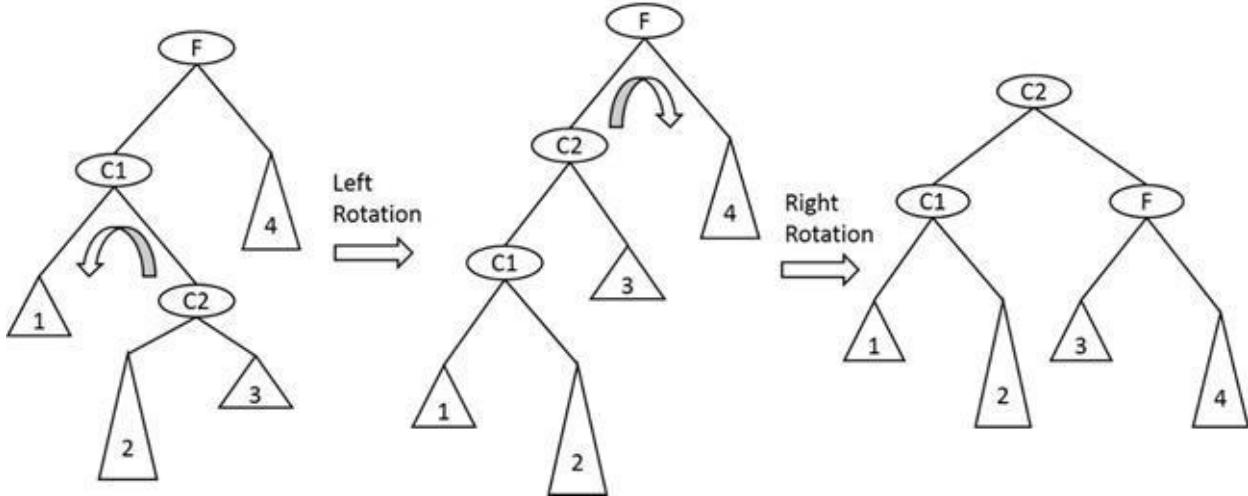
Case 3: The new node is left child of the right child of the current node.

Case 4: The new node is right child of the right child of the current node.



Case 1 can be re-balanced using a single Right Rotation.

Case 4 is symmetrical to Case 1: can be re-balanced using a single Left Rotation



Case 2 can be re-balanced using a double rotation. First, rotate left than rotation right.

Case 3 is symmetrical to Case 2: can be re-balanced using a double rotation. First, rotate right than rotation left.

Time Complexity of Insertion: To search the location where a new node needs to be added is done in $O(\log(n))$. Then on the way back, we look for the AVL balanced property and fixes them with rotation. Since the rotation at each node is done in constant time, the total amount of word is proportional to the length of the path. Therefore, the final time complexity of insertion is $O(\log(n))$.

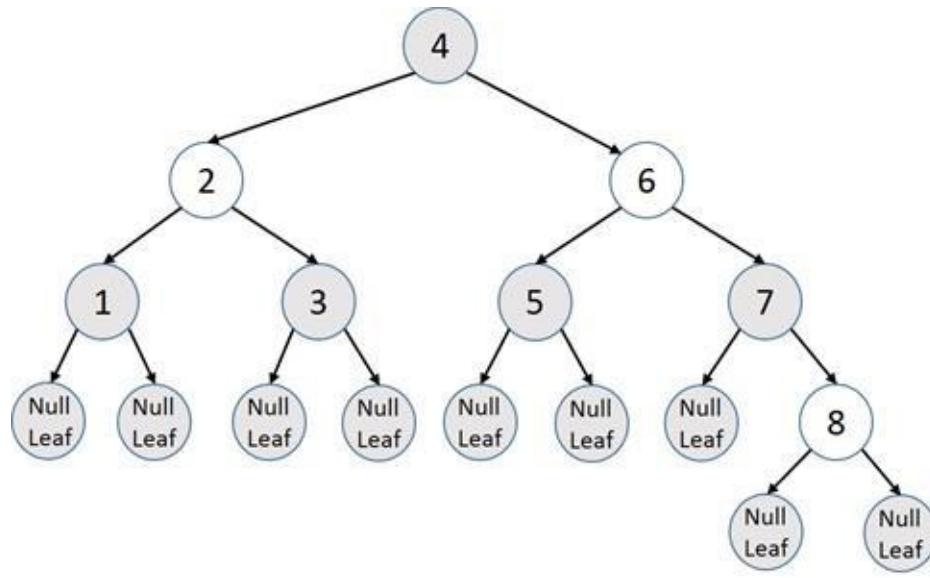
Red-Black Tree

The red-black tree contains its data, left and right children like any other binary tree. In addition to this its node also contains an extra bit of information which represents colour which can either red or black. Red-Black tree also contains a specialized type of nodes called null nodes. null nodes are pseudo nodes that exists at the leaf of the tree. All internal nodes have their own data associated with them.

Red-Black tree has the following properties:

1. Root of tree is black.
2. Every leaf node (null node) is black.
3. If a node is red then both of its children are black.
4. Every path from a node to a descendant leaf contains the same number of black nodes.

The first three properties are self-explanatory. The forth property states that, from any node in the tree to any leaf (null), the number of black nodes must be the same.



In the above figure, from the root node to the leaf node (null) the number of black node is always three nodes.

Like the AVL tree, red-black trees are also self-balancing binary search tree.

Whereas the balance property of an AVL tree has a direct relationship between the heights of left and right subtrees of each node. In red-black trees, the balancing property is governed by the four rules mentioned above. Adding or removing a node from red-black tree may violate the properties of a red-black tree. The red-black properties are restored through recolouring and rotation. Insert, delete, and search operation time complexity is $O(\log(n))$

Splay tree

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $O(\log n)$ amortized time. Elements of the tree are rearranged so that the recently accessed element is placed at the top of the tree. When an element is searched then we use standard BST search and then use rotation to bring the element to the top.

	Average Case	Worst Case
Space complexity	$O(n)$	$O(n)$
Time complexity search	$O(\log(n))$	Amortized $O(\log(n))$
Time complexity insert	$O(\log(n))$	Amortized $O(\log(n))$
Time complexity delete	$O(\log(n))$	Amortized $O(\log(n))$

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap.

B-Tree

As we had already seen various types of binary tree for searching, insertion and deletion of data in the main memory. However, these data structures are not appropriate for huge data that cannot fit into main memory, the data that is stored in the disk.

A B-tree is a self-balancing search tree that allows searches, insertions, and deletions in logarithmic time. The B-tree is a tree in which a node can have multiple children. Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write entire blocks (page) of data. The read - write operation from disk is very slow as compared with the main memory. The main purpose of B-Tree is to reduce the number of disk access. The node in a B-Tree has a huge number of pointers to the children nodes. Thereby reducing the size of the tree. While accessing data from disk, it makes sense to read an entire block of data and store into a node of tree. B-Tree nodes are designed such that entire block of data (page) fits into it. It is commonly used in databases and file systems.

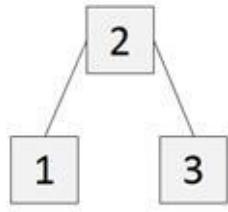
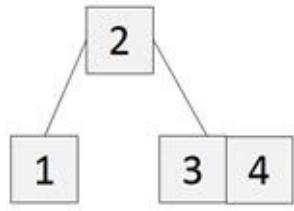
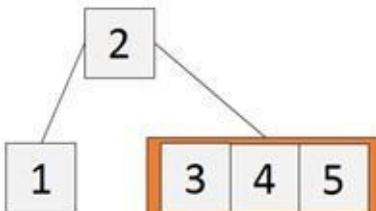
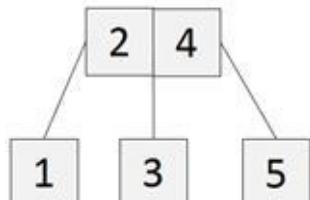
B-Tree of minimum degree d has the following properties:

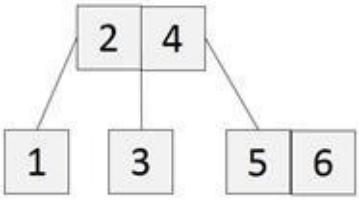
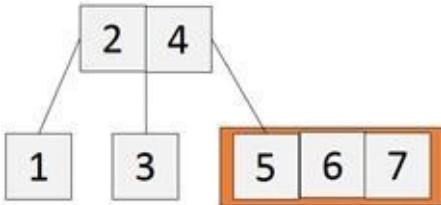
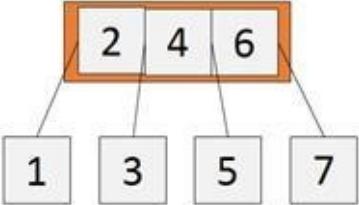
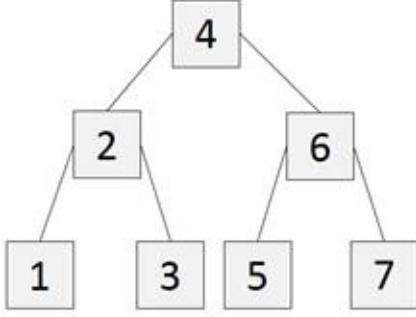
1. All the leaf nodes must be at same level.
2. All nodes except root must have at least $(d-1)$ keys and maximum of $(2d-1)$ keys. Root may contains minimum 1 key.
3. If the root node is a non-leaf node, then it must have at least 2 children.
4. A non-leaf node with N keys must have $(N+1)$ number of children.
5. All the key values within a node must be in Ascending Order.
6. All keys of a node are sorted in ascending order. The child between two keys, K1 and K2 contains all keys in range from K1 and K2.

B-Tree	Average Case	Worst Case
Space complexity	$O(n)$	$O(n)$
Time complexity search	$O(\log(n))$	$O(\log(n))$
Time complexity insert	$O(\log(n))$	$O(\log(n))$
Time complexity delete	$O(\log(n))$	$O(\log(n))$

Below is the steps of creation of B-Tree by adding value from 1 to 7.

1	Insert 1 to the tree.	Stable
---	-----------------------	--------

			
2		Insert 2 to the tree.	Stable
3		Insert 3 to the tree.	Intermediate
4		New node is created and data is distributed.	Stable
5		Insert 4 to the tree.	Stable
6		Insert 5 to the tree.	Intermediate
7		New node is created and data is distributed.	Stable

8		Insert 6 to the tree.	Stable
9		Insert 7 to the tree. New node is created and data is distributed.	Intermediate
10		After rearranging the intermediate node, still another intermediate node have more keys than maximum number of allowed keys.	Intermediate
11		New node is created and data is distributed. The height of the tree is increased.	Stable

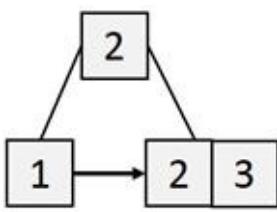
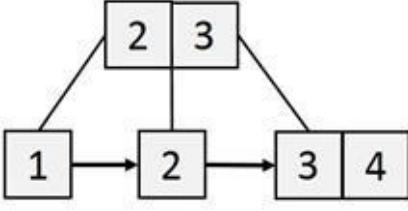
Note: 2-3 tree is a B-tree of degree three.

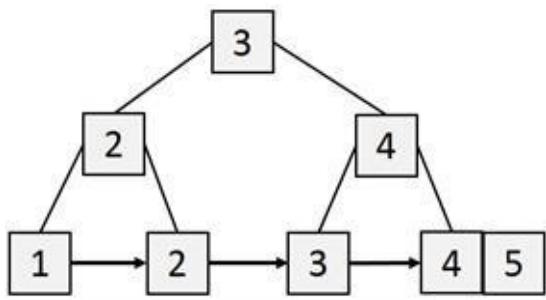
B+ Tree

B+ Tree is a variant of B-Tree. The B+ Tree stores records only at the leaf nodes. The internal nodes store keys. These keys are used for insertion, deletion and search. The rules of splitting and merging of nodes are same as B-Tree.

<i>b</i> -order B+ tree	Average Case	Worst Case
Space complexity	$O(n)$	$O(n)$
Time complexity search	$O(\log_b(n))$	$O(\log_b(n))$
Time complexity insert	$O(\log_b(n))$	$O(\log_b(n))$
Time complexity delete	$O(\log_b(n))$	$O(\log_b(n))$

Below is the B+ Tree created by adding value from 1 to 5.

1.		Value 1 is inserted to leaf node.
2.		Value 2 is inserted to leaf node.
3.		Value 3 is inserted to leaf node. Content of the leaf node passed the maximum number of elements. Therefore, node is split and intermediate / key node is created.
4.		Value 4 is further inserted to the leaf node. Which further splits the leaf node.
5.		Value 5 is added to the leaf node the number of nodes in the leaf passed the maximum number of nodes limit that it could contain so it is divided into 2. One more key is added to the intermediate node, which also make it passed maximum number of nodes it



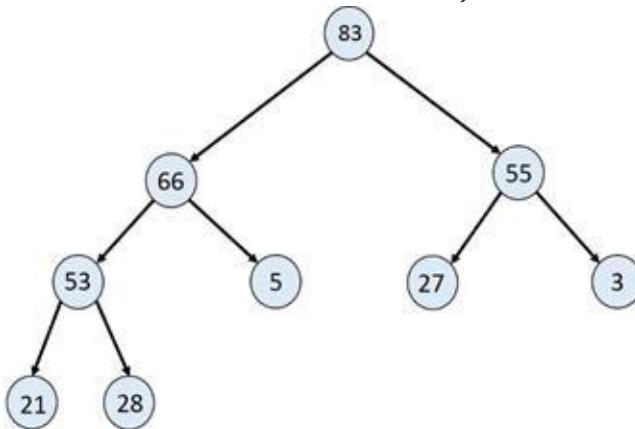
can contain, and finally divided and a new node is created.

B* Tree

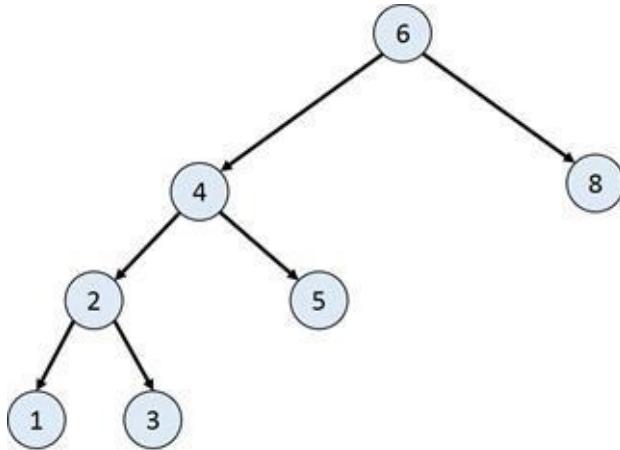
The B* tree is identical to the B+ tree, except for the rules for splitting and merging of nodes. Instead of splitting a node into two halves when it overflows, the B* tree node tries to give some of its records to its neighbouring sibling. If the sibling is also full, then a new node is created and records are distributed into three.

Exercise

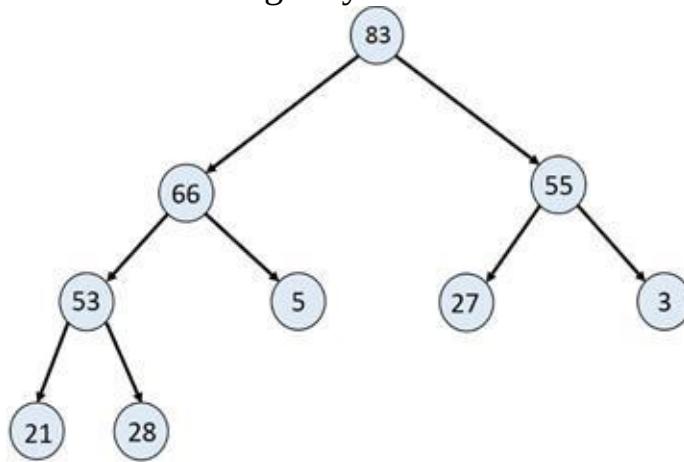
1. Construct a tree given its in-order and pre-order traversal strings.
 - o inorder: 1 2 3 4 5 6 7 8 9 10
 - o pre-order: 6 4 2 1 3 5 8 7 9 10
2. Construct a tree given its in-order and post-order traversal strings.
 - o inorder: 1 2 3 4 5 6 7 8 9 10
 - o post-order: 1 3 2 5 4 7 10 9 8 6
3. Write a delete node function in Binary tree.
4. Write a function print depth first in a binary tree without using system stack
Hint: you may want to keep another element to tree node like visited flag.
5. Check whether a given Binary Tree is Complete or not
 - o In a complete binary tree, every level except the last one is completely filled. All nodes in the left are filled first, then the right one.



6. Check whether a given Binary Tree is Full/ Strictly binary tree or not. The full binary tree is a binary tree in which each node has zero or two children.



7. Check whether a given Binary Tree is a Perfect binary tree or not. The perfect binary tree- is a type of full binary trees in which each non-leaf node has exactly two child nodes.
8. Check whether a given Binary Tree is Height-balanced Binary Tree or not. A height-balanced binary tree is a binary tree such that the left & right subtrees for any given node differs in height by not more than one



9. Isomorphic: two trees are isomorphic if they have the same shape, it does not matter what the value is. Write a program to find if two given tree are isomorphic or not.
10. The worst-case runtime Complexity of building a BST with n nodes
- o $O(n^2)$
 - o $O(n * \log n)$
 - o $O(n)$
 - o $O(\log n)$

11. The worst-case runtime Complexity of insertion into a BST with n nodes is
- O(n^2)
 - O($n * \log n$)
 - O(n)
 - O(log n)
12. The worst-case runtime Complexity of a search of a value in a BST with n nodes is:
- O(n^2)
 - O($n * \log n$)
 - O(n)
 - O(log n)
13. Which of the following traversals always gives the sorted sequence of the elements in a BST?
- Preorder
 - Ignored
 - Postorder
 - Undefined
14. The height of a Binary Search Tree with n nodes in the worst case?
- O($n * \log n$)
 - O(n)
 - O(log n)
 - O(1)
15. Try to optimize the above solution to give a DFS traversal without using recursion use some stack or queue.
16. This is an open exercise for the readers. Every algorithm that is solved using recursion (system stack) can also be solved using user defined or library defined stack. So try to figure out what all algorithms that uses recursion and try to figure out how you will do this same using user defined stack.
17. In a binary tree, print the nodes in zigzag order. In the first level, nodes are printed in the left to right order. In the second level, nodes are printed in right to left and in the third level again in the order left to right.

Hint: Use two stacks. Pop from first stack and push into another stack. Swap the stacks alternatively.

18. Find nth smallest element in a binary search tree.

Hint: Nth inorder in a binary tree.

19. Find the floor value of key that is inside a BST.

20. Find the Ceil value of key, which is inside a BST.

21. What is Time Complexity of the below pseudo code:

Function **DFS**(head):

```
curr = head
count = 0;
while (curr != None && curr.visited == False):
    count++;
    if (curr.lChild != None && curr.lChild.visited == False):
        curr= curr.lChild
    else if (curr.rChild != None && curr.rChild.visited == False):
        curr= curr.rChild;
    else
        print curr.value
        curr.visited = 1;
        curr = head;

    print "count is : ", count
```

CHAPTER 10: PRIORITY QUEUE / HEAPS

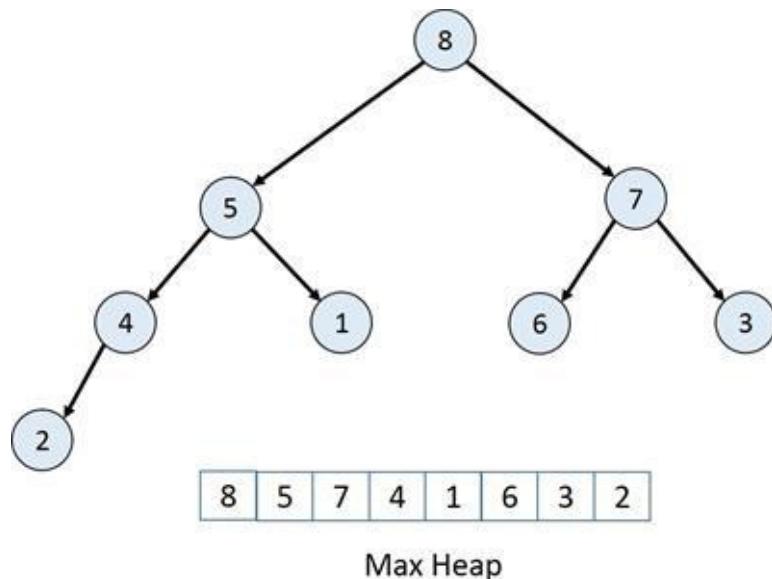
Introduction

A Priority-Queue, also known as Heap, is a variant of queue. Items are removed from the beginning of the queue. However, in a Priority-Queue the logical ordering of objects is determined by their priority. The highest priority item is at the front of the Priority-Queue. When you add an item to the Priority-Queue, the new item moves to its proper position according to its priority. A Priority-Queue is a very important data structure. Priority-Queue is used in various Graph algorithms like [Prim's Algorithm](#) and [Dijkstra's algorithm](#). Priority-Queue is also used in the timer implementation etc.

A Priority-Queue is implemented using Heap. A Heap data structure is an array of elements that can be observed as a complete binary tree.

A heap is a binary tree that satisfies the following properties:

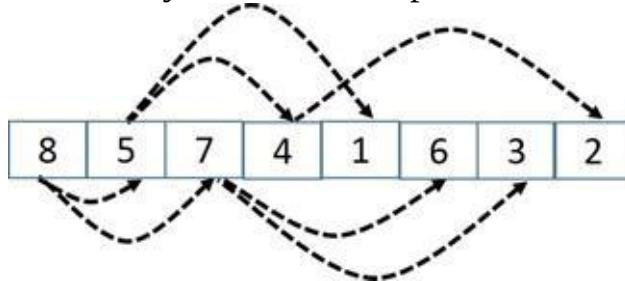
1. The tree is a complete binary tree. A heap is a complete binary tree so the height of tree with N nodes is always **O(logn)**.
2. Heap satisfies the heap ordering property. In max-heap, the parent's value is greater than or equal to its children's value. In min-heap, the parent's value is less than or equal to its children's value.



A heap is not a sorted data structure and can be regarded as partially ordered. As you can see in the picture, there is no relationship among nodes at any given

level, even among the siblings.

Heap is implemented using an array. Moreover, because heap is a complete binary tree, the left child of a parent (at position x) is the node that is found in position $(2x+1)$ in the array. Similarly, the right child of the parent is at position $(2x+2)$ in the array. To find the parent of any node in the heap, we can simply make division. In given index y of a node, the parent index will be $(y-1)/2$.

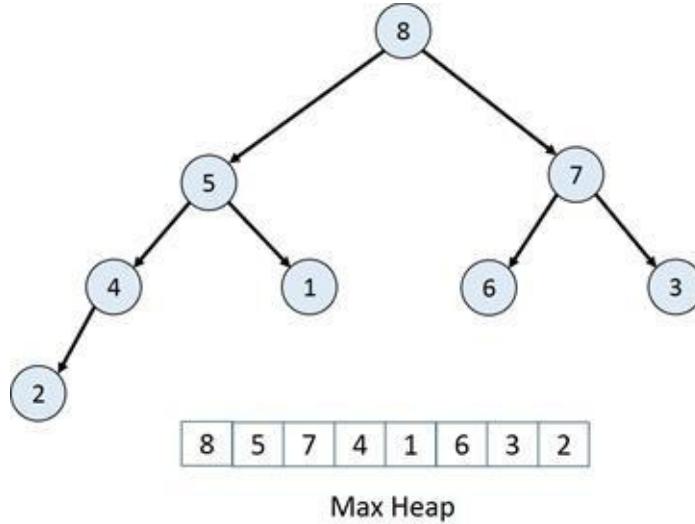


Types of Heap

There are two types of heap and the type depends on the ordering of the elements. The ordering can be done in two ways: Min-Heap and Max-Heap

Max Heap

Max-Heap: the value of each node is less than or equal to the value of its parent, with the largest-value element at the root.

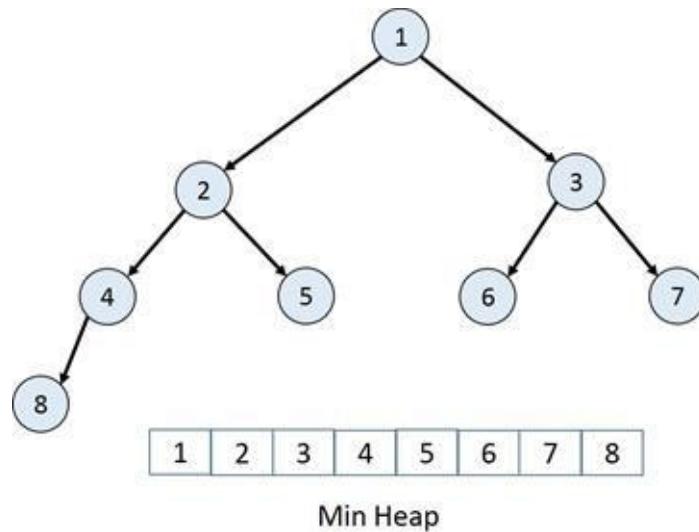


Max Heap Operations

Insert	$O(\log n)$
DeleteMax	$O(\log n)$
Remove	$O(\log n)$
FindMax	$O(1)$

Min Heap

Min-Heap: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.



Use it whenever you need quick access to the smallest item, because that item will always be at the root of the tree or the first element in the array. However, the remainder of the array is kept partially sorted. Thus, instant access is only possible for the smallest item.

Min Heap Operations

Insert	$O(\log n)$
DeleteMin	$O(\log n)$
Remove	$O(\log n)$
FindMin	$O(1)$

Heap ADT Operations

The basic operations of binary heap are as follows:

Binary Heap	Creates a new empty binary heap	O(1)
Insert	Adding a new element to the heap	O(logn)
DeleteMax	Deletes the maximum element form the heap.	O(logn)
FindMax	Finds the maximum element in the heap.	O(1)
isEmpty	Returns true if the heap is empty else return false	O(1)
Size	Returns the number of elements in the heap.	O(1)
BuildHeap	Builds a new heap from the array of elements	O(logn)

Operation on Heap

Before looking in detail of how to initialize heap, add or remove elements into it we need to understand two important operations involved in heap. These operations are used to restore heap order property when single element is out of its position.

There are two cases are:

1. When a node (parent node) is not following heap property with its children. This is resolved by proclateDown() operation in which parent node value is swapped with one of the child value and heap property is restored recursively.
2. When a node (child node) is not following heap property with its parent. This is resolved by proclateUp() or bubbleUp() operation in which child node value is swapped with parent node value and heap property is restored recursively.

Example 10.1: Below is the code of proclateUp and proclateDown operation.

```
private void proclateDown(int position) {  
    int lChild = 2 * position + 1;  
    int rChild = lChild + 1;  
    int child = -1;  
    int temp;  
  
    if (lChild < size) {  
        child = lChild;  
    }  
  
    if (rChild < size && compare(arr, rChild, lChild)) {  
        child = rChild;  
    }  
  
    if (child != -1 && compare(arr, child, position)) {  
        temp = arr[position];  
        arr[position] = arr[child];  
        arr[child] = temp;  
        proclateDown(child);  
    }  
}
```

```
}
```

```
private void proclateUp(int position) {  
    int parent = (position - 1) / 2;  
    int temp;  
  
    if (parent == 0) {  
        return;  
    }  
  
    if (compare(arr, parent, position)) {  
        temp = arr[position];  
        arr[position] = arr[parent];  
        arr[parent] = temp;  
        proclateUp(parent); }  
}
```

```
boolean compare(int[] arr, int first, int second) {  
    if (isMinHeap)  
        return (arr[first] < arr[second]); // Min heap compare  
    return (arr[first] > arr[second]); // Max heap compare  
}
```

Heap is represented using a class. The following elements of heap class are:

- “array” that is used to store heap.
- “capacity”, which is capacity of array
- “size”, which is the number of elements in the heap.
- “isMinHeap” will be 1 for min heap and will be 0 for max heap

Example 10.2:

```
public class Heap {  
    private static final int CAPACITY = 32;  
    private int size; // Number of elements in heap  
    private int[] arr; // The heap array  
    boolean isMinHeap;  
  
    // Methods.  
    public boolean isEmpty() {
```

```

return (size == 0);
}

public int size() {
    return size;
}

public int peek() {
    if (isEmpty()) {
        throw new IllegalStateException();
    }
    return arr[0];
}

```

`isEmpty()` function return true if heap is empty else return false.

`size()` function return the number of elements in the heap.

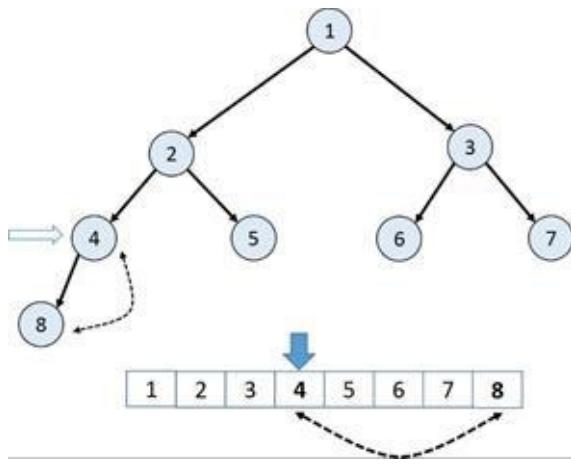
`peek()` function will return highest priority element form the heap without deleting it.

Create Heap from an array

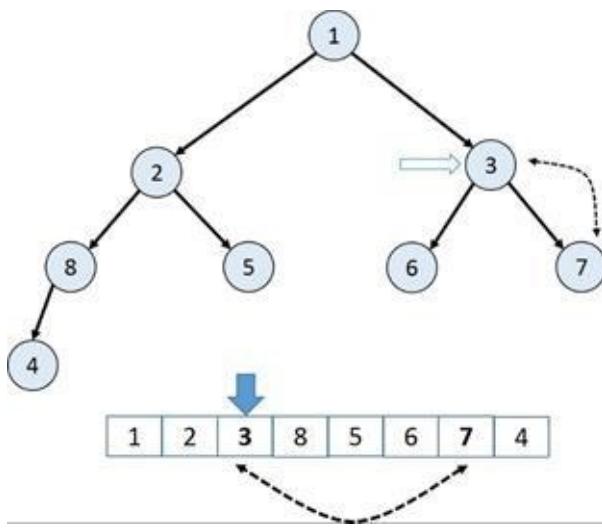
Heapify is the process of converting an array into Heap. The various steps are:

1. Values are present in the array.
2. Starting from middle of the array move downward towards the start of the array. At each step, compare parent value with its left child and right child. In addition, restore the heap property by shifting the parent value with its largest-value child. Such that the parent value will always be greater than or equal to left child and right child.
3. For all elements from middle of the array to the start of the array. We make comparisons and shift, until we reach the leaf nodes of the heap.

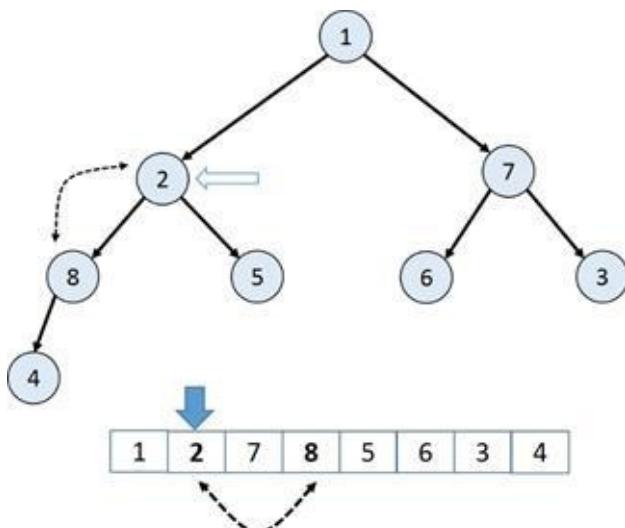
	<p>Given an array as input to create heap function. Value of index i is compared with value of its children nodes that is at index $(i*2 + 1)$ and $(i*2 + 2)$. Middle of list $N/2$,</p>
--	---



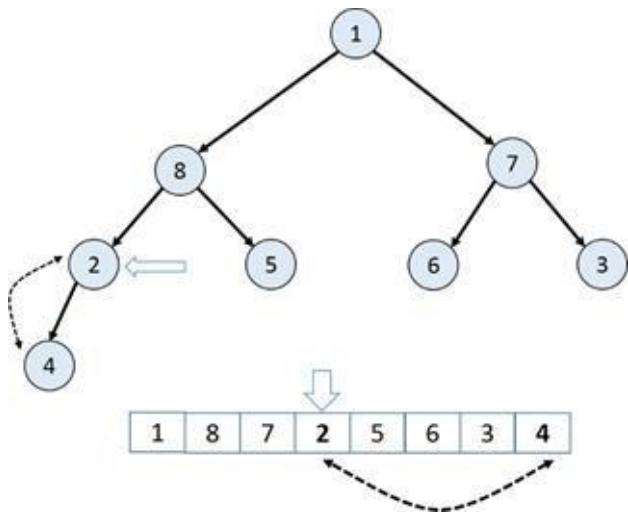
that is index 3, is compared with index 7. If the children node value is greater than parent node then the value will be swapped.



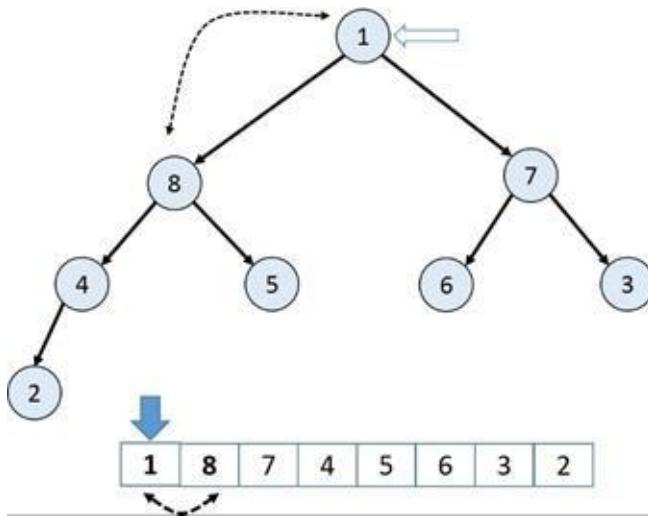
Similarly, value of index 2 is compared with index 5 and 6. The largest of all the values is 7 will be swapped with the value at the index 2.



Similarly, value of index 1 is compared with index 3 and 4. The largest of all the values is 8 which will be swapped with the value at the index 1.

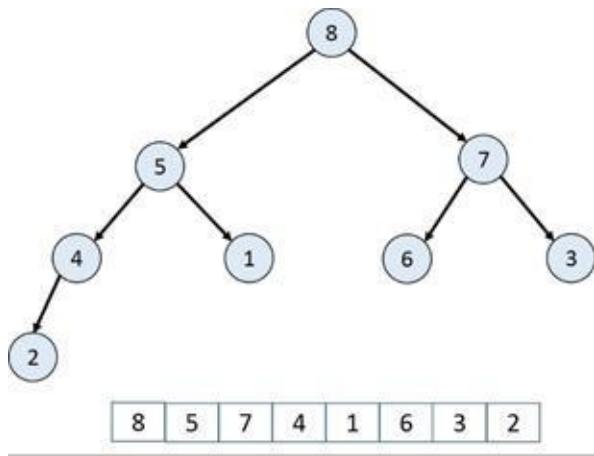
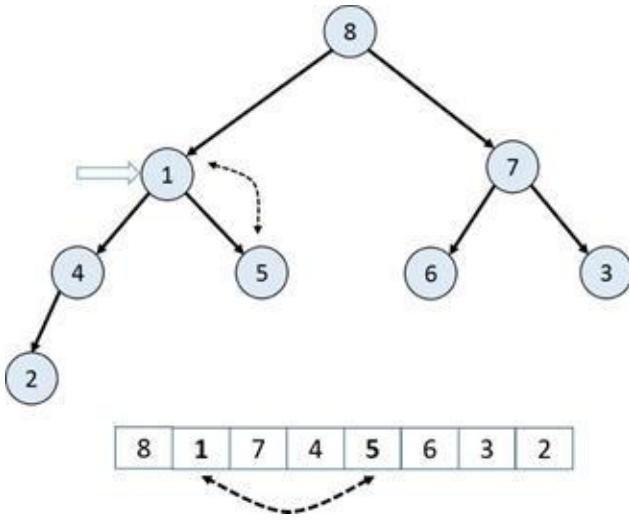


Percolate down function is used to subsequently adjust the value replaced in the previous step by comparing it with its children nodes.



Now value at index 0 is compared with index 1 and 2. 8 is the largest value so it is swapped with the value at index 0.

Percolate down function is used to further compare the value at index 1 with its children nodes at index 3 and 4.



In the end max heap is created.

Example 10.3: Initializing heap from array

```
public Heap(int[] array, boolean isMin) {
    size = array.length;
    arr = array;
    isMinHeap = isMin;

    // Build Heap operation over array
    for (int i = (size / 2); i >= 0; i--) {
        proclateDown(i);
    }
}
```

Analysis: Heap can be initialized by passing an array to it. Heapify() function is

called over the array to make it Heap.

Time Complexity of build heap is **O(N)**.

The number of comparison is related to the height of the node. Total number of comparison will be represented as: $(1*n/4) + (2*n/8) + (3*n/16) \dots ((h-1)*1) == N$

Where “h” is height of the heap.

Initializing an empty Heap

Example 10.4:

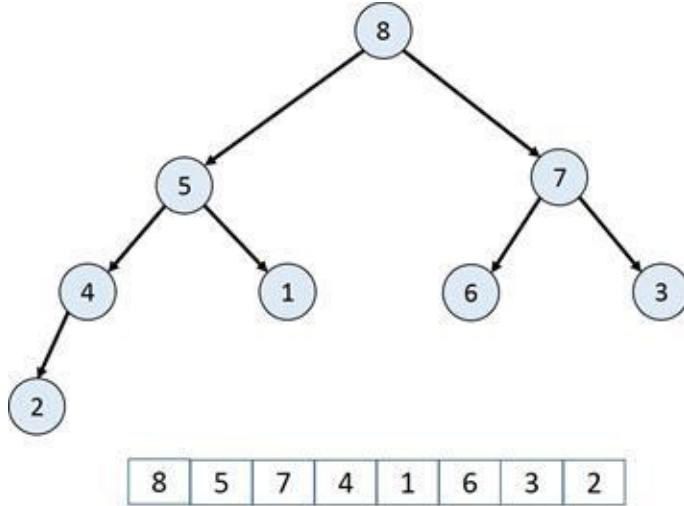
```
public Heap(boolean isMin) {  
    arr = new int[CAPACITY];  
    size = 0;  
    isMinHeap = isMin;  
}
```

Analysis: Empty heap is created by creating an empty array. Since the heap is empty we do not need any heapify operations.

Enqueue / add

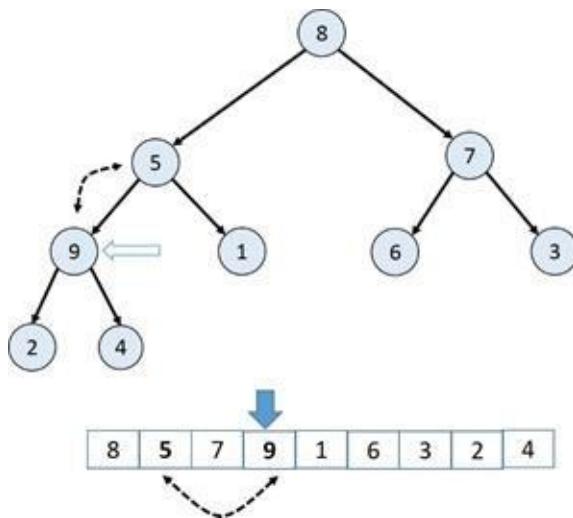
1. Add the new element at the end of the array. This keeps the values as a complete binary tree, but it might no longer be a heap since the new element might have a value greater than its parent's value.
2. Swap the new element with its parent until it has value greater than its parent's value.
3. Step 2 will be terminated when the new element reaches the root or when the new element's parent has a value greater than or equal to the new element's value.

Let us take an example of the Max heap created in the above example.

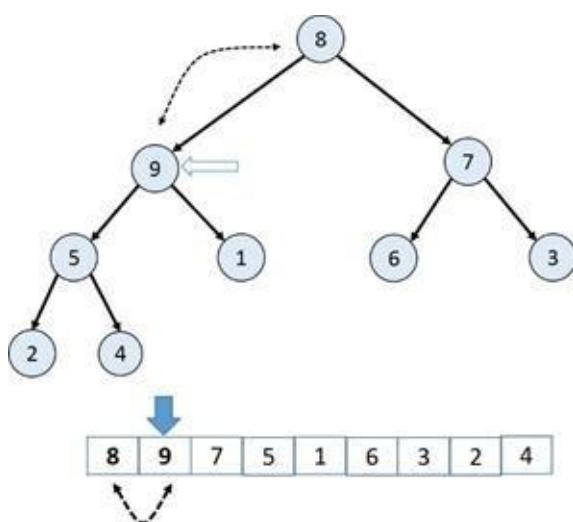


Let us take an example by inserting element with value 9 to the heap. The element is added to the end of the Heap array. Now the value will be percolated up by comparing it with the parent. The value is added to index 8 and its parent will be $(N-1)/2 =$ index 3.

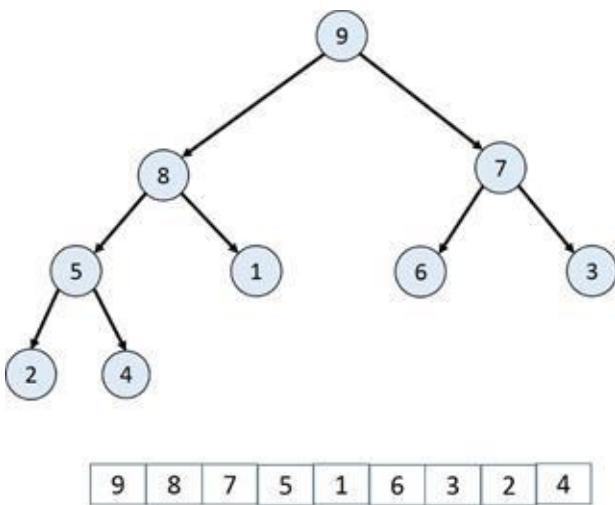
<p>New bvalue 9 added to the end of array. Since the value 9 is greater than 4 it will be swapped with it.</p>	
	<p>Percolate up is used and the value is moved up until heap property is satisfied.</p>



Now the value at index 1 is compared with index 0 and to satisfy heap property it is further swapped.



Now, finally max heap is created by inserting new node.



Example 10.5:

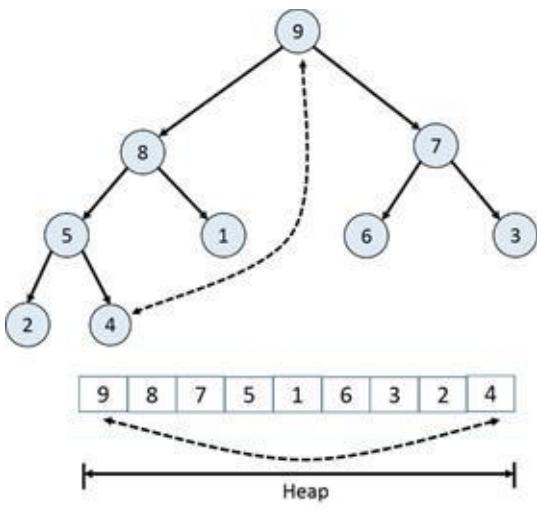
```
public void add(int value) {  
    if (size == arr.length) {  
        doubleSize();  
    }  
  
    arr[size++] = value;  
    proclateUp(size - 1);  
}  
  
private void doubleSize() {  
    int[] old = arr;  
    arr = new int[arr.length * 2];  
    System.arraycopy(old, 0, arr, 0, size);  
}
```

Analysis: We check if the heap is full. In case when heap is already full then we create another array of twice capacity. Copy the content of heap into it. Then add the new element to the heap and then use proclateUp() operation to restore heap property.

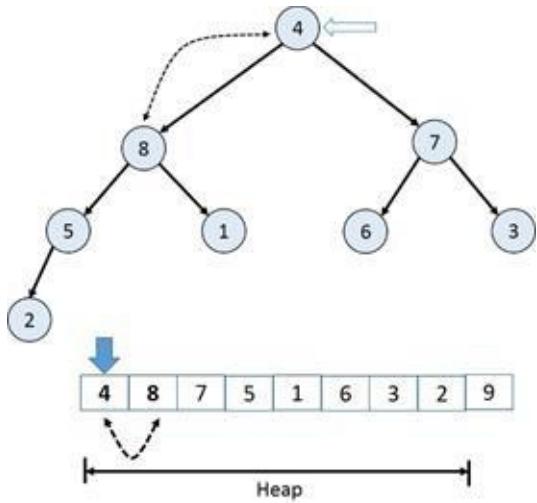
Time complexity of insertion in heap is $O(\log n)$

Dequeue / remove

1. Copy the value at the root of the heap to the variable that will be used to return that value.
2. Copy the last element of the heap to the root, and then reduce the size of heap by 1. This element is called the "out-of-place" element.
3. Restore heap property by swapping the out-of-place element with its greatest-value child. Repeat this process until the out-of-place element reaches a leaf or it has a value that is greater or equal to all its children.
4. Return the answer that was saved in Step 1.

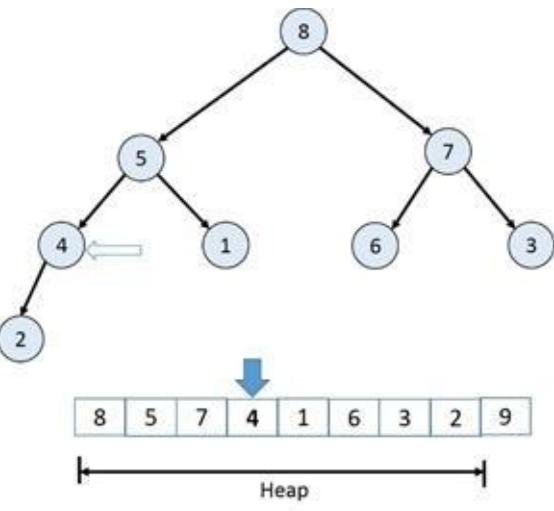
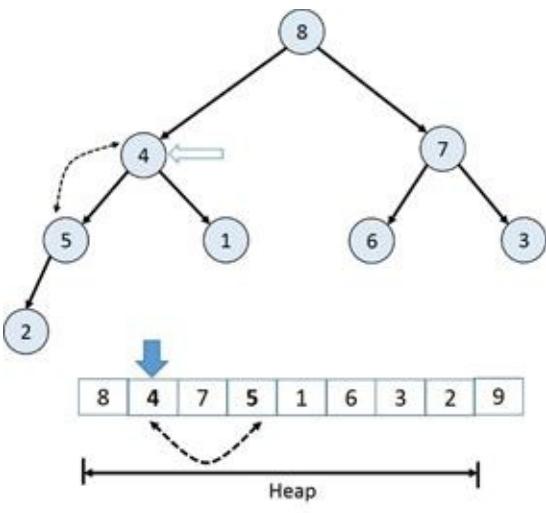


Value at start and end of the heap are swapped. Heap size is reduced by one. Heap property is disturbed so we need to percolate down by comparing node with its children nodes and restore heap property.



Percolate down is continued by comparing with its children nodes.

Percolate down



Percolate down Complete

Example 10.6:

```

public int remove() {
    if (isEmpty()) {
        throw new IllegalStateException();
    }

    int value = arr[0];
    arr[0] = arr[size - 1];
    size--;
    procladeDown(0);
    return value;
}
  
```

```
}
```

Analysis: Value at the top of heap is swapped with the value at the end of the heap array. Size of heap is reduced by one and proclateDown is called to restore heap property.

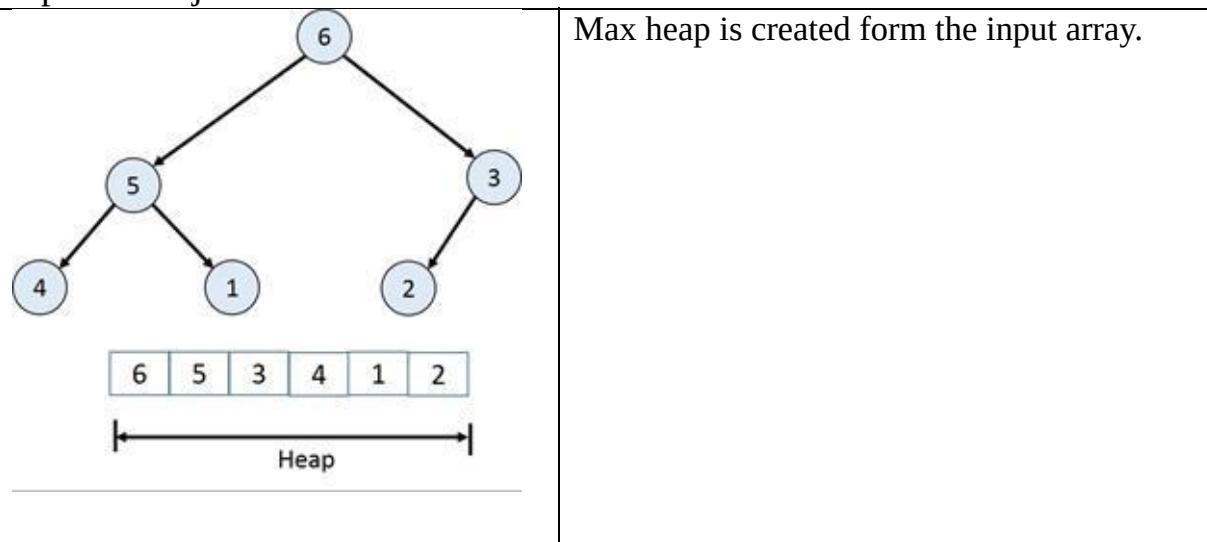
Time complexity of remove element of heap is $O(\log n)$

```
public static void main(String[] args) {  
    int[] a = { 1, 9, 6, 7, 8, 0, 2, 4, 5, 3 };  
    Heap hp = new Heap(a, false);  
    hp.print(); System.out.println();  
    while (!hp.isEmpty()) {  
        System.out.print(hp.remove() + " ");  
    }  
}
```

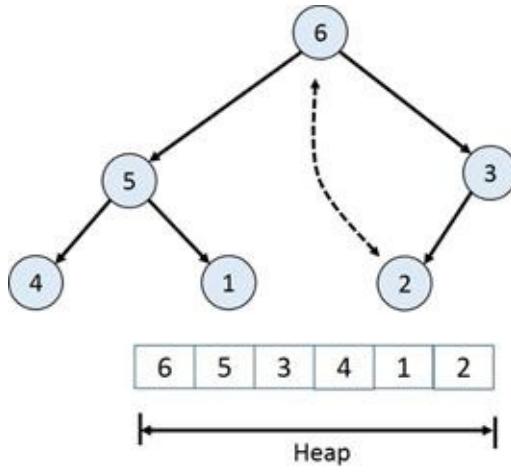
Heap-Sort

1. Use create heap function to build a max heap from the given list of elements.
This operation will take **O(N)** time.
2. Dequeue the max value from the heap and store this value to the end of the array at location arr[size-1]
 - a) Copy the value at the root of the heap to end of the array.
 - b) Copy the last element of the heap to the root, and then reduce the size of heap by 1. This element is called the "out-of-place" element.
 - c) Restore heap property by swapping the out-of-place element with its greatest-value child. Repeat this process until the out-of-place element reaches a leaf or it has a value that is greater or equal to all its children
3. Repeat this operation until there is just one element in the heap.

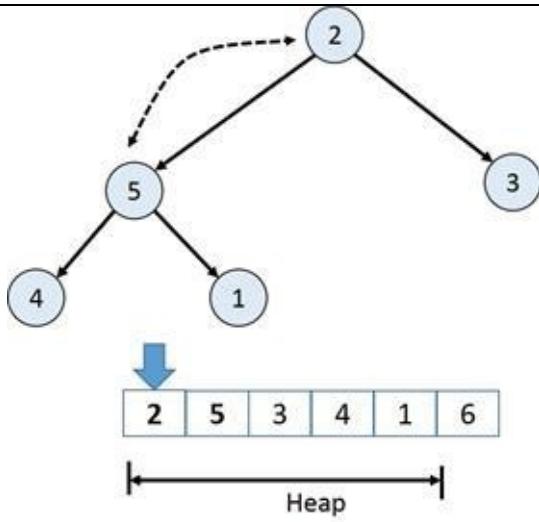
Let us take an example of the heap that we had already created at the start of the chapter. Heap sort is algorithm starts by creating a heap of the given list, which is done in linear time. Then at each step head of the heap is swapped with the end of the heap and the heap size is reduced by 1. Then percolate down is used to restore the heap property. Moreover, the same is done multiple times until the heap contain just one element.



The maximum value, which is the first element of the Heap

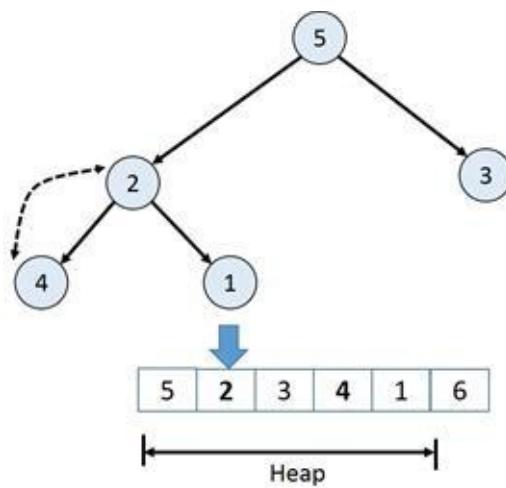


array, is swapped with the last element of the array. Now the largest value is at the end of the array. Then we will reduce the size of the heap by one.

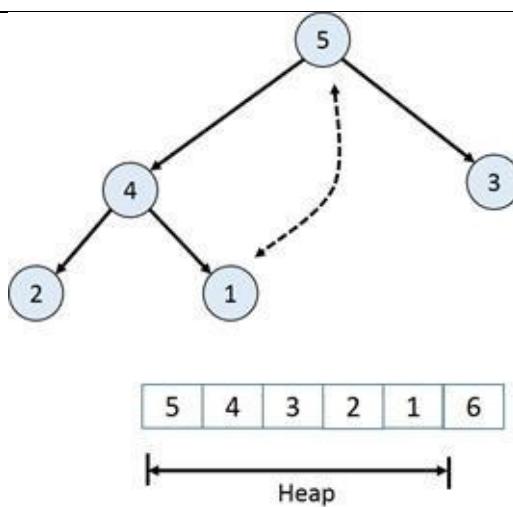
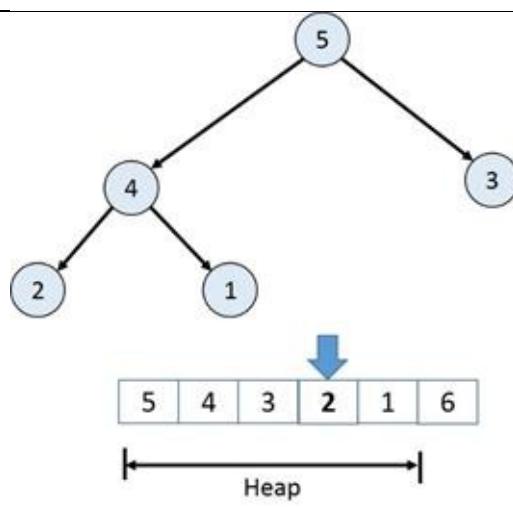


Since 2 is at the top of the heap. Its, heap property is lost. We will use Percolate down method to regain the heap property.

Percolate down continued.

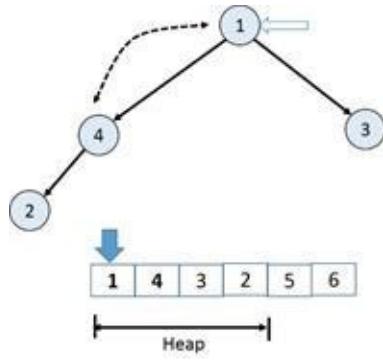


Heap property is regained.

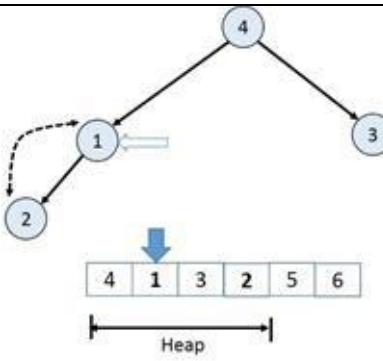


We will copy the first element of the heap array to the second last position.
Size of heap is further reduced by one.

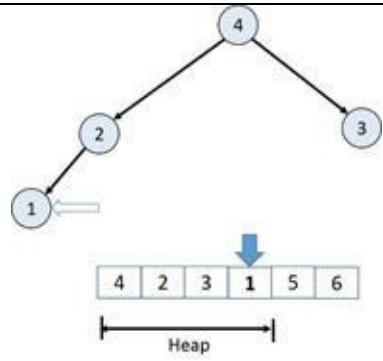
Heap



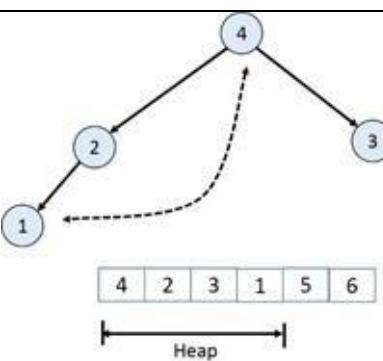
property
is lost.
Perform
percolate
down to
regain
heap
property.



Percolate
down
continued.

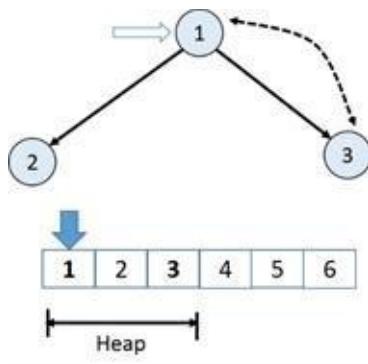


Heap
property
regained.

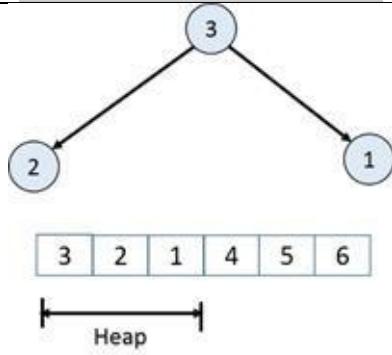


Swap first
and the
last value
in heap
array.
Reduce
size of
heap.

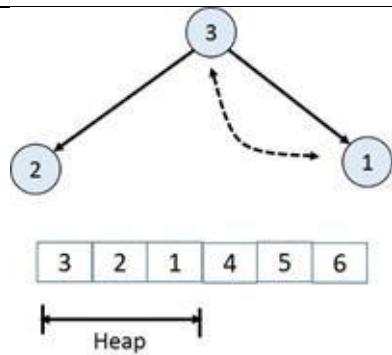
Percolate



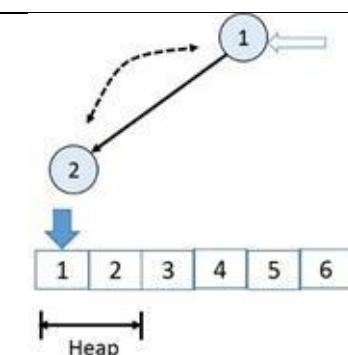
down.



Heap
property
regained.

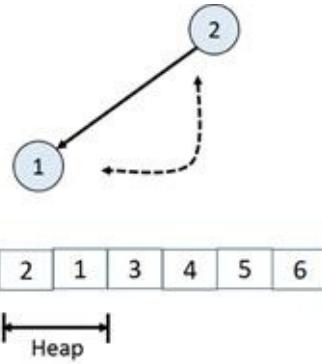


Swap first
and last
element of
heap
array.
Reduce
size of
heap by
one.



Heap
property
lost so
Percolate
down.

Swap first
and last
value



stored in
Heap
array.
Reduce
size of
heap by
one.



End

Example 10.7:

```
public static void heapSort(int[] array, boolean inc) {
    Heap hp = new Heap(array, !inc);
    for (int i = 0; i < array.length; i++) {
        array[array.length - i - 1] = hp.remove();
    }
}

public static void main(String[] args) {
    int[] a1 = { 1, 9, 6, 7, 8, 0, 2, 4, 5, 3 };
    Heap.heapSort(a1, true);
    for (int i = 0; i < a1.length; i++) {
        System.out.print(a1[i] + " ");
    }
}
```

Data structure	List
Worst Case time complexity	$O(n \log n)$
Best Case time complexity	$O(n \log n)$
Average case time complexity	$O(n \log n)$
Space Complexity	$O(1)$

Time complexity analysis: Build heap of the input array takes $O(n)$ time and each element removal and adding to the end of heap take $O(\log n)$. So the total time complexity will be $O(n) + O(n \log n)$. Final time complexity of this sorting is $O(n \cdot \log n)$

Note: Heap-Sort is not a Stable sort and does not require any extra space for

sorting an array.

Uses of Heap

1. **Heapsort:** One of the best sorting methods being in-place and $\log(N)$ time complexity in all scenarios.
2. **Selection algorithms:** Finding the min, max, both the min and max, median, or even the k th largest element can be done in linear time (often constant time) using heaps.
3. **Priority Queues:** Heap Implemented priority queues are used in Graph algorithms like [Prim's Algorithm](#) and [Dijkstra's algorithm](#). A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority. Schedulers, timers
4. **Graph algorithms:** By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal
5. Because of the lack of pointers, the operations are faster than a binary tree. In addition, some more complicated heaps (such as binomial) can be merged efficiently, which is not easy to do for a binary tree.

Problems in Heap

Kth Smallest using Min Heap

Problem: Given an unsorted array, find kth smallest value.

First Solution: Sort the given array and then give the value at index K-1. Time complexity is O(NlogN).

Example 10.8:

```
public static int KthSmallest(int[] arr, int size, int k) {  
    Arrays.sort(arr);  
    return arr[k - 1];  
}
```

Second solution: Create a min heap from the array, call DeleteMin() operation K times and the last operation will give Kth smallest value. Time Complexity O(KlogN)

Example 10.9:

```
public static int KthSmallest2(int[] arr, int size, int k) {  
    int i = 0;  
    int value = 0;  
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();  
    for (i = 0; i < size; i++) {  
        pq.add(arr[i]);  
    }  
    while (i < size && i < k) {  
        value = pq.remove();  
        i += 1;  
    }  
    return value;  
}
```

Kth Largest using Max Heap

Problem: Given an unsorted array, find kth largest element.

Solution: Same as above problem we will create a Max Heap from the input array. Then we call DeleteMax() operation K times and the last operation will give Kth largest value. Time Complexity $O(K \log N)$

Kth largest/ smallest in an infinite stream of number

Problem: Find kth largest value in an infinite stream of number.

First solution: Keep a sorted array, which will contain k largest elements. For each element of the stream, compare it with the smallest value in the array. If it is smaller than ignore it. If it is greater than the smallest value in the array, then remove the smallest value and insert the new value in its proper position in the array to keep it sorted. This solution will take $O(K)$ time for each compare and insert operation.

Second solution: We can keep the elements in Balanced Binary Search tree. First k elements are added to the tree. After this for each new element of the stream we first find the smallest element in the binary tree this operation will take $O(\log k)$ time then we compare the smallest value with the new value. If the new value is smaller than the smallest value then we ignore it. If the new value is greater than the smallest value then we delete the smallest value and insert the new value these operation also take $O(\log n)$ time. This solution will take $O(\log k)$ time for each compare and insert operation.

100 Largest in a Stream

Problem: There are billions of integers coming out of a stream some getInt() function is providing integers one by one. How would you determine the largest 100 numbers?

Solution: Large hundred (or smallest hundred etc.), such problems are solved very easily using a Heap. In this case, we will create a min heap.

1. First, from 100 integers build a min heap.
2. Then for each coming integer compare if it is greater than the top of the min heap.

3. If not, then look for next integer. If yes, then remove the top min value from the min heap, insert the new value at the top of the heap, use procolateDown, and move it to its proper position, down the heap.
4. Every time you have largest 100 values stored in your head

Merge two Heap

Problem: Given two heaps, you need to merge them to create a single heap.

Solution: There is no single solution for this. Let us suppose the size of the bigger heap is N and the size of the smaller heap is M.

1. If both heaps are comparable in size, then put both Heap arrays in same bigger Lists. Alternatively, in one of the arrays if they are big enough, then apply CreateHeap() function which will take theta($N+M$) time.
2. If M is much smaller than N then add() each element of M list one by one to N heap. This will take $O(M\log N)$ the worst case or $O(M)$ the best case.

Is Min Heap

Problem: In given list, find if it represent a binary Min Heap

Solution: Min Heap property we are going to check is that value at parent index is always less than or equal to its children index.

Example 10.10:

```
public static boolean isMinHeap(int[] arr, int size) {
    int lchild, rchild;
    // last element index size - 1
    for (int parent = 0; parent < (size / 2 + 1); parent++) {
        lchild = parent * 2 + 1;
        rchild = parent * 2 + 2;
        // heap property check.
        if (((lchild < size) && (arr[parent] > arr[lchild])) || ((rchild < size) &&
        (arr[parent] > arr[rchild])))
            return false;
    }
    return true;
}
```

Is Max Heap

Problem: In given list, find if it represent a binary Max Heap

Solution: Max Heap property we are going to check is that the value at parent index is always greater than or equal to its children index.

Example 10.11:

```
public static boolean isMaxHeap(int[] arr, int size) {  
    int lchild, rchild;  
    // last element index size - 1  
    for (int parent = 0; parent < (size / 2 + 1); parent++) {  
        lchild = parent * 2 + 1;  
        rchild = lchild + 1;  
        // heap property check.  
        if (((lchild < size) && (arr[parent] < arr[lchild])) || ((rchild < size) &&  
        (arr[parent] < arr[rchild])))  
            return false;  
    }  
    return true;  
}
```

Analysis: If each parent value is greater than its children value then heap property is true. We will traverse from start to half of the array and compare the value of index node with its left child and right child node.

Traversal in Heap

Heaps are not designed to traverse, to find some element. They are made to get min or max element quickly. Still if you want to traverse a heap just traverse the array sequentially. This traversal will be level order traversal. This traversal will have linear Time Complexity.

Deleting Arbiter element from Min Heap

Again, heap is not designed to delete an arbitrary element, still if you want to do

so. Find the element by linear search in the Heap array. Replace it with the value stored at the end of the Heap value. Reduce the size of the heap by one. Compare the new inserted value with its parent. If its value is smaller than the parent value, then percolate up. Else if its value is greater than its left and right child then percolate down. Time Complexity is **O(logn)**

Deleting Kth element from Min Heap

Again, heap is not designed to delete an arbitrary element, still if you want to do so. Replace the kth value with the value stored at the end of the Heap value. Reduce the size of the heap by one. Compare the new inserted value with its parent. If its value is smaller than the parent value, then percolate up. Else if its value is greater than its left and right child then percolate down. Time Complexity is **O(logn)**

Print value in Range in Min Heap

Linearly traverse through the heap and print the value that are in the given range.

Product of K minimum elements

Problem: Given an array of positive elements. Find product of k minimum elements in array.

First solution: Sorting, Sort the array and find product of first K elements. Sorting will take $O(n.logn)$ time. And then finding product will take $O(n)$ time so total time complexity is $O(n.logn)$.

Example 10.12:

```
public static int KSmallestProduct(int[] arr, int size, int k) {  
    Arrays.sort(arr); // , size, 1);  
    int product = 1;  
    for (int i = 0; i < k; i++)  
        product *= arr[i];  
    return product;  
}
```

Second solution: Min Heap, Create min heap from the array. Then pop K elements from heap and find their product. Heapify will take $O(n)$ time. Pop elements from heap take $O(\log n)$ time. K elements are popped from heap. So total time complexity is $O(k \cdot \log n)$

Example 10.13:

```
public static int KSmallestProduct2(int[] arr, int size, int k) {  
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();  
    int i = 0;  
    int product = 1;  
    for (i = 0; i < size; i++) {  
        pq.add(arr[i]);  
    }  
  
    while (i < size && i < k) {  
        product *= pq.remove();  
        i += 1;  
    }  
    return product;  
}
```

Third solution: Max heap, Max heap of size K can be created from first K elements. Then traverse the rest of array if the value at top of array is greater than the current value traversed. Then pop from the heap and then add the current value to the heap. Repeat this process till all the elements of array are traversed. In the end the heap contain the K minimum values then find their product.

Fourth solution: Quick select method, this method is the bi-product of quick select method to find the kth element of an array. Run quick select method to find kth element of an array. The values at the left of kth index is less than the value at kth index. Find product of first k element. Average case time complexity of this solution is $O(n)$.

Example 10.14:

```
public static int KSmallestProduct3(int[] arr, int size, int k) {  
    QuickSelectUtil(arr, 0, size - 1, k);
```

```
int product = 1;
for (int i = 0; i < k; i++)
product *= arr[i];
return product;
}
```

Print larger half of array

Problem: Given an array, print the larger half of the array when it will be sorted.

First solution: Sorting, Sort the array and then print the larger half of the array. Sorting will take $O(n \log n)$ time. And then print the larger half will take $O(n)$ time so total time complexity is $O(n \log n)$.

Example 10.15:

```
public static void PrintLargerHalf(int[] arr, int size) {
    Arrays.sort(arr); // , size, 1);
    for (int i = size / 2; i < size; i++)
        System.out.print(arr[i]); System.out.println(); }
```

Second solution: Min Heap, Create min heap from the array. Then pop first half $n/2$ elements from heap and find print the rest part. Heapify will take $O(n)$ time. Pop elements from heap take $O(\log n)$ time. $n/2$ time elements are popped from heap. So total time complexity is $O(n \log n)$.

Example 10.16:

```
public static void PrintLargerHalf2(int[] arr, int size) {
    int product = 1;
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
    for (int i = 0; i < size; i++) {
        pq.add(arr[i]);
    }
    for (int i = 0; i < size / 2; i++)
        pq.remove();
    System.out.println(pq); }
```

Third solution: Quick select method, run quick select method to find the middle

element of the array. The values at the right half of the array will be greater than the middle element's value. Finally print the value at the middle and right of it. Average case time complexity of this solution is $O(n)$.

Example 10.17:

```
public static void PrintLargerHalf3(int[] arr, int size) {  
    QuickSelectUtil(arr, 0, size - 1, size / 2);  
    for (int i = size / 2; i < size; i++)  
        System.out.print(arr[i]); System.out.println(); }
```

Nearly sorted array

Problem: Given a nearly sorted array, in which an element is at max k units away from its sorted position.

First solution: If you use sorting then it will take $O(N \log N)$ time

Second solution: There is one algorithm by which it can be done in $O(N \log K)$ time.

1. You can create a min Heap of size $K+1$ from first $K+1$ elements of input array.
2. Create an empty output array.
3. Pop an elements from heap and store it into output array.
4. Push next element from array to heap.
5. Repeat this process 3 and 4 till all the elements of array are consumed and heap is empty.
6. In the end you have sorted array.

Example 10.18:

```
public static void sortK(int[] arr, int size, int k) {  
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();  
    int i = 0;  
    for (i = 0; i < size; i++) {  
        pq.add(arr[i]);  
    }  
  
    int[] output = new int[size];  
    int index = 0;
```

```

for (i = k; i < size; i++) {
    output[index++] = pq.remove();
    pq.add(arr[i]);
}
while (pq.size() > 0)
    output[index++] = pq.remove();

for (i = k; i < size; i++) {
    arr[i] = output[i];
}
System.out.println(output); }

// Testing Code
public static void main(String[] args) {
    int k = 3;
    int[] arr = { 1, 5, 4, 10, 50, 9 };
    int size = arr.length;
    sortK(arr, size, k);
}

```

Chota bhim

Problem: A boy named Chota bhim had visited his grandfather Takshak. Takshak had offered bhim to drink elixir of power. He had offered number of cups filled with different quantity of elixir. Bhim is instructed to drink maximum quantity of elixir from the cups one by one. The cups are magical when bhim empties a cup, it refill itself with half the previous quantity it uses ceil [previous amount / 2] function to fill itself. Bhim is given 1 minute's time. He is efficient, in each second he drink from one cup and puts it back. Bhim had consumed lot of elixir by always picking the cup with maximum elixir. Now takshak had called you to find how much he had consumed.

Example:

5 Cups

Day 1 Cups: 2, 1, 7, 4, 2

>> 7 is selected

Day 2 Cups: 2, 1, 4, 4, 2
>> 4 is selected two times
Day 3 Cups: 2, 1, 2, 2, 2
>> 2 is selected four times
Day 4 and rest of days Cups: 1, 1, 1, 1, 1
>> rest all 1

Total: $7+4+4+2+2+2+(60 - 7) = 76$ units of elixir

First Solution: Sorted list method, First sort the ropes list. The minimum two will be at the start of the array. We remove the first two and then join them and add them to the sorted list in such a way that it remain sorted. Now repeat this process till all the arrays are added.

Each insertion into proper position such that the array remain sorted takes linear time so the total time complexity is $O(n^2)$

Example 10.19:

```
public static int ChotaBhim(int cups[], int size) {  
    int time = 60;  
    Arrays.sort(cups);  
    int total = 0;  
    int index, temp;  
    while (time > 0) {  
        total += cups[0];  
        cups[0] = (int) Math.ceil(cups[0] / 2.0);  
        index = 0;  
        temp = cups[0];  
        while (index < size - 1 && temp < cups[index + 1]) {  
            cups[index] = cups[index + 1];  
            index += 1;  
        }  
        cups[index] = temp;  
        time -= 1;  
    }  
    System.out.println("Total %d " + total);  
    return total;
```

```

}

public static void main(String[] args) {
    int cups[] = { 2, 1, 7, 4, 2 };
    ChotaBhim(cups, cups.length);
}

```

Second Solution: This performance can be further improved by using a heap to store the cups. Each deletion will take $\log K$ time and each insertion will again take $\log K$ time so the final Time complexity is $O(N.\log K)$

Second Solution: Performance can be improved by using a heap to store values. Create a min heap of all the ropes. Then remove the lowest two values from the heap. Add these two values and then insert them into the heap. Deletion and Insertion will take $O(\log N)$ time, so the total time complexity of this algorithm will be $O(N \log N)$

Example 10.20:

```

public static int ChotaBhim3(int cups[], int size) {
    int time = 60;
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
    int i = 0;
    for (i = 0; i < size; i++) {
        pq.add(cups[i]);
    }

    int total = 0;
    int value;
    while (time > 0) {
        value = pq.remove();
        total += value;
        value = (int) Math.ceil(value / 2.0);
        pq.add(value);
        time -= 1;
    }
    System.out.println("Total : " + total);
    return total;
}

```

```
}
```

Join Rope

Problem: Given N number of ropes of various length. You need to join these ropes to make a single rope. The cost of joining two ropes of length X and Y is $(X+Y)$ which is their combined length. You need to find the minimum cost of joining all the ropes. Minimum cost of joining rope is obtained when we always join two smallest ropes.

First Solution: Sorted list method, First sort the ropes list. The minimum two will be at the start of the array. We remove the first two and then join them and add them to the sorted list in such a way that it remain sorted. Now repeat this process till all the arrays are added.

Each insertion into proper position such that the array remain sorted takes linear time so the total time complexity is $O(n^2)$

Example 10.21:

```
public static int JoinRopes(int ropes[], int size) {  
    Arrays.sort(ropes);  
    System.out.println(ropes); int total = 0;  
    int value = 0;  
    int temp, index;  
    int length = size;  
  
    while (length >= 2) {  
        value = ropes[length - 1] + ropes[length - 2];  
        total += value;  
        index = length - 2;  
  
        while (index > 0 && ropes[index - 1] < value) {  
            ropes[index] = ropes[index - 1];  
            index -= 1;  
        }  
        ropes[index] = value;  
        length--;  
    }  
}
```

```

    }
    System.out.println("Total : " + total);
    return total;
}

public static void main(String[] args) {
    int ropes[] = { 2, 1, 7, 4, 2 };
    JoinRopes(ropes, ropes.length);
}

```

Second Solution: Performance can be improved by using a heap to store values. Create a min heap of all the ropes. Then remove the lowest two values from the heap. Add these two values and then insert them into the heap. Deletion and Insertion will take $O(\text{Log}N)$ time, so the total time complexity of this algorithm will be $O(N\text{log}N)$

Example 10.22:

```

public static int JoinRopes2(int ropes[], int size) {
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
    int i = 0;
    for (i = 0; i < size; i++) {
        pq.add(ropes[i]);
    }

    int total = 0;
    int value = 0;
    while (pq.size() > 1) {
        value = pq.remove();
        value += pq.remove();
        pq.add(value);
        total += value;
    }
    System.out.println("Total : %d " + total);
    return total;
}

```

kth Largest Stream

Problem: Given an infinite number of integer stream.

Solution: Heap is created, than elements are added to it till it contain k elements. Then as more and more elements are processed the values are added to the heap and Kth larges element is printed to screen.

Example 10.23:

```
public static int kthLargestStream(int k) {  
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>();  
    int size = 0;  
    int data = 0;  
    while (true) {  
        System.out.println("Enter data: ");  
        // data = System.in.read();  
  
        if (size < k - 1)  
            pq.add(data);  
        else {  
            if (size == k - 1)  
                pq.add(data);  
            else if (pq.peek() < data) {  
                pq.add(data);  
                pq.remove();  
            }  
            System.out.println("Kth largest element is :: " + pq.peek());  
        }  
        size += 1;  
    }  
}
```

Get Median function

Problem: Give a data structure that will provide median of given values in constant time.

Solution: We will use two heaps, one min heap and other max heap. Max heap will contain the first half of data and min heap will contain the second half of the data. Max heap will contain the smaller half of the data and its max value that is at the top of the heap will be the median contender. Similarly, the Min heap will contain the larger values of the data and its min value that is at its top will contain the median contender. We will keep track of the size of heaps. Whenever we insert a value to heap, we will make sure that the size of two heaps differs by max one element, otherwise we will pop one element from one and insert into another to keep them balanced.

Example 10.24:

```
public class MedianHeap {  
    PriorityQueue<Integer> minHeap;  
    PriorityQueue<Integer> maxHeap;  
  
    public MedianHeap() {  
        minHeap = new PriorityQueue<Integer>();  
        maxHeap = new PriorityQueue<Integer>(Collections.reverseOrder());  
    }  
  
    // Other Methods.  
    public void insert(int value) {  
        if (maxHeap.size() == 0 || maxHeap.peek() >= value) {  
            maxHeap.add(value);  
        } else {  
            minHeap.add(value);  
        }  
  
        // size balancing  
        if (maxHeap.size() > minHeap.size() + 1) {  
            value = maxHeap.remove();  
            minHeap.add(value);  
        }  
  
        if (minHeap.size() > maxHeap.size() + 1) {  
            value = minHeap.remove();  
            maxHeap.add(value);  
        }  
    }  
}
```

```
}

}

public int getMedian() {
if (maxHeap.size() == 0 && minHeap.size() == 0)
return Integer.MAX_VALUE;

if (maxHeap.size() == minHeap.size())
return (maxHeap.peek() + minHeap.peek()) / 2;
else if (maxHeap.size() > minHeap.size())
return maxHeap.peek();
else
return minHeap.peek();
}

public static void main(String[] args) {
int arr[] = { 1, 9, 2, 8, 3, 7, 4, 6, 5, 1, 9, 2, 8, 3, 7, 4, 6, 5, 10, 10 };
MedianHeap hp = new MedianHeap();

for (int i = 0; i < 20; i++) {
hp.insert(arr[i]);
System.out.println("Median after insertion of " + arr[i] + " is " +
hp.getMedian());
}
}
}
```

Exercise

1. What is the worst-case runtime Complexity of finding the smallest item in a min-heap?
2. Find max in a min heap.
Hint: normal search in the complete list. There is one more optimization you can search from the mid of the array at index $N/2$
3. What is the worst-case time complexity of finding the largest item in a min-heap?
4. What is the worst-case time complexity of deleteMin in a min-heap?
5. What is the worst-case time complexity of building a heap by insertion?
6. Is a heap full or complete binary tree?
7. What is the worst time runtime Complexity of sorting an array of N elements using heapsort?
8. In given sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9
 - a. Draw a binary Min-heap by inserting the above numbers one by one
 - b. Also draw the tree that will be formed after calling Dequeue() on this heap
9. In given sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9
 - a. Draw a binary Max-heap by inserting the above numbers one by one
 - b. Also draw the tree that will be formed after calling Dequeue() on this heap
10. In given sequence of numbers: 3, 9, 5, 4, 8, 1, 5, 2, 7, 6. Construct a Min-heap by calling CreateHeap function.
11. Show an array that would be the result after the call to deleteMin() on this heap

12. In given list: [3, 9, 5, 4, 8, 1, 5, 2, 7, 6]. Apply heapify over this to make a min heap and sort the elements in decreasing order?
13. In Heap-Sort once a root element has been put in its final position, how much time, does it take to re-heapify the array so that the next removal can take place? In other words, what is the Time Complexity of a single element removal from the heap of size N?
14. What do you think the overall Time Complexity for heapsort is? Why do you feel this way?

CHAPTER 11: HASH-TABLE

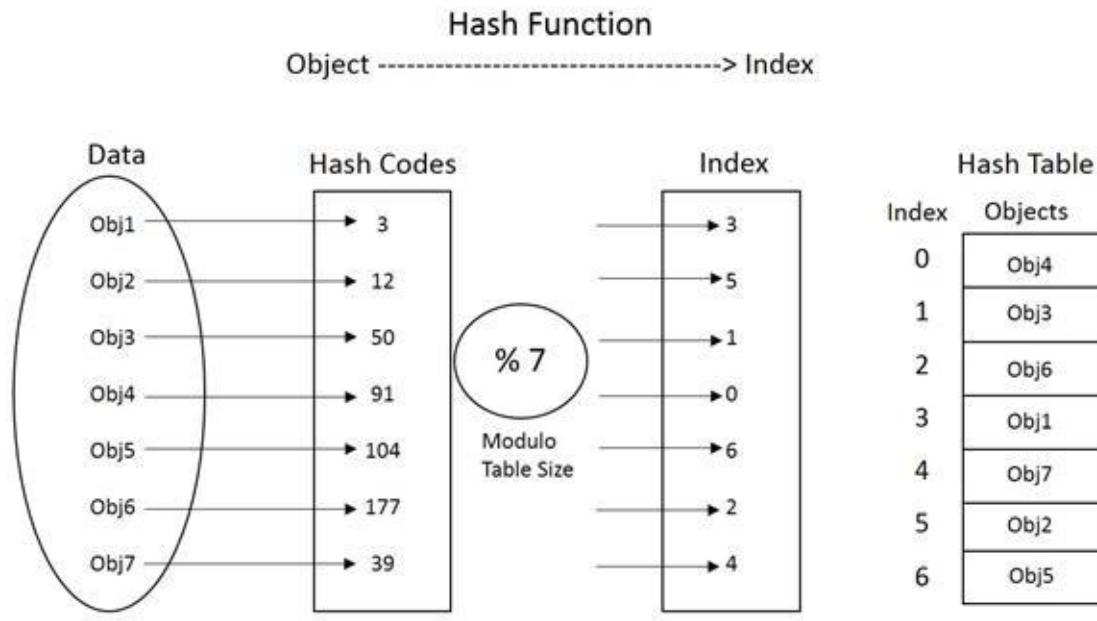
Introduction

In the searching chapter, we have gone through various searching techniques. Consider a problem of searching a value in an array. If the array is not sorted then we have no other option, but to look into each element one by one so the searching time complexity will be **O(n)**. If the array is sorted then we can search the value in **O(logn)** logarithmic time using binary search.

What if it is possible to get location / index of value we are looking in the array by a magic function that return index in constant time? We can directly go into that location and see whether the value we are searching for is present or not in **O(1)** constant time. Hash function works just like that of course, there is no magic involved.

Hash-Table

A Hash-Table is a data structure that maps keys to values. Each position of the Hash-Table is called a slot. The Hash-Table uses a hash function to calculate an index of an array. We use the Hash-Table when the number of keys, actually stored, is small relatively to the number of possible keys.



The process of storing data using a hash table is as follows:

1. Create an array of size M to store data; this array is called Hash-Table.
2. Find a hash code of a data by passing it through the hash function.
3. Take module of hash code by the size of Hashtable to get the index of the table where data will be stored.
4. Finally store this data in the designated index.

The process of searching value in Hash-Table using a hash function is as follows:

1. Find a hash code of the key we are searching for by passing it through the hash function.
2. Take module of hash code by the size of Hashtable to get the index of the table where value is stored.
3. Finally, retrieve the value from the designated index.

Hash-Table Abstract Data Type (ADT)

ADT of Hash-Table contains the following functions:

1. Insert(x), add x to the data set.
2. Delete(x), delete x from the data set.
3. Search(x), search x in data set.

Hash Function

A hash function is a function that generates an index in a table for a given key. An ideal hash function that generate a unique index for every key is called the perfect hash function.

Example 11.1: Most simple hash function

```
private int computeHash(int key)// division method
{
    int hashValue = key;
    return hashValue % tableSize;
}
```

There are many hash functions. The above function is a very simple hash function. Various hash generation logics will be added to this function to generate a better hash.

Collisions

When a hash function generates the same index for the two or more different keys, the problem is known as the collision. Ideally, hash function should return a unique address for each key, but practically it is not possible.

Properties of good hash function:

1. It should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them.
2. Choose a hash function, which can be computed quickly and returns values within the range of the Hash-Table.

3. Choose a hash function with a good collision resolution algorithm which can be used to compute alternative index if the collision occurs.
4. Choose a hash function, which uses the necessary information provided in the key.
5. It should have high load factor for a given set of keys.

Load Factor

Load factor = (Maximum Allowed Number of elements in Hash-Table) / Hash-Table size

Based on the above definition, Load factor is a measure of how many elements hash table is allowed to contain before its capacity is increased. When the number of elements exceeds the product of load factor and current capacity, the hash table is rehashed. Normally the hash table size is doubled in this case.

For example if load factor is 0.8 and hash table capacity is 1000 then we are allowed to add 800 elements to Hashtable without rehashing it.

Collision Resolution Techniques

Hash collisions are practically unavoidable when hashing large number of keys. Techniques that are used to find the alternate location in the Hash-Table is called collision resolution. There are a number of collision resolution techniques to handle the collision in hashing.

Most common and widely used techniques are:

- Open addressing
- Separate chaining

Hashing with Open Addressing

When using linear open addressing, the Hash-Table is represented by a one-dimensional array with indices that range from 0 to the table size-1.

One method of resolving collision is to look into a Hash-Table and find another free slot to hold the value that have caused the collision. A simple way is to

move from one slot to another in some sequential order until we find a free space. This collision resolution process is called Open Addressing.

Linear Probing

In Linear Probing, we try to resolve the collision of an index of a Hash-Table by sequentially searching the Hash-Table free location. Let us assume, if k is the index retrieved from the hash function. If the k th index is already filled then we will look for $(k+1) \% M$, then $(k+2) \% M$ and so on. When we get a free slot, we will insert the data into that free slot.

Example 11.2: The resolver function of linear probing

```
int resolverFun(int i) {  
    return i;  
}
```

Quadratic Probing

In Quadratic Probing, we try to resolve the collision of the index of a Hash-Table by quadratic increasing the search index of free location. Let us assume, if k is the index retrieved from the hash function. If the k th index is already filled then we will look for $(k+1^2) \% M$, then $(k+2^2) \% M$ and so on. When we get a free slot, we will insert the data into that free slot.

Example 11.3: The resolver function of quadratic probing

```
int resolverFun2(int i) {  
    return i * i;  
}
```

Note: Table size should be a prime number to prevent early looping it should not be too close to 2^{powN}

Linear Probing implementation

Hash table class consist of an array which is used to store data, and another array flags which is used to track if some slot is occupied or not. It also contain size of the array.

Example 11.4: Below is Hash-Table class and its initialization function.

```
public class HashTableLP {  
  
    private static int EMPTY_VALUE = 0;  
    private static int DELETED_VALUE = 1;  
    private static int FILLED_VALUE = 2;  
  
    private int tableSize;  
    int[] Arr;  
    int[] Flag;  
  
    public HashTableLP(int tSize) {  
        tableSize = tSize;  
        Arr = new int[tSize + 1];  
        Flag = new int[tSize + 1]; // flag value 0 is considered empty.  
    }  
  
    /* Other Methods */  
}
```

Example 11.5: Below is hash code generation and collision resolution function.

```
int computeHash(int key) {  
    return key % tableSize;  
}  
  
int resolverFun(int index) {  
    return index;  
}
```

When the hash index is already occupied, the resolver function is used to get new index.

Example 11.6:

```
boolean add(int value) {  
    int hashValue = computeHash(value);  
    for (int i = 0; i < tableSize; i++) {
```

```

if (Flag[hashValue]==EMPTY_VALUE ||
Flag[hashValue]==DELETED_VALUE) {
    Arr[hashValue] = value;
    Flag[hashValue] = FILLED_VALUE;
    return true;
}
hashValue += resolverFun(i);
hashValue %= tableSize;
}
return false;
}

```

add() function is used to add values to the hash table. First hash is calculated. Then we try to place that value in the Hash-Table. We look for empty node or lazy deleted node to insert value. In case insert did not success, we try new location using a resolver function.

Example 11.7:

```

boolean find(int value) {
    int hashValue = computeHash(value);
    for (int i = 0; i < tableSize; i++) {
        if (Flag[hashValue] == EMPTY_VALUE) {
            return false;
        }
        if (Flag[hashValue] == FILLED_VALUE && Arr[hashValue] == value) {
            return true;
        }
        hashValue += resolverFun(i);
        hashValue %= tableSize;
    }
    return false;
}

```

find() function is used to search values in the array. First hash is calculated. Then we try to find that value in the Hash-Table. We look for over desired value or empty node. In case we find the value that we are looking for, then we return that value or in case it is not found we return -1. We use a resolver function to find

the next probable index to search.

Example 11.8:

```
boolean remove(int value) {  
    int hashValue = computeHash(value);  
    for (int i = 0; i < tableSize; i++) {  
        if (Flag[hashValue] == EMPTY_VALUE) {  
            return false;  
        }  
        if (Flag[hashValue] == FILLED_VALUE && Arr[hashValue] == value) {  
            Flag[hashValue] = DELETED_VALUE;  
            return true;  
        }  
        hashValue += resolverFun(i);  
        hashValue %= tableSize;  
    }  
    return false;  
}
```

remove() function is used to delete values from a Hashtable. We do not actually delete the value we just mark that value as DELETED_NODE. Same as the insert and search we use resolverFun to find the next probable location of the key.

Example 11.9:

```
void print() {  
    for (int i = 0; i < tableSize; i++) {  
        if (Flag[i] == FILLED_VALUE) {  
            System.out.println("Node at index [" + i + "] :: " + Arr[i]);  
        }  
    }  
}  
  
public static void main(String[] args) {  
    HashTableLP ht = new HashTableLP(1000);  
    ht.add(1);  
    ht.add(2);
```

```
ht.add(3);
ht.print();
System.out.println(ht.remove(1));
System.out.println(ht.remove(4));
ht.print();
}
```

Output:

```
Node at index [1] :: 1
Node at index [2] :: 2
Node at index [3] :: 3
true
false
Node at index [2] :: 2
Node at index [3] :: 3
```

Print method print the content of hash table. Main function demonstrating how to use hash table.

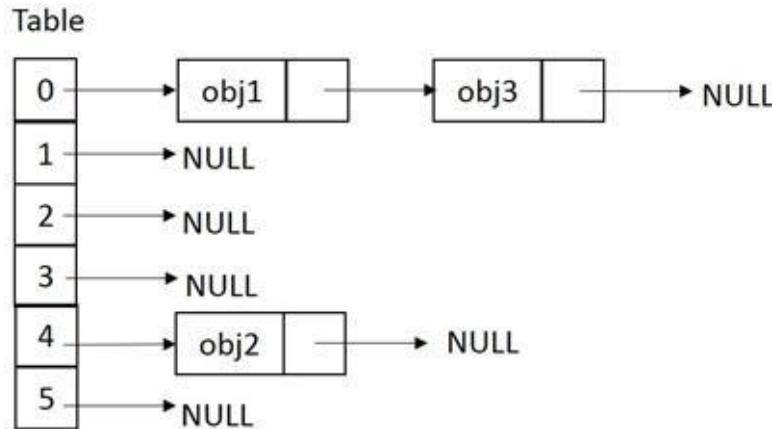
Quadratic Probing implementation.

Everything will be same as linear probing implementation only resolver function will be changed.

```
int resolverFun2(int index) {
    return index*index;
}
```

Hashing with separate chaining

Another method for collision resolution is based on an idea of putting the keys that collide in a linked list. This method is called separate chaining. To speed up search we use Insertion-Sort or keeping the linked list sorted.



Separate Chaining implementation

Example 11.10: Below is separate chaining implementation of hash tables.

```
public class HashTableSC {  
    private int tableSize;  
    Node[] listArray;  
  
    private class Node {  
        private int value;  
        private Node next;  
  
        public Node(int v, Node n) {  
            value = v;  
            next = n;  
        }  
    };  
  
    public HashTableSC() {  
        tableSize = 512;  
        listArray = new Node[tableSize];  
        for (int i = 0; i < tableSize; i++) {
```

```
listArray[i] = null;
}
}

private int computeHash(int key)// division method
{
int hashValue = key;
return hashValue % tableSize;
}

public void add(int value) {
int index = computeHash(value);
listArray[index] = new Node(value, listArray[index]);
}

public boolean remove(int value) {
int index = computeHash(value);
Node nextNode, head = listArray[index];
if (head != null && head.value == value) {
listArray[index] = head.next;
return true;
}
while (head != null) {
nextNode = head.next;
if (nextNode != null && nextNode.value == value) {
head.next = nextNode.next;
return true;
} else {
head = nextNode;
}
}
return false;
}

public void print() {
for (int i = 0; i < tableSize; i++) {
System.out.println("printing for index value :: " + i + "List of value printing ::
```

```

");
Node head = listArray[i];
while (head != null) {
System.out.println(head.value); head = head.next;
}
}
}

public boolean find(int value) {
int index = computeHash(value);
Node head = listArray[index];
while (head != null) {
if (head.value == value) {
return true;
}
head = head.next;
}
return false;
}

public static void main(String[] args) {
HashTableSC ht = new HashTableSC();

for (int i = 100; i < 110; i++) {
ht.add(i); }
System.out.println("search 100 :: " + ht.find(100));
System.out.println("remove 100 :: " + ht.remove(100));
System.out.println("search 100 :: " + ht.find(100));
System.out.println("remove 100 :: " + ht.remove(100));
}
}
}

```

Note: It is important to note that the size of the table should be such that all the slots in the table will eventually be occupied. Otherwise, part of the table will be unused. It is suggested that the table size should be a prime number for better distribution of keys.

Problems in Hashing

Anagram solver

Problem: Find if two strings are anagrams, an anagram is a word or phrase formed by reordering the letters of another word or phrase.

Example 11.11: Two words are anagram if they are of same size and their characters are same.

```
public static boolean isAnagram(char[] str1, char[] str2) {  
    int size1 = str1.length;  
    int size2 = str2.length;  
  
    if (size1 != size2)  
        return false;  
  
    HashMap<Character, Integer> hm = new HashMap<Character, Integer>();  
  
    for (char ch : str1) {  
        if (hm.containsKey(ch)) hm.put(ch, hm.get(ch) + 1);  
        else  
            hm.put(ch, 1);  
    }  
  
    for (char ch : str2) {  
        if (hm.containsKey(ch) == false || hm.get(ch) == 0)  
            return false;  
        else  
            hm.put(ch, hm.get(ch) - 1);  
    }  
  
    return true;  
}
```

Remove Duplicate

Problem: Remove duplicates in an array of numbers.

Solution: We can use a second array or the same array, as the output array. In the following example, Hash-Table is used to solve this problem.

Example 11.12:

```
public static String removeDuplicate(char[] str) {  
    HashSet<Character> hs = new HashSet<Character>();  
    String out = new String();  
  
    for (char ch : str) {  
        if (hs.contains(ch) == false) {  
            out += ch;  
            hs.add(ch);  
        }  
    }  
    return out;  
}
```

Find Missing

Problem: Given an array of integers, you need to find the missing number in the array.

Example 11.13:

```
public static int findMissing(int[] arr, int start, int end) {  
    HashSet<Integer> hs = new HashSet<Integer>();  
    for (int i : arr) {  
        hs.add(i);  
    }  
  
    for (int curr = start; curr <= end; curr++) {  
        if (hs.contains(curr) == false)  
            return curr;  
    }  
    return Integer.MAX_VALUE;  
}
```

Analysis: All the elements in the array is added to a HashTable. The missing element is found by searching into HashTable and final missing value is returned.

Print Repeating

Problem: Given an array of integers, print the repeating integers in the array.

Example 11.14:

```
public static void printRepeating(int[] arr) {  
    HashSet<Integer> hs = new HashSet<Integer>();  
  
    System.out.print("Repeating elements are:");  
    for (int val : arr) {  
        if (hs.contains(val)) System.out.print(" " + val);  
        else  
            hs.add(val);  
    }  
}
```

Analysis: All the values are added to the hash table, when some value came which is already in the hash table then that is the repeated value.

Print First Repeating

Problem: It is same as the above problem in this we need to print the first repeating number. Care should be taken to find the first repeating number. It should be the one number that is repeating. For example, 1, 2, 3, 2, 1. The answer should be 1 as it is the first number, which is repeating.

Example 11.15:

```
public static void printFirstRepeating(int[] arr) {  
    int i;  
    int size = arr.length;  
    HashSet<Integer> hs = new HashSet<Integer>();  
    int firstRepeating = Integer.MAX_VALUE;
```

```
for (i = size - 1; i >= 0; i--) {  
    if (hs.contains(arr[i])) {  
        firstRepeating = arr[i];  
    }  
    hs.add(arr[i]);  
}  
System.out.println("First Repeating number is : " + firstRepeating);
```

Analysis: Add values to the count map the one that is repeating will have multiple count. Now traverse the array again and see if the count is more than one. Therefore, that is the first repeating.

Exercise

1. Design a number (ID) generator system that generates numbers between 0-99999999 (8-digits).

The system should support two functions:

- a. int getNumber();
- b. boolean requestNumber();

getNumber() function should find out a number that is not assigned, then mark it as assigned and return that number. requestNumber() function checks the number if it is assigned or not. If it is assigned returns false, else marks it as assigned and return true.

2. In given large string, find the most occurring words in the string. What is the Time Complexity of the above solution?

Hint:-

- a. Create a Hashtable which will keep track of <word, frequency>
- b. Iterate through the string and keep track of word frequency by inserting into Hash-Table.
- c. When we have a new word, we will insert it into the Hashtable with frequency 1. For all repetition of the word, we will increase the frequency.
- d. We can keep track of the most occurring words whenever we are increasing the frequency we can see if this is the most occurring word or not.
- e. Time Complexity is **O(n)** where n is the number of words in the string and Space Complexity is the **O(m)** where m is the unique words in the string.

3. In the above question, What if you are given whole work of OSCAR WILDE, most popular playwrights in the early 1890s.

Hint:-

- a. Who knows how many books are there, let us assume there is a lot and we cannot put everything in memory. First, we need a Streaming Library so that we can read section by section in each document. Then we need a tokenizer that will give words to our program. In addition,

we need some sort of dictionary let us say we will use HashTable.

- b. What you need is - 1. A streaming library tokenizer, 2. A tokenizer 3. A hashmap

Method:

1. Use streamers to find a stream of the given words
2. Tokenize the input text

3. If the stemmed word is in hash map, increment its frequency count else add a word to hash map with frequency 1

- c. We can improve the performance by looking into parallel computing. We can use the map-reduce to solve this problem. Multiple nodes will read and process multiple documents. Once they are done with their processing, then we can do the reduce operation by merging them.

4. In the above question, What if we want to find the most common PHRASE in his writings.

Hint: - We can keep <phrase, frequency> Hash-Table and do the same process of the 2nd and 3rd problems.

5. Write a hashing algorithm for strings.

Hint: Use Horner's method

```
public static int hornerHash(char[] key, int tableSize) {  
    int size = key.length;  
    int h = 0;  
    int i;  
    for (i = 0; i < size; i++) {  
        h = (32 * h + key[i]) % tableSize;  
    }  
    return h;  
}
```

6. Pick two data structures to use in implementing a Map. Describe lookup, insert, & delete operations. Give time & Space Complexity for each. Give pros & cons for each.

Hint:-

- a) Linked List

- I. Insert is **O(1)**
- II. Delete is **O(1)**
- III. Lookup is **O(1)** auxiliary and **O(N)** worst case.
- IV. Pros: Fast inserts and deletes, can use for any data type.
- V. Cons: Slow lookups.

b) Balanced Search Tree (RB Tree)

- I. Insert is **O(logn)**
- II. Delete is **O(logn)**
- III. Lookup is **O(logn)**
- IV. Pros: Reasonably fast inserts/deletes and lookups.
- V. Cons: Data needs to have order defined on it.

CHAPTER 12: GRAPHS

Introduction

In this chapter, we will study about Graph Data Structure.

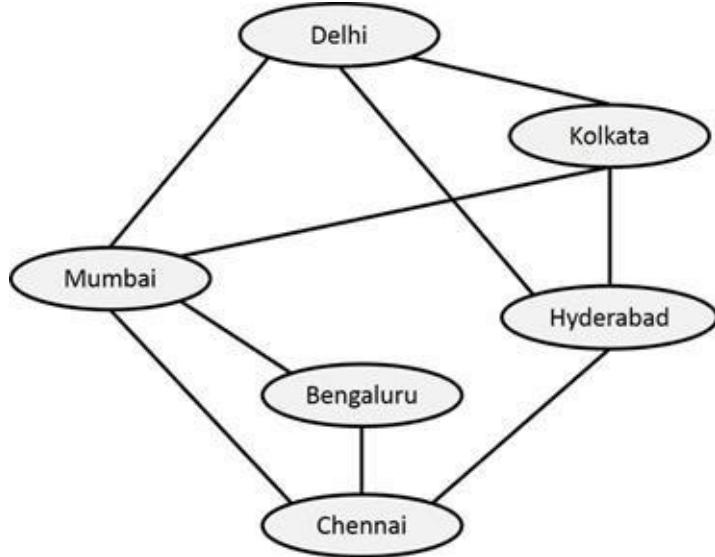
Graph data structure consists of following two components:

1. A finite set of nodes called **vertices**.
2. A finite set of pair of vertices called **edges**. Edges are connection between two vertices.

Some of the real world examples of graphs are:

1. Google Maps: Various locations can be represented as Vertices of Graph and path between them are represented by Edges of Graph. Graph theory, algorithms are used to suggest a shortest and quickest path between two locations.
2. Facebook Friend Suggestion: Each user profile is represented as vertices of graphs and their friend relationship is represented by the edges of Graph. Graph theory, algorithms are used for friend suggestion.
3. Topology Sorting: Topology sorting is a method to find sequence in which some event need be performed to complete some task by looking into the dependency of various events on each other. For example, a sequence of classes that we take to become a graduate in computer science. Or a sequence of steps that we take to become ready for jobs daily
4. Transportation Network: Map of airlines is represented as Graph. Various airports are represented as nodes of the graph and if there is a non-stop flight from airport u and airport v then then in the graph there is an edge from node u to node v. You may want to go from one location to another, through graph theory algorithms we can compute shortest, quickest or cheapest path from source to destination.

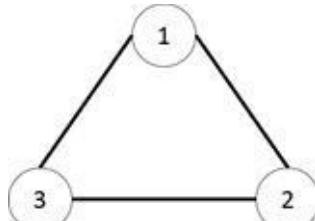
The flight connection between major cities of India can also be represented by the graph given below. Each city is represented as vertices and flight between cities is represented as edges.



Graph Definition: A Graph is represented by ordered pair G where $G = (V, E)$, where V is a finite set of points called **Vertices** and E is a finite set of **Edges**. Each **edge** is a pair (u, v) where $u, v \in V$.

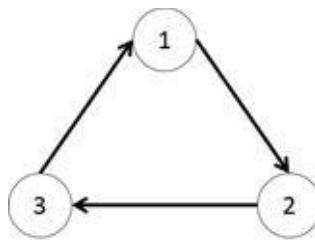
Graph Terminology

Undirected Graph: An Undirected Graph is a graph in which edges have no directions. Here the edges of the graph are two ways. An edge (x, y) is identical to edge (y, x) .



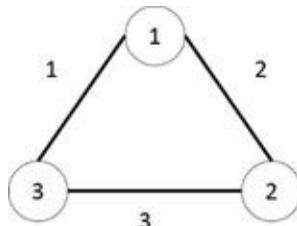
Undirected Graph

Directed Graph or Digraph: A Directed Graph is a graph in which edges have a direction. Here the edges of graph are one way. An edge (x, y) is not identical to edge (y, x) .



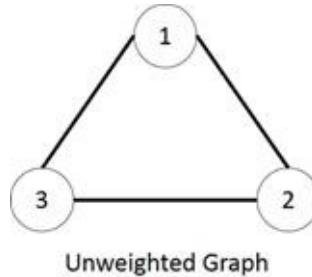
Directed Graph

Weighted Graph: A Graph is weighted if its edges have some value or weight associated with them.



Weighted Graph

Unweighted Graph: A Graph in which edges do not have any weight associated with them.



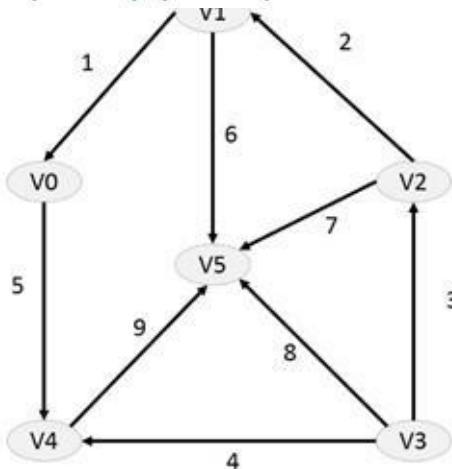
Path: A Path is a sequence of edges between two vertices. The length of a path is defined as the sum of the weight of all the edges in the path.

Simple Path: A path with distinct vertices is called simple path.

Adjacent Vertex: Two vertices u and v are **adjacent** if there is an edge whose endpoints are u and v .

In the below graph: $V = \{V1, V2, V3, V4, V5, V6, V7, V8, V9\}$

$$E = \{(V1, V0, 1), (V2, V1, 2), (V3, V2, 3), (V3, V4, 4), (V5, V4, 5), \\ (V1, V5, 6), (V2, V5, 7), (V3, V5, 8), (V4, V5, 9)\}$$



The **in-degree** of a vertex v , denoted by $\text{indeg}(v)$ is the number of incoming edges to the vertex v .

The **out-degree** of a vertex v , denoted by $\text{outdeg}(v)$ is the number of outgoing edges of a vertex v .

The **degree** of a vertex v , denoted by $\text{deg}(v)$ is the total number of edges whose one endpoint is v .

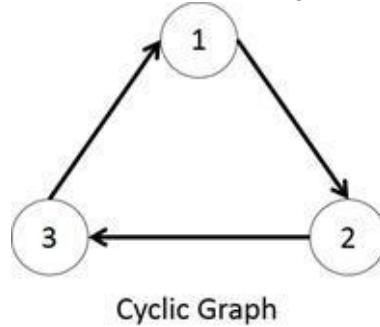
$$\text{deg}(v) = \text{Indeg } (v) + \text{outdeg } (v)$$

In the above graph: $\text{deg}(V4) = 3$, $\text{indeg}(V4) = 2$ and $\text{outdeg}(V4) = 1$

A **Cycle** is a path that starts and ends at the same vertex and includes at least one vertex.

An edge is a **Self-Loop** if two if its two endpoints coincide. This is a form of a cycle.

Cyclic Graph: A Graph that has one or more cycles is called a Cyclic Graph.



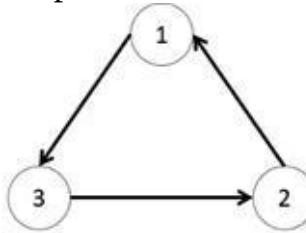
Acyclic Graph: A Graph that has no cycle is called an Acyclic Graph.

Directed Acyclic Graph / DAG: A directed graph that does not have any cycle is called Directed Acyclic Graph.

Reachable: A vertex v is **Reachable** from vertex u or u reaches v, if there is a path from u to v. In an undirected graph if v is reachable from u then u is reachable from v. However, in a directed graph it is possible that u reaches v but there is no path from v to u.

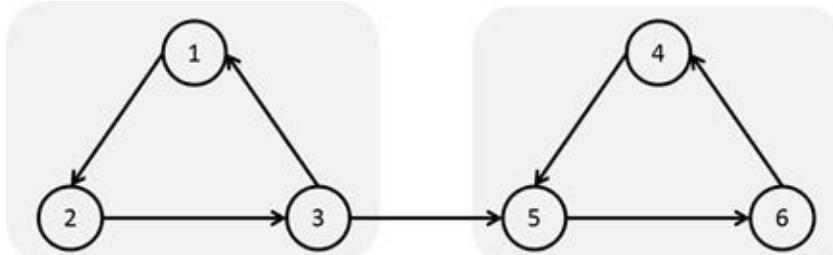
Connected Graph: A graph is a **Connected** if for any two vertices there is a path between them.

Strongly Connected Graph: A directed graph is strongly connected if for each pair of vertices u and v, there is a path from u to v and a path from v to u.



Strongly Connected Graph

Strongly Connected Components: A directed graph may have different sub-graphs that are strongly connected. These sub-graphs are called strongly connected components.



Strongly Connected Component

Weakly Connected Graph: A directed graph is weakly connected if for each pair of vertices u and v, there is a path from u to v or a path from v to u.

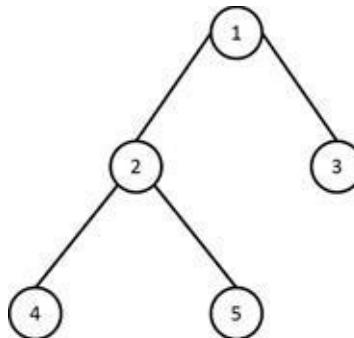
Complete Graph: A graph is complete if any vertex is connected by an edge to all the other vertices of the graph. A complete graph has maximum number of edges. For an undirected graph of n vertices the total number of edges will be $n(n-1)/2$ and for a directed graph the total number of edges will be $n(n-1)$.

A **Sub-Graph** of a graph G is a graph whose vertices and edges are a subset of the vertices and edges of G.

A **Spanning Sub-Graph** of G is a graph that connects all the vertices of G.

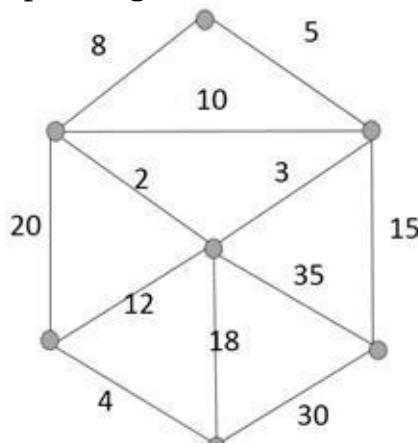
A **Forest** is a graph without cycles.

A **Tree** is a connected graph with no cycles. There is only one simple path between any two vertices u and v. If we remove any edge of a tree, it becomes a forest.

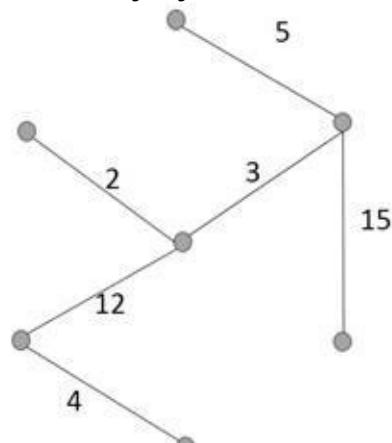


Tree

A **Spanning tree** of a graph is a tree that connects all the vertices of the graph. Since a Spanning-Tree is a tree, so it should not have any cycle.



Graph



Minimum Spanning Tree

Hamiltonian path is a path in which every vertex is visited exactly once with no

repeats, it does not have to start and end at the same vertex.

Hamiltonian circuit is a Hamiltonian Path such that there is an edge from its last vertex to its first vertex. A **Hamiltonian circuit** is a circuit that visits every vertex exactly once and it must start and end at the same vertex.

Eulerian Path is a path in the graph that visits every edge exactly once.

Eulerian Circuit is an Eulerian Path, which starts and ends on the same vertex. Or **Eulerian Circuit** is a path in the graph that visits every edge exactly one and it starts and ends on the same vertex.

Travelling salesman problem (TSP) a salesperson needs to visit various cities. He looks up the distance between each city, and puts the distance in a graph. In what order should he travel to visit each city and in the end return home city with the least distance travelled.

This problem can be rephrased as find the lowest cost Hamiltonian circuit

Graph Representation

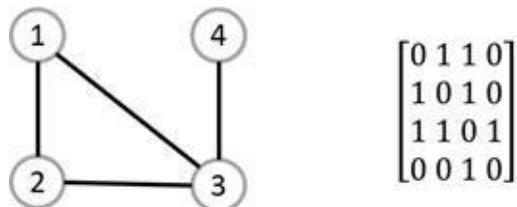
The two are the most common way of representing a graph are:

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

Adjacency Matrix is a two dimensional matrix of size V rows and V columns where V is the number of vertices in a graph. Adjacency matrix is represented using V x V size two-dimensional array.

Let us suppose the array is “adj”, so the node adj[i][j] corresponds to the edge from vertex i to vertex j. If adj[i][j] is not zero, then it means that there is a path from vertex i to vertex j. For an unweighted graph, the values in the array are either 0 for no path or 1 for a path exist. But in case of weighted graph the value in the matrix indicates the cost / weight of a path from vertex i to j.



In the above graph, each node has weight 1 so the adjacency matrix has just 1s or 0s. If the edges are of different weights, then that weight will be filled in the matrix.

Example 12.1: adjacency matrix representation of a graph

```
public class GraphAM {
```

```
    int count;  
    int[][] adj;
```

```
    GraphAM(int cnt) {
```

```
        count = cnt;  
        adj = new int[count][count];  
    }
```

```

public void addDirectedEdge(int src, int dst, int cost) {
    adj[src][dst] = cost;
}

public void addUndirectedEdge(int src, int dst, int cost) {
    addDirectedEdge(src, dst, cost);
    addDirectedEdge(dst, src, cost);
}

public void print() {
    for (int i = 0; i < count; i++) {
        System.out.print("Node index " + i + " is connected with : ");
        for (int j = 0; j < count; j++) {
            if (adj[i][j] != 0)
                System.out.print(j + " ");
        }
        System.out.println("");
    }
}

public static void main(String[] args) {
    GraphAM graph = new GraphAM(4);
    graph.addUndirectedEdge(0, 1, 1);
    graph.addUndirectedEdge(0, 2, 1);
    graph.addUndirectedEdge(1, 2, 1);
    graph.addUndirectedEdge(2, 3, 1);
    graph.print();
}
}

```

Output:

```

Vertex 0 is connected to : 1 2
Vertex 1 is connected to : 0 2
Vertex 2 is connected to : 0 1 3
Vertex 3 is connected to : 2

```

Analysis:

- In the constructor of graph of N vertices, an NxN size of two dimensional array adj is created.
- AddDirectedEdge function is used to add a directed edge from source to destination by setting adj[source][destination] field of adj array as one.
- AddUndirectedEdge function is used to add an undirected edge. It call's AddDirectedEdge function twice to join source and destination both ways.

Complexity Analysis:

- Space complexity of Adjacency Matrix representation is $O(n^2)$ because two dimensional array is created.
- Time complexity of search is $O(1)$. To find if there is an edge from vertex u to vertex v can be done in constant time.

Pros and cons of adjacency matrix

Pros of adjacency matrix:

1. Queries if there is an edge from vertex 'u' to vertex 'v' is takes $O(1)$ time.
2. Removing an edge takes $O(1)$ time.

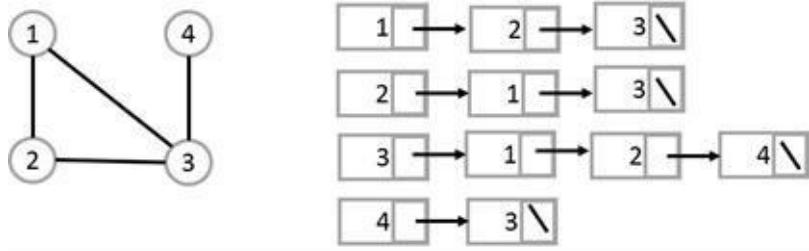
Cons of adjacency matrix:

1. Space complexity is $O(n^2)$.
2. Even if the graph is sparse (with few edges), space complexity remains same.
3. Adding a new vertex takes $O(n^2)$ time.

Sparse Matrix: In a huge graph, each node is connected with fewer nodes. So most of the places in the adjacency matrix remain empty. Such matrix is called sparse matrix. In most of the real world problems adjacency matrix is not a good choice for sore graph data.

Adjacency List

A more space efficient way of storing graph is adjacency list. **An adjacency list is an array of linked list. Each list element corresponds an edge of Graph.** Size of the array is equal to the number of vertices in the graph. Let the array be Arr[]. An entry Arr[k] corresponds to Kth vertex and is a list of vertices directly connected to kth vertex. The weights of edges can be stored in nodes of linked lists.



The adjacency list helps us to represent a sparse graph. An adjacency list representation also allows us to find all the vertices that are directly connected to any vertices by just one link list scan. In all our programs, we are going to use the adjacency list to store the graph.

Example 12.2: adjacency list representation of a graph

```
public class Graph {
    int count;
    private LinkedList<LinkedList<Edge>> Adj;

    private static class Edge {
        private int dest;
        private int cost;

        public Edge(int dst, int cst) {
            dest = dst;
            cost = cst;
        }
    }

    public Graph(int cnt) {
        count = cnt;
        Adj = new LinkedList<LinkedList<Edge>>();
        for (int i = 0; i < cnt; i++) {
            Adj.add(new LinkedList<Edge>());
        }
    }

    private void addDirectedEdge(int source, int dest, int cost) {
```

```
Edge edge = new Edge(dest, cost);
Adj.get(source).add(edge); }

public void addDirectedEdge(int source, int dest) {
    addDirectedEdge(source, dest, 1);
}

public void addUndirectedEdge(int source, int dest, int cost) {
    addDirectedEdge(source, dest, cost);
    addDirectedEdge(dest, source, cost);
}

public void addUndirectedEdge(int source, int dest) {
    addUndirectedEdge(source, dest, 1);
}

public void print() {
    for (int i = 0; i < count; i++) {
        LinkedList<Edge> ad = Adj.get(i);
        System.out.print("\n Vertex " + i + " is connected to : ");
        for (Edge adn : ad) {
            System.out.print("(" + adn.dest + ", " + adn.cost + ") ");
        }
    }
}

public static void main() {
    Graph gph = new Graph(5);
    gph.addDirectedEdge(0, 1, 3);
    gph.addDirectedEdge(0, 4, 2);
    gph.addDirectedEdge(1, 2, 1);
    gph.addDirectedEdge(2, 3, 1);
    gph.addDirectedEdge(4, 1, -2);
    gph.addDirectedEdge(4, 3, 1);
    gph.print();
}
}
```

Analysis:

- In the constructor of graph of N vertices, an array of lists of size N is created.
- `addDirectedEdge()` function is used to add a directed edge from source to destination by adding a tuple (destination, cost) to the corresponding vertex list.
- `addUndirectedEdge()` function is used to add an undirected edge. It call's `addDirectedEdge()` function twice to join source and destination both ways.

Complexity Analysis:

Space Complexity of Adjacency list is $O(E+V)$, to create vertices array and to store edges from each vertices.

Time complexity of search if there is an edge from vertex u to vertex v is done in `outdegree(u)`. We need to traverse the neighbours list of vertex u. In worst case, it can be $O(E)$

Graph traversals

Traversal is the process of exploring a graph by examining all its edges and vertices.

A list of some of the problems that are solved using graph traversal are:

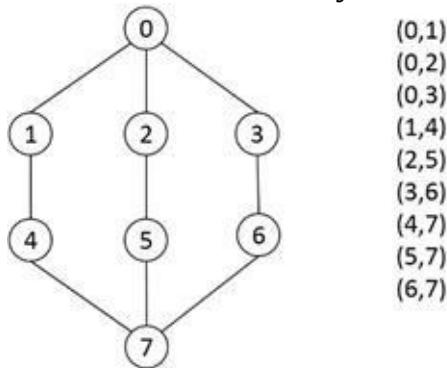
1. Determining a path from vertex u to vertex v, or report an error if there is no such path.
2. Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph.
3. Testing if a graph G is connected.
4. Finding a spanning tree of a Graph.
5. Finding if there is some cycle in the graph.

The **Depth first search (DFS)** and **Breadth first search (BFS)** are the two algorithms used to traverse a graph. These same algorithms can also be used to find some node in the graph, find if a node is reachable etc.

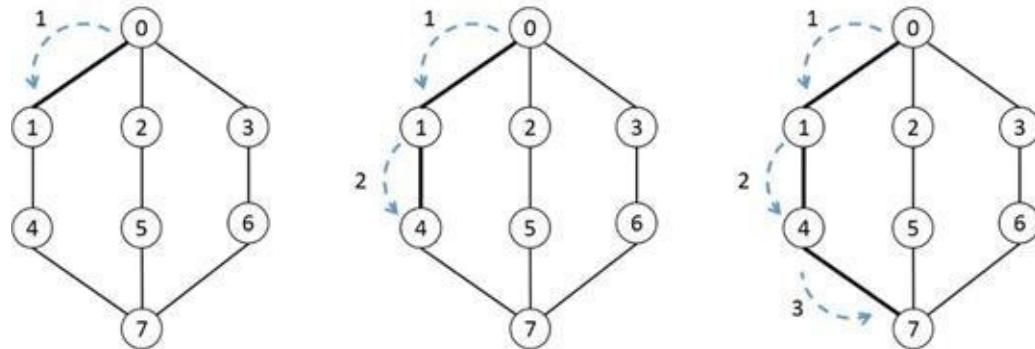
Depth First Traversal

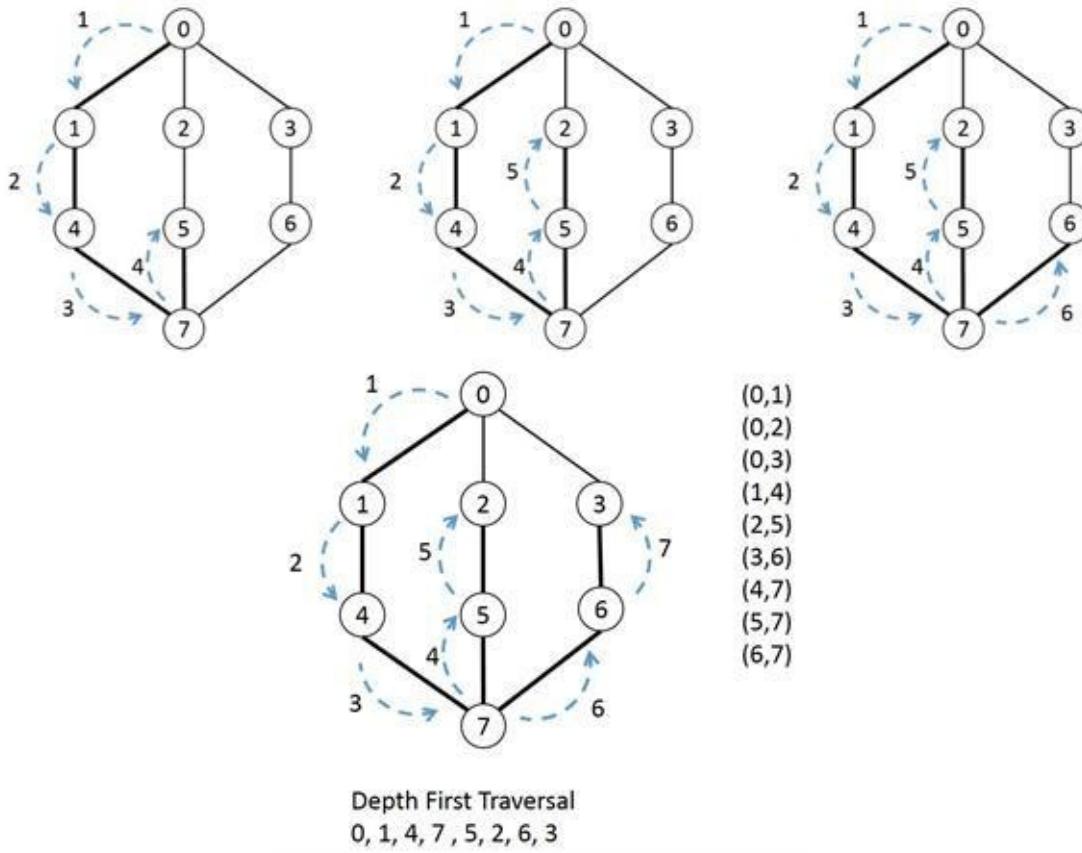
We start the DFS algorithm from starting point and go into depth of graph until we reach a dead end and then move up to parent node (Backtrack). In DFS, we use stack to get the next vertex to start a search. Alternatively, we can use recursion (system stack) to do the same.

Depth First Traversal of a graph is similar to Depth First Traversal of a tree. The only difference is that graphs may contain cycles (trees do not have cycles). So while traversing we may come back to the same node again. To avoid processing same node again, we use a Boolean visited array.



Below diagram, demonstrate DFS traversal.





Stack based implementation of DFS

Algorithm steps for DFS

1. Push the starting node in the stack.
2. Loop until the stack is empty.
3. Pop node from the stack call this node current.
4. Process the current node. //Print, etc.
5. Traverse all the child nodes of the current node and push them into stack.
6. Repeat steps 3 to 5 until the stack is empty.

Example 12.3: Stack based implementation of DFS.

```
public static boolean dfsStack(Graph gph, int source, int target) {
    int count = gph.count;
    boolean[] visited = new boolean[count];
    Stack<Integer> stk = new Stack<Integer>();
    stk.push(source); visited[source] = true;
```

```

while (stk.isEmpty() == false) {
    int curr = stk.pop();
    LinkedList<Edge> adl = gph.Adj.get(curr);
    for (Edge adn : adl) {
        if (visited[adn.dest] == false) {
            visited[adn.dest] = true;
            stk.push(adn.dest); }
        }
    }
    return visited[target];
}

```

Complexity Analysis:

- Time complexity of DFS algorithm when the graph is represented as adjacency list is $O(V+E)$ where V is the total number of vertices and E are the total number of edges in the graph.
- When the graph is represented as adjacency matrix the time complexity of algorithms is $O(V^2)$. We need to traverse adjacent vertices of a vertex, which is efficient in the graph when it is represented by adjacency list.

Recursion based implementation of DFS

Algorithm steps for DFS

1. In DFS() function, create a visited array to keep track of visited nodes.
2. In DFS() function, source node is passed as current node to DFSRec() recursion function.
3. All the nodes which are visited from index node are further passed to DFSRec() function as recursion.
4. This recursion will return to DFS() function when all the nodes which are visited from source are visited. Finally, we can find if target is visited or not by looking into visited array.

Example 12.4:

```

public static boolean dfs(Graph gph, int source, int target) {
    int count = gph.count;
    boolean[] visited = new boolean[count];
    dfsUtil(gph, source, visited);
}

```

```
    return visited[target];
}
public static void dfsUtil(Graph gph, int index, boolean[] visited) {
    visited[index] = true;
    LinkedList<Edge> adl = gph.Adj.get(index);
    for (Edge adn : adl) {
        if (visited[adn.dest] == false)
            dfsUtil(gph, adn.dest, visited);
    }
}
```

Complexity Analysis:

- Time complexity of DFS algorithm when the graph is represented as adjacency list is $O(V+E)$ where V is the total number of vertices and E are the total number of edges in the graph.
- When the graph is represented as adjacency matrix the time complexity of algorithms become $O(V^2)$.

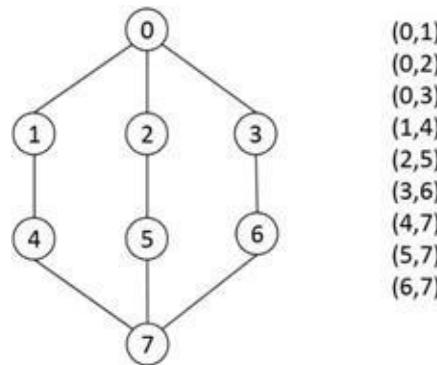
Breadth First Traversal

In BFS algorithm, a graph is traversed in layer-by-layer fashion. The graph is traversed, closer to the starting point. The queue is used to implement BFS.

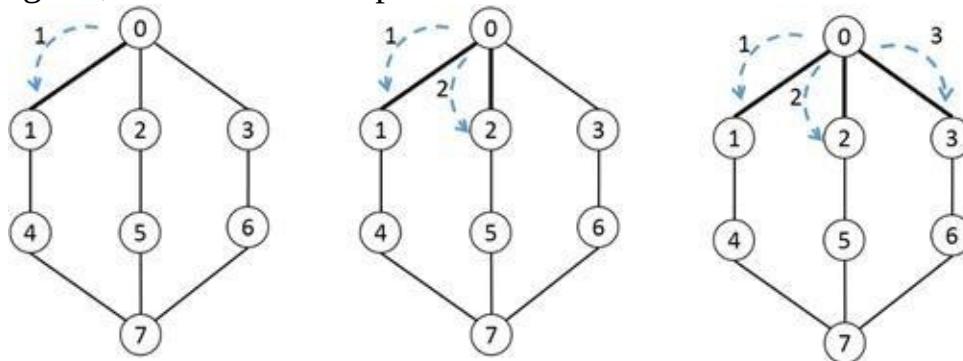
Breadth First Traversal of a graph is similar to Breadth First Traversal of a tree. The only difference is that graphs may contain cycles (trees do not have cycles). So while traversing we may come back to the same node again. To avoid processing same node again, we use a Boolean visited array.

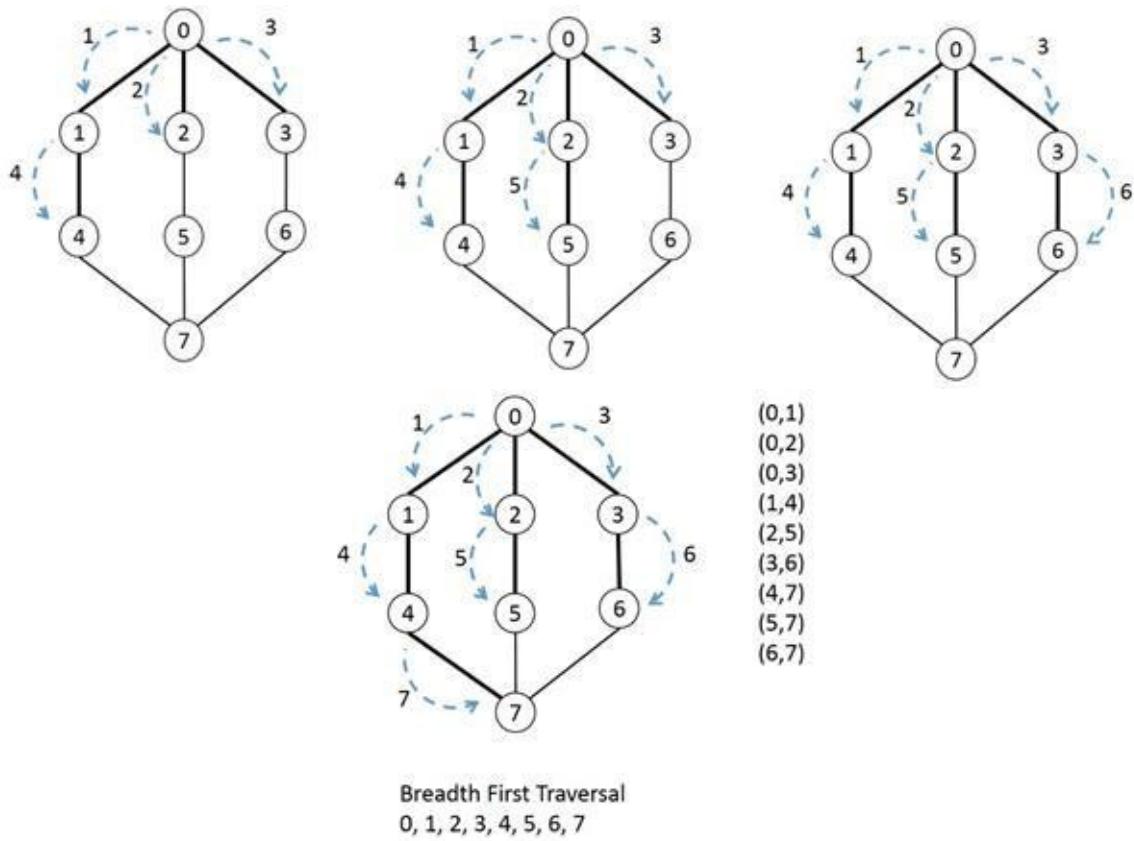
Algorithm steps for BFS

1. Push the starting node into the Queue.
2. Loop until the Queue is empty.
3. Remove a node from the Queue inside loop, and call this node current.
4. Process the current node.//print etc.
5. Traverse all the child nodes of the current node and push them into the Queue.
6. Repeat steps 3 to 5 until Queue is empty.



Below diagram, demonstrate Graph Breadth First Traversal.





Example 12.5:

```
public static boolean bfs(Graph gph, int source, int target) {
    int count = gph.count;
    boolean[] visited = new boolean[count];
    LinkedList<Integer> que = new LinkedList<Integer>();
    que.add(source); visited[source] = true;

    while (que.isEmpty() == false) {
        int curr = que.remove();
        LinkedList<Edge> adl = gph.Adj.get(curr);
        for (Edge adn : adl) {
            if (visited[adn.dest] == false) {
                visited[adn.dest] = true;
                que.add(adn.dest); }
        }
    }
    return visited[target];
}
```

Complexity Analysis: A runtime analysis of DFS and BFS traversal is $O(V+E)$ time, where V is the number of edges reachable from source node and E is the number of edges in the graph.

Uses of BFS and DFS

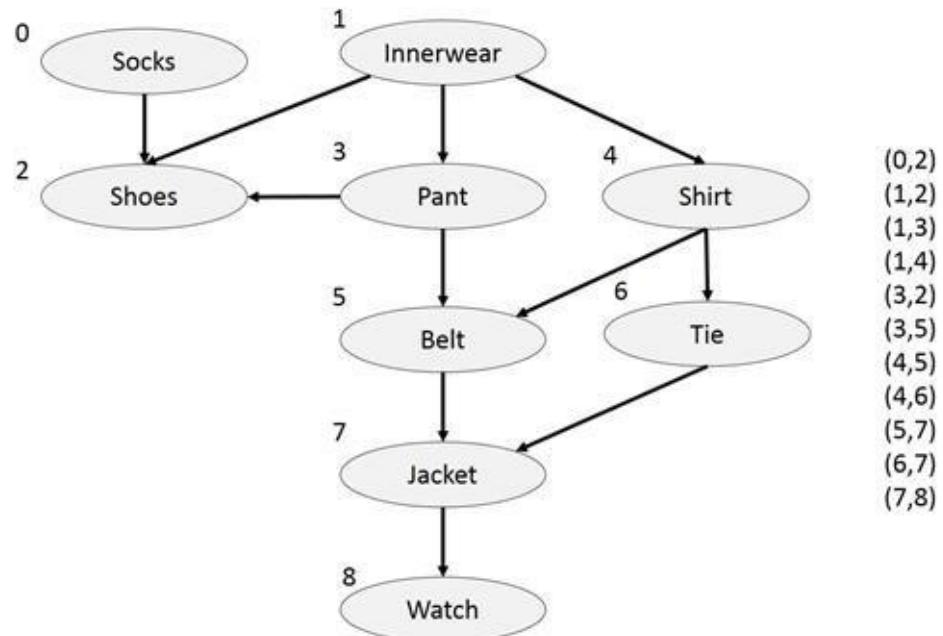
The following problems are solved using DFS:

1. Find if path exists between two vertices can be done using BFS or DFS.
2. Given a starting vertex u , finding the minimum number of edges from vertex s to all the other vertices of the graph is done using BFS.
3. Testing of a graph G is connected can be done using BFS or DFS.
4. Finding if there is a cycle in the graph Or check if given graph is a tree is done using DFS.
5. Topological Sorting is done using DFS.
6. Strongly connected components or graph in directed graph is done using DFS.

DFS & BFS based problems

Directed Acyclic Graph and Topological Sort

A Directed Acyclic Graph (DAG) is a directed graph with no cycle. A DAG represents a relationship, which is more general than a tree. Below is an example of DAG, this is how someone becomes ready for work. There are N other real life examples of DAG such as coerces selection to be a graduate from college



A topological sort is a method of ordering the nodes of a directed graph in which nodes represent activities and the edges represent dependency among those tasks. For topological sorting to work it is required that the graph should be a DAG which means it should not have any cycle. Just use DFS to get topological sorting.

Example 12.6:

```
public static void topologicalSort(Graph gph) {  
    Stack<Integer> stk = new Stack<Integer>();  
    int count = gph.count;  
    boolean[] visited = new boolean[count];
```

```

for (int i = 0; i < count; i++) {
    if (visited[i] == false) {
        dfsUtil2(gph, i, visited, stk);
    }
}
System.out.print("topologicalSort :: ");
while (stk.isEmpty() != true) {
    System.out.print(" " + stk.pop());
}
}

public static void dfsUtil2(Graph gph, int index, boolean[] visited,
Stack<Integer> stk) {
    visited[index] = true;
    LinkedList<Edge> adl = gph.Adj.get(index);
    for (Edge adn : adl) {
        if (visited[adn.dest] == false) {
            dfsUtil2(gph, adn.dest, visited, stk);
        }
    }
    stk.push(index);
}

public static void main() {
    Graph gph = new Graph(6); gph.addDirectedEdge(5, 2, 1);
    gph.addDirectedEdge(5, 0, 1);
    gph.addDirectedEdge(4, 0, 1);
    gph.addDirectedEdge(4, 1, 1);
    gph.addDirectedEdge(2, 3, 1);
    gph.addDirectedEdge(3, 1, 1);
    gph.print();
    topologicalSort(gph);
}

```

Topology sort is a DFS traversal of topology graph. First, all the children of a node are added to the stack, then only the current node is added. So the sorting order is maintained. Reader is requested to run some examples to understand this

algorithm.

Determining a path from vertex u to vertex v

Problem: Find if there is a path from vertex u to vertex v.

Solution: If there is a path from vertex u to vertex v then when we perform DFS from vertex u we will visit vertex v.

Example 12.7:

```
public static boolean pathExist(Graph gph, int source, int dest) {  
    int count = gph.count;  
    boolean[] visited = new boolean[count];  
  
    dfsUtil(gph, source, visited);  
    return visited[dest];  
}
```

Complexity Analysis: Time Complexity same as DFS for adjacency list implementation of the graph it is $O(V+E)$ and for adjacency matrix implementation it is $O(V^2)$

Count All Path DFS

Problem: Given a source vertex and a destination vertex, find all the possible paths from source to destination.

Example 12.8:

```
public static int countAllPathDFS(Graph gph, boolean[] visited, int source, int dest) {  
    if (source == dest) {  
        return 1;  
    }  
    int count = 0;  
    visited[source] = true;  
    LinkedList<Edge> adl = gph.Adj.get(source);  
    for (Edge adn : adl) {
```

```

        if (visited[adn.dest] == false) {
            count += countAllPathDFS(gph, visited, adn.dest, dest);
        }
        visited[source] = false;
    }
    return count;
}

public static int countAllPath(Graph gph, int src, int dest) {
    int count = gph.count;
    boolean[] visited = new boolean[count];
    return countAllPathDFS(gph, visited, src, dest);
}

```

Analysis: DFS traversal of a graph and count from various paths is added.

Print All Path

Problem: Print all the paths from source vertex to the destination vertex.

Example 12.9:

```

public static void printAllPathDFS(Graph gph, boolean[] visited, int source, int dest, Stack<Integer> path) {
    path.push(source);

    if (source == dest) {
        System.out.println(path); path.pop();
        return;
    }
    visited[source] = true;
    LinkedList<Edge> adl = gph.Adj.get(source);
    for (Edge adn : adl) {
        if (visited[adn.dest] == false) {
            printAllPathDFS(gph, visited, adn.dest, dest, path);
        }
    }
    visited[source] = false;
}

```

```

    path.pop();
}

public static void printAllPath(Graph gph, int src, int dest) {
    boolean[] visited = new boolean[gph.count];
    Stack<Integer> path = new Stack<Integer>();
    printAllPathDFS(gph, visited, src, dest, path);
}

```

Analysis: DFS traversal of a graph is performed by keeping track of the visited vertices. When the destination is found, then the path is printed.

Root Vertex

Problem: Find Root vertex in a graph. The root vertex is the vertex, which have path of all other vertices in a graph. If there are multiple root vertex, then return any one of them.

Hint: You need to return the top node of stack in topology sort.

Example 12.10:

```

public static int rootVertex(Graph gph) {
    int count = gph.count;
    boolean[] visited = new boolean[count];
    int retVal = -1;
    for (int i = 0; i < count; i++) {
        if (visited[i] == false) {
            dfsUtil(gph, i, visited);
            retVal = i;
        }
    }
    System.out.print("Root vertex is :: " + retVal);
    return retVal;
}

```

Transitive Closure

Problem: Given a directed graph, construct a transitive closure matrix or reachability matrix. Vertex v is reachable from vertex u if there is a path from u to v.

Transitive closure of a graph G is a graph G', which contains the same set of vertices as G and whenever there is a path from vertex u to vertex v in G there is an edge from u to v in G'.

Example 12.11:

```
public static void transitiveClosureUtil(Graph gph, int source, int dest, int[][] tc) {
    tc[source][dest] = 1;
    LinkedList<Edge> adl = gph.Adj.get(dest);
    for (Edge adn : adl) {
        if (tc[source][adn.dest] == 0)
            transitiveClosureUtil(gph, source, adn.dest, tc);
    }
}

public static int[][] transitiveClosure(Graph gph) {
    int count = gph.count;
    int tc[][] = new int[count][count];
    for (int i = 0; i < count; i++) {
        transitiveClosureUtil(gph, i, i, tc);
    }
    return tc;
}
```

Analysis: We create a two-dimensional array tc and assign all it to zero. Traverse the graph by keeping track of source vertex u and when we are able to traverse to a vertex v we will mark $tc[u][v]$ as reachable.

BFS Level Node

Problem: Perform BFS traversal of the graph. Along with the nodes, also print their distance from the starting source vertex.

Example 12.12:

```
public static void bfsLevelNode(Graph gph, int source) {  
    int count = gph.count;  
    boolean[] visited = new boolean[count];  
    int[] level = new int[count];  
    visited[source] = true;  
  
    LinkedList<Integer> que = new LinkedList<Integer>();  
    que.add(source);  
    level[source] = 0;  
    System.out.println("\nNode - Level");  
  
    while (que.isEmpty() == false) {  
        int curr = que.remove();  
        int depth = level[curr];  
        LinkedList<Edge> adl = gph.Adj.get(curr);  
        System.out.println(curr + " - " + depth);  
        for (Edge adn : adl) {  
            if (visited[adn.dest] == false) {  
                visited[adn.dest] = true;  
                que.add(adn.dest);  
                level[adn.dest] = depth + 1;  
            }  
        }  
    }  
}
```

Analysis: Along with the adjacent nodes, their distance from source is also added to the queue.

BFS Distance

Problem: Find the distance between source and destination vertex in a Graph.

Example 12.13:

```
public static int bfsDistance(Graph gph, int source, int dest) {  
    int count = gph.count;
```

```

boolean[] visited = new boolean[count];
LinkedList<Integer> que = new LinkedList<Integer>();
que.add(source);
visited[source] = true;
int[] level = new int[count];
level[source] = 0;

while (que.isEmpty() == false) {
    int curr = que.remove();
    int depth = level[curr];
    LinkedList<Edge> adl = gph.Adj.get(curr);
    for (Edge adn : adl) {
        if (adn.dest == dest) {
            return depth;
        }
        if (visited[adn.dest] == false) {
            visited[adn.dest] = true;
            que.add(adn.dest);
            level[adn.dest] = depth + 1;
        }
    }
}
return -1;
}

```

Analysis: Perform BFS traversal of the graph starting from source vertex and keep track of distance of adjacent nodes before adding them to the queue. If we find the destination vertex, then return its distance from the source vertex. And if the destination vertex is not reachable then return -1.

Find cycle in an Undirected Graph.

Problem: Find if there is a cycle in an undirected graph.

Solution: Complete connected component is traversed in a single DFS traversal. So keeping track of visited vertices is sufficient to find cycles in undirected graph.

Example 12.14:

```
public static boolean isCyclePresentUndirectedDFS(Graph graph, int index, int parentIndex, boolean[] visited) {
    visited[index] = true;
    int dest;
    LinkedList<Edge> adl = graph.Adj.get(index);
    for (Edge adn : adl) {
        dest = adn.dest;
        if (visited[dest] == false) {
            if (isCyclePresentUndirectedDFS(graph, dest, index, visited))
                return true;
        } else if (parentIndex != dest)
            return true;
    }
    return false;
}

public static boolean isCyclePresentUndirected(Graph graph) {
    int count = graph.count;
    boolean[] visited = new boolean[count];
    for (int i = 0; i < count; i++)
        if (visited[i] == false)
            if (isCyclePresentUndirectedDFS(graph, i, -1, visited))
                return true;
    return false;
}
```

Complexity Analysis: Time Complexity same as DFS for adjacency list implementation of the graph it is $O(V+E)$ and for adjacency matrix implementation it is $O(V^2)$

Find cycle in a Directed Graph.

Problem: Given a directed graph, find if there is a cycle in Graph. In a single traversal, if some node is traversed twice then there is a cycle.

Solution 1: We will be using DFS traversal for a single traversal; we will use a marked array to keep track of the number of nodes visited in a single traversal.

Example 12.15: Finding if there is a cycle in a graph using marked array.

```
public static boolean isCyclePresentDFS(Graph graph, int index, boolean[]
visited, int[] marked) {
    visited[index] = true;
    marked[index] = 1;
    LinkedList<Edge> adl = graph.Adj.get(index);

    for (Edge adn : adl) {
        int dest = adn.dest;
        if (marked[dest] == 1)
            return true;

        if (visited[dest] == false)
            if (isCyclePresentDFS(graph, dest, visited, marked))
                return true;
        }
        marked[index] = 0;
        return false;
    }

    public static boolean isCyclePresent(Graph graph) {
        int count = graph.count;
        boolean[] visited = new boolean[count];
        int[] marked = new int[count];
        for (int index = 0; index < count; index++) {
            if (visited[index] == false)
                if (isCyclePresentDFS(graph, index, visited, marked))
                    return true;
            }
            return false;
    }
```

Complexity Analysis: Time Complexity same as DFS for adjacency list implementation of the graph it is $O(V+E)$ and for adjacency matrix

implementation it is $O(V^2)$

Solution 2: Find if there is a cycle in a graph using colour method.

In colour method, initially visited array is assigned the value “white” which means that nodes are not visited. When we visit a node, we mark its colour as “Grey”. Nodes that are currently in visited path remain “Grey” and when all the connected nodes are traversed, and then the colour is changed to “Black”. If a node that is marked “Grey” is visited again, then there is a cycle in that path.

Example 12.16: Cycle detection using coloring method.

```
public static boolean isCyclePresentDFSColor(Graph graph, int index, int[] visited) {
    visited[index] = 1; // 1 = grey
    int dest;
    LinkedList<Edge> adl = graph.Adj.get(index);
    for (Edge adn : adl) {
        dest = adn.dest;
        if (visited[dest] == 1) // "Grey":
            return true;

        if (visited[dest] == 0) // "White":
            if (isCyclePresentDFSColor(graph, dest, visited))
                return true;
    }
    visited[index] = 2; // "Black"
    return false;
}

public static boolean isCyclePresentColor(Graph graph) {
    int count = graph.count;
    int[] visited = new int[count];
    for (int i = 0; i < count; i++) {
        if (visited[i] == 0) // "White"
            if (isCyclePresentDFSColor(graph, i, visited))
                return true;
    }
    return false;
}
```

Complexity Analysis: Time Complexity same as DFS for adjacency list implementation of the graph it is $O(V+E)$ and for adjacency matrix implementation it is $O(V^2)$

Transpose Graph

Problem: Transpose of a Graph G is a graph G' that has the same set of vertices, but the direction of edges is reversed.

Example 12.17:

```
public static Graph transposeGraph(Graph gph) {  
    int count = gph.count;  
    Graph g = new Graph(count); for (int i = 0; i < count; i++) {  
        LinkedList<Edge> adl = gph.Adj.get(i);  
        for (Edge adn : adl) {  
            int dest = adn.dest;  
            g.addDirectedEdge(dest, i);  
        }  
    }  
    return g;  
}
```

Complexity Analysis: Time Complexity for adjacency list implementation of the graph it is $O(V+E)$ and for adjacency matrix implementation it is $O(V^2)$

Test if an undirected graph is connected.

Problem: Given an undirected graph. Start from any vertex if we can visit all the other vertices using DFS or BFS then the graph is connected.

Example 12.18:

```
public static boolean isConnectedUndirected(Graph gph) {  
    int count = gph.count;  
    boolean[] visited = new boolean[count];  
  
    dfsUtil(gph, 0, visited);
```

```

for (int i = 0; i < count; i++) {
if (visited[i] == false) {
return false;
}
}
return true;
}

```

Complexity Analysis: Time Complexity same as DFS for adjacency list implementation of the graph it is $O(V+E)$ and for adjacency matrix implementation it is $O(V^2)$

Strongly Connected Graph

A directed graph is strongly connected if for each pair of vertices u and v, there is a path from u to v and a path from v to u.

To prove that the graph is connected graph we need to prove two conditions as true for any one vertex:

- 1) First condition, that every vertex can be visited from some vertex u. Or every vertex is reachable from vertex u.
- 2) Second condition, that vertex u is reachable from every other vertex.

First conditions can be verified by doing a DFS from some vertex u. We need to check that all the vertices of the graph are visited. Second condition can be verified by first creating transpose graph G' . Then perform DFS over G' from vertex u. If all other vertices are visited from vertex u in graph G' . It means that vertex u is reachable from all the vertices of the original graph G.

Kosaraju's Algorithm to find if the graph is connected based on DFS:

- 1) Create a visited array of size V, and Initialize all vertices in the visited array as False.
- 2) Choose any vertex and perform a DFS traversal of the graph. For all visited vertices, mark them visited by setting their values as True in visited array.
- 3) If DFS traversal does not mark all vertices as True, then return false.
- 4) Find transpose or reverse of the graph
- 5) Repeat step 1, 2 and 3 for the reversed graph.

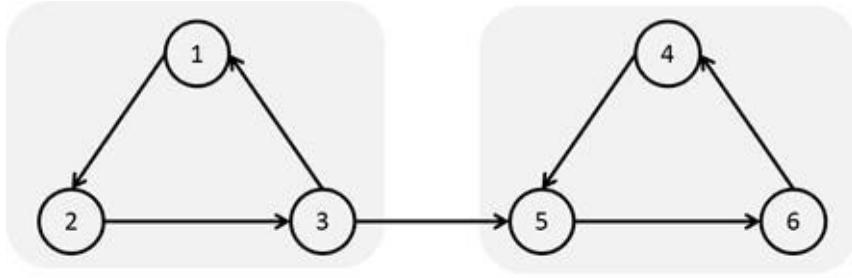
6) If DFS traversal mark all the vertices as True, then return true.

Example 12.19:

```
public static boolean isStronglyConnected(Graph gph) {  
    int count = gph.count;  
    boolean visited[] = new boolean[count];  
    for (int i = 0; i < count; i++) {  
        visited[i] = false;  
    }  
    dfsUtil(gph, 0, visited);  
    for (int i = 0; i < count; i++) {  
        if (visited[i] == false) {  
            return false;  
        }  
    }  
    Graph gReversed = transposeGraph(gph);  
    for (int i = 0; i < count; i++) {  
        visited[i] = false;  
    }  
    dfsUtil(gReversed, 0, visited);  
    for (int i = 0; i < count; i++) {  
        if (visited[i] == false) {  
            return false;  
        }  
    }  
    return true;  
}
```

Strongly Connected Components

Strongly Connected Components: A directed graph may have different sub-graphs that are strongly connected. These sub-graphs are called strongly connected components. In the below graph the whole graph is not strongly connected but its two subgraphs are strongly connected components.



Strongly Connected Component

Algorithm to find Strongly Connected Component

- 1) Create an empty stack, do DFS traversal on the graph G. Call DFS for adjacent vertices, and push them into the stack.
- 2) Reverse the graph G to get new graph G'
- 3) Perform the DFS traversal on the graph G' by picking vertices from the top of the stack.
- 4) Each strongly connected component is traversed in one single iteration.

Example 12.20:

```

public static void stronglyConnectedComponent(Graph gph) {
    int count = gph.count;
    boolean[] visited = new boolean[count];

    Stack<Integer> stk = new Stack<Integer>();
    for (int i = 0; i < count; i++) {
        if (visited[i] == false) {
            dfsUtil2(gph, i, visited, stk);
        }
    }

    Graph gReversed = transposeGraph(gph);
    for (int i = 0; i < count; i++) {
        visited[i] = false;
    }

    Stack<Integer> stk2 = new Stack<Integer>();
    while (stk.isEmpty() == false) {
        int index = stk.pop();
        if (visited[index] == false) {
            stk2.clear();
            dfsUtil2(gReversed, index, visited, stk2);
        }
    }
}

```

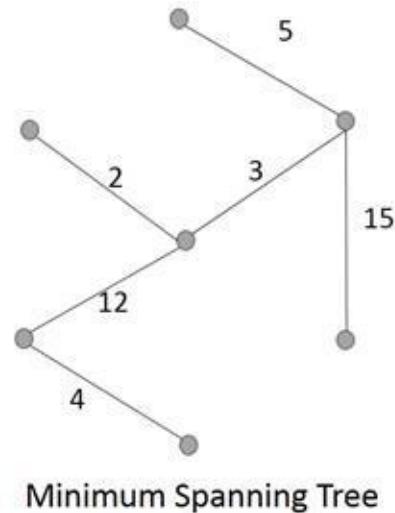
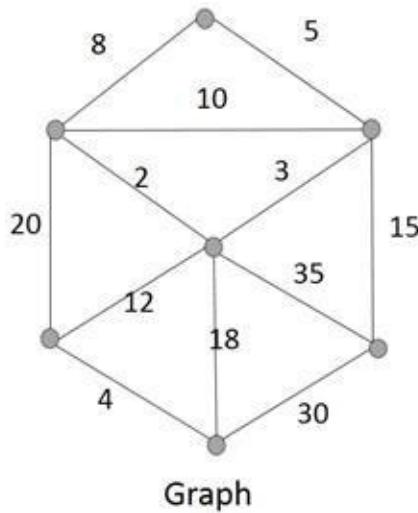
```
    System.out.println(stk2); }  
}  
}
```

Minimum Spanning Trees (MST)

A **Spanning Tree** of a graph G is a tree that contains all the vertices of the Graph.

A **Minimum Spanning Tree** is a spanning-tree whose sum of length / weight of edges is minimum as possible.

For example, if you want to setup communication between a set of cities, then you may want to use the least amount of wire as possible. MST can be used to find the network path and wire cost estimate.



Prim's Algorithm for MST

Prim's algorithm grows a single tree T , one edge at a time, until it becomes a spanning tree.

We initialize T with zero edges and U with a single node. Where T is spanning tree edges set and U is spanning tree vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other not in U . Since each edge adds one new vertex to U , after $n - 1$ additions, U contains all the vertices of the spanning tree and T becomes a spanning tree.

Example 12.21:

```
// Returns the MST by Prim's Algorithm  
// Input: A weighted connected graph G = (V, E)
```

```
// Output: Set of edges comprising a MST
```

```
Algorithm Prim(G)
T = {}
Let r be any vertex in G
U = {r}
for i = 1 to |V| - 1 do
  e = minimum-weight edge (u, v)
  With u in U and v in V-U
  U = U + {v}
  T = T + {e}
return T
```

Prim's Algorithm using a priority queue to get the closest next vertex
Time complexity is $O(E \log V)$ where V vertices and E edges of the MST.

Example 12.22: Prims algorithm implementation for adjacency list representation of graph.

```
public static void prims(Graph gph) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    boolean[] visited = new boolean[gph.count];
    int source = 1;

    for (int i = 0; i < gph.count; i++) {
        previous[i] = -1;
        dist[i] = 999999; // infinite
    }

    dist[source] = 0;
    previous[source] = -1;
    EdgeComparator comp = new EdgeComparator();
    PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);
    Edge node = new Edge(source, 0);
    queue.add(node);
    while (queue.isEmpty() != true) {
        node = queue.peek();
```

```

queue.remove(); visited[source] = true;
source = node.dest;
LinkedList<Edge> adl = gph.Adj.get(source);
for (Edge adn : adl) {
    int dest = adn.dest;
    int alt = adn.cost;
    if (dist[dest] > alt && visited[dest] == false) {
        dist[dest] = alt;
        previous[dest] = source;
        node = new Edge(dest, alt);
        queue.add(node); }
    }
}

// printing result.
int count = gph.count;
for (int i = 0; i < count; i++) {
    if (dist[i] == Integer.MAX_VALUE) {
        System.out.println(" node id " + i + " prev " + previous[i] + " distance :
Unreachable");
    } else {
        System.out.println(" node id " + i + " prev " + previous[i] + " distance : " +
dist[i]);
    }
}
}

```

Example 12.23: Prims algorithm implementation for adjacency matrix representation of graph.

```

public static void prims(GraphAM gph) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    int source = 0;
    boolean[] visited = new boolean[gph.count];

    for (int i = 0; i < gph.count; i++) {

```

```

previous[i] = -1;
dist[i] = Integer.MAX_VALUE; // infinite
visited[i] = false;
}

dist[source] = 0;
previous[source] = -1;

EdgeComparator comp = new EdgeComparator();
PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);

Edge node = new Edge(source, 0);
queue.add(node);
while (queue.isEmpty() != true) {
node = queue.peek();
queue.remove(); source = node.dest;
visited[source] = true;
for (int dest = 0; dest < gph.count; dest++) {
int cost = gph.adj[source][dest];
if (cost != 0) {
int alt = cost;
if (dist[dest] > alt && visited[dest] == false) {

dist[dest] = alt;
previous[dest] = source;
node = new Edge(dest, alt);
queue.add(node); }
}
}
}

int count = gph.count;
for (int i = 0; i < count; i++) {
if (dist[i] == Integer.MAX_VALUE) {
System.out.println("\n node id " + i + " prev " + previous[i] + " distance :
Unreachable");
} else {
}
}

```

```

        System.out.println(" node id " + i + " prev " + previous[i] + " distance : " +
dist[i]);
    }
}
}

private static class Edge {
    private int dest;
    private int cost;

    public Edge(int dst, int cst) {
        dest = dst;
        cost = cst;
    }
}

static class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge x, Edge y) {
        if (x.cost < y.cost) {
            return -1;
        }
        if (x.cost > y.cost) {
            return 1;
        }
        return 0;
    }
}

```

Kruskal's Algorithm

Kruskal's Algorithm repeatedly chooses the smallest-weight edge that does not form a cycle.

Sort the edges in non-decreasing order of cost: $c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$.

Set T to be the empty tree. Add edges to tree one by one, if it does not create a cycle.

```
// Returns the MST by Kruskal's Algorithm
```

```
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST
```

```
Algorithm Kruskal(G)
    Sort the edges E by their weights
    T = { }
    while |T| + 1 < |V| do
        e = next edge in E
        if T + {e} does not have a cycle then
            T = T + {e}
    return T
```

Kruskal's Algorithm is $O(E \log V)$ using efficient cycle detection.

Shortest Path Algorithms in Graph

Single Source Shortest Path

For a graph $G = (V, E)$, the single source shortest path problem is to find the shortest path from a given source vertex s to all the vertices of V .

Single Source Shortest Path for unweighted Graph.

Find single source shortest path for unweighted graph or a graph with all the vertices of same weight.

Or

Given a starting vertex s , finding the minimum number of edges from vertex s to all the other vertices of the graph.

Example 12.24:

```
public static void shortestPath(Graph gph, int source)// unweighted graph
{
    int curr;
    int count = gph.count;
    int[] distance = new int[count];
    int[] path = new int[count];
    for (int i = 0; i < count; i++) {
        distance[i] = -1;
    }
    Queue<Integer> que = new LinkedList<Integer>();
    que.add(source); distance[source] = 0;
    while (que.isEmpty() == false) {
        curr = que.remove();
        LinkedList<Edge> adl = gph.Adj.get(curr);
        for (Edge adn : adl) {
            if (distance[adn.dest] == -1) {
                distance[adn.dest] = distance[curr] + 1;
                path[adn.dest] = curr;
                que.add(adn.dest); }
        }
    }
}
```

```

    }
}

for (int i = 0; i < count; i++) {
    System.out.println(path[i] + " to " + i + " weight " + distance[i]);
}
}

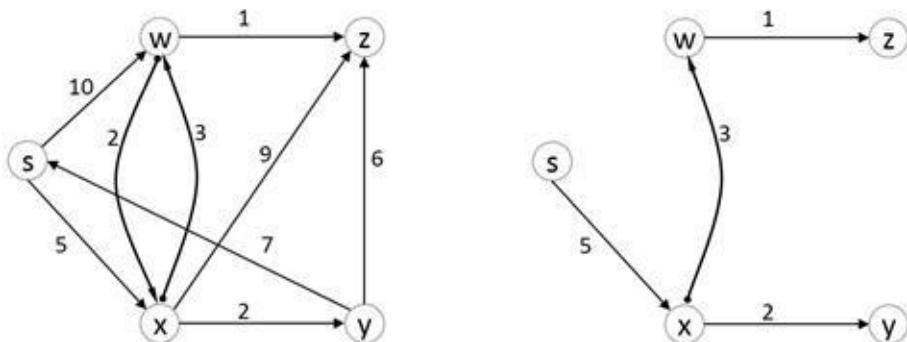
```

Analysis:

- First, the starting point source is added to a queue.
- Breadth first traversal is performed
- Nodes that are closer to the source are traversed first and processed.

Dijkstra's algorithm

Dijkstra's algorithm is used for single-source shortest path problem for weighted edges with no negative weight. Given a weighted connected graph G, find shortest paths from the source vertex s to each of the other vertices. Dijkstra's algorithm is similar to prims algorithm. It maintains a set of nodes for which shortest path is known.



Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning as we do not know the actual path to the nodes and parents of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)

At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the

distance and a parent list fully populated.

```
// Solves SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph G = (V, E)
// with no negative weights, and source vertex v
// Output: The length and path from s to every v

Algorithm Dijkstra(G, s)
for each v in V do
    D[v] = infinite // Unknown distance
    P[v] = null //unknown previous node
    add v to PQ //adding all nodes to priority queue

D[source] = 0 // Distance from source to source

while (PQ is not empty)
    u = vertex from PQ with smallest D[u]
    remove u from PQ
    for each v adjacent from u do
        alt = D[u] + length ( u , v )
        if alt < D[v] then
            D[v] = alt
            P[v] = u
Return D[], P[]
```

Time complexity is $O(|E|\log|V|)$ Where V is the number of vertices and E is the number of edges in the given graph.

Note: Dijkstra's algorithm does not work for graphs with negative edge weight.
Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Example 12.25: Dijkstra's algorithm for adjacency list implementation of graph.

```
public static void dijkstra(Graph gph, int source) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    boolean[] visited = new boolean[gph.count];
```

```

for (int i = 0; i < gph.count; i++) {
    previous[i] = -1;
    dist[i] = 999999; // infinite
}

dist[source] = 0;
previous[source] = -1;
EdgeComparator comp = new EdgeComparator();
PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);
Edge node = new Edge(source, 0);
queue.add(node);
while (queue.isEmpty() != true) {
    node = queue.peek();
    queue.remove(); source = node.dest;
    visited[source] = true;
    LinkedList<Edge> adl = gph.Adj.get(source);
    for (Edge adn : adl) {
        int dest = adn.dest;
        int alt = adn.cost + dist[source];
        if (dist[dest] > alt && visited[dest] == false) {

            dist[dest] = alt;
            previous[dest] = source;
            node = new Edge(dest, alt);
            queue.add(node); }
    }
}

int count = gph.count;
for (int i = 0; i < count; i++) {
    if (dist[i] == Integer.MAX_VALUE) {
        System.out.println("\n node id " + i + " prev " + previous[i] + " distance :
Unreachable");
    } else {
        System.out.println(" node id " + i + " prev " + previous[i] + " distance : " +
dist[i]); }
}

```

```

    }
}
}

static class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge x, Edge y) {
        if (x.cost < y.cost) {
            return -1;
        }
        if (x.cost > y.cost) {
            return 1;
        }
        return 0;
    }
}

```

Example 12.26: Dijkstra's algorithm for adjacency matrix implementation of graph.

```

public static void dijkstra(GraphAM gph, int source) {
    int[] previous = new int[gph.count];
    int[] dist = new int[gph.count];
    boolean[] visited = new boolean[gph.count];

    for (int i = 0; i < gph.count; i++) {
        previous[i] = -1;
        dist[i] = Integer.MAX_VALUE; // infinite
        visited[i] = false;
    }

    dist[source] = 0;
    previous[source] = -1;
    EdgeComparator comp = new EdgeComparator();
    PriorityQueue<Edge> queue = new PriorityQueue<Edge>(100, comp);

    Edge node = new Edge(source, 0);
    queue.add(node);
    while (queue.isEmpty() != true) {

```

```

node = queue.peek();
queue.remove(); source = node.dest;
visited[source] = true;
for (int dest = 0; dest < gph.count; dest++) {
    int cost = gph.adj[source][dest];
    if (cost != 0) {
        int alt = cost + dist[source];
        if (dist[dest] > alt && visited[dest] == false) {

            dist[dest] = alt;
            previous[dest] = source;
            node = new Edge(dest, alt);
            queue.add(node); }
    }
}
}

int count = gph.count;
for (int i = 0; i < count; i++) {
    if (dist[i] == Integer.MAX_VALUE) {
        System.out.println("\n node id " + i + " prev " + previous[i] + " distance :
Unreachable");
    } else {
        System.out.println(" node id " + i + " prev " + previous[i] + " distance : " +
dist[i]); }

}
}

private static class Edge {
    private int dest;
    private int cost;

    public Edge(int dst, int cst) {
        dest = dst;
        cost = cst;
    }
}

```

```

    }
}

static class EdgeComparator implements Comparator<Edge> {
    public int compare(Edge x, Edge y) {
        if (x.cost < y.cost) {
            return -1;
        }
        if (x.cost > y.cost) {
            return 1;
        }
        return 0;
    }
}

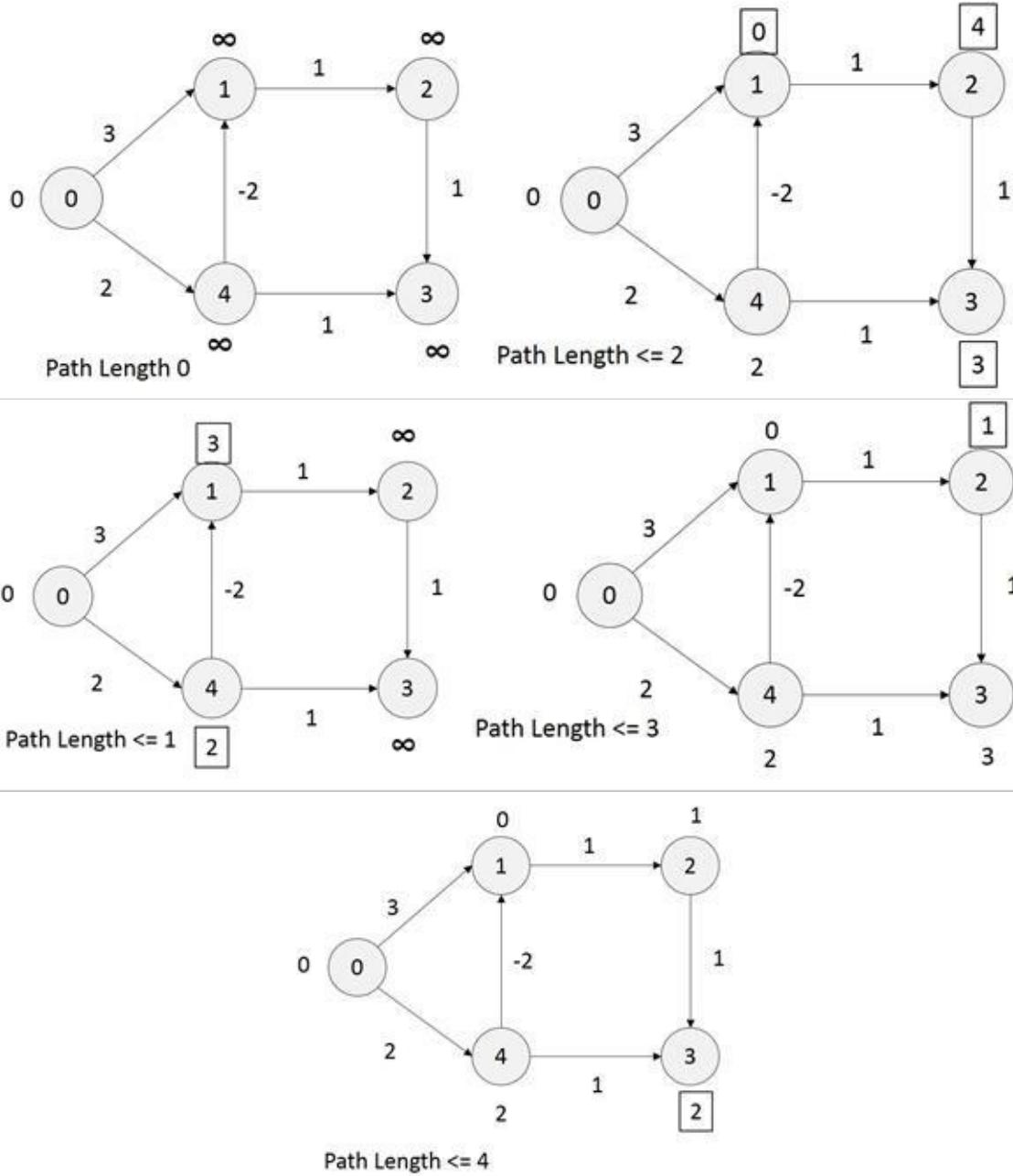
```

Bellman Ford Shortest Path

Single source shortest path from a source vertex s to all other vertices in a graph containing negative weight edges but no negative cycle is done using Bellman Ford algorithm. It does not work if there is some cycle in the graph whose total weight is negative. In this algorithm, distance of all the vertices is assigned to ∞ and the source vertex distance is assigned as 0. Then $V-1$ passes (V is total no of vertices) over all the edges is performed and the distance to destination is updated for each vertice.

We can checks over all the edges again and if there is change in the weights in the distance then it detect a negative weight cycle. If distance is changed after $V-1$ relaxation then we have a negative cycle. We can stop the algorithm if an iteration does not modify distance estimates. This is beneficial if shortest paths are likely to be less than $V-1$.

Time complexity is $O(V.E)$, where V is number of vertices and E is the total number of edges.



Example 12.27:

```
public static void bellmanFordshortestPath(Graph gph, int source) {
    int count = gph.count;
    int[] distance = new int[count];
    int[] path = new int[count];

    for (int i = 0; i < count; i++) {
        distance[i] = 999999; // infinite
        path[i] = -1;

        for (int j = 0; j < count; j++) {
            if (gph.getEdge(i, j) != null) {
                if (distance[i] + gph.getEdge(i, j).weight < distance[j]) {
                    distance[j] = distance[i] + gph.getEdge(i, j).weight;
                    path[j] = i;
                }
            }
        }
    }
}
```

```

}

distance[source] = 0;
// Outer loop will run (V-1) number of times.
// Inner for loop and while loop runs combined will
// run for Edges number of times.
// Which make the total complexity as O(V*E)

for (int i = 0; i < count - 1; i++) {
    for (int j = 0; j < count; j++) {
        LinkedList<Edge> adl = gph.Adj.get(j);
        for (Edge adn : adl) {
            int newDistance = distance[j] + adn.cost;
            if (distance[adn.dest] > newDistance) {
                distance[adn.dest] = newDistance;
                path[adn.dest] = j;
            }
        }
    }
}
for (int i = 0; i < count; i++) {
    System.out.println(path[i] + " to " + i + " weight " + distance[i]);
}
}

```

All Pairs Shortest Paths

Given a weighted graph $G(V, E)$, the all pair shortest path problem is used to find the shortest path between all pairs of vertices $u, v \in V$. Execute V instances of single source shortest path algorithm for each vertex of the graph.

The complexity of this algorithm will be $O(V^3)$

Hamiltonian Path and Hamiltonian Circuit

Hamiltonian path is a path in which every vertex is visited exactly once with no repeats, it does not have to start and end at the same vertex.

Hamiltonian path is a Np-Complete problem, so the only solution is possible using backtracking, which starts from a vertex s and try all adjacent vertices recursively. If we do not find the path, then we backtrack and try other vertices.

Example 12.28:

```
public static boolean hamiltonianPathUtil(GraphAM graph, int path[], int pSize, int added[]) {  
    // Base case full length path is found  
    if (pSize == graph.count) {  
        return true;  
    }  
    for (int vertex = 0; vertex < graph.count; vertex++) {  
        // there is a path from last element and next vertex  
        // and next vertex is not already included in path.  
        if (pSize == 0 || (graph.adj[path[pSize - 1]][vertex] == 1 && added[vertex] == 0)) {  
            path[pSize++] = vertex;  
            added[vertex] = 1;  
            if (hamiltonianPathUtil(graph, path, pSize, added))  
                return true;  
            // backtracking  
            pSize--;  
            added[vertex] = 0;  
        }  
    }  
    return false;  
}  
  
public static boolean hamiltonianPath(GraphAM graph) {  
    int[] path = new int[graph.count];  
    int[] added = new int[graph.count];
```

```

if (hamiltonianPathUtil(graph, path, 0, added)) {
    System.out.println("Hamiltonian Path found :: ");
    for (int i = 0; i < graph.count; i++)
        System.out.println(" " + path[i]);

    return true;
}
System.out.println("Hamiltonian Path not found");
return false;
}

```

Hamiltonian circuit is a Hamiltonian Path such that there is an edge from its last vertex to its first vertex. A **Hamiltonian circuit** is a circuit that visits every vertex exactly once and it must start and end at the same vertex.

Solution of Hamiltonian circuit is also NP-complete problem. The only difference is the base condition in which there should be a path from last node of the Hamiltonian path to the first element.

Example 12.29:

```

public static boolean hamiltonianCycleUtil(GraphAM graph, int path[], int
pSize, int added[]) {
    // Base case full length path is found
    // this last check can be modified to make it a path.
    if (pSize == graph.count) {
        if (graph.adj[path[pSize - 1]][path[0]] == 1) {
            path[pSize] = path[0];
            return true;
        } else
            return false;
    }
    for (int vertex = 0; vertex < graph.count; vertex++) {
        // there is a path from last element and next vertex
        if (pSize == 0 || (graph.adj[path[pSize - 1]][vertex] == 1 && added[vertex] ==
0)) {
            path[pSize++] = vertex;
            added[vertex] = 1;

```

```

if (hamiltonianCycleUtil(graph, path, pSize, added))
    return true;
// backtracking
pSize--;
added[vertex] = 0;
}
}
return false;
}

public static boolean hamiltonianCycle(GraphAM graph) {
    int[] path = new int[graph.count + 1];
    int[] added = new int[graph.count];
    if (hamiltonianCycleUtil(graph, path, 0, added)) {
        System.out.println("Hamiltonian Cycle found :: ");
        for (int i = 0; i <= graph.count; i++)
            System.out.print(" " + path[i]);
        return true;
    }
    System.out.println("Hamiltonian Cycle not found");
    return false;
}

public static void main(String[] args) {
    int count = 5;
    GraphAM graph = new GraphAM(count);
    int[][] adj = {
        { 0, 1, 0, 1, 0 },
        { 1, 0, 1, 1, 0 },
        { 0, 1, 0, 0, 1 },
        { 1, 1, 0, 0, 1 },
        { 0, 1, 1, 1, 0 } };

    for (int i = 0; i < count; i++)
        for (int j = 0; j < count; j++)
            if (adj[i][j] == 1)
                graph.addDirectedEdge(i, j, 1);
}

```

```
System.out.println("hamiltonianPath : " + hamiltonianPath(graph));
System.out.println("hamiltonianCycle : " + hamiltonianCycle(graph));

GraphAM graph2 = new GraphAM(count);
int[][] adj2 = {
{ 0, 1, 0, 1, 0 },
{ 1, 0, 1, 1, 0 },
{ 0, 1, 0, 0, 1 },
{ 1, 1, 0, 0, 0 },
{ 0, 1, 1, 0, 0 } };

for (int i = 0; i < count; i++)
for (int j = 0; j < count; j++)
if (adj2[i][j] == 1)
graph2.addEdge(i, j, 1);

System.out.println("hamiltonianPath : " + hamiltonianPath(graph2));
System.out.println("hamiltonianCycle : " + hamiltonianCycle(graph2));
}
```

Euler path and Euler Circuit

Eulerian Path is a path in the graph that visits every edge exactly once.

Eulerian Circuit is an Eulerian Path, which starts and ends on the same vertex. Or **Eulerian Circuit** is a path in the graph that visits every edge exactly one and it starts and ends on the same vertex.

A graph is called Eulerian if there is an Euler circuit in it. A graph is called Semi-Eulerian if there is an Euler Path in the graph. If there is no Euler path possible in the graph, then it is called non-Eulerian.

A graph is Eulerian if all the edges have even number of number of edges in it. A graph is Semi-Eulerian if it has exactly two vertices with odd number of edges or odd degree. In all other cases, the graph is Non-Eulerian.

Example 12.30: Check if the graph is Eulerian.

```
public static int isEulerian(Graph graph) {
    int count = graph.count;
    int odd;
    int[] inDegree;
    int[] outDegree;
    LinkedList<Edge> adl;
    // Check if all non - zero degree nodes are connected
    if (isConnected(graph) == false) {
        System.out.println("graph is not Eulerian");
        return 0;
    } else {
        // Count odd degree
        odd = 0;
        inDegree = new int[count];
        outDegree = new int[count];

        for (int i = 0; i < count; i++) {
            adl = graph.Adj.get(i);
            for (Edge adn : adl) {
                outDegree[i] += 1;
```

```
inDegree[adn.dest] += 1;
}
}
for (int i = 0; i < count; i++) {
if ((inDegree[i] + outDegree[i]) % 2 != 0) {
odd += 1;
}
}
}

if (odd == 0) {
System.out.println("graph is Eulerian");
return 2;
} else if (odd == 2) {
System.out.println("graph is Semi-Eulerian");
return 1;
} else {
System.out.println("graph is not Eulerian");
return 0;
}
}
```

Travelling Salesman Problem (TSP)

Problem: The travelling salesperson problem tries to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

Alternatively, find the shortest Hamiltonian circuit in a weighted connected graph. A cycle that passes through all the vertices of the graph exactly once.

Algorithm TSP

Select a city

MinTourCost = infinite

For (All permutations of cities) do

If(LengthOfPathSinglePermutation < MinTourCost)

 MinTourCost = LengthOfPath

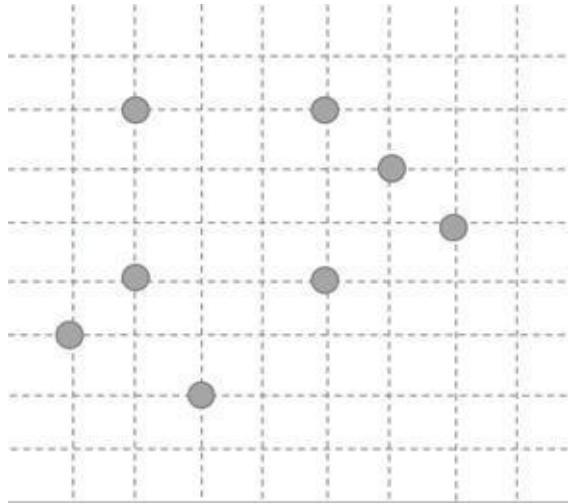
Total number of possible combinations = $(n-1)!$

Cost for calculating the path: $\Theta(n)$

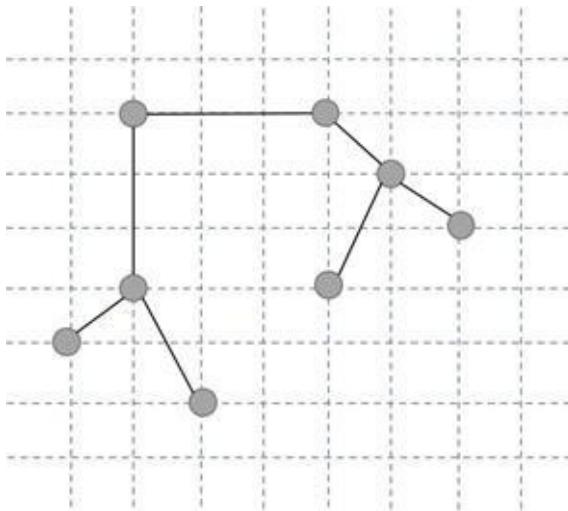
So the total cost of finding the shortest path: $\Theta(n!)$

It is an NP-Hard problem there is no efficient algorithm to find its solution. Even if some solution is given, it is equally hard to verify that this is a correct solution or not. However, some approximate algorithms can be used to find a good solution. We will not always get the best solution, but will get a good solution.

Our approximate algorithm is based on the minimum spanning tree problem. In which we have to construct a tree from a graph such that every node is connected by edges of the graph and the total sum of the cost of all the edges is minimum.



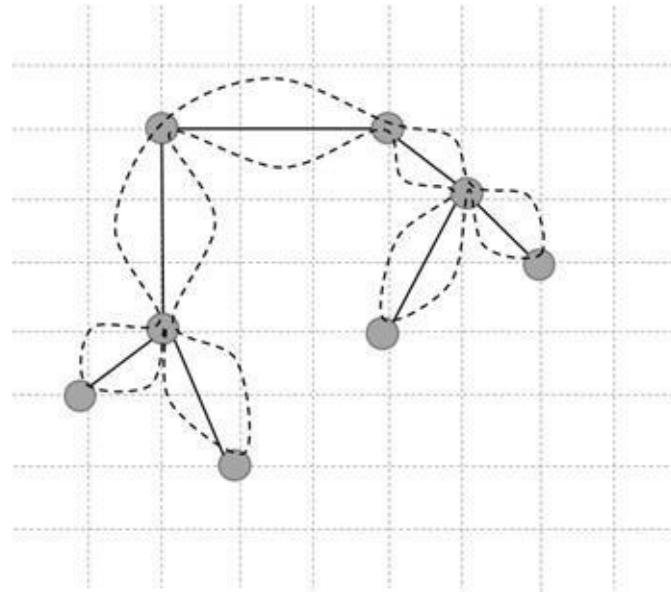
In the above diagram, we have a group of cities (each city is represented by a circle.) Which are located in the grid and the distance between the cities is same as per the actual distance. And there is a path from each city to another city which is a straight path from one to another.



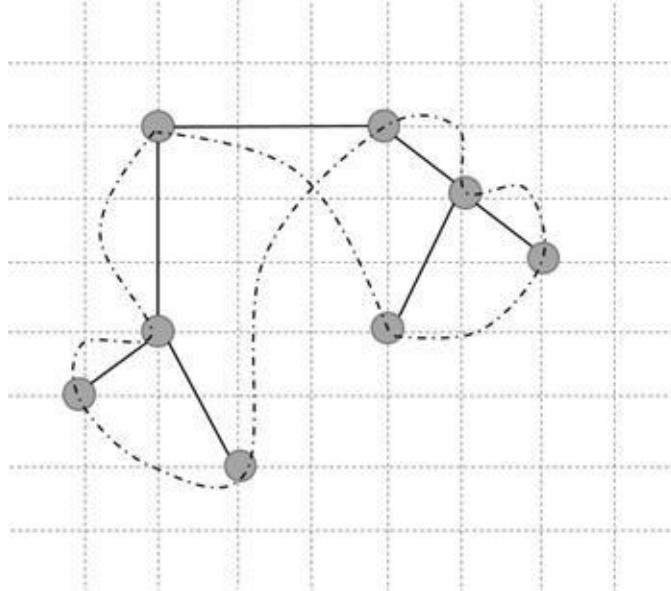
We have made a minimum spanning tree for the above city graph.

What we want to prove that the shortest path in a TSP will always be greater than the length of MST. Since in MST all nodes are connected to the next node, which is also the minimum distance from the group of node. Therefore, to make it a path without repeating the nodes we need to go directly from one node to other without following MST. At that point, when we are not following MST we are choosing an edge, which is greater, then the edges provided by MST. So TSP

path will always be greater than or equal to MST path.



Now let us take a path from starting node and traverse each node on the way given above and then come back to the starting node. The total cost of the path is 2MST . The only difference is that we are visiting many nodes multiple times.



Now let us change our traversal algorithm so that it will become TSP in our traversal, we do not visit an already visited node we will skip them and will visit the next unvisited node. In this algorithm, we will reach the next node by as shorter path. (The sum of the length of all the edges of a polygon is always

greater than a single edge.) Ultimately, we will get the TSP and its path length is not more than twice the optimal solution. Therefore, the proposed algorithm gives a good result.

Exercise

1. In the various path-finding algorithms, we have created a path array that just stores immediate parent of a node, print the complete path for it.
2. All the functions are implemented considering as if the graph is represented by adjacency list. Write all those functions for graph representation as adjacency matrix.
3. In a given start string, end string and a set of strings, find if there exists a path between the start string and end string via the set of strings.

A path exists if we can get from start string to end the string by changing (no addition/removal) only one character at a time. The restriction is that the new string generated after changing one character has to be in the set.

Start: "cog"

End: "bad"

Set: ["bag", "cag", "cat", "fag", "con", "rat", "sat", "fog"]

One of the paths: "cog" -> "fog" -> "fag" -> "bag" -> "bad"

CHAPTER 13: STRING ALGORITHMS

Introduction

String in Java language is an array of character. We use string algorithm in so many tasks, when we are using some copy-paste, some string replacement, and some string search. When we are using some dictionary program, we are using string algorithms. When we are searching something in google we are passing some information that is also a string and that will further convert and processed by google.

Note: This chapter is very important for the interview point of view as many interview problems are from this chapter.

String Matching

Every word processing program has a search function in which you can search all occurrences of any particular word in a long text file. For this, we need string-matching algorithms.

Problem: Search a pattern in a given text. The pattern is of length m and the text is of length n. Where $m < n$.

Approach 1: Brute Force Search

The brute force search algorithm will check the pattern at all possible value of “i” in the text where the value of “i” ranges from 0 to $n-m$. The pattern is compared with the text, character by character from left to right. When a mismatch is detected, then pattern is compared by shifting the compare window by one character.

Example 13.1:

```
int bruteForceSearch(String text, String pattern) {  
    return bruteForceSearch(text.toCharArray(), pattern.toCharArray());  
}  
  
int bruteForceSearch(char[] text, char[] pattern) {  
    int i = 0, j = 0;  
    final int n = text.length;  
    final int m = pattern.length;  
    while (i <= n - m) {  
        j = 0;  
        while (j < m && pattern[j] == text[i + j]) {  
            j++;  
        }  
        if (j == m) {  
            return (i);  
        }  
        i++;  
    }  
    return -1;  
}
```

Worst case time complexity of the algorithm is $O(m*n)$, we get the pattern at the end of the text or we do not get the pattern at all.

Approach 2: Robin-Karp algorithm

Robin-Karp algorithm is somewhat similar to the brute force algorithm in which the pattern is compared to each portion of the text of length m . Instead of comparing pattern, character by character its hash code is compared. The hash code of the pattern is compared with the hash code of the text window. We try to keep the hash code as unique as possible. So that when hash code matches then the text should also match.

The two features of good hash code are:

- The collision should be excluded as much as possible. A collision occurs when hash code matches, but the pattern does not.
- The hash code of text must be calculated in constant time.

In this algorithm, hash code of some window is calculated from hash code of previous window in constant time. In the start, hash value of text of length m is calculated. We compare its hash code with the hash code of pattern string. To get hash code of next window we exclude one character and include next character. The portion of text that need to be compared moves as a window of characters. For each window calculation of hash is done in constant time, one member leaves the window and a new number enters the window.

Multiplication by 2 is same as left shift operation. Multiplication by 2^{m-1} is same as left shift $m-1$ times. If the pattern is “ m ” character long. Then when we want to remove the left most character from hash we will subtract its ASCII value multiplied by 2^{m-1} . We shift the whole hash calculation by multiplying it by 2. Finally, the hash value of the new window is calculated by adding the ASCII value of right most element of this window.

We do not want to do large multiplication operations so modular operation with a prime number is used.

Example 13.2:

```
int robinKarp(String text, String pattern) {
```

```
        return robinKarp(text.toCharArray(), pattern.toCharArray());
    }

int robinKarp(char[] text, char[] pattern) {
    int n = text.length;
    int m = pattern.length;
    int i, j;
    int prime = 101;
    int powm = 1;
    int TextHash = 0, PatternHash = 0;
    if (m == 0 || m > n) {
        return -1;
    }

    for (i = 0; i < m - 1; i++) {
        powm = (powm << 1) % prime;
    }

    for (i = 0; i < m; i++) {
        PatternHash = ((PatternHash << 1) + pattern[i]) % prime;
        TextHash = ((TextHash << 1) + text[i]) % prime;
    }

    for (i = 0; i <= (n - m); i++) {
        if (TextHash == PatternHash) {
            for (j = 0; j < m; j++) {
                if (text[i + j] != pattern[j]) {
                    break;
                }
            }
            if (j == m)
                return i;
        }
        TextHash = (((TextHash - text[i] * powm) << 1) + text[i + m]) % prime;
        if (TextHash < 0) {
            TextHash = (TextHash + prime);
        }
    }
}
```

```

    }
    return -1;
}

```

Worst case time complexity of the algorithm is $O(n)$.

Approach 3: Knuth-Morris-Pratt algorithm

There is an inefficiency in the brute force method of string matching. After a shift of the pattern, the brute force algorithm forgotten all the information about the previous matched symbols. This is because of which its worst case time complexity is $O(mn)$.

The Knuth-Morris-Pratt algorithm makes use of this information that is computed in the previous comparison. It never re compares the whole text. It uses preprocessing of the pattern. The preprocessing takes $O(m)$ time and whole algorithm is $O(n)$

Preprocessing step: we try to find the border of the pattern at a different prefix of the pattern.

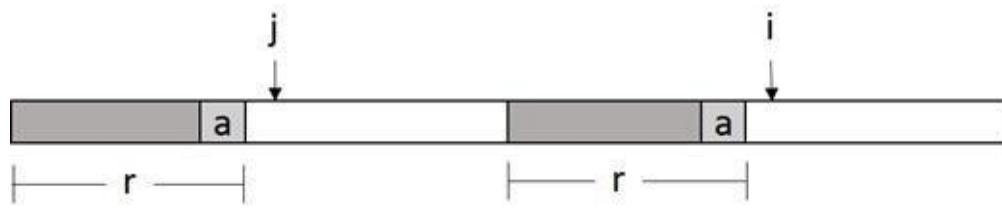
A **prefix** is a string that comes at the beginning of a string.

A **proper prefix** is a prefix that is not the complete string. Its length is less than the length of the string.

A **suffix** is a string that comes at the end of a string.

A **proper suffix** is a suffix that is not a complete string. Its length is less than the length of the string.

A **border** is a string that is both proper prefix and a proper suffix.



Example 13.3:

```

void KMPPreprocess(char[] pattern, int[] ShiftArr) {
    final int m = pattern.length;

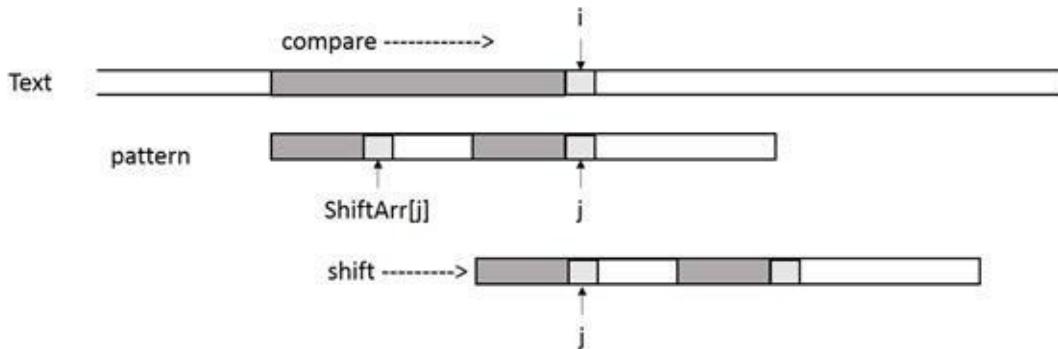
```

```

int i = 0, j = -1;
ShiftArr[i] = -1;
while (i < m) {
    while (j >= 0 && pattern[i] != pattern[j]) {
        j = ShiftArr[j];
    }
    i++;
    j++;
    ShiftArr[i] = j;
}

```

We have to loop outer loop for the text and inner loop for the pattern when we have matched the text and pattern mismatch, we shift the text such that the widest border is considered and then the rest of the pattern matching is resumed after this shift. If again a mismatch happens then the next mismatch is taken.



Example 13.4:

```

int KMP(String text, String pattern) {
    return KMP(text.toCharArray(), pattern.toCharArray());
}

```

```

int KMP(char[] text, char[] pattern) {
    int i = 0, j = 0;
    final int n = text.length;
    final int m = pattern.length;
    int[] ShiftArr = new int[m + 1];
    KMPPreprocess(pattern, ShiftArr);
    while (i < n) {

```

```

        while (j >= 0 && text[i] != pattern[j])
            j = ShiftArr[j];
            i++;
            j++;
        if (j == m) {
            return (i - m);
        }
    }
    return -1;
}

/* Testing Code */
public static void main(String[] args) {
    String st1 = "hello, world!";
    String st2 = "world";
    Algo algo = new Algo();
    System.out.println("BruteForceSearch return : " +
algo.bruteForceSearch(st1, st2));
    System.out.println("RobinKarp return : " + algo.robinKarp(st1, st2));
    System.out.println("KMP return : " + algo.KMP(st1, st2));
}

```

Problem: Use the same KMP algorithm to find the number of occurrences of the pattern in a text.

Example 13.5:

```

int KMPFindCount(char[] text, char[] pattern) {
    int i = 0, j = 0, count = 0;
    final int n = text.length;
    final int m = pattern.length;
    int[] ShiftArr = new int[m + 1];
    KMPPreprocess(pattern, ShiftArr);
    while (i < n) {
        while (j >= 0 && text[i] != pattern[j]) {
            j = ShiftArr[j];
        }
        i++;
    }
}
```

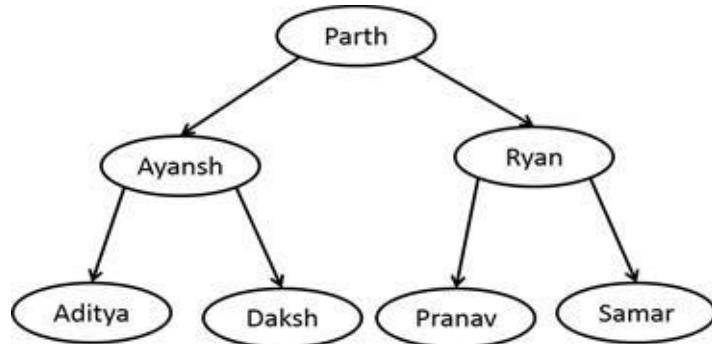
```
j++;
if (j == m) {
    count++;
    j = ShiftArr[j];
}
}
return count;
}
```

Dictionary / Symbol Table

A symbol table is a mapping between a string (key) and a value that can be of any type. A value can be an integer such as occurrence count, dictionary meaning of a word and so on. Dictionary can be implemented in various ways. We will be studying Binary Search Tree of strings, Hash-Table, Tries and Ternary Search Tree.

Binary Search Tree (BST) for Strings

Binary Search Tree (BST) is the simplest way to implement symbol table. Simple `strcmp()` function can be used to compare two strings. If all the keys are random and the tree is balanced, then on an average key lookup can be done in $O(\log n)$ time.



Binary Search Tree as Dictionary

Below is an implementation of binary search tree to store string as key. This will keep track of the occurrence count of words in a text.

Example 13.6:

```
public class StringTree {  
    Node root = null;  
  
    class Node {  
        String value;  
        int count;  
        Node lChild;
```

```
Node rChild;
};

// Other Methods.

public void print() {
print(root); }

public void print(Node curr) /* pre order */
{
if (curr != null) {
System.out.print(" value is ::" + curr.value);
System.out.println(" count is :: " + curr.count);
print(curr.lChild); print(curr.rChild); }
}

public void add(String value) {
root = add(value, root);
}

Node add(String value, Node curr) {
int compare;
if (curr == null) {
curr = new Node();
curr.value = value;
curr.lChild = curr.rChild = null;
curr.count = 1;
} else {
compare = curr.value.compareTo(value);
if (compare == 0)
curr.count++;
else if (compare == 1)
curr.lChild = add(value, curr.lChild);
else
curr.rChild = add(value, curr.rChild);
}
return curr;
```

```
}

boolean find(String value) {
    boolean ret = find(root, value);
    System.out.println("Find " + value + " Return " + ret);
    return ret;
}

boolean find(Node curr, String value) {
    int compare;
    if (curr == null)
        return false;
    compare = curr.value.compareTo(value);
    if (compare == 0)
        return true;
    else {
        if (compare == 1)
            return find(curr.lChild, value);
        else
            return find(curr.rChild, value);
    }
}

int frequency(String value) {
    return frequency(root, value);
}

int frequency(Node curr, String value) {
    int compare;
    if (curr == null)
        return 0;

    compare = curr.value.compareTo(value);
    if (compare == 0)
        return curr.count;
    else {
        if (compare > 0)
```

```

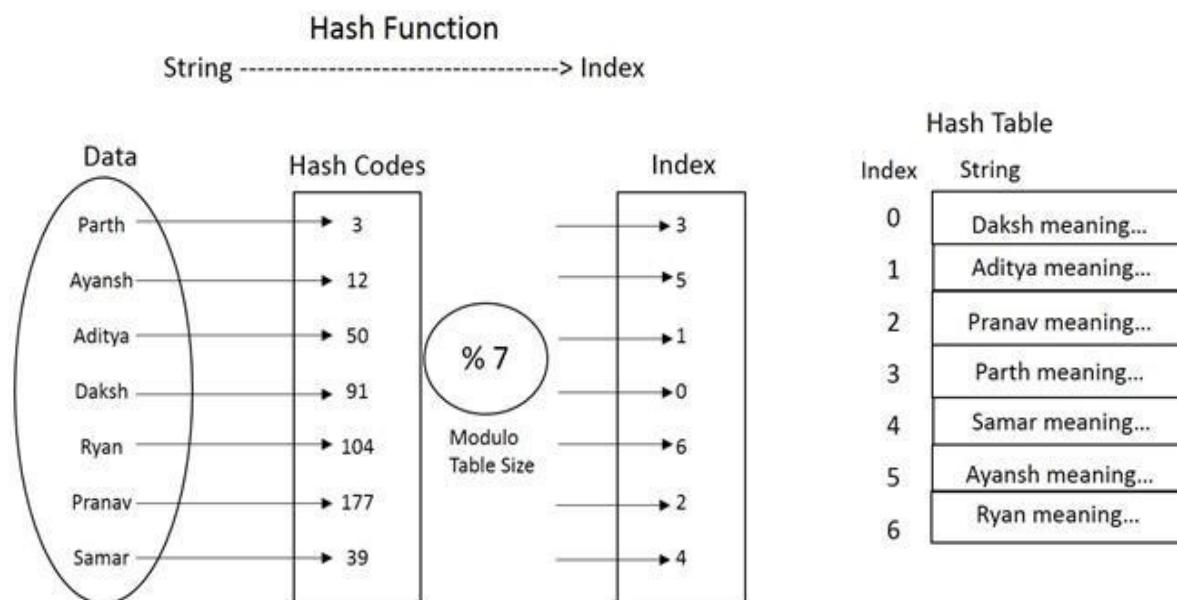
return frequency(curr.lChild, value);
else
return frequency(curr.rChild, value);
}
}

void freeTree() {
root = null;
}
}

```

Hash-Table

The Hash-Table is another data structure that can be used for symbol table implementation. Below Hash-Table diagram, we can see the name of that person is taken as key, and their meaning is the value of the search. The first key is converted into a hash code by passing it to appropriate hash function. Inside hash function the size of Hash-Table is also passed, which is used to find the actual index where values will be stored. Finally, the value, which is the meaning of name, is stored in the Hash-Table, or you can store a reference to the string which stores meaning is stored into the Hash-Table.



Hash-Table has an excellent lookup of $O(1)$.

Let us suppose we want to implement autocomplete the box feature of Google search. When you type some string to search in google search, it propose some complete string even before you have done typing. BST cannot solve this problem as related strings can be in both right and left subtree.

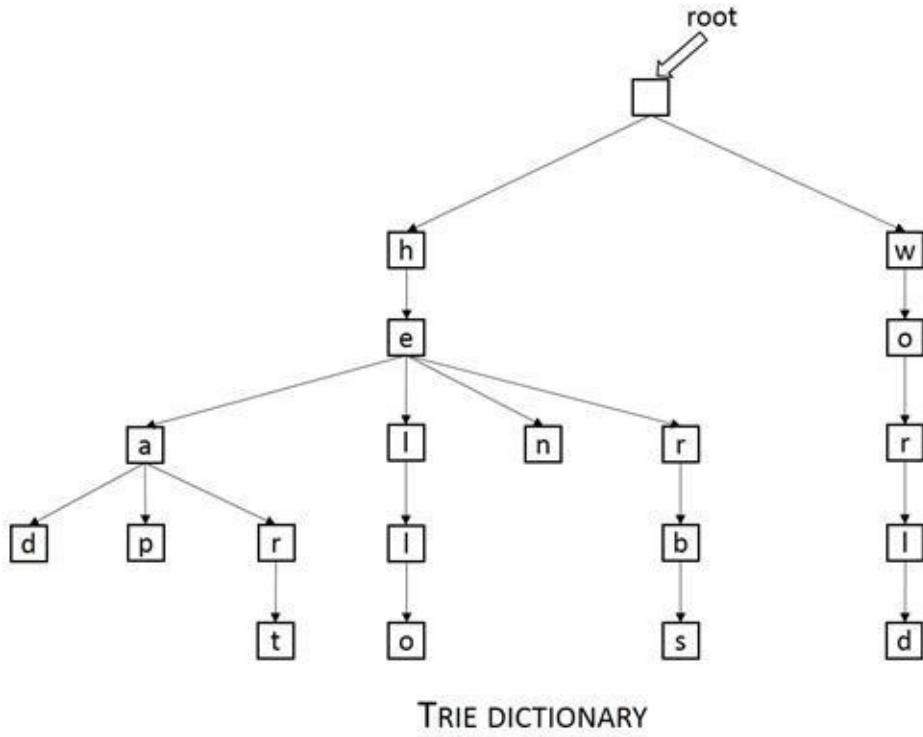
The Hash-Table is also not suited for this job. One cannot perform a partial match or range query on a Hash-Table. Hash function transforms string to a number. Moreover, a good hash function will give a distributed hash code even for partial string and there is no way to relate two strings in a Hash-Table.

Trie and Ternary Search tree are a special kind of tree that solves partial match and range query problem efficiently.

Trie

Trie is a tree, in which we store only one character at each node. The final key value pair is stored in the leaves. Each node has R children, one for each possible character. For simplicity purpose, let us consider that the character set is 26, corresponds to different characters of English alphabets.

Trie is an efficient data structure. Using Trie, we can search the key in $O(M)$ time. Where M is the maximum string length. Trie is also suitable for solving partial match and range query problems.



Example 13.7:

```

public class Trie {
    private static final int CharCount = 26;
    Node root = null;

    private class Node {
        boolean isLastChar;
        Node[] child;

        public Node(char c) {
            child = new Node[CharCount];
            for (int i = 0; i < CharCount; i++) {
                child[i] = null;
            }
            isLastChar = false;
        }
    };

    public Trie() {
        root = new Node(' '); // first node with dummy value.
    }
}

```

```
}

Node add(String str) {
    if (str == null) {
        return root;
    }
    return add(root, str.toLowerCase(), 0);
}

Node add(Node curr, String str, int index) {
    if (curr == null) {
        curr = new Node(str.charAt(index - 1));
    }

    if (str.length() == index) {
        curr.isLastChar = true;
    } else {
        curr.child[str.charAt(index) - 'a'] = add(curr.child[str.charAt(index) - 'a'], str,
index + 1);
    }
    return curr;
}

void remove(String str) {
    if (str == null) {
        return;
    }
    str = str.toLowerCaseremove(root, str, 0);
}

void remove(Node curr, String str, int index) {
    if (curr == null) {
        return;
    }
    if (str.length() == index) {
        if (curr.isLastChar) {
            curr.isLastChar = false;
```

```

    }
    return;
}
remove(curr.child[str.charAt(index) - 'a'], str, index + 1);
}

boolean find(String str) {
if (str == null) {
return false;
}
str = str.toLowercase(); return find(root, str, 0);
}

boolean find(Node curr, String str, int index) {
if (curr == null) {
return false;
}
if (str.length() == index) {
return curr.isLastChar;
}
return find(curr.child[str.charAt(index) - 'a'], str, index + 1);
}

public static void main(String[] args) {
Trie t = new Trie();
String a = "hemant";
String b = "heman";
String c = "hemantjain";
String d = "jain";
t.add(a); t.add(d); System.out.println(t.find(a)); t.remove(a); t.remove(d);
System.out.println(t.find(a)); System.out.println(t.find(c));
System.out.println(t.find(d)); }
}

```

Output:

true
false

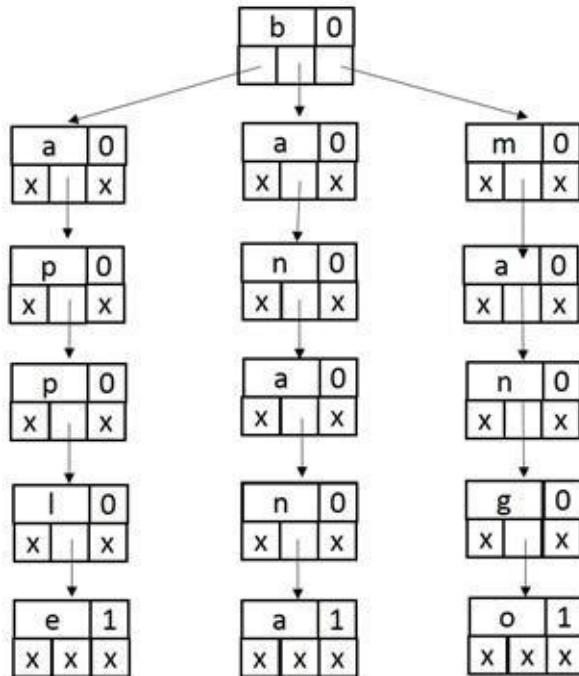
```
false  
false
```

Ternary Search Trie/ Ternary Search Tree

Tries have a very good search performance of $O(M)$ where M is the maximum size of the search string. However, tries have a very high space requirement. In every node, Trie contains pointers to multiple nodes which are pointer corresponds to possible characters of the key. To avoid this high space requirement Ternary Search Trie (TST) is used.

A TST avoids heavy space requirement of traditional Trie, still keeping many of its advantages. In a TST, each node contains a character, an end of key indicator and three pointers. The three pointers are corresponding to current char hold by the node(equal), characters less than and character greater than.

Time Complexity of ternary search tree operation is proportional to the height of the ternary search tree. In the worst case, we need to traverse up to 3 times the length of largest string. However, this case is rare. Therefore, TST is a very good solution for implementing Symbol Table, Partial match and range query.



Ternary Search Tree

Example 13.8:

```
public class TST {
    Node root;

    private class Node {
        char data;
        boolean isLastChar;
        Node left, equal, right;
    }

    private Node(char d) {
        data = d;
        isLastChar = false;
        left = equal = right = null;
    }
}

public void add(String word) {
    root = add(root, word, 0);
}
```

```
private Node add(Node curr, String word, int wordIndex) {  
    if (curr == null)  
        curr = new Node(word.charAt(wordIndex));  
    if (word.charAt(wordIndex) < curr.data)  
        curr.left = add(curr.left, word, wordIndex);  
    else if (word.charAt(wordIndex) > curr.data)  
        curr.right = add(curr.right, word, wordIndex);  
    else {  
        if (wordIndex < word.length() - 1)  
            curr.equal = add(curr.equal, word, wordIndex + 1);  
        else  
            curr.isLastChar = true;  
    }  
    return curr;  
}
```

```
private boolean find(Node curr, String word, int wordIndex) {  
    if (curr == null)  
        return false;  
    if (word.charAt(wordIndex) < curr.data)  
        return find(curr.left, word, wordIndex);  
    else if (word.charAt(wordIndex) > curr.data)  
        return find(curr.right, word, wordIndex);  
    else {  
        if (wordIndex == word.length() - 1)  
            return curr.isLastChar;  
        return find(curr.equal, word, wordIndex + 1);  
    }  
}
```

```
public boolean find(String word) {  
    boolean ret = find(root, word, 0);  
    System.out.print(word + " :: ");  
    if (ret)  
        System.out.println(" Found ");  
    else  
        System.out.println("Not Found ");
```

```
return ret;
}

public static void main(String[] args) {
TST tt = new TST();
tt.add("banana"); tt.add("apple"); tt.add("mango");
System.out.println("\nSearch results are:");
tt.find("apple"); tt.find("banana"); tt.find("mango"); tt.find("grapes");
}
```

Output:

```
Search results are:
```

```
apple :: Found
```

```
banana :: Found
```

```
mango :: Found
```

```
grapes :: Not Found
```

Problems in String

Regular Expression Matching

Problem: Implement regular expression matching with the support of ‘?’ and ‘*’ special character.

‘?’ Matches any single character.

‘*’ Matches zero or more of the preceding element.

Example 13.9:

```
public static boolean matchExpUtil(char[] exp, char[] str, int i, int j) {  
    if (i == exp.length && j == str.length) {  
        return true;  
    }  
    if ((i == exp.length && j != str.length) || (i != exp.length && j == str.length)) {  
        return false;  
    }  
    if (exp[i] == '?' || exp[i] == str[j]) {  
        return matchExpUtil(exp, str, i + 1, j + 1);  
    }  
    if (exp[i] == '*') {  
        return matchExpUtil(exp, str, i + 1, j) || matchExpUtil(exp, str, i, j + 1) ||  
matchExpUtil(exp, str, i + 1, j + 1);  
    }  
    return false;  
}  
  
public static boolean matchExp(String exp, String str) {  
    return matchExpUtil(exp.toCharArray(), str.toCharArray(), 0, 0);  
}  
/* Testing code */  
public static void main() {  
    System.out.println(matchExp("*llo,?World?", "Hello, World!"));  
}
```

Time complexity: O(n), where n is length of text string.

Order Matching

Problem: In given long text string and a pattern string, find if the characters of pattern string are in the same order in text string. Eg. Text String: ABCDEFGHIJKLMNOPQRSTUVWXYZ Pattern string: JOST

Example 13.10:

```
public static boolean match(String src, String ptn) {  
    char[] source = src.toCharArray();  
    char[] pattern = ptn.toCharArray();  
    int iSource = 0;  
    int iPattern = 0;  
    int sourceLen = source.length;  
    int patternLen = pattern.length;  
  
    for (iSource = 0; iSource < sourceLen; iSource++) {  
        if (source[iSource] == pattern[iPattern]) {  
            iPattern++;  
        }  
        if (iPattern == patternLen) {  
            return true;  
        }  
    }  
    return false;  
}  
  
public static void main() {  
    System.out.println(match("harrypottermustnotgotoschool", "pottergo"));  
}
```

Time complexity: $O(n)$, where n is length of text string.

ASCII to Integer Conversion

Problem: Write a function that takes integer as an ascii char list and converts it into an int.

Example 13.11:

```
public static int myAtoi(String str) {  
    int value = 0;  
    int size = str.length();  
  
    for (int i = 0; i < size; i++) {  
        char ch = str.charAt(i);  
        value = (value << 3) + (value << 1) + (ch - '0');  
    }  
    return value;  
}  
  
public static void main() {  
    System.out.println(myAtoi("1000"));  
}
```

Time complexity: O(n), where n is length of ascii string.

Unique Characters

Problem: Write a function that will take a string as input and return true if it contains all unique characters, else return false.

Example 13.12:

```
public static boolean isUniqueChar(String str) {  
    int[] bitarr = new int[26];  
    int index;  
    for (int i = 0; i < 26; i++) {  
        bitarr[i] = 0;  
    }  
    int size = str.length();  
    for (int i = 0; i < size; i++) {  
        char c = str.charAt(i);  
        if ('A' <= c && 'Z' >= c) {  
            index = (c - 'A');  
        } else if ('a' <= c && 'z' >= c) {  
            index = (c - 'a');
```

```

} else {
    System.out.println("Unknown Char!\n");
    return false;
}
if (bitarr[index] != 0) {
    System.out.println("Duplicate detected!");
    return false;
}
bitarr[index] += 1;
}
System.out.println("No duplicate detected!");
return true;
}

public static void main() {
    System.out.println(isUniqueChar("apple"));
    System.out.println(isUniqueChar("apple"));
}

```

Time complexity: O(n), where n is number of character in input text.

ToUpper

Problem: Write a function to convert small case character of English alphabets to large case character.

Example 13.13:

```

public static char ToUpper(char s) {
    if (s >= 97 && s <= (97 + 25)) {
        s = (char) (s - 32);
    }
    return s;
}

```

Time complexity: O(1).

ToLower

Problem: Write a function to convert large case character of English alphabets to small case character.

Example 13.14:

```
public static char ToLower(char s) {  
    if (s >= 65 && s <= (65 + 25)) {  
        s = (char) (s + 32);  
    }  
    return s;  
}
```

Time complexity: O(1).

Permutation Check

Problem: Write a function to check if two strings are permutation of each other.

Example 13.15:

```
public static boolean isPermutation(String s1, String s2) {  
    int[] count = new int[256];  
    int length = s1.length();  
    if (s2.length() != length) {  
        System.out.println("is permutation return false\n");  
        return false;  
    }  
    for (int i = 0; i < 256; i++) {  
        count[i] = 0;  
    }  
    for (int i = 0; i < length; i++) {  
        char ch = s1.charAt(i);  
        count[ch]++;  
        ch = s2.charAt(i);  
        count[ch]--;  
    }  
    for (int i = 0; i < length; i++) {
```

```

if (count[i] != 0) {
    System.out.println("is permutation return false\n");
    return false;
}
}
System.out.println("is permutation return true\n");
return true;
}

public static void main() {
    System.out.println(isPermutation("apple", "plepa"));
}

```

Time complexity: O(n), where n is number of character in text.

Palindrome Check

Problem: Find if the string is a palindrome or not

Example 13.16:

```

public static boolean isPalindrome(String str) {
    int i = 0, j = str.length() - 1;
    while (i < j && str.charAt(i) == str.charAt(j)) {
        i++;
        j--;
    }
    if (i < j) {
        System.out.println("String is not a Palindrome");
        return false;
    } else {
        System.out.println("String is a Palindrome");
        return true;
    }
}

public static void main() {
    System.out.println(isPalindrome("hello"));
}

```

```
    System.out.println(isPalindrome("eoloe"));
}
```

Time Complexity is **O(n)** and Space Complexity is **O(1)**

Power function

Problem: Write a function which will calculate x^n , Taking x and n as argument.

Example 13.19:

```
public static int pow(int x, int n) {
    int value;
    if (n == 0) {
        return (1);
    } else if (n % 2 == 0) {
        value = pow(x, n / 2);
        return (value * value);
    } else {
        value = pow(x, n / 2);
        return (x * value * value);
    }
}

public static void main() {
    System.out.println(pow(5, 2));
}
```

Time complexity: $O(\log N)$, where N is exponent in the desired value .

String Compare function

Problem: Write a function strcmp() to compare two strings. The function return values should be:

- The return value is 0 indicates that both first and second strings are equal.
- The return value is negative it indicates that the first string is less than the second string.
- The return value is positive it indicates that the first string is greater than

the second string.

Example 13.20:

```
public static int mystrcmp(String a, String b) {  
    int index = 0;  
    int len1 = a.length();  
    int len2 = b.length();  
    int minlen = len1;  
    if (len1 > len2) {  
        minlen = len2;  
    }  
  
    while (index < minlen && a.charAt(index) == b.charAt(index)) {  
        index++;  
    }  
  
    if (index == len1 && index == len2) {  
        return 0;  
    } else if (len1 == index) {  
        return -1;  
    } else if (len2 == index) {  
        return 1;  
    } else {  
        return a.charAt(index) - b.charAt(index);  
    }  
}  
  
public static void main() {  
    System.out.println(mystrcmp("abs", "abs"));  
}
```

Time complexity: O(n), where n is length of smaller string.

Reverse String

Problem: Reverse all the characters of a string.

Example 13.22:

```

public static String reverseString(String str) {
    char[] a = str.toCharArray();
    reverseStringUtil(a);
    String expn = new String(a);
    return expn;
}

public static void reverseStringUtil(char[] a) {
    int lower = 0;
    int upper = a.length - 1;
    char tempChar;
    while (lower < upper) {
        tempChar = a[lower];
        a[lower] = a[upper];
        a[upper] = tempChar;
        lower++;
        upper--;
    }
}

public static void reverseStringUtil(char[] a, int lower, int upper) {
    char tempChar;
    while (lower < upper) {
        tempChar = a[lower];
        a[lower] = a[upper];
        a[upper] = tempChar;
        lower++;
        upper--;
    }
}

```

Time complexity: O(n), where n is number of characters in input string.

Reverse Words

Problem: Reverse order of words in a sentence.

Example 13.23:

```
public static String reverseWords(String str) {  
    char[] a = str.toCharArray();  
    int length = a.length;  
    int lower = 0, upper = -1;  
    for (int i = 0; i <= length; i++) {  
        if (i == length || a[i] == ' ') {  
            reverseStringUtil(a, lower, upper);  
            lower = i + 1;  
            upper = i;  
        } else {  
            upper++;  
        }  
    }  
    reverseStringUtil(a, 0, length - 1);  
    String expn = new String(a);  
    return expn;  
}  
  
public static void main() {  
    System.out.println(reverseString("apple"));  
    System.out.println(reverseWords("hello world"));  
}
```

Time complexity: O(n), where n is number of characters in input string.

Print Anagram

Problem: Given a string as character list, print all the anagram of the string.

Example 13.24:

```
public static void printAnagram(String str) {  
    char[] a = str.toCharArray();  
    int n = a.length;  
    printAnagram(a, n, n);  
}
```

```

public static void printAnagram(char[] a, int max, int n) {
    if (max == 1) {
        System.out.println(a);
    }
    char temp;
    for (int i = -1; i < max - 1; i++) {
        if (i != -1) {
            temp = a[i];
            a[i] = a[max - 1];
            a[max - 1] = temp;
        }
        printAnagram(a, max - 1, n);
        if (i != -1) {
            temp = a[i];
            a[i] = a[max - 1];
            a[max - 1] = temp;
        }
    }
}

public static void main() {
    printAnagram("123");
}

```

Time complexity: $O(n!)$, where n is number of characters in input string.

Shuffle String

Problem: Write a program to convert list ABCDE12345 to A1B2C3D4E5.

Example 13.25:

```

public static void shuffle(String str) {
    char[] ar = str.toCharArray();
    int n = ar.length / 2;
    int count = 0;
    int k = 1;
    char temp = '\0';

```

```

for (int i = 1; i < n; i = i + 2) {
    temp = ar[i];
    k = i;
    do {
        k = (2 * k) % (2 * n - 1);
        temp ^= ar[k] ^= temp ^= ar[k];
        count++;
    } while (i != k);
    if (count == (2 * n - 2)) {
        break;
    }
}
}

public static void main() {
    shuffle("ABCDE12345");
}

```

Time complexity: $O(n^2)$, where n is number of characters in input string.

Binary Addition

Problem: Given two binary string, find the sum of these two binary strings.

Example 13.26:

```

public static char[] addBinary(String firstStr, String secondStr) {
    char[] first = firstStr.toCharArray();
    char[] second = secondStr.toCharArray();
    int size1 = first.length;
    int size2 = second.length;
    int totalIndex;
    char[] total;

    if (size1 > size2) {
        total = new char[size1 + 2];
        totalIndex = size1;
    } else {

```

```

total = new char[size2 + 2];
totalIndex = size2;
}

total[totalIndex + 1] = '\0';
int carry = 0;
size1--;
size2--;

while (size1 >= 0 || size2 >= 0) {
    int firstValue = (size1 < 0) ? 0 : first[size1] - '0';
    int secondValue = (size2 < 0) ? 0 : second[size2] - '0';
    int sum = firstValue + secondValue + carry;
    carry = sum >> 1;
    sum = sum & 1;
    total[totalIndex] = (sum == 0) ? '0' : '1';
    totalIndex--;
    size1--;
    size2--;
}

total[totalIndex] = (carry == 0) ? '0' : '1';
return total;
}

public static void main() {
    System.out.println(addBinary("1000", "1111111"));
}

```

Time complexity: O(n), where n is number of characters larger size input string.

Exercise

1. In given string, find the longest substring without repeated characters.
2. The function `memset()` copies `ch` into the first '`n`' characters of the string
3. Serialize a collection of strings into a single string and de serializes the string into that collection of strings.
4. Write a smart input function, which takes 20 characters as input from the user.
Without cutting some word.
User input: "Harry Potter must not go"
First 20 chars: "Harry Potter must no"
Smart input: "Harry Potter must"
5. Write a code that finds if a string is a palindrome and it should return true for below inputs too.
Stella won no wallets.
No, it is open on one position.
Rise to vote, Sir.
Won't lovers revolt now?
6. Write an ASCII to integer function, which ignores the non-integral character and gives the integer. For example, if the input is "12AS5" it should return 125.
7. Write code that would parse a Bash brace expansion.
Example: the expression "(a, b, c) d, e" and would give output all the possible strings: ad, bd, cd, e
8. In given string write a function to return the length of the longest substring with only unique characters
9. [Replace all occurrences of "a" with "the"](#)
10. Replace all occurrences of "%20" with ' '.
E.g. Input: www.Hello%20World.com

Output: www.Hello World.com

11. Write an expansion function that will take an input string like "1..5,8,11..14,18,20,26..30" and will print "1, 2 , 3, 4, 5, 8, 11, 12, 13, 14, 18, 20, 26, 27, 28, 29, 30"
12. Suppose you have a string like "Thisisasentence". Write a function that would separate these words. Moreover, will print whole sentence with spaces.
13. In given three string str1, str2 and str3. Write a complement function to find the smallest sub-sequence in str1 which contains all the characters in str2 and but not those in str3.
14. In given two strings A and B, find whether any anagram of string A is a sub string of string B.
For eg: If A = xyz and B = afdgzyxksldfm then the program should return true.
15. In given string, find whether it contains any permutation of another string.
For example, given "abcdefgh" and "ba", the function should return true, because "abcdefgh" has substring "ab", which is a permutation of the given string "ba".
16. In give algorithm which removes the occurrence of “a” by “bc” from a string? The algorithm must be in-place.
17. In given string "1010101010" in base2 convert it into string with base4. Do not use an extra space.
18. In Binary Search tree to store strings, delete() function is not implemented, implement it.
19. If you implement delete() function, then you need to make changes in find() function. Do the needful.

CHAPTER 14: ALGORITHM DESIGN TECHNIQUES

Introduction

In real life, when we are asked to do some work, we try to correlate it with our experience and then try to solve it. Similarly, when we get a new problem to solve. We first try to find the similarity of the current problem with some problems for which we already know the solution. Then solve the current problem and get our desired result.

This method provides following benefits:

- 1) It provides a template for solving a wide range of problems.

- 2) It provides us an idea of the suitable data structure for the problem.
- 3) It helps us in analyzing space and Time Complexity of algorithms.

In the previous chapters, we have used various algorithms to solve different kind of problems. In this chapter, we will read about various techniques of solving algorithmic problems.

Various Algorithm design techniques are:

- 1) Brute Force
- 2) Greedy Algorithms
- 3) Divide-and-Conquer, Decrease-and-Conquer
- 4) Dynamic Programming
- 5) Reduction / Transform-and-Conquer
- 6) Backtracking and Branch-and-Bound

Brute Force Algorithm

Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem. It is useful for solving small size dataset problem.

Some examples of brute force algorithms are:

- Bubble-Sort
- Selection-Sort
- Sequential search in an array
- Computing $\text{pow}(a, n)$ by multiplying a, n times.
- Convex hull problem
- String matching
- Exhaustive search: Traveling salesman, Knapsack, and Assignment problems

Greedy Algorithm

Greedy algorithms are generally used to solve optimization problems. In greedy algorithm, solution is constructed through a sequence of steps. At each step, choice is made which is locally optimal.

Note: Greedy algorithms does not always give optimum solution.

Some examples of greedy algorithms are:

- Minimal spanning tree: Prim's algorithm, Kruskal's algorithm
- Dijkstra's algorithm for single-source shortest path problem
- Greedy algorithm for the Knapsack problem
- The coin exchange problem
- Huffman trees for optimal encoding

Divide-and-Conquer, Decrease-and-Conquer

Divide-and-Conquer algorithms involve basic three steps. First, split the problem into several smaller sub-problems. Second, solve each sub-problem. Finally, combine the sub-problems results to produce the desired result.

In divide-and-conquer the size of the problem is reduced by a factor (half, one-third, etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.

Examples of divide-and-conquer algorithms:

- Merge-Sort algorithm (using recursion)
- Quicksort algorithm (using recursion)
- Computing the length of the longest path in a binary tree (using recursion)
- Computing Fibonacci numbers (using recursion)
- Quick-hull

Examples of decrease-and-conquer algorithms:

- Computing $\text{pow}(a, n)$ by calculating $\text{pow}(a, n/2)$ using recursion.
- Binary search in a sorted list (using recursion)
- Searching in BST
- Insertion-Sort
- Graph traversal algorithms (DFS and BFS)
- Topological sort
- Warshall's algorithm (using recursion)
- Permutations (Minimal change approach, Johnson-Trotter algorithm)
- Computing a median, Topological sorting, Fake-coin problem (Ternary search)

Consider the problem of exponentiation Compute x^n

Brute Force:	n-1 multiplications
Divide and conquer:	$T(n) = 2*T(n/2) + 1 = n-1$
Decrease by one:	$T(n) = T(n-1) + 1 = n-1$
Decrease by constant factor:	$\begin{aligned} T(n) &= T(n/a) + a-1 \\ &= (a-1)n \\ &= n \text{ when } a = 2 \end{aligned}$

Dynamic Programming

While solving problems using Divide-and-Conquer method, there may be a case when recursively sub-problems can result in the same computation being performed multiple times. This problem arises when there are identical sub-problems arise repeatedly in a recursion.

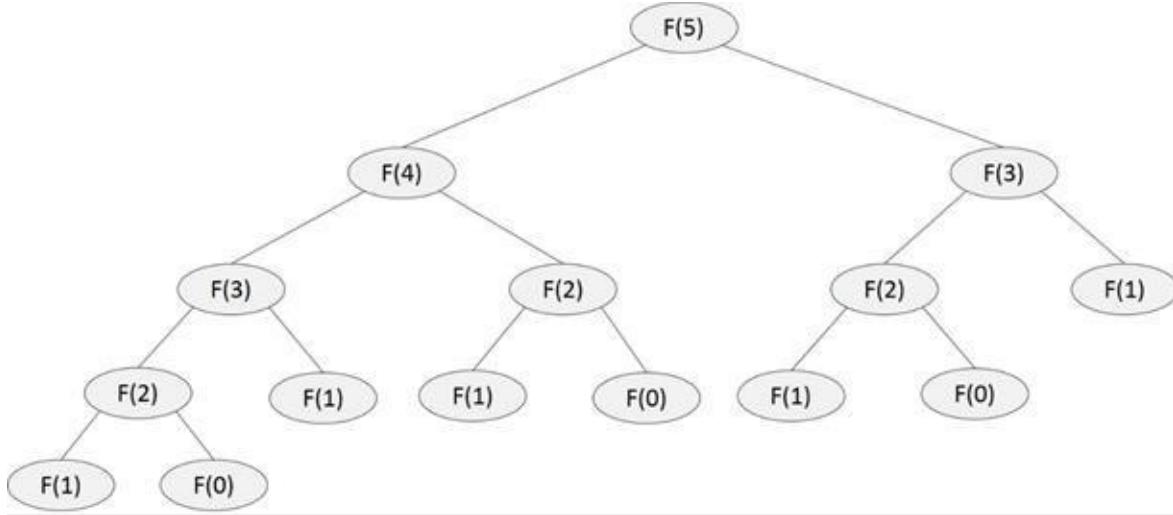
Dynamic programming is used to avoid the requirement of repeated calculation of same sub-problem. In this method, we usually store the result of sub - problems in a table and refer that table to find if we have already calculated the solution of sub - problems before calculating it again.

Dynamic programming is a bottom up technique in which the smaller sub-problems are solved first and the result of these are sued to find the solution of the larger sub-problems.

Examples:

- Fibonacci numbers computed by iteration.
- Warshall's algorithm for transitive closure implemented by iterations
- Floyd's algorithms for all-pairs shortest paths

```
public static int fibonacci(int n) {  
    if (n <= 1) {  
        return n;  
    }  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}
```



Using divide and conquer the same sub problem is solved again and again, which reduce the performance of the algorithm. This algorithm has an exponential Time Complexity and linear Space Complexity.

```

public static int fibonacci2(int n)
{
    int first = 0;
    int second = 1;
    int i, temp=0;

    if (n == 0)
        return first;
    else if (n == 1)
        return second;
    i = 2;
    while (i <= n)
    {
        temp = first + second;
        first = second;
        second = temp;
        i += 1;
    }
    return temp;
}

```

Using this algorithm, we will get Fibonacci in linear Time Complexity and constant Space Complexity.

Reduction / Transform-and-Conquer

These methods work as a two-stage procedure. First, the problem is transformed into a known problem for which we know optimal solution. In the second stage, the problem is solved.

The most common type of transformation is sorting of an array. For example, in a given list of numbers finds the two closest number.

Brute-force solution, we will find distance between each element in the array and will keep the minimum distance pair. In this approach, total time complexity is $O(n^2)$

Transform and conquer solution, we will first sort the array in $O(n \log n)$ time and then find the closest number by scanning the array in another single pass with time complexity $O(n)$. Thus, the total time complexity is $O(n \log n)$.

Examples:

- Gaussian elimination
- Heaps and Heapsort

Backtracking

In real life, let us suppose someone has given you a lock with a number (three digit lock, number range from 1 to 9). Moreover, you do not have the exact password key for the lock. You need to test every combination until you got the right one. Obviously, you need to test starting from something like “111”, then “112” and so on. You will get your key before you reach “999”. Therefore, what you are doing is backtracking.

Suppose the lock produces some sound “click” if correct digit is selected for any level. If we can listen to this sound such intelligence/ heuristics will help you to reach your goal much faster. These functions are called Pruning function or bounding functions.

Backtracking is a method by which solution is found by exhaustively searching through large but finite number of states, with some pruning or bounding function, we can narrow down our search.

For all the problems (like NP hard problems) for which there does not exist any other efficient algorithm we use backtracking algorithm.

Backtracking problems have the following components:

1. Initial state
2. Target / Goal state
3. Intermediate states
4. Path from the initial state to the target / goal state
5. Operators to get from one state to another
6. Pruning function (optional)

The solving process of backtracking algorithm starts with the construction of state’s tree, whose nodes represents the states. The root node is the initial state and one or more leaf node will be our target state. Each edge of the tree represents some operation. The solution is obtained by searching the tree until a Target state is found.

Backtracking uses depth-first search:

- 1) Store the initial state in a stack
- 2) While the stack is not empty, repeat:

- 3) Read a node from the stack.
- 4) While there are available operators, do:
 - a. Apply an operator to generate a child
 - b. If the child is a goal state – return solution
 - c. If it is a new state and pruning function does not discard it, than push the child into the stack.

There are three monks and three demons at one side of a river. We want to move all of them to the other side using a small boat. The boat can carry only two persons at a time. Given if on any shore the number of demons will be more than monks then they will eat the monks. How can we move all of these people to the other side of the river safely?

Same as the above problem there is a farmer who has a goat, a cabbage and a wolf. If the farmer leaves, goat with cabbage, goat will eat the cabbage. If the farmer leaves wolf alone with goat, wolf will kill the goat. How can the farmer move all his belongings to the other side of the river?

You are given two jugs, a 4-gallon one and a 3-gallon one. There are no measuring markers on jugs. A tap can be used to fill the jugs with water. How can you get 2 gallons of water in the 4-gallon jug?

Branch-and-bound

Branch and bound method is used when we can evaluate cost of visiting each node by a utility functions. At each step, we choose the node with the lowest cost to proceed further. Branch-and bound algorithms are implemented using a priority queue. In branch and bound, we traverse the nodes in breadth-first manner.

A* Algorithm

A* is a sort of an elaboration on branch-and-bound. In branch-and-bound, at each iteration we expand the shortest path that we have found so far. In A*, instead of just picking the path with the shortest length so far, we pick the path with the shortest estimated total length from start to goal, where the total length is estimated as length traversed so far plus a heuristic estimate of the remaining distance from the goal.

Branch-and-bound will always find an optimal solution, which is the shortest path. A* will always find an optimal solution if the heuristic is correct. Choosing a good heuristic is the most important part of A* algorithm.

Conclusion

Usually a given problem can be solved using a number of methods; however, it is not wise to settle for the first method that comes to our mind. Some methods result in a much more efficient solution than others do.

For example, the Fibonacci numbers calculated recursively (decrease-and-conquer approach), and computed by iterations (dynamic programming). In the first case, the complexity is **O(2ⁿ)**, and in the other case, the complexity is **O(n)**.

Another example, consider sorting based on the Insertion-Sort and basic bubble sort. For almost sorted files, Insertion-Sort will give almost linear complexity, while bubble sort sorting algorithms have quadratic complexity.

So the most important question is how to choose the best method?

First, you should understand the problem statement.

Second by knowing various problems and their solutions.

CHAPTER 15: BRUTE FORCE ALGORITHM

Introduction

Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem. It is useful for solving small size dataset problem.

Most of the cases there are other algorithm techniques can be used to get a better solution of the same problem.

Some examples of brute force algorithms are:

- Bubble-Sort
- Selection-Sort · Sequential search in an array · Computing pow (a, n) by multiplying a, n times.
- Convex hull problem · String matching · Exhaustive search · Traveling salesman · Knapsack
- Assignment problems

Problems in Brute Force Algorithm

Bubble-Sort

In Bubble-Sort, adjacent elements of the array are compared and are exchanged if they are out of order.

```
// Sorts a given list by Bubble Sort  
// Input: An array A of orderable elements  
// Output: List A[0..n - 1] sorted in ascending order
```

```
Algorithm BubbleSort(A[0..n - 1])  
    sorted = false  
    while !sorted do sorted = true  
        for j = 0 to n - 2 do  
            if A[j] > A[j + 1] then  
                swap A[j] and A[j + 1]  
            sorted = false
```

Time Complexity is $\Theta(n^2)$

Selection-Sort

The entire given list of N elements is traversed to find its smallest element and exchange it with the first element. Then, the array is traversed again to find the second element and exchange it with the second element. After N-1 passes, the array will be completely sorted.

```
//Sorts a given list by selection sort  
//Input: An array A[0..n-1] of orderable elements  
//Output: List A[0..n-1] sorted in ascending order
```

```
Algorithm SelectionSort (A[0..n-1])  
    for i = 0 to n - 2 do  
        min = i
```

```
for j = i + 1 to n - 1 do
if A[j] < A[min]
min = j
swap A[i] and A[min]
```

Time Complexity is $\Theta(n^2)$

Sequential Search

The algorithm compares consecutive elements of a given list with a given search keyword until either a match is found or the array is exhausted.

```
Algorithm SequentialSearch (A[0..n], K)
```

```
i = 0
While A [i] ≠ K do
    i = i + 1
    if i < n
        return i
    else
        return -1
```

Worst case time complexity is $\Theta(n)$.

Computing pow (a, n)

Computing a^n ($a > 0$, and n is a nonnegative integer) based on the definition of exponentiation.

$N-1$ multiplications are required in brute force method.

```
// Input: A real number a and an integer n = 0
// Output: a power n
```

```
Algorithm Power(a, n)
```

```
result = 1
for i = 1 to n do
    result = result * a
return result
```

The algorithm requires $\Theta(n)$

String matching

A brute force string matching algorithm takes two inputs, first text consists of n characters and a pattern consist of m character ($m \leq n$). The algorithm starts by comparing the pattern with the beginning of the text. Each character of the pattern is compared to the corresponding character of the text. Comparison starts from left to right until all the characters are matched or a mismatch is found. The same process is repeated until a match is found. Each time the comparison starts from one position to the right.

```
//Input: An array T[0..n - 1] of n characters representing a text  
// an array P[0..m - 1] of m characters representing a pattern  
//Output: The position of the first character in the text that starts the first  
// matching substring if the search is successful and -1 otherwise.
```

```
Algorithm BruteForceStringMatch (T[0..n - 1], P[0..m - 1])  
    for i = 0 to n - m do  
        j = 0  
        while j < m and P[j] = T[i + j] do  
            j = j + 1  
        if j = m then  
            return i  
    return -1
```

In the worst case, the algorithm is $O(mn)$.

Closest-Pair Brute-Force Algorithm

The closest-pair problem is to find the two closest points in a set of n points in a 2-dimensional space.

A brute force implementation of this problem computes the distance between each pair of distinct points and find the smallest distance pair.

```
// Finds two closest points by brute force
```

```
// Input: An array P of n >= 2 points  
// Output: The closest pair
```

```
Algorithm BruteForceClosestPair(P)  
    dmin = infinite  
    for i = 1 to n - 1 do  
        for j = i + 1 to n do  
            d = (xi - xj)2 + (yi - yj)2  
            if d < dmin then  
                dmin = d  
                imin = i  
                jmin = j  
    return imin, jmin
```

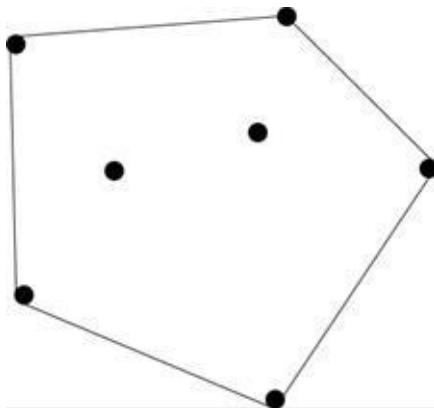
Time Complexity is $\Theta(n^2)$

Convex-Hull Problem

Convex-hull of a set of points is the smallest convex polygon that contains all the points. All the points of the set will lie on the convex hull or inside the convex hull. The convex-hull of a set of points is a subset of points in the given sets.

How to find this subset?

Answer: Subset points are the boundary of the Convex Hull. We take any two consecutive points of the boundary, and the rest of the points of the set will lie on its one side.



Two points $(x_1, y_1), (x_2, y_2)$ make the line $ax + by = c$
Where $a = y_2 - y_1$, $b = x_1 - x_2$, and $c = x_1y_2 - y_1x_2$

And divides the plane by $ax + by - c < 0$ and $ax + by - c > 0$
So, we need to check $ax + by - c$ for the rest of the points

If we find all the points in the set lies one side of the line with either all have $ax + by - c < 0$ or all the points have $ax + by - c > 0$ then we will add these points to the desired convex hull point set.

For each of $n(n-1)/2$ pairs of distinct points, one needs to find the sign of $ax + by - c$ in each of the other $n-2$ points.

What is the worst-case cost of the algorithm: $O(n^3)$

```
Algorithm ConvexHull
for i=0 to n-1
    for j=0 to n-1
        if (xi,yi) !=(xj,yj)
            draw a line from (xi,yi) to (xj,yj)
            for k=0 to n-1
                if(i!=k and j!=k)
                    if ( all other points lie on the same side of the
                        line (xi,yi) and (xj,yj))
                        then add (xi,yi) to (xj,yj) to the convex hull set
```

Exhaustive Search

Exhaustive search is a brute force approach applies to combinatorial problems. In exhaustive search, we generate all the possible combinations. At each step, we try to find if the combinations satisfy the problem constraints. Either, we get a desired solution, which satisfy the problem constraint or there is no solution.

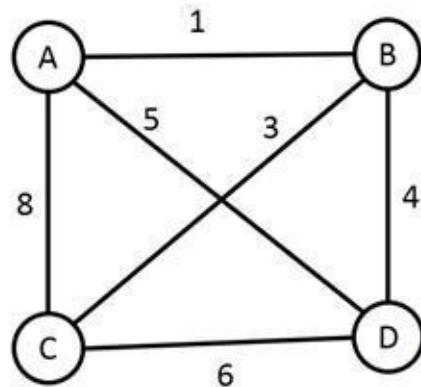
Examples of exhaustive search are:

- Traveling salesman problem
- Knapsack problem
- Assignment problem

Traveling Salesman Problem (TSP)

In the traveling salesman problem we need to find the shortest tour through a given set of N cities that salesperson visits each city exactly once before returning to the city where he has started.

Alternatively, finding the shortest Hamiltonian circuit in a weighted connected graph. A cycle that passes through all the vertices of the graph exactly once.



Tours where A is starting city:

Tour Cost

A → B → C → D → A $1+3+6+5 = 15$

A → B → D → C → A $1+4+6+8 = 19$

A → C → B → D → A $8+3+4+5 = 20$

A → C → D → B → A $8+6+4+1 = 19$

A → D → B → C → A $5+4+3+8 = 20$

A → D → C → B → A $5+6+3+1 = 15$

Algorithm TSP

Select a city

MinTourCost = infinite

For (All permutations of cities) do

If(LengthOfPathSinglePermutation < MinTourCost)

MinTourCost = LengthOfPath

Total number of possible combinations = $(n-1)!$

Cost for calculating the path: $\Theta(n)$

So the total cost for finding the shortest path: $\Theta(n!)$

Knapsack Problem

Given an item with cost C_1, C_2, \dots, C_n , and volume V_1, V_2, \dots, V_n and knapsack of capacity V_{max} , find the most valuable ($\max \sum C_j$) that fits in the knapsack ($\sum V_j \leq V_{max}$).

The solution is one of the subset of the set of object taking 1 to n objects at a time, so the Time complexity is $O(2^n)$

Algorithm KnapsackBruteForce

MaxProfit = 0

For (All permutations of objects) do

CurrProfit = sum of objects selected

If(MaxProfit < CurrProfit)

MaxProfit = CurrProfit

Store the current set of objects selected

Conclusion

Brute force is the first algorithm that comes into mind when we see some problem. They are the simplest algorithms that are very easy to understand. However, these algorithms rarely provide an optimum solution. In many cases, we will find other effective algorithm that is more efficient than the brute force method.

CHAPTER 16: GREEDY ALGORITHM

Introduction

Greedy algorithms are generally used to solve optimization problems. To find the solution that minimizes or maximizes some value (cost/profit/count etc.).

In greedy algorithm, solution is constructed through a sequence of steps. At each step, choice is made which is locally optimal. We always take the next data to be processed depending upon the dataset which we have already processed and then choose the next optimum data to be processed.

Greedy algorithms does not always give optimum solution. For some problems, greedy algorithm gives an optimal solution. They are useful for fast approximations.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. Greedy choice: A global optimum can be arrived at by selecting a local optimum.
2. Optimal substructure: An optimal solution to the problem is made from optimal solutions of sub-problems.

Some examples of brute force algorithms are:

Optimal solutions:

- Minimal spanning tree:
 - o Prim's algorithm,
 - o Kruskal's algorithm
- Dijkstra's algorithm for single-source shortest path
- Huffman trees for optimal encoding
- Scheduling problems

Approximate solutions:

- Greedy algorithm for the Knapsack problem
- Coin exchange problem

Problems on Greedy Algorithm

Coin exchange problem

How can a given amount of money N be made with the least number of coins of given denominations $D = \{d_1, d_2, \dots, d_n\}$?

The Indian coin system $\{5, 10, 20, 25, 50, 100\}$

Suppose we want to give change of a certain amount of 40 paisa.

We can make a solution by repeatedly choosing a coin \leq to the current amount, resulting in a new amount. In the greedy algorithm, we always choose the largest coin value possible without exceeding the total amount.

For 40 paisa: $\{25, 10, \text{ and } 5\}$

The optimal solution will be $\{20, 20\}$

The greedy algorithm did not give us optimal solution, but it gave us a fair approximation.

Algorithm MAKE-CHANGE (N)

$C = \{5, 20, 25, 50, 100\}$ // constant denominations.

$S = \{\}$ // set that will hold the solution set.

$\text{Value} = N$

WHILE $\text{Value} \neq 0$

$x = \text{largest item in set } C \text{ such that } x < \text{Value}$

IF no such item THEN

RETURN "No Solution"

$S = S + x$

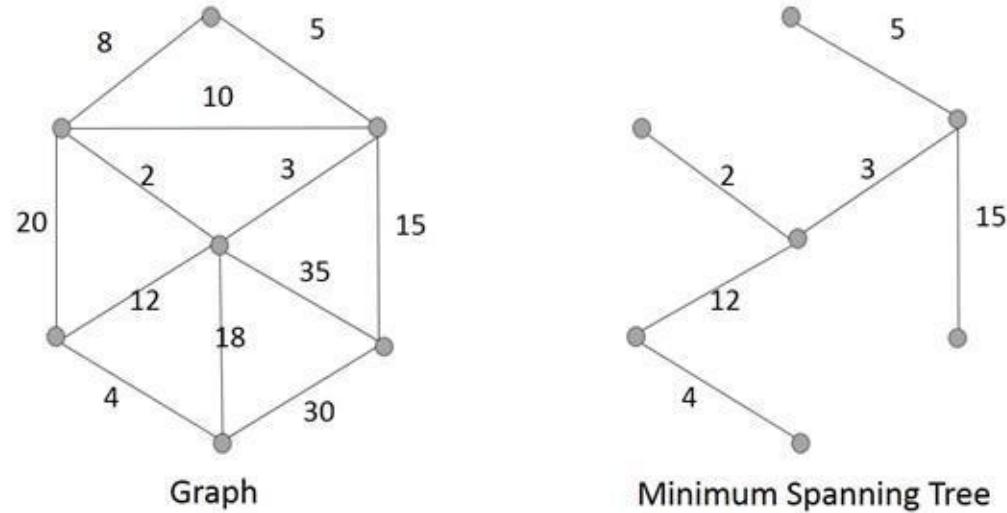
$\text{Value} = \text{Value} - x$

RETURN S

Minimum Spanning Tree

A spanning tree of a connected graph is a tree containing all the vertices.

A minimum spanning tree of a weighted graph is a spanning tree with the smallest sum of the edge weights.



Prim's Algorithm

Prim's algorithm grows a single tree T , one edge at a time, until it becomes a spanning tree.

We initialize T with zero edges and U with single node. Where T is spanning tree edges set and U is spanning tree vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other not in U .

Since each edge adds one new vertex to U , after $n - 1$ additions, U contains all the vertices of the spanning tree and T becomes a spanning tree.

```
// Returns the MST by Prim's Algorithm  
// Input: A weighted connected graph  $G = (V, E)$   
// Output: Set of edges comprising a MST
```

Algorithm $\text{Prim}(G)$

$T = \{\}$

Let r be any vertex in G

$U = \{r\}$

for $i = 1$ to $|V| - 1$ do

```
e = minimum-weight edge (u, v)
```

```
With u in U and v in V-U
```

```
U = U + {v}
```

```
T = T + {e}
```

```
return T
```

Prim's Algorithm using a priority queue (min heap) to get the closest fringe vertex

Time complexity is $O(m \log n)$ where n vertices and m edges of the MST.

Kruskal's Algorithm

Kruskal's Algorithm is used to create minimum spanning tree. Spanning tree is created by choosing smallest weight edge that does not form a cycle. And repeat this process until all the edges from the original set is exhausted.

Sort the edges in non-decreasing order of cost: $c(e1) \leq c(e2) \leq \dots \leq c(em)$.

Set T to be the empty tree. Add edges to tree one by one, if it does not create a cycle. (If the new edge form cycle then ignore that edge.)

```
// Returns the MST by Kruskal's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST
```

Algorithm Kruskal(G)

```
Sort the edges E by their weights
```

```
T = {}
```

```
while |T| + 1 < |V| do
```

```
    e = next edge in E
```

```
    if T + {e} does not have a cycle then
```

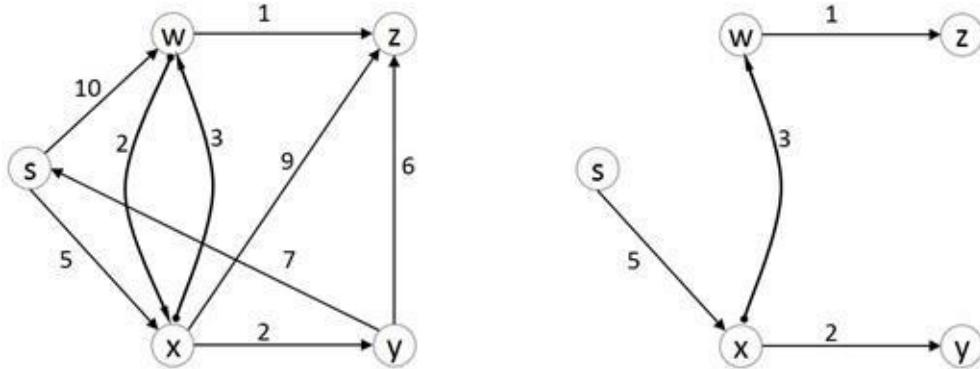
```
        T = T + {e}
```

```
return T
```

Kruskal's Algorithm is $O(E \log V)$ using efficient cycle detection.

Dijkstra's algorithm

Dijkstra's algorithm is used for single-source shortest path problem for weighted edges with no negative weight. It determines the length of the shortest path from the source to each of the other nodes of the graph. In a given weighted graph G , we need to find shortest paths from the source vertex s to each of the other vertices.



Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning, as we do not know the actual path to the nodes and parent of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)

At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then, update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the distance and parent list fully populated.

```
// Solves SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph  $G = (V, E)$ 
// with no negative weights, and source vertex  $v$ 
// Output: The length and path from  $s$  to every  $v$ 
```

```
Algorithm Dijksta( $G, s$ )
  for each  $v$  in  $V$  do
     $D[v] = \text{infinite}$  // Unknown distance
     $P[v] = \text{null}$  // Unknown previous node
    add  $v$  to  $PQ$  // Adding all nodes to priority queue
     $D[\text{source}] = 0$  // Distance from source to source
```

```

while (PQ is not empty)
u = vertex from PQ with smallest D[u]
remove u from PQ
for each v adjacent from u do
alt = D[u] + length ( u , v )
if alt < D[v] then
D[v] = alt
P[v] = u
Return D[] , P[]

```

Time complexity is $O(|E|\log|V|)$.

Note: Dijkstra's algorithm does not work for graphs with negative edges weight.

Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Huffman trees for optimal encoding

Encoding is an assignment of bit strings of alphabet characters.

There are two types of encoding:

- Fixed-length encoding (eg., ASCII)
- Variable-length encoding (eg., Huffman code)

Variable length encoding can only work on prefix free encoding. Which means that no code word is a prefix of another code word.

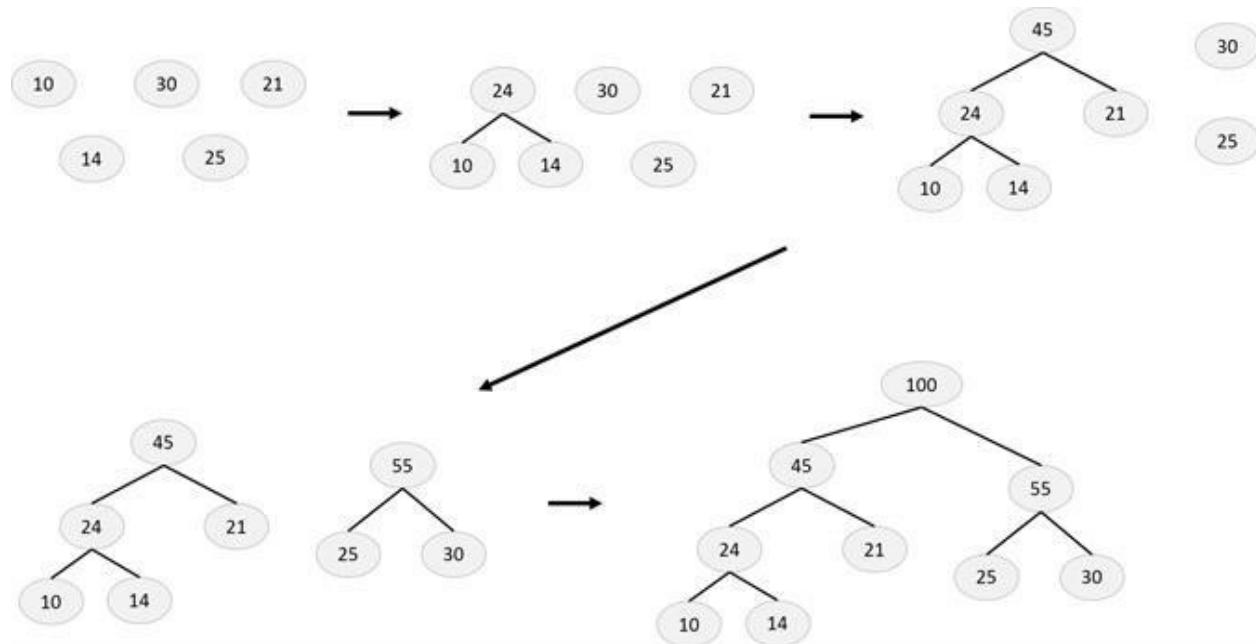
Huffman codes are the best prefix free code. Any binary tree with edges labeled as 0 and 1 will produce a prefix free code of characters assigned to its leaf nodes.

Huffman's algorithm is used to construct a binary tree whose leaf value is assigned a code, which is optimal for the compression of the whole text need to be processed. For example, the most frequently occurring words will get the smallest code so that the final encoded text is compressed.

Initialize n one-node trees with words and the tree weights with their frequencies. Join the two binary tree with smallest weight into one and the weight of the new formed tree as the sum of weight of the two small trees. Repeat the above process $N-1$ times and when there is just one big tree left you

are done.

Mark edges leading to left and right subtrees with 0's and 1's, respectively.



Word	Frequency
Apple	30
Banana	25
Mango	21
Orange	14
Pineapple	10

Word	Value	Code
Apple	30	11
Banana	25	10
Mango	21	01
Orange	14	001
Pineapple	10	000

It is clear that more frequency words gets smaller Huffman's code.

```

// Computes optimal prefix code.
// Input: List W of character probabilities
// Output: The Huffman tree.
  
```

```
Algorithm Huffman(C[0..n - 1], W[0..n - 1])
```

```
    PQ = {} // priority queue
```

```
    for i = 0 to n - 1 do
```

```
        T.char = C[i]
```

```
        T.weight = W[i]
```

```
        add T to priority queue PQ
```

```
    for i = 0 to n - 2 do
```

```
        L = remove min from PQ
```

```
        R = remove min from PQ
```

```
        T = node with children L and R
```

```
        T.weight = L.weight + R.weight
```

```
        add T to priority queue PQ
```

```
    return T
```

Time Complexity is **O(n log n)**.

Activity Selection Problem

Suppose that activities require exclusive use of common resources, and you want to schedule as many activities as possible.

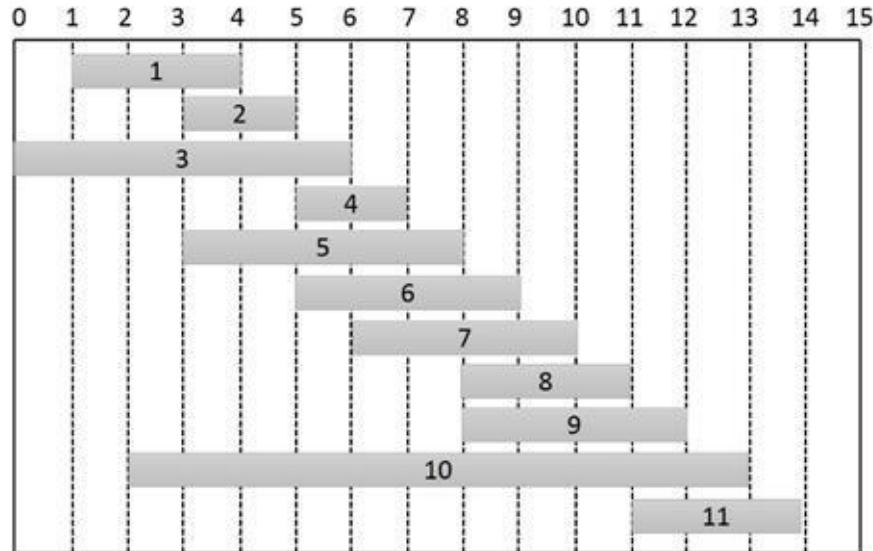
Let $S = \{a_1, \dots, a_n\}$ be a set of n activities.

Each activity a_i needs the resource during a time period starting at s_i and finishing before f_i , i.e., during $[s_i, f_i]$.

The optimization problem is to select the non-overlapping largest set of activities from S . We assume that activities $S = \{a_1, \dots, a_n\}$ are sorted in finish time $f_1 \leq f_2 \leq \dots \leq f_{n-1} \leq f_n$ (this can be done in $\Theta(n \lg n)$).

Example: Consider these activities:

I	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	11
F[i]	4	5	6	7	8	9	10	11	12	13	14



We chose an activity that starts first, and then look for the next activity that starts after it is finished. This could result in $\{a_4, a_7, a_8\}$, but this solution is not optimal.

An optimal solution is $\{a_1, a_3, a_6, a_8\}$. (It maximizes the objective function of a number of activities scheduled.)

Another one is $\{a_2, a_5, a_7, a_9\}$. (Optimal solutions are not necessarily unique.)

How do we find (one of) these optimal solutions? Let us consider it as a dynamic programming problem.

We are trying to optimize the number of activities. Let us be greedy!

- The time left after running an activity can be used to run subsequent activities.
- If we choose the first activity to finish, the more time will be left.
- Since activities are sorted by finish time, we will always start with a_1 .
- Then we can solve the single sub problem of activity scheduling in this remaining time.

Algorithm ActivitySelection($S[]$, $F[]$, N)

Sort $S[]$ and $F []$ in increasing order of finishing time

$A = \{a_1\}$

$K = 1$

For $m = 2$ to N do

If $S[m] \geq F[k]$

$A = A + \{a_m\}$

$K = m$

Return A

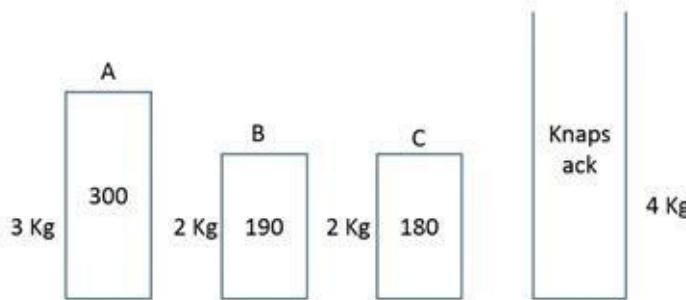
Knapsack Problem

A thief enters a store and sees a number of items with their mentioned cost and weight. His Knapsack can hold a max weight. What should he steal to maximize profit?

Fractional Knapsack problem

A thief can take a fraction of an item (they are divisible substances, like gold powder).

The fractional knapsack problem has a greedy solution one should first sort the items in term of cost density against weight. Then fill up as much of the most valuable substance by weight as one can hold, then as much of the next most valuable substance, etc. Until W is reached.



Item	A	B	C
Cost	300	190	180
Weight	3	2	2
Cost/weight	100	95	90

For a knapsack of capacity of 4 kg.

The optimum solution of the above will take 3kg of A and 1 kg of B.

Algorithm FractionalKnapsack(W[], C[], Wk)

For i = 1 to n do

X[i] = 0

```

Weight = 0
//Use Max heap
H = BuildMaxHeap(C/W)
While Weight < Wk do
    i = H.GetMax()
    If(Weight + W[i] <= Wk) do
        X[i] = 1
        Weight = Weight + W[i]
    Else
        X[i] = (Wk - Weight)/W[i]
        Weight = Wk
Return X

```

0/1 Knapsack Problem

A thief can only take or leave the item. He cannot take a fraction.

A greedy strategy same as above could result in empty space, reducing the overall cost density of the knapsack.

In the above example, after choosing object A there is no place for B or C. Therefore, there leaves empty space of 1kg. Moreover, the result of the greedy solution is not optimal.

The optimal solution will be when we take object B and C. This problem can be solved by dynamic programming that we will see in the coming chapter.

CHAPTER 17: DIVIDE-AND-CONQUER, DECREASE-AND-CONQUER

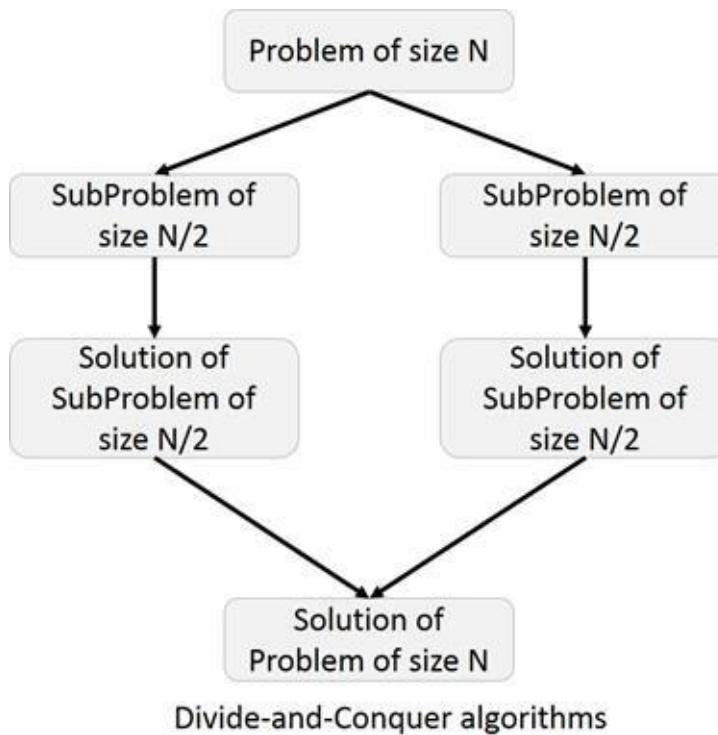
Introduction

Divide-and-Conquer algorithms works by recursively breaking down a problem into two or more sub-problems (divide step), until these sub-problems become simple enough so that they can be solved directly (conquer step). The solution of these sub-problems is then combined to give a solution of the original problem.

Divide-and-Conquer algorithms involve basic three steps

1. Divide the problem into smaller problems.
2. Conquer by solving these problems.
3. Combine these results together.

In divide-and-conquer the size of the problem is reduced by a factor (half, one-third etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.



Examples of divide-and-conquer algorithms:

- Merge-Sort algorithm (recursion)
- Quicksort algorithm (recursion)
- Computing the length of the longest path in a binary tree (recursion)

- Computing Fibonacci numbers (recursion) · Convex Hull

Examples of decrease-and-conquer algorithms:

- Computing POW (a, n) by calculating POW ($a, n/2$) using recursion · Binary search in a sorted list (recursion) · Searching in BST
- Insertion-Sort
- Graph traversal algorithms (DFS and BFS) · Topological sort · Warshall's algorithm (recursion) · Permutations (Minimal change approach, Johnson-Trotter algorithm) · Fake-coin problem (Ternary search) · Computing a median

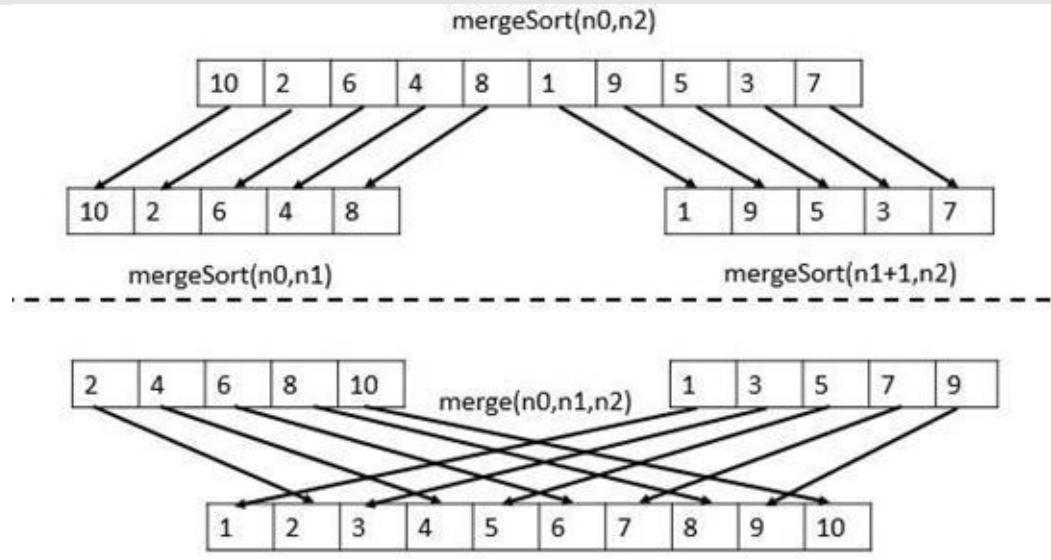
General Divide-and-Conquer Recurrence

$$T(n) = aT(n/b) + f(n)$$

- Where $a \geq 1$ and $b > 1$.
- "n" is the size of a problem.
- "a" is a number of sub-problem in the recursion.
- "n/b" is the size of each sub-problem.
- "f(n)" is the cost of the division of the problem into sub problem or merge of the results of sub-problem to get the final result.

Problems on Divide-and-Conquer Algorithm

Merge-Sort algorithm



```
// Sorts a given list by mergesort  
// Input: An array A of orderable elements  
// Output: List A[0..n - 1] in ascending order
```

```
Algorithm Mergesort(A[0..n - 1])  
  if n ≤ 1 then  
    return;  
  copy A[0..[n/2] - 1] to B[0..[n/2] - 1]  
  copy A[[n/2]..n - 1] to C[0..[n/2] - 1]  
  Mergesort(B)  
  Mergesort(C)  
  Merge(B, C, A)
```

```
// Merges two sorted arrays into one list  
// Input: Sorted arrays B and C  
// Output: Sorted list A
```

```
Algorithm Merge(B[0..p - 1], C[0..q - 1], A[0..p + q - 1])
```

```

i = 0
j = 0
for k = 0 to p + q - 1 do
if i < p and (j = q or B[i] ≤ C[j]) then
A[k] = B[i]
i = i + 1
else
A[k] = C[j]
j = j + 1

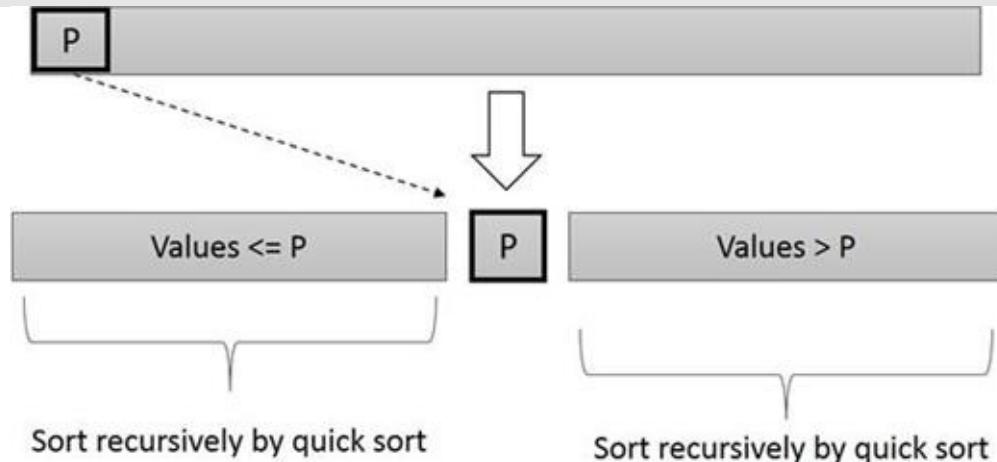
```

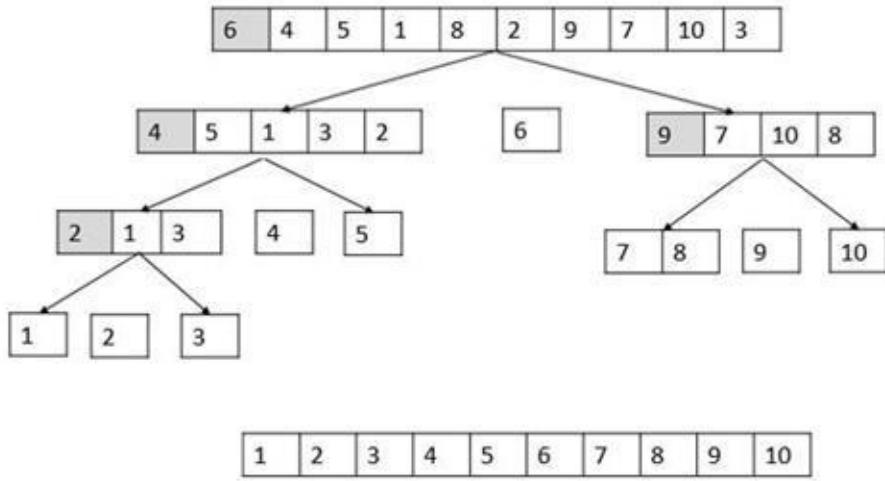
Time Complexity: **O(nlogn)** & Space Complexity: **O(n)**

Time Complexity of Merge-Sort is **O(nlogn)** in all 3 cases (worst, average and best) as Merge-Sort always divides the array into two halves and take linear time to merge two halves.

It requires equal amount of additional space as the unsorted list. Hence, it is not at all recommended for searching large unsorted lists.

Quick-Sort





```
// Sorts a subarray by quicksort
// Input: An subarray of A
// Output: List A[l..r] in ascending order
```

```
Algorithm Quicksort(A[l..r])
  if l < r then
    p ← Partition(A[l..r]) // p is index of pivot
    Quicksort(A[l..p - 1])
    Quicksort(A[p + 1..r])
```

```
// Partitions a subarray using A[..] as pivot
// Input: Subarray of A
// Output: Final position of pivot
```

```
Algorithm Partition(A[], left, right)
  pivot = A[left]
  lower = left
  upper = right
  while lower < upper
    while A[lower] <= pivot
      lower = lower + 1
    while A[upper] > pivot
      upper = upper - 1
    if lower < upper then
      swap A[lower] and A[upper]
```

```
swap A[lower] and A[upper] //upper is the pivot position  
return upper
```

Worst Case time complexity : $O(n^2)$,

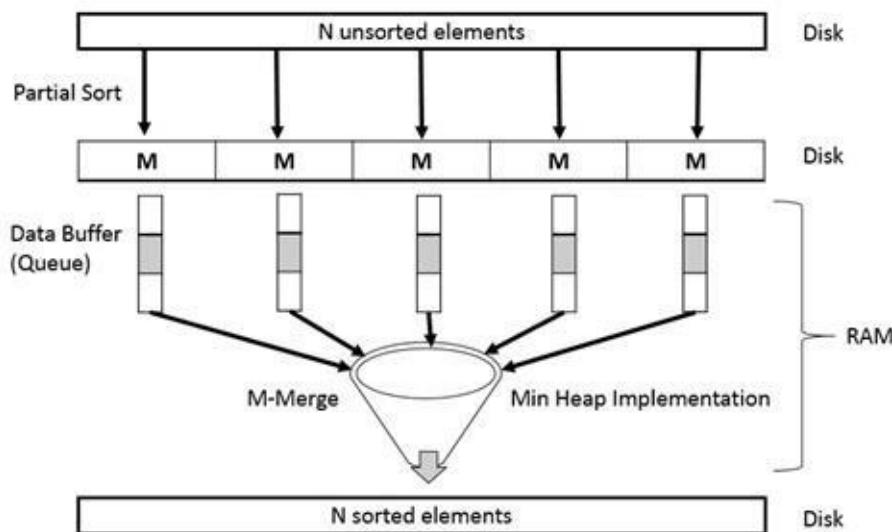
Average case time complexity : $O(n \log n)$,

Space Complexity: $O(n \log n)$, The space required by Quick-Sort is very less, only $O(n \log n)$ additional space is required.

Quicksort is not a stable sorting technique, so it might change the occurrence of two similar elements in the array while sorting.

External Sorting

External sorting is also done using divide and conquer algorithm.



Binary Search

We get the middle point from the sorted list and start comparing with the desired value.

Note: Binary search requires the array to be sorted otherwise binary search cannot be applied.

```
// Searches a value in a sorted list using binary search  
// Input: An sorted list A and a key K
```

```

// Output: The index of K or -1
Algorithm BinarySearch(A[0..N - 1], N, K) // iterative solution
    low = 0
    high = N-1
    while low <= high do
        mid = ⌊ (low + high)/2⌋
        if K = A[mid] then
            return mid
        else if A[mid] < K
            low = mid + 1
        else
            high = mid - 1
    return -1

```

```

// Searches a value in a sorted list using binary search
// Input: An sorted list A and a key K
// Output: The index of K or -1

```

```

Algorithm BinarySearch(A[], low, high, K) //Recursive solution
    If low > high
        return -1
    mid = ⌊ (low + high)/2⌋
    if K = A[mid] then
        return mid
    else if A[mid] < K
        return BinarySearch(A[],mid + 1, high, K)
    else
        return BinarySearch(A[],low, mid - 1, K)

```

Time Complexity: **O(logn)**. If you notice the above programs, you should keep in mind that we always take half input and throwing out the other half. So the recurrence relation for binary search is $T(n) = T(n/2) + c$. Using a divide and conquer master theorem, we get $T(n) = \Theta(\log n)$.

Space Complexity: **O(1)**

Power function

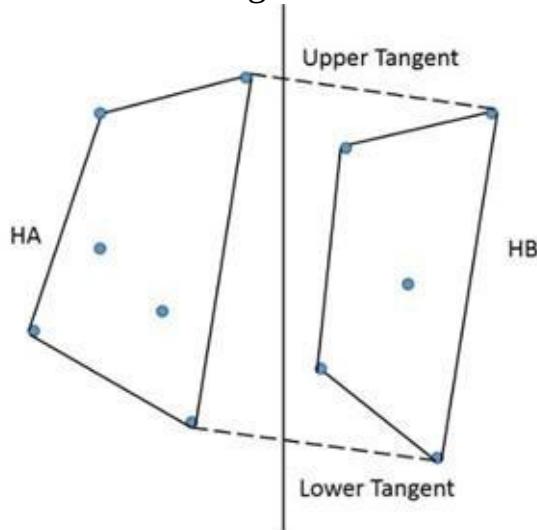
```
// Compute Nth power of X using divide and conquer using recursion
```

```
// Input: Value X and power N  
// Output: Power( X, N)
```

```
Algorithm Power( X, N)  
    If N = 0  
        Return 1  
    Else if N % 2 == 0  
        Value = Power(X, N/2)  
        Return Value * Value  
    Else  
        Value = Power(X, N/2)  
        Return Value * Value * X
```

Convex Hull

Sort points by X-coordinates. Divide points into equal halves A and B. Recursively compute HA and HB. Merge HA and HB to obtain convex hull



LowerTangent(HA, HB)

A = rightmost point of HA

B = leftmost point of HB

While ab is not a lower tangent for HA and HB do

 While ab is not a lower tangent to HA do

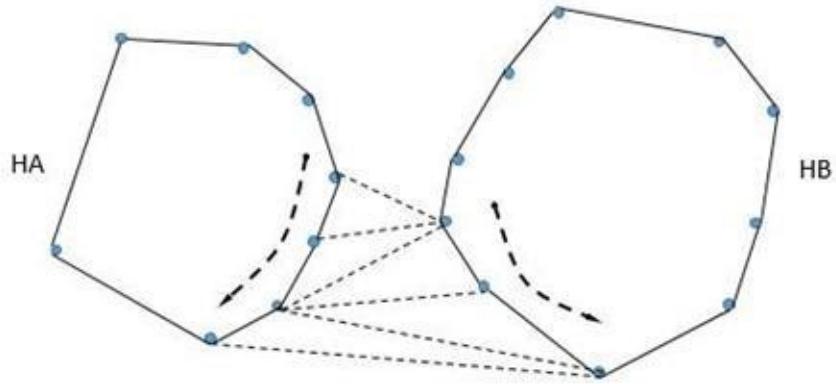
 a = a - 1 (move a clockwise)

 While ab is not a lower tangent to HB do

$b = b + 1$ (move b counterclockwise)

Return ab

Similarly find upper tangent and combine the two hulls.



Initial sorting takes **O(nlogn)** time

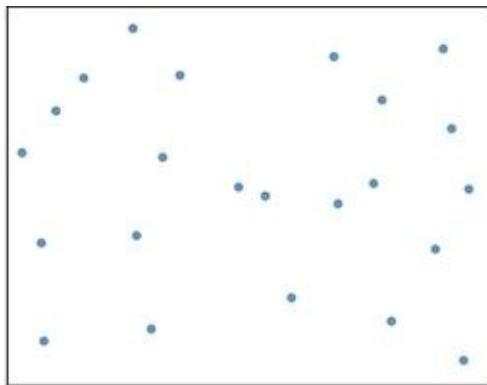
Recurrence relation $T(N) = 2T(N/2) + O(N)$

Where, **O(N)** time is for tangent computation inside the merge step.

Final Time complexity is $T(N) = O(nlogn)$.

Closest Pair

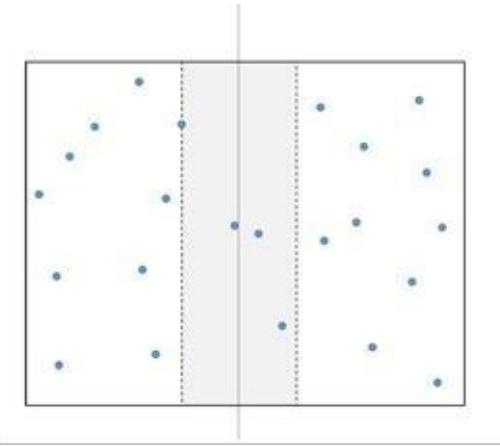
Given N points in 2-dimensional plane, find two points whose mutual distance is smallest.



A brute force algorithm takes every point and find its distance with all the other points in the plane. In addition, keep track of the minimum distance points and

minimum distance. The closest pair will be found in $O(n^2)$ time.

Let us suppose that there is a vertical line, which divides the graph into two separate parts (let us call it left and right part). In the brute force algorithm, we will notice that we are comparing all the points in the left half with the points in the right half. This is the point where we are doing some extra work.



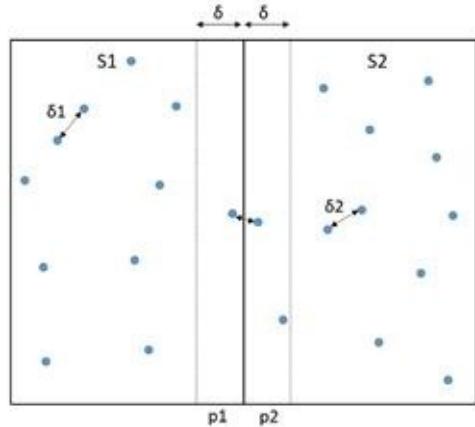
To find the minimum we need to consider only three cases:

- 1) Closest pair in the right half
- 2) Closest pair in the left half.
- 3) Closest pair in the boundary region of the two halves. (Gray)

Every time we will divide the space S into two parts S_1 and S_2 by a vertical line. Recursively we will compute the closest pair in both S_1 and S_2 . Let us call minimum distance in space S_1 as δ_1 and minimum distance in space S_2 as δ_2 . We will find $\delta = \min(\delta_1, \delta_2)$

Now we will find the closest pair in the boundary region. By taking one point each from S_1 and S_2 in the boundary range of δ width on both sides.

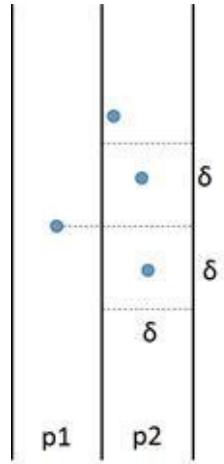
The candidate pair of point (p, q) where $p \in S_1$ and $q \in S_2$.



We can find the points that lie in this region in linear time $O(N)$ by just scanning through all the points and finding that all points lie in this region.

Now we can sort them in increasing order in Y-axis in just **$O(n \log n)$** time. Then scan through them and get the minimum in just one linear pass. Closest pair cannot be far apart from each other.

Let us look into the next figure.



Then the question is how many points we need to compare. We need to compare the points sorted in Y-axis only in the range of δ . Therefore, the number of points will come down to only 6 points.



By doing this, we are getting equation.

$$T(N) = 2T(N/2) + N + N\log N + 6N = O(n(\log n)^2)$$

Can we optimize this further?

Yes

Initially, when we are sorting the points in X coordinate we are sorting them in Y coordinate too.

When we divide the problem, then we traverse through the Y coordinate list too, and construct the corresponding Y coordinate list for both S1 and S2. And pass that list to them.

Since we have the Y coordinate list passed to a function the δ region points can be found sorted in the Y coordinates in just one single pass in just **O(N)** time.

$$T(N) = 2T(N/2) + N + N + 6N = \mathbf{O(nlogn)}$$

```
// Finds closest pair of points
// Input: A set of n points sorted by coordinates
// Output: Distance between closest pair
```

Algorithm ClosestPair(P)

```
if n < 2 then
    return ∞
else if n = 2 then
    return distance between pair
else
    m = median value for x coordinate
    δ1 = ClosestPair(points with x < m)
    δ2 = ClosestPair(points with x > m)
    δ = min(δ1, δ2)
    δ3 = process points with m - δ < x < m + δ
    return min(δ, δ3)
```

First pre-process the points by sorting them in X and Y coordinates. Use two separate lists to keep these sorted points.

Before recursively solving sub-problem pass the sorted list for that sub-problem.

CHAPTER 18: DYNAMIC PROGRAMMING

Introduction

While solving problems using Divide-and-Conquer method, there may be a case when recursively sub-problems can result in the same computation being performed multiple times. This problem arises when there are identical sub-problems arise repeatedly in a recursion.

Dynamic programming is used to avoid the requirement of repeated calculation of same sub-problem. In this method, we usually store the result of sub - problems in some data structure (like a table) and refer it to find if we have already calculated the solution of sub - problems before calculating it again.

Dynamic programming is applied to solve problems with the following properties:

1. Optimal Substructure: An optimal solution constructed from the optimal solutions of its sub-problems.
2. Overlapping Sub-problems: While calculating the optimal solution of sub-problems same computation is repeated repeatedly.

Examples:

- Fibonacci numbers computed by iteration.
- Assembly-line Scheduling
- Matrix-chain Multiplication · 0/1 Knapsack Problem
- Longest Common Subsequence · Optimal Binary Tree
- Warshall's algorithm for transitive closure implemented by iterations
- Floyd's algorithms for all-pairs shortest paths · Optimal Polygon Triangulation · Floyd-Warshall's Algorithm

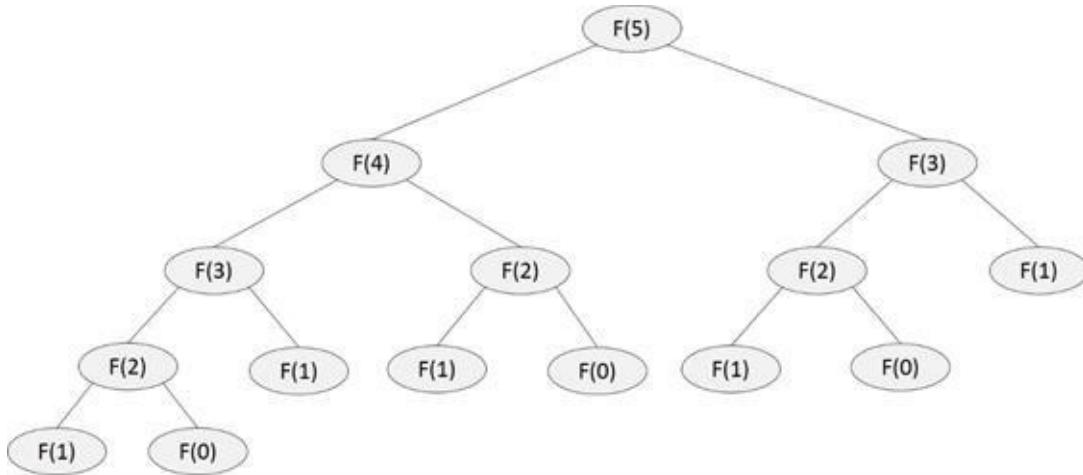
Steps for solving / recognizing if DP applies.

- 1) Optimal Substructure: Try to find if there is a recursive relation between problem and sub-problem.
- 2) Write recursive relation of the problem. (Observe Overlapping Sub-problems at this step.)
- 3) Compute the value of sub-problems in a bottom up fashion and store this value in some table.
- 4) Construct the optimal solution from the value stored in step 3.
- 5) Repeat step 3 and 4 until you get your solution.

Problems on Dynamic programming Algorithm

Fibonacci numbers

```
public static int fibonacci(int n) {  
    if (n <= 1) {  
        return n;  
    }  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}
```



Using divide and conquer same sub-problem is solved again and again, which reduce the performance of the algorithm. This algorithm has an exponential Time Complexity.

Same problem of Fibonacci can be solved in linear time if we sort the results of sub-problems.

```
public static int fibonacci2(int n) {  
    int first = 0;  
    int second = 1;  
    int i, temp = 0;  
  
    if (n == 0)  
        return 0;  
    else if (n == 1)  
        return 1;
```

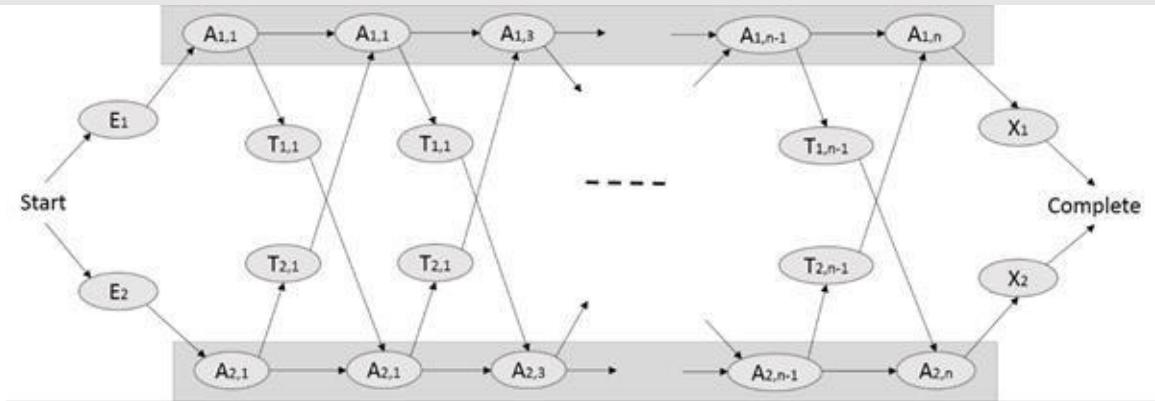
```

return first;
else if (n == 1)
return second;
i = 2;
while (i <= n) {
temp = first + second;
first = second;
second = temp;
i += 1;
}
return temp;
}

```

Using this algorithm, we will get Fibonacci in linear Time Complexity and constant Space Complexity.

Assembly-line Scheduling



We consider the problem of calculating the least amount of time necessary to build a car when using a manufacturing chain with two assembly lines, as shown in the figure

The problem variables:

- e[i]: entry time in assembly line i
- x[i]: exit time from assembly line i
- a[i,j]: Time required at station S[i,j] (assembly line i, stage j)
- t[i,j]: Time required to transit from station S[i,j] to the other assembly line

Your program must calculate:

- The least amount of time needed to build a car
- The array of stations to traverse in order to assemble a car as fast as possible.

The manufacturing chain will have no more than 50 stations.

If we want to solve this problem in the brute force approach, there will be in total 2^n Different combinations so the Time complexity is $O(2^n)$

Step 1: Characterizing the class of the optimal solution

To calculate the fastest assembly time, we only need to know the fastest time to $S1_n$ and the fastest time to $S2_n$, including the assembly time for the nth part. Then we choose between the two exit points by taking into consideration the extra time required, x_1 and x_2 . To compute the fastest time to $S1_n$ we only need to know the fastest time to $S1_{n-1}$ and to $S2_{n-1}$. Then there are only two choices.

Step 2: A recursive definition of the values to be computed

$$f1[j] = \begin{cases} e1 + a1, & \text{if } j = 1 \\ \min(f1[j - 1] + a1, j, f2[j - 1] + t2, j - 1 + a1, j) & \text{if } j \geq 2 \end{cases}$$

$$f2[j] = \begin{cases} e2 + a2, & \text{if } j = 1 \\ \min(f2[j - 1] + a2, j, f1[j - 1] + t1, j - 1 + a2, j) & \text{if } j \geq 2 \end{cases}$$

Step 3: Computing the fastest time finally, compute f^* as

Step 4: Computing the fastest path compute as $l_i[j]$ as the choice made for $f_i[j]$ (whether the first or the second term gives the minimum). Also, compute the choice for f^* as l^* .

FASTEST-WAY(a, t, e, x, n)

```
f1[1] ← e1 + a1,1  
f2[1] ← e2 + a2,1  
for j ← 2 to n do  
  if f1[j - 1] + a1,j ≤ f2[j - 1] + t2,j-1 + a1,j  
    then f1[j] ← f1[j - 1] + a1, j  
    l1[j] ← 1
```

```

else f1[j] ← f2[j - 1] + t2,j-1 + a1,j
l1[j] ← 2
if f2[j - 1] + a2,j ≤ f1[j - 1] + t1,j-1 + a2,j
then f2[j] ← f2[j - 1] + a2,j
l2[j] ← 2
else f2[j] ∞ f1[j - 1] + t1,j-1 + a2,j
l2[j] ← 1
if f1[n] + x1 ≤ f2[n] + x2
then f* = f1[n] + x1
l* = 1
else f* = f2[n] + x2
l* = 2

```

Largest Increasing subsequence

Given an array of integers, you need to find largest subsequence in increasing order. You need to find largest increasing sequence by selecting the elements from the given array such that their relative order does not change.

Input = [10, 12, 9, 23, 25, 55, 49, 70]

Output will be 6 corresponds to [10, 12, 23, 25, 49, 70]

Let V be the input array.

The optimal substructure in this problem for ith is defined as:

$LIS(k) = \max (LIS(j)) + 1$ such that $j < k$ and $v[j] < v[k]$

```

function LIS(v[], n)
    for i ← 1 to n do
        length[i] ← 1
        for j ← 0 to i do
            if v[i] < v[j] and length[i] < length[j]+1 then
                length[ i ] ← length[i] +1

    return max (length[i] where 1≤i≤n)

```

function **LIS**(arr):

 maxValue = 0

```

size = len(arr)
lis = [1]*size
for i in range(size):
    for j in range(i):
        if arr[j] < arr[i] and lis[i] < lis[j] + 1:
            lis[i] = lis[j] + 1

    if maxValue < lis[i]:
        maxValue = lis[i]

return maxValue

```

arr = [10, 12, 9, 23, 25, 55, 49, 70]

print LIS(arr)

Time Complexity: $O(n^2)$, Space Complexity: $O(n)$.

Longest Bitonic Subsequence

Given an array, you need to find longest bitonic subsequence.

Function **LBS**(arr):

```

maxValue = 0
size = len(arr)
lis = [1]*size
lds = [1]*size
for i in range(size):
    for j in range(i):
        if arr[j] < arr[i] and lis[i] < lis[j] + 1:
            lis[i] = lis[j] + 1

    for i in reversed(range(size)):
        for j in reversed(range(i, size)):
            if arr[j] < arr[i] and lds[i] < lds[j] + 1:
                lds[i] = lds[j] + 1

for i in range(size):

```

```

maxValue = max((lis[i] + lds[i] - 1), maxValue)
return maxValue

arr = [1 , 6 , 3, 11, 1, 9 , 5 , 12 , 3 , 14 , 6 , 17, 3, 19 , 2 , 19]
print LBS(arr)

```

Matrix chain multiplication

Same problem is also known as Matrix Chain Ordering Problem or Optimal-parenthesization of matrix problem.

Given a sequence of matrices, $M = M_1, \dots, M_n$. The goal of this problem is to find the most efficient way to multiply these matrices. The goal is not to perform the actual multiplication, but to decide the sequence of the matrix multiplications, so that the result will be calculated in minimal operations.

To compute the product of two matrices of dimensions $p \times q$ and $q \times r$, pqr number of operations will be required. Matrix multiplication operations are associative in nature. Therefore, matrix multiplication can be done in many ways.

For example, M_1, M_2, M_3 and M_4 , can be fully parenthesized as:

- ($M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$)
- ($M_1 \cdot ((M_2 \cdot M_3) \cdot M_4)$)
- ($(M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$)
- ($((M_1 \cdot M_2) \cdot M_3) \cdot M_4$)
- ($(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$)

For example,

Let M_1 dimensions are 10×100 , M_2 dimensions are 100×10 , and M_3 dimensions are 10×50 .

$$((M_1 \cdot M_2) \cdot M_3) = (10 \cdot 100 \cdot 10) + (10 \cdot 10 \cdot 50) = 15000$$

$$(M_1 \cdot (M_2 \cdot M_3)) = (100 \cdot 10 \cdot 50) + (10 \cdot 100 \cdot 50) = 100000$$

Therefore, in this problem we need to parenthesize the matrix chain so that total multiplication cost is minimized.

Given a sequence of n matrices M_1, M_2, \dots, M_n . And their dimensions are $p_0, p_1, p_2, \dots, p_n$.

Where matrix A_i has dimension $p_i - 1 \times p_i$ for $1 \leq i \leq n$. Determine the order of multiplication that minimizes the total number of multiplications.

If you try to solve this problem using the brute-force method, then you will find all possible parenthesization. Then you will compute the cost of multiplications. Thereafter you will pick the best solution. This approach will be exponential in nature.

There is an insufficiency in the brute force approach. Take an example of M_1, M_2, \dots, M_n . When you have calculated that $((M_1 \cdot M_2) \cdot M_3)$ is better than $(M_1 \cdot (M_2 \cdot M_3))$ so there is no point of calculating then combinations of $(M_1 \cdot (M_2 \cdot M_3))$ with (M_4, M_5, \dots, M_n) .

Optimal substructure:

Assume that $M(1, N)$ is the optimum cost of production of the M_1, \dots, M_n .

An array $p[]$ to record the dimensions of the matrices.

$P[0] = \text{row of } M_1$

$p[i] = \text{col of } M_i \quad 1 \leq i \leq N$

For some k

$$M(1, N) = M(1, K) + M(K+1, N) + p_0 * p_k * p_N$$

If $M(1, N)$ is minimal then both $M(1, K)$ & $M(K+1, N)$ are minimal.

Otherwise, if there is some $M'(1, K)$ is there whose cost is less than $M(1..K)$, then $M(1..N)$ can't be minimal and there is a more optimal solution possible.

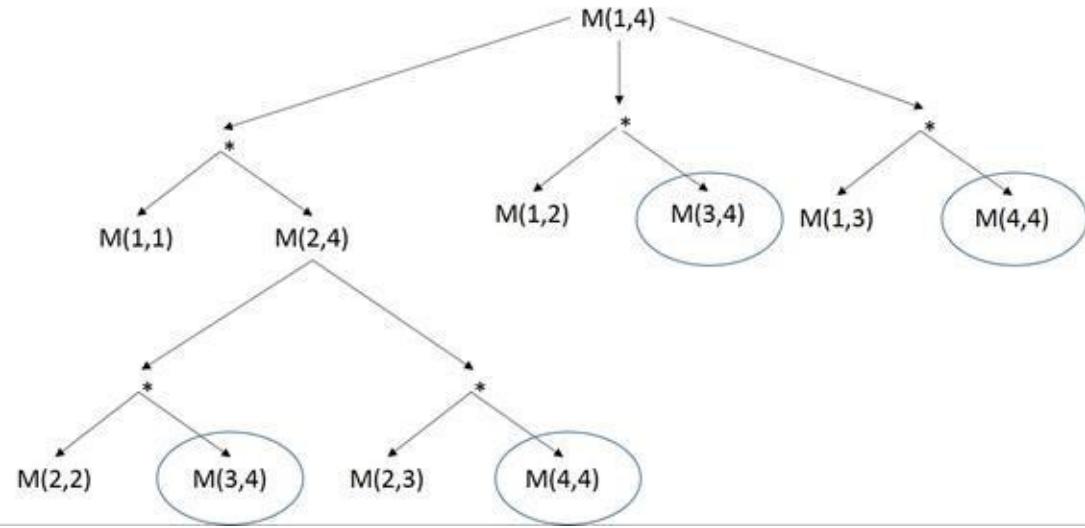
For some general i and j .

$$M(i, j) = M(i, K) + M(K+1, j) + p_{i-1} * p_k * p_j$$

Recurrence relation:

$$M(i, j) = \begin{cases} 0 & \text{if } i = j \\ \min \{M(i, k) + M(k, j) + p_{i-1} * p_k * p_j\} & i \leq k < j \end{cases}$$

Overlapping Sub-problems:



Directly calling recursive function will lead to calculation of same sub-problem multiple times. This will lead to exponential solution.

```
Algorithm MatrixChainMultiplication(p[])
for i := 1 to n
```

```
    M[i, i] := 0;
```

```
    for l = 2 to n // l is the moving line
```

```
        for i = 1 to n - l + 1
```

```
            j = i + l - 1;
```

```
            M[i, j] = min {M(i, k) + M(k, j) + pi * ... * pk * pj }
```

```
i <= k < j
```

Time Complexity will $O(n^3)$

Constructing optimal parenthesis Solution

Use another table $s[1..n, 1..n]$. Each entry $s[i, j]$ records the value of k such that the optimal parenthesization of $M_i M_{i+1} \dots M_j$ splits the product between M_k and M_{k+1} .

```
Algorithm MatrixChainMultiplication(p[])
for i := 1 to n
```

```
    M[i, i] := 0;
```

```
    for l = 2 to n // l is the moving line
```

```
        for i = 1 to n - l + 1
```

```
            j = i + l - 1;
```

```


$$M[i, j] = \min_{i \leq k < j} \{M(i, k) + M(k, j) + p_{i-1} * p_k * p_j\}$$


$$S[i, j] = k \text{ for } \min_{i \leq k < j} \{M(i, k) + M(k, j) + p_{i-1} * p_k * p_j\}$$


```

Algorithm MatrixChainMultiplication(p[])

```

for i := 1 to n
  M[i, i] := 0;
  for l = 2 to n // l is the moving line
    for i = 1 to n - l + 1
      j = i + l - 1;
      for k = i to j
        if( (M[i, k] + M(k, j) + p_{i-1} * p_k * p_j) < M[i, j])
          M[i, j] = (M[i, k] + M(k, j) + p_{i-1} * p_k * p_j)
          S[i, j] = k

```

Algorithm PrintOptimalParenthesis(s[], i, j)

```

If i = j
  Print A_i
Else
  Print "("
  PrintOptimalParenthesis(s[], i, s[i, j])
  PrintOptimalParenthesis(s[], s[i, j], j)
  Print ")"

```

Longest Common Subsequence

Let X = {x₁, x₂, ..., x_m} is a sequence of characters and Y = {y₁, y₂, ..., y_n} is another sequence.

Z is a subsequence of X if it can be derived by deleting some elements of X. Z is a subsequence of Y if it can be derived by deleting some elements from Y. Z is LCS of it is subsequence to both X and Y, and length of all the subsequence is less than Z.

Optimal Substructure:

Let X = <x₁, x₂, ..., x_m> and Y = <y₁, y₂, ..., y_n> be two sequences, and let Z

= $< z_1, z_2, \dots, z_k >$ be a LCS of X and Y.

- If $x_m = y_n$, then $z_k = x_m = y_n \Rightarrow Z_{k-1}$ is a LCS of X_{m-1} and Y_{n-1}
- If $x_m \neq y_n$, then:
 - o $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.
 - o $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1} .

Recurrence relation

Let $c[i, j]$ be the length of the longest common subsequence between $X = \{x_1, x_2, \dots, x_i\}$ and $Y = \{y_1, y_2, \dots, y_j\}$.

Then $c[n, m]$ contains the length of an LCS of X and Y

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i - 1, j], c[i, j - 1]) & \text{otherwise} \end{cases}$$

Algorithm LCS($X[], m, Y[], n$)

```
for i = 1 to m
    c[i,0] = 0
    for j = 1 to n
        c[0,j] = 0;
    for i = 1 to m
        for j = 1 to n
            if X[i] == Y[j]
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = ↗
            else
                if c[i-1,j] ≥ c[i,j-1]
                    c[i,j] = c[i-1,j]
                    b[i,j] = ↑
                else
                    c[i,j] = c[i,j-1]
                    b[i,j] = ←
```

Algorithm PrintLCS($b[], X[], i, j$)

```
if i = 0
    return
```

```

if j = 0
return
if b[i, j] = ↗
PrintLCS (b[],X[], i - 1, j - 1)
print X[i]
else if b[i, j] = ↑
PrintLCS (b[],X[], i - 1, j)
else
PrintLCS (b[],X[], i, j - 1)

```

Coin Exchanging problem

How can a given amount of money N be made with the least number of coins of given denominations D= {d₁... d_n}?

For example, Indian coin system {5, 10, 20, 25, 50,100}. Suppose we want to give change of a certain amount of 40 paisa.

We can make a solution by repeatedly choosing a coin \leq to the current amount, resulting in a new amount. The greedy solution always choose the largest coin value possible.

For 40 paisa: {25, 10, and 5}

This is how billions of people around the globe make change every day. That is an approximate solution of the problem. But this is not the optimal way, the optimal solution for the above problem is {20, 20}

Step (I): Characterize the class of a coin-change solution.

Define C [j] to be the minimum number of coins we need to make a change for j cents.

If we knew that an optimal solution for the problem of making change for j cents used a coin of denomination d_i, we would have:

$$C[j] = 1 + C[j - d_i]$$

Step (II): Recursively defines the value of an optimal solution.

$$c[j] = \begin{cases} \text{infinite if } j < 0 \\ 0 \text{ if } j = 0 \\ 1 + \min(c[j - d_i]) \text{ if } 1 \leq i \leq k \text{ and } j \geq 1 \end{cases}$$

Step (III): Compute values in a bottom-up fashion.

Algorithm CoinExchange($n, d[], k$)

```
C[0] = 0
for j = 1 to n do
    C[j] = infinite
    for i = 1 to k do
        if j < di and 1+C[j - di] < C[j] then
            C[j] = 1+C[j - di]
return C
```

Complexity: $O(nk)$

Step (iv): Construct an optimal solution

We use an additional list Deno[1.. n], where Deno[j] is the denomination of a coin used in an optimal solution.

Algorithm CoinExchange($n, d[], k$)

```
C[0] = 0
for j = 1 to n do
    C[j] = infinite
    for i = 1 to k do
        if j < di and 1+C[j - di] < C[j] then
            C[j] = 1+C[j - di]
            Deno[j] = di
return C
```

Algorithm PrintCoins(Deno[], j)

```
if j > 0
    PrintCoins (Deno, j - Deno[j])
    print Deno[j]
```

CHAPTER 19: BACKTRACKING

Introduction

Suppose there is a lock, which produce some “click” sound when correct digit is selected for any level. To open it you just need to find the first digit, then find the second digit, then find the third digit and done. This will be a greedy algorithm and you will find the solution very quickly.

However, let us suppose the lock is some old one and it creates same sound not only at the correct digit, but also at some other digits. Therefore, when you are trying to find the digit of the first ring, then it may product sound at multiple instances. So, at this point you are not directly going straight to the solution, but you need to test various states and in case those states are not the solution you are looking for, then you need to backtrack one step at a time and find the next solution. Sure, this intelligence/ heuristics of click sound will help you to reach your goal much faster. These functions are called Pruning function or bounding functions.

Problems on Backtracking Algorithm

N Queens Problem

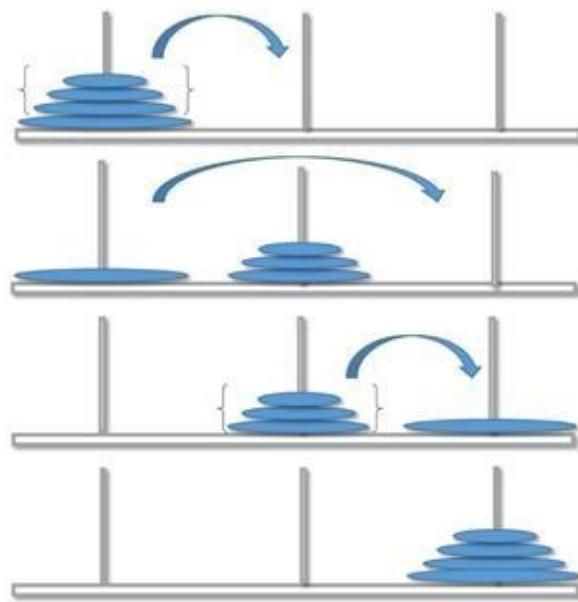
There are N queens given, you need to arrange them in a chessboard on NxN such that no queen should attack each other.

```
public static boolean Feasible(int[] Q, int k) {  
    for (int i = 0; i < k; i++) {  
        if (Q[k] == Q[i] || Math.abs(Q[i] - Q[k]) == Math.abs(i - k)) {  
            return false;  
        }  
    }  
    return true;  
}  
public static void NQueens(int[] Q, int k, int n) {  
    if (k == n) {  
        print(Q, n);  
        return;  
    }  
    for (int i = 0; i < n; i++) {  
        Q[k] = i;  
        if (Feasible(Q, k)) {  
            NQueens(Q, k + 1, n);  
        }  
    }  
}  
public static void main(String[] args) {  
    int[] Q = new int[8];;  
    NQueens(Q, 0, 8); }
```

Tower of Hanoi

The Tower of Hanoi puzzle, disks need to be moved from one pillar to another such that any large disk cannot rest above any small disk.

This is a famous puzzle in the programming world; its origins can be tracked back to India. "There is a story about an Indian temple in Kashi Viswanathan which contains a large room with three timeworn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient Hindu prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma the creator of the universe, since the beginning of time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the prophecy, when the last move of the puzzle will be completed, the world will end." ;) ;) ;)



```

public static void TOHUtil(int num, char from, char to, char temp) {
    if (num < 1) {
        return;
    }

    TOHUtil(num - 1, from, temp, to);
    System.out.println("Move disk " + num + " from peg " + from + " to peg " +
to);
    TOHUtil(num - 1, temp, to, from);
}

public static void TowersOfHanoi(int num) {
    System.out.println("The sequence of moves involved in the Tower of Hanoi
are :");
}

```

```
    TOHUtil(num, 'A', 'C', 'B');
```

```
}
```

```
public static void main(String[] args) {
```

```
    TowersOfHanoi(3);
```

```
}
```

CHAPTER 20: COMPLEXITY THEORY

Introduction

Computational complexity is the measurement of how much resources are required to solve some problem.

There are two types of resources:

1. Time: Number of steps required to solve a problem
2. Space: Amount of memory required to solve a problem

Decision problem

Much of Complexity theory deals with decision problems. A decision problem always has a yes or no answer.

Many problems can be converted to a decision problem that have answer as yes or no. For example:

1. Searching: The problem of searching element can be a decision problem if we ask, “Find if a particular number is there in the array?”.
2. Sorting of list and find if the array is sorted, you can make a decision problem, “Is the array is sorted in increasing order?”.
3. Graph coloring algorithms: this is can also be converted to a decision problem. Can we do the graph coloring by using X number of colors?
4. Hamiltonian cycle: Is there is a path from all the nodes, each node is visited exactly once and comes back to the starting node without breaking?

Complexity Classes

Problems are divided into many classes based on their difficulty. Or how difficult it is to find if the given solution is correct or not.

Class P problems

The class P consists of a set of problems that can be solved in polynomial time. The complexity of a P problem is $O(n^k)$ where n is input size and k is some constant (it cannot depend on n).

Class P Definition: The class P contains all decision problems for which a Turing machine algorithm leads to the “yes/no” answer in a definite number of steps bounded by a polynomial function.

For example:

Given a sequence $a_1, a_2, a_3, \dots, a_n$. Find if a number X is there in this list.

We can search, the number X in this list in linear time (polynomial time)

Another example:

Given a sequence $a_1, a_2, a_3, \dots, a_n$. If we are asked to sort the sequence.

We can sort an array in polynomial time using Bubble-Sort, this is also linear time.

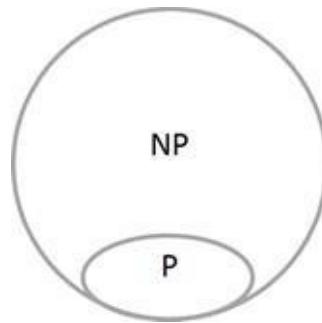
Note: **$O(\log n)$** is also polynomial. Any algorithm, which has complexity less than some $O(n^k)$, is also polynomial.

- Some problem of P class is:
 - 1. Shortest path
 - 2. Minimum spanning tree
 - 3. Maximum problem.
 - 4. Max flow graph problem.
 - 5. Convex hull

Class NP problems

Set of problems for which there is a polynomial time checking algorithm. Given a solution if we can check in a polynomial time if that solution is correct or not then, the problem is NP problem.

Class NP Definition: The class NP contains all decision problems for which, given a solution, there exists a polynomial time “proof” or “certificate” that can verify if the solution is the right “yes/no” answer



Note: There is no guarantee that you will be able to solve this problem in polynomial time. However, if a problem is an NP problem, then you can verify an answer in polynomial time.

NP does not mean “non-polynomial”. Actually, it is “Non-Deterministic Polynomial” type of problem. They are the kind of problems that can be solved in polynomial time by a Non-Deterministic Turing machine. At each point, all the possibilities are executed in parallel. If there are n possible choices, then all n cases will be executed in parallel. We do not have non-deterministic computers. Do not be confused with parallel computing because the number of CPU is limited in parallel computing it may be 16 core or 32 core, but it cannot be N -Core.

In short NP problems are those problems for which, if a solution is given, we can verify that solution (if it is correct or not) in polynomial time.

Boolean Satisfiability problem

A Boolean formula is satisfied if there exist some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.

$$(A_1 \vee A_2 \dots) \wedge (A_2 \vee A_4 \dots) \dots \wedge (\dots \vee A_N)$$

There are in total N Different Boolean Variables A₁, A₂... A_N. There are an M number of brackets. Each bracket has K variables.

There is N variable so the number of solutions will be 2ⁿ

And to verify if the solutions really evaluate the equation to 1 will take total (2ⁿ * km) steps

In given solution of this problem you can find if the formula satisfies or not in KM steps.

Hamiltonian cycle

Hamiltonian cycle is a path from all the nodes of a graph, each node is visited exactly once and come back to the starting node without breaking.

This is an NP problem, if you have a solution to it, then you just need to see if all the nodes are there in the path and you came back to where you have started and you are done? The checking is done in linear time and you are done.

Determining whether a directed graph has a Hamiltonian cycle or not does not have a polynomial time algorithm. O(n!)

However, if someone has given you a sequence of vertices, determining whether that sequence forms a Hamiltonian cycle can be done in polynomial time (Linear time).

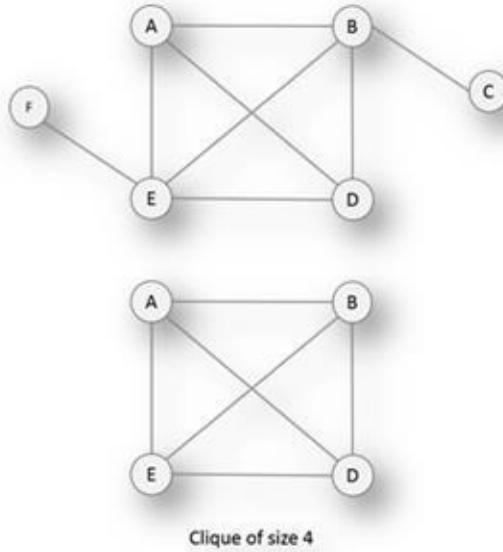
Hamiltonian cycles are in NP

Clique Problem

In a given graph find if there is a clique of size K or more. A clique is a subset of nodes, which are completely connected to each other.

This problem is NP problem. Given a set of nodes, you can very easily find out whether it is a clique or not.

For example:



Prime Number

Finding Prime number is NP. Given a solution, it is easy to find if it is a Prime or not in polynomial time. Finding prime numbers is important which is heavily used in cryptography.

```
int isPrime(int n) {
    int answer = (n > 1) ? 1 : 0;
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            answer = 1;
            break;
        }
    }
    return answer;
}
```

Checking will happen until the square root of number so the Time complexity is $O(\sqrt{n})$. Hence, prime number finding is an NP problem as we can verify the solution in polynomial time.

Graph theory have wonderful set of problems

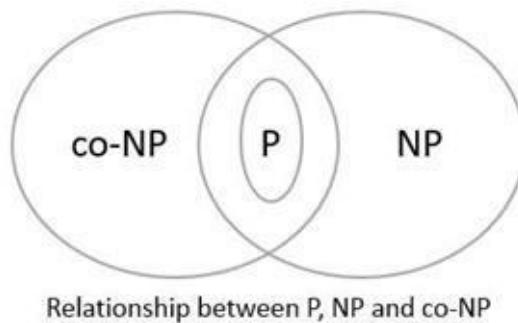
- Shortest path algorithms?
- Longest path is NP complete.

- Eulerian tours is a polynomial time problem.
- Hamiltonian tours is a NP complete

Class co-NP

Set of problems for which there is a polynomial time checking algorithm. Given a solution, if we can check in a polynomial time if that solution is incorrect the problem is co-NP problem.

Class co-NP Definition: The class co-NP contains all decision problems such that there exists a polynomial time proof that can verify if the problem does not have the right “yes/no” answer.



Relationship between P, NP and co-NP

Class P is Subset of Class NP

All problems that are P also are NP ($P \subseteq NP$). Problem set P is a subset of problem set NP.

Searching

If we have some number sequence $a_1, a_2, a_3, \dots, a_n$. We already know that searching a number X inside this list is of type P.

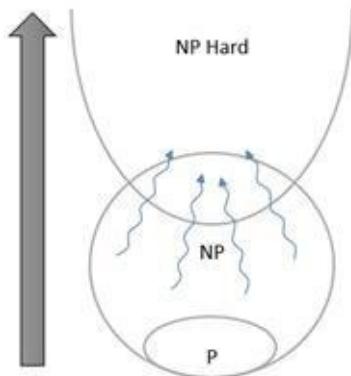
If it is given that number X is inside this sequence, then we can verify by looking into every entry again and find if the answer is correct in polynomial time (linear time.)

Sorting

Another example of sorting a number sequence, if it is given that the array $b_1, b_2, b_3, \dots, b_n$ is sorted then we can loop through this given list and find if the array is really sorted in polynomial time (linear time again.)

NP-Hard:

A problem is NP-Hard if all the problems in NP can be reduced to it in polynomial time.

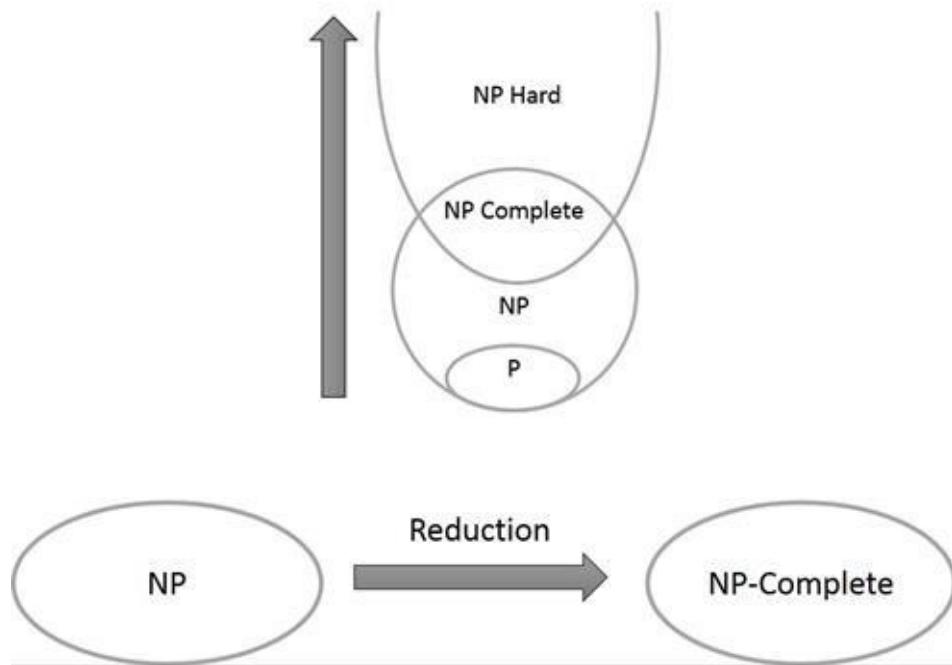


NP-Complete Problems

Set of problem is NP-Complete if it is an NP problem and an NP-Hard problem.
It should follow both the properties:

- 1) Its solutions can be verified in a polynomial time.
- 2) All problems of NP are reduced to NP complete problems in polynomial time.

You can always reduce any NP problem into NP-Complete in polynomial time.
In addition, when you get the answer to the problem, then you can verify this solution in polynomial time.



Any NP problem is polynomial reduced to NP-Complete problem, if we can find a solution to a single NP-Complete problem in polynomial time, then we can solve all the NP problems in polynomial time. However, so far no one is able to find any solution of NP-Complete problem in polynomial time.

$P \neq NP$

Reduction

It is a process of transformation of one problem into another problem. The transformation time should be polynomial. If a problem A is transformed into B and we know the solution of B in polynomial time, then A can also be solved in polynomial time.

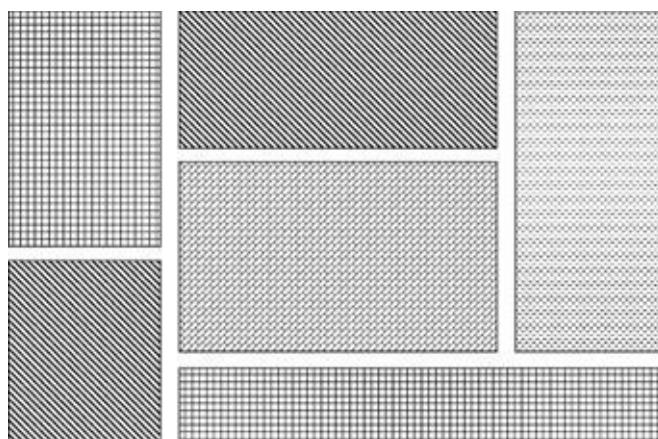
For example, Quadratic Equation Solver: We have a Quadratic Equation Solver, which solves equation of the form $ax^2 + bx + c = 0$. It takes Input a, b, c and generates output r₁, r₂.

Now try to solve a linear equation $2x+4=0$. Using reduction second equation can be transformed to the first equation.

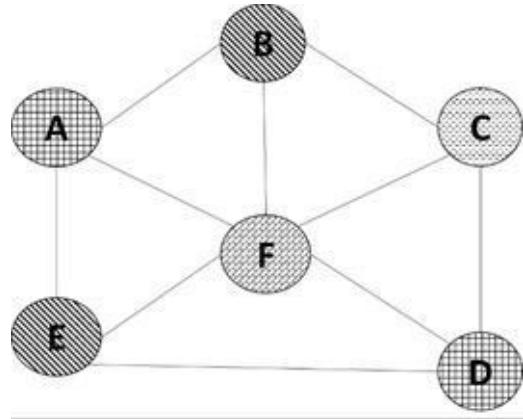
$$2x+4 = 0$$

$$X^2 + 2x + 4 = 0$$

ATLAS: We have an atlas and we need to color maps so that no two countries have the same color. Let us suppose below is the various countries. Moreover, different patterns represent different color.



We can see that same problem of atlas coloring can be reduced to graph coloring and if we know the solution of graph coloring then same solution can work for atlas coloring too. Where each node of the graph represents one country and the adjacent country relation is represented by the edges between nodes.



The sorting problem reduces (\leq) to Convex Hull problem.

SAT reduces (\leq) to 3SAT

End Note

Nobody has come up with such a polynomial-time algorithm to solve a NP-Complete problem. Many important algorithms depends upon it. However, at the same time nobody has proven that no polynomial time algorithm is possible. There is a million US dollars for anyone who can solve any NP Complete problem in polynomial time. The whole economy of the world will fall as most of the banks depend on public key encryption which will be easy to break if P=NP solution is found.

APPENDIX

Appendix A

Algorithms	Time Complexity
Binary Search in a sorted array of N elements	$O(\log N)$
Reversing a string of N elements	$O(N)$
Linear search in an unsorted array of N elements	$O(N)$
Compare two strings with lengths L1 and L2	$O(\min(L1, L2))$
Computing the Nth Fibonacci number using dynamic programming	$O(N)$
Checking if a string of N characters is a palindrome	$O(N)$
Finding a string in another string using the Aho-Corasick algorithm	$O(N)$
Sorting an array of N elements using Merge-Sort/Quick-Sort/Heap-Sort	$O(N * \log N)$
Sorting an array of N elements using Bubble-Sort	$O(N!)$
Two nested loops from 1 to N	$O(N!)$
The Knapsack problem of N elements with capacity M	$O(N * M)$
Finding a string in another string – the naive approach	$O(L1 * L2)$
Three nested loops from 1 to N	$O(N^3)$
Twenty-eight nested loops ... you get the idea	$O(N^{28})$
Stack	
Adding a value to the top of a stack	$O(1)$
Removing the value at the top of a stack	$O(1)$
Reversing a stack	$O(N)$
Queue	
Adding a value to end of the queue	$O(1)$
Removing the value at the front of the queue	$O(1)$
Reversing a queue	$O(N)$
Heap	
Adding a value to the heap	$O(\log N)$
Removing the value at the top of the heap	$O(\log N)$
Hash	
Adding a value to a hash	$O(1)$
Checking if a value is in a hash	$O(1)$