DATA STRUCTURES AND ALGORITHMS MADE EASY

Success Key for:

- Campus Preparation
- Degree/Masters Course Preparation
- Instructor's
- GATE Preparation
- Big Job hunters: Microsoft, Google, Amazon, Yahoo, Facebook, Adobe and many more
- Reference Manual for Working People

Multiple smart solutions with different complexities

Narasimha Karumanchi

M-Tech, IIT Bombay

Founder of CareerMonk.com

CareerMonk Publications

To My Parents
~Laxmi and Modaiah

To My Family Members

To My Friends

To IIT Bombay

To All Hard Workers

Designed by Narasimha Karumanchi

Printed in India

Acknowledgements

I would like to express my gratitude to the many people who saw me through this book, to all those who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and design. In particular, I would like to thank the following individuals.

I would like to thank *Ram Mohan Mullapudi* for encouraging me when I was at IIT Bombay. He is the first person who taught me the importance of *algorithms* and its *design*. From that day I keep on updating myself.

I would like to thank *Vamshi Krishna* [*Mentor Graphics*] and *Kalyani Tummala* [*Xilinx*] for spending time in reviewing this book and providing me the valuable suggestions almost every day.

I would like to thank *Sobhan* [*Professor IIT, Hyderabad*] for spending his valuable time in reviewing the book and suggestions. His review gave me the confidence in the quality of the book.

I would like to thank *Kiran* and *Laxmi* [Founder's of *TheGATEMATE.com*] for approaching me for teaching Data Structures and Algorithms at their training centers. They are the primary reason for initiation of this book.

My *friends* and *colleagues* have contributed greatly to the quality of this book. I thank all of you for your help and suggestions.

Special thanks should go to my wife, *Sailaja* for her encouragement and help during writing of this book.

Last but not least, I would like to thank Director's of *Guntur Vikas College*, *Gopala Krishna Murthy* [Director of *ACE Engineering Academy*], *TRC Bose* [Former Director of *APTransco*] and *Venkateswara Rao* [*VNR Vignanajyothi Engineering College*, *Hyderabad*] for helping me and my family during our studies.

-Narasimha Karumanchi
M-Tech, IIT Bombay
Founder of CareerMonk.com

Preface

Dear Reader,

Please Hold on! I know many people do not read preface. But I would like to strongly recommend reading preface of this book at least. This preface has *something different* from regular prefaces.

As a *job seeker* if you read complete book with good understanding, I am sure you will challenge the interviewer's and that is the objective of this book.

If you read as an *instructor*, you will give better lectures with easy go approach and as a result your students will feel proud for selecting Computer Science / Information Technology as their degree.

This book is very much useful for the *students* of *Engineering* and *Masters* during their academic preparations. All the chapters of this book contain theory and their related problems as many as possible. There a total of approximately 700 algorithmic puzzles and all of them are with solutions.

If you read as a *student* preparing for competition exams for Computer Science/Information Technology], the content of this book covers *all* the *required* topics in full details. While writing the book, an intense care has been taken to help students who are preparing for these kinds of exams.

In all the chapters you will see more importance given to problems and analyzing them instead of concentrating more on theory. For each chapter, first you will see the basic required theory and then followed by problems.

For many of the problems, *multiple* solutions are provided with different complexities. We start with *brute force* solution and slowly move towards the *best solution* possible for that problem. For each problem we will try to understand how much time the algorithm is taking and how much memory the algorithm is taking.

It is *recommended* that, at least one complete reading of this book is required to get full understanding of all the topics. In the subsequent readings, readers can directly go to any chapter and refer. Even though, enough readings were given for correcting the errors, due to human tendency there could be some minor typos in the book. If any such typos found, they will be updated at *www.CareerMonk.com*. I request readers to constantly monitor this site for any corrections, new problems and solutions. Also, please provide your valuable suggestions at: *Info@CareerMonk.com*.

Wish you all the best. Have a nice reading.

-Narasimha Karumanchi
M-Tech, IIT Bombay
Founder of CareerMonk.com

INTRODUCTION

Chapter-1

The objective of this chapter is to explain the importance of analysis of algorithms, their notations, relationships and solving as many problems as possible. We first concentrate on understanding the basic elements of algorithms, importance of analysis and then slowly move towards analyzing the algorithms with different notations and finally the problems. After completion of this chapter you should be able to find the complexity of any given algorithm (especially recursive functions).

1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the below equation:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of above equation. The important thing that we need to understand is, the equation has some names (x and y) which hold values (data). That means, the *names* (x and y) are the place holders for representing data. Similarly, in computer science we need something for holding data and *variables* are the facility for doing that.

1.2 Data types

In the above equation, the variables x and y can take any values like integral numbers (10, 20 etc...), real numbers (0.23, 5.5 etc...) or just 0 and 1. To solve the equation, we need to relate them to kind of values they can take and *data type* is the name being used in computer science for this purpose.

A *data type* in a programming language is a set of data with values having predefined characteristics. Examples of data types are: integer, floating point unit number, character, string etc...

Computer memory is all filled with zeros and ones. If we have a problem and wanted to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers are providing the facility of data types.

For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes etc... This says that, in memory we are combining 2 bytes (16 bits) and calling it as *integer*. Similarly, combining 4 bytes (32 bits) and calling it as *float*. A data type reduces the coding effort. Basically, at the top level, there are two types of data types:

- System defined data types (also called *Primitive* data types)
- User defined data types

System defined data types (Primitive data types)

Data types which are defined by system are called *primitive* data types. The primitive data types which are provided by many programming languages are: int, float, char, double, bool, etc...

The number of bits allocated for each primitive data type depends on the programming languages, compiler and operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types the total available values (domain) will also changes. For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits) then the total possible values are -32,768 to +32,767 (-2^{15} *to* $2^{15}-1$). If it takes, 4 bytes (32 bits), then the possible values are between −2,147,483,648 to +2,147,483,648 (-2^{31} *to* $2^{31}-1$). Same is the case with remaining data types too.

User defined data types

If the system defined data types are not enough then most programming languages allows the users to define their own data types called as user defined data types. Good example of user defined data types are: structures in $C/C++$ and classes in $Java$.

For example, in the below case, we are combining many system defined data types and called it as user defined data type with name "*newType*". This gives more flexibility and comfort in dealing with computer memory.

```
struct newType {
        int data1;
        float data 2;
        ...
        char data;
};
```

1.3 Data Structure

Based on the above discussion, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. That means, a *data structure* is a specialized format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) *Linear data structures*: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). *Examples*: Linked Lists, Stacks and Queues.
2) *Non − linear data structures*: Elements of this data structure are stored/accessed in a non-linear order. *Examples*: Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system defined data types. We all know that, by default, all primitive data types (int, float, et..) supports basic operations like addition, subtraction etc... The system is providing the implementations for the primitive data types. For user defined data types also we need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general user defined data types are defined along with their operations.

To simplify the process of solving the problems, we generally combine the data structures along with their operations and are called *Abstract Data Types* (ADTs). An ADT consists of *two* parts:

1. Declaration of data
2. Declaration of operations

Commonly used ADTs *include*: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many other. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack etc...

While defining the ADTs do not care about implementation details. They come in to picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

1.5 What is an Algorithm?

Let us consider the problem of preparing an omelet. For preparing omelet, general steps we follow are:

1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
            1. If yes, then go out and buy.
            2. If no, we can terminate.
3) Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelet), giving step by step procedure for solving it. Formal definition of an algorithm can be given as:

An algorithm is the step-by-step instructions to solve a given problem.

Note: we do not have to prove each step of the algorithm.

1.6 Why Analysis of Algorithms?

To go from city "*A*" to city "*B*", there can be many ways of accomplishing this: by flight, by bus, by train and also by cycle. Depending on the availability and convenience we choose the one which suits us. Similarly, in computer science there can be multiple algorithms exist for solving the same problem (for example, sorting problem has many algorithms like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us determining which of them is efficient in terms of time and space consumed.

1.7 Goal of Analysis of Algorithms

The goal of *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developers effort etc.)

1.8 What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is number of elements in the input and depending on the problem type the input may be of different types. In general, we encounter the following types of inputs.

- Size of an array

- Polynomial degree
- Number of elements in a matrix
- Number of bits in binary representation of the input
- Vertices and edges in a graph

1.9 How to Compare Algorithms?

To compare algorithms, let us define few *objective measures*:
Execution times? *Not a good measure* as execution times are specific to a particular computer.

Number of statements executed? *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal Solution? Let us assume that we expressed running time of given algorithm as a function of the input size n (i.e., $f(n)$) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc...

1.10 What is Rate of Growth?

The rate at which the running time increases as a function of input is called *rate of growth*. Let us assume that you went to a shop for buying a car and a cycle. If your friend sees you there and asks what you are buying then in general we say *buying a car*. This is because, cost of car is too big compared to cost of cycle (approximating the cost of cycle to cost of car).

$$Total\ Cost = cost\_of\_car + cost\_of\_cycle$$
$$Total\ Cost \approx cost\_of\_car\ (approximation)$$

For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example in the below case, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate it to n^4. Since, n^4 is the highest rate of growth.

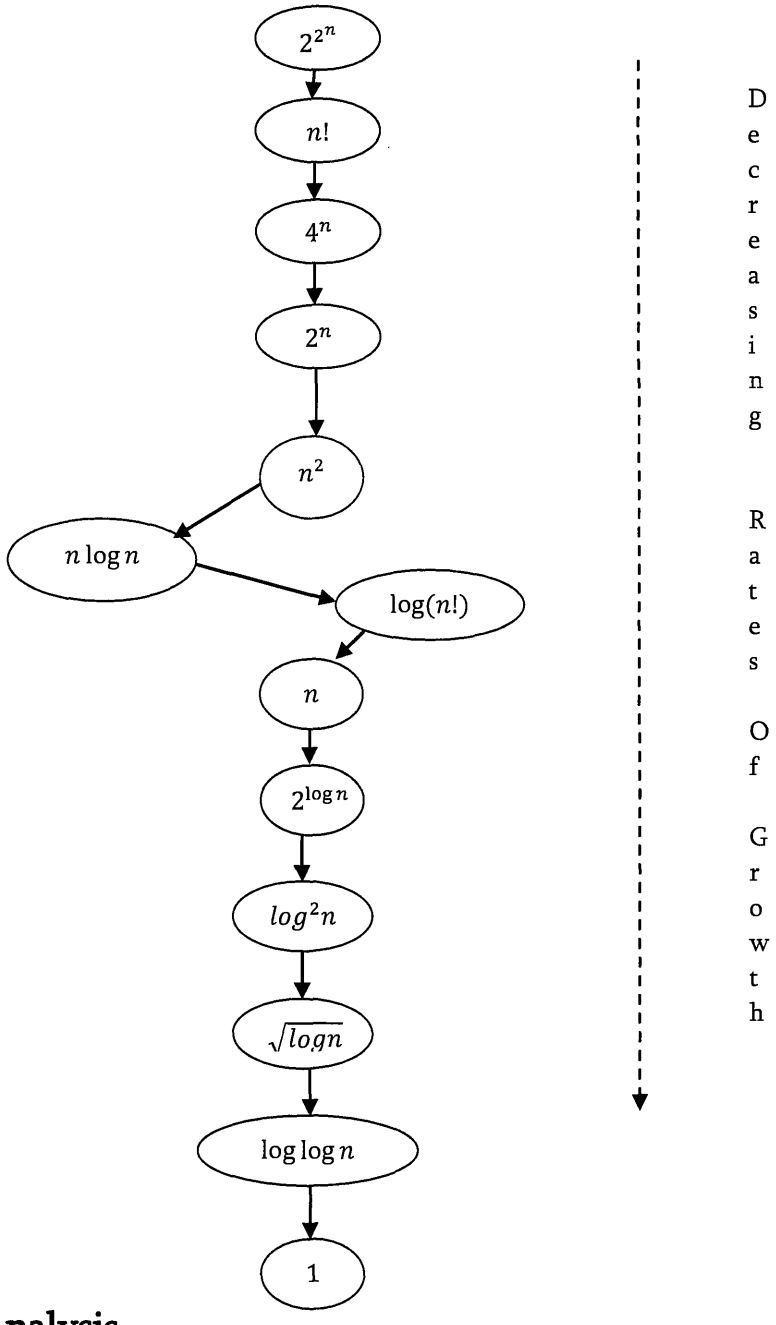
$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

1.11 Commonly used Rate of Growths

Below is the list of rate of growths which come across in remaining chapters.

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Below diagram shows the relationship between different rates of growth.



1.12 Types of Analysis

To analyze the given algorithm we need to know on what inputs the algorithm is taking less time (performing well) and on what inputs the algorithm is taking huge time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for case where it is taking the less time and other for case where it is taking the more time. In general the first case is called the *best case* and second case is called the *worst case* of the algorithm. To analyze an algorithm we need some kind of syntax and that forms the base for asymptotic analysis/notation. There are three types of analysis:

- Worst case
  - Defines the input for which the algorithm takes huge time.
  - Input is the one for which the algorithm runs the slower.
- Best case
  - Defines the input for which the algorithm takes lowest time.

- Input is the one for which the algorithm runs the fastest.

- **Average case**
  - Provides a prediction about the running time of the algorithm
  - Assumes that the input is random

$$Lower\ Bound <= Average\ Time <= Upper\ Bound$$

For a given algorithm, we can represent best, worst and average cases in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

$f(n) = n^2 + 500$, for worst case

$f(n) = n + 100n + 500$, for best case

Similarly, for average case too. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

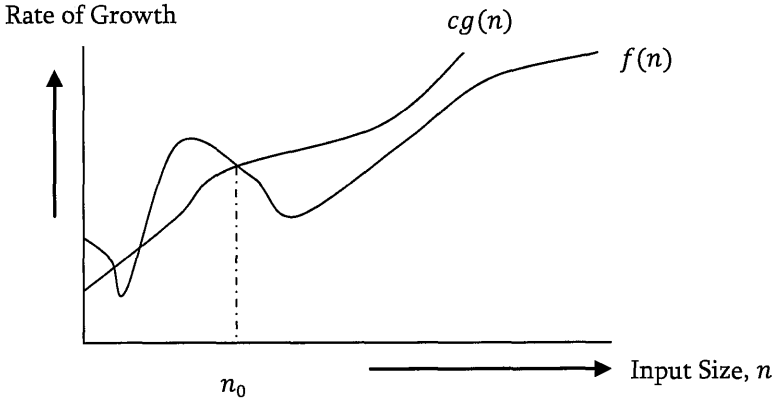
Having the expressions for best, average case and worst cases, for all the three cases we need to identify the upper and lower bounds. In order to represent these upper and lower bounds we need some kind syntax and that is the subject of following discussion. Let us assume that the given algorithm is represented in the form of function $f(n)$.

1.14 Big-O Notation

This notation gives the *tight* upper bound of the given function. Generally, it is represented as $f(n) = O(g(n))$. That means, at larger values of n, the upper bound of $f(n)$ is $g(n)$. For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is $g(n)$. That means, $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.

Let us see the O–notation with little more detail. O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give smallest rate of growth $g(n)$ which is greater than or equal to given algorithms rate of growth $f(n)$.

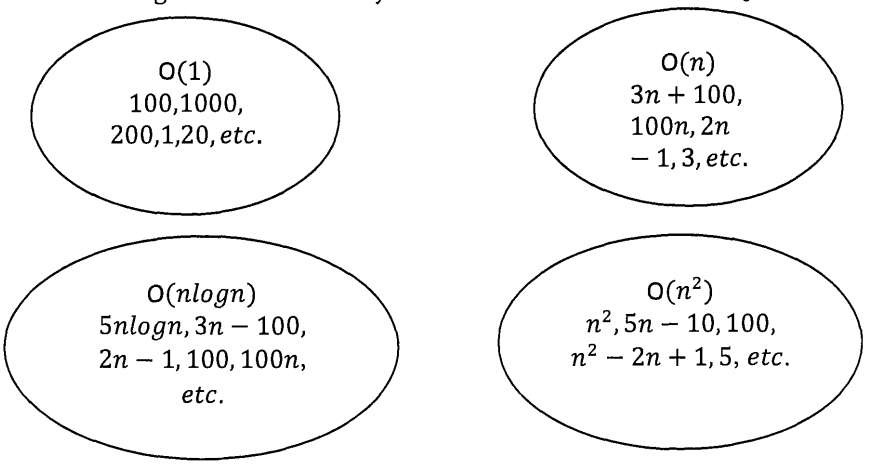
In general, we discard lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we need to consider the rate of growths for a given algorithm. Below n_0 the rate of growths could be different.



Big-O Visualization

$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$. For example, $O(n^2)$ includes $O(1)$, $O(n)$, $O(nlogn)$ etc..

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care for rate of growth.



Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$
Solution: $3n + 8 \leq 4n$, for all $n \geq 1$
$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$
Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$
$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$
Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$
Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(2n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$
Solution: $n \leq n^2$, for all $n \geq 1$
$\therefore n = O(n^2)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$
Solution: $410 \leq 410$, for all $n \geq 1$
$\therefore 100 = O(1)$ with $c = 1$ and $n_0 = 1$

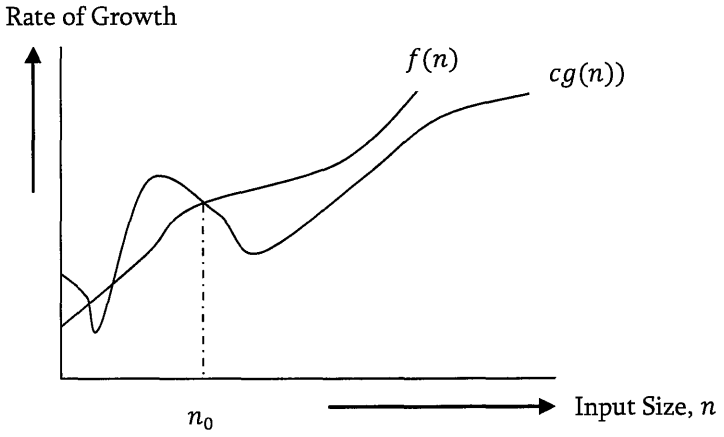
No Uniqueness?

There are no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n^2)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n^2$ for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.
Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$ for all $n \geq 1, n_0 = 1$ and $c = 105$ is also a solution.

1.15 Omega-Ω Notation

Similar to O discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.



The Ω notation can be defined as Ω(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all n $\geq n_0$}. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give largest rate of growth $g(n)$ which is less than or equal to given algorithms rate of growth $f(n)$.

Ω Examples

Example-1 Find lower bound for $f(n) = 5n^2$

Solution: $\exists\ c,\ n_0$ Such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$
$\therefore 5n^2 = \Omega(n)$ with $c = 1$ and $n_0 = 1$

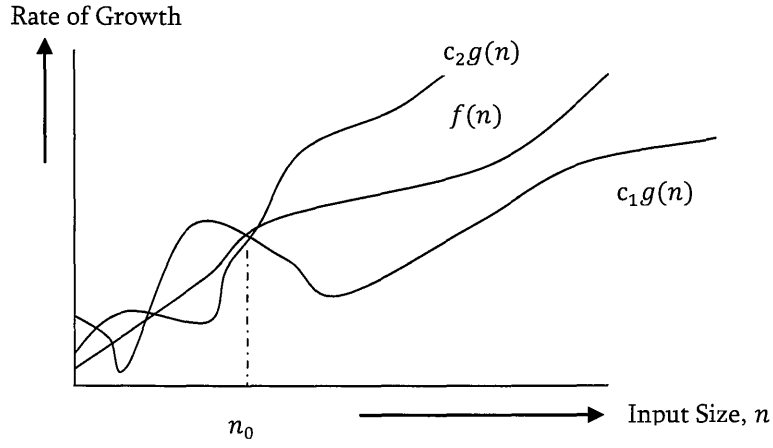
Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$

Solution: $\exists$ c, n_0 Such that: $0 \leq cn^2 \leq 100n + 5$
$100n + 5 \leq 100n + 5n\ (\forall n \geq 1) = 105n$
$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$
$\Rightarrow$ Contradiction: n cannot be smaller than a constant

Example-3 $n = \Omega(2n),\ n^3 = \Omega(n^2),\ n = \Omega(logn)$

1.16 Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given function (algorithm) are same or not. The average running time of algorithm is always between lower bound and upper bound. If the upper bound (O) and lower bound (Ω) gives the same result then Θ notation will also have the same rate of growth. As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is O(n). The rate of growth in best case is $g(n)$ =O(n). In this case, rate of growths in best case and worst are same. As a result, the average case will also be same. For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth Θ case may not be same.



Now consider the definition of Θ notation. It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

Θ Examples

Example-1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$, for all, $n \geq 1$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

Example-2 Prove $n \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example-3 Prove $6n^3 \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example-4 Prove $n \neq \Theta(logn)$

Solution: $c_1 logn \leq n \leq c_2 log n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ – Impossible

Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (Θ) may not be possible always. For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance and we use Θ notation if upper bound (O) and lower bound (Ω) are same.

1.17 Why is it called Asymptotic Analysis?

From the above discussion (for all the three notations: worst case, best case and average case), we can easily understand that, in every case for a given function $f(n)$ we are trying to find other function $g(n)$ which approximates $f(n)$ at higher values of n. That means, $g(n)$ is also a curve which approximates $f(n)$ at higher values of n. In

mathematics we call such curve as *asymptotic curve*. In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis as *asymptotic analysis*.

1.18 Guidelines for Asymptotic Analysis

There are some general rules to help us in determining the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
   m = m + 2; // constant time, c
```

Total time = a constant $c \times n = cn = O(n)$.

2) **Nested loops:** Analyze from inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executed n times
    for (j=1; j<=n; j++)
            k = k+1; //constant time
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time
// executed n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executed n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
            k = k+1; //constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
//test: constant
if(length( ) == 0 ) {
    return false; //then part: constant
}
else {// else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++) {
            // another if : constant + constant (no else part)
            if(!list[n].equals(otherList.list[n]))
                    //constant
                    return false;
    }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is O($logn$) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for (i=1; i<=n;)
        i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^i = n$ and we come out of loop. Taking logarithm on both sides, gives

$log(2^i) = logn$
$ilog2 = logn$
$i = logn$ //if we assume base-2

Total time = O($logn$).

Note: Similarly, for the below case also, worst case rate of growth is O($logn$). The same discussion holds good for decreasing sequence as well.

```
for (i=n; i<=1;)
    i = i/2;
```

Another example: binary search (finding a word in a dictionary of n pages)

- Look at the center point in the dictionary
- Is word towards left or right of center?
- Repeat process with left or right part of dictionary until the word is found

1.19 Properties of Notations

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω also.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

1.20 Commonly used Logarithms and Summations

Logarithms

$log\ x^y = y\ log\ x$ $\quad$ $logn = log_{10}^n$
$log\ xy = logx + logy$ $\quad$ $log^k n = (logn)^k$
$log\ logn = log(logn)$ $\quad$ $log\frac{x}{y} = logx - logy$
$a^{log_b^x} = x^{log_b^a}$ $\quad$ $log_b^x = \frac{log_a^x}{log_a^b}$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx log\ n$$

Other important formulae

$$\sum_{k=1}^{n} log\, k \approx nlogn$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

1.21 Master Theorem for Divide and Conquer

All divide and conquer algorithms (In detail, we will discuss them in *Divide and Conquer* chapter) divides the problem into subproblems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, merge sort algorithm [for details, refer *Sorting* chapter] operates on two subproblems, each of which is half the size of the original and then performs O(n) additional work for merging. This gives the running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we can directly give the answer without fully solving it.

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1$, $b > 1, k \geq 0$ and p is a real number, then:

1) If $a > b^k$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

1.22 Problems on Divide and Conquer Master Theorem

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$

Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + n^2$

Solution: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$

Solution: $T(n) = 2^n T(n/2) + n^n \Rightarrow$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + nlogn$

Solution: $T(n) = 2T(n/2) + nlogn \Rightarrow T(n) = \Theta(nlog^2n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/logn$

Solution: $T(n) = 2T(n/2) + n/logn \Rightarrow T(n) = \Theta(nloglogn)$ (Master Theorem Case 2.b)

Problem-8 $T(n) = 2T(n/4) + n^{0.51}$

Solution: $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T(n) = O(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$

Solution: $T(n) = 0.5T(n/2) + 1/n \Rightarrow$ Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$

Solution: $T(n) = 6T(n/3) + n^2logn \Rightarrow T(n) = \Theta(n^2logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2logn$

Solution: $T(n) = 64T(n/8) - n^2logn \Rightarrow$ Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$

Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$

Solution: $T(n) = 4T(n/2) + logn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$

Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$

Solution: $T(n) = \sqrt{2}T(n/2) + logn \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$

Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$

Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$

Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$

Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$

Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

1.23 Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants $c, a > 0, b > 0, k \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if a} < 1 \\ O(n^{k+1}), & \text{if a} = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if a} > 1 \end{cases}$$

1.24 Variant of subtraction and conquer master theorem

The solution to the equation $T(n) = T(\alpha\, n) + T((1-\alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(nlogn)$.

1.25 Amortized Analysis

Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not "bad" (e.g., some sorting algorithms do well on "average" over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst case analysis, but for a sequence of operations, rather than for individual operations.

The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can "charge them" to the cheap operations, and only bound the cheap operations.

The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. In order to analyze the running time, the amortized cost thus is a correct way of understanding the overall running time — but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

When one event in a sequence affects the cost of later events:

- One particular task may be expensive.
- But it may leave data structure in a state that next few operations becomes easier.

Example: Let us consider an array of elements from which we want to find k^{th} smallest element. We can solve this problem using sorting. After sorting the given array, we just need to return the k^{th} element from it. Cost of performing sort (assuming comparison based sorting algorithm) is $O(nlogn)$. If we perform n such selections then the average cost of each selection is $O(nlogn/n) = O(logn)$. This clearly indicates that sorting once is reducing the complexity of subsequent operations.

1.26 Problems on Algorithms Analysis

Note: From the following problems, try to understand the cases which give different complexities ($O(n), O(logn), O(loglogn)$ etc...).

Problem-21 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 3T(n-1), if\ n > 0, \\ 1, \quad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$$T(n) = 3T(n-1)$$

$$T(n) = 3\big(3T(n-2)\big) = 3^2T(n-2)$$

$$T(n) = 3^2(3T(n-3))$$

.

.

$$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$$

This clearly shows that the complexity of this function is $O(3^n)$.
Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-22 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 2T(n-1) - 1, if\ n > 0, \\ 1, \qquad\qquad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$$T(n) = 2T(n-1) - 1$$

$$T(n) = 2(2T(n-2) - 1) - 1 = 2^2T(n-2) - 2 - 1$$

$$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$$

$$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots . 2^2 - 2^1 - 2^0$$

$$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots . 2^2 - 2^1 - 2^0$$

$$T(n) = 2^n - (2^n - 1)\ [note: 2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n]$$

$$T(n) = 1$$

∴Complexity is O(1). Note that while the recurrence relation looks exponential the solution to the recurrence relation here gives a different result.

Problem-23 What is the running time of the following function?

```
void Function(int n) {
        int i=1, s=1;
        while( s <= n) {
                i++;
                s= s+i;
                printf("*");
        }
}
```

Solution: Consider the comments in below function:

```
void Function (int n) {
        int i=1, s=1;
        // s is increasing not at rate 1 but i
        while( s <= n) {
                i++;
                s= s+i;
                printf("*");
        }
}
```

We can define the terms '*s*' according to the relation $s_i = s_{i-1} + i$. The value of '*i*' increases by one for each iteration. The value contained in '*s*' at the i^{th} iteration is the sum of the first '*i*' positive integers. If k is the total number of iterations taken by the program, then *while* loop terminates if:

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

Problem-24 Find the complexity of the function given below.

```
void Function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
}
```

Solution:

```
void Function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
}
```

In the above function the loop will end, if $i^2 \leq n \Rightarrow T(n) = O(\sqrt{n})$. The reasoning is same as that of Problem-23.

Problem-25 What is the complexity of the below program:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j++)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n) {
        int i, j, k , count =0;
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
                //Middle loop executes n/2 times
                for(j=1; j + n/2<=n; j= j++)
                        //outer loop execute logn times
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

The complexity of the above function is $O(n^2 logn)$.

Problem-26 What is the complexity of the below program:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n) {
        int i, j, k , count =0;
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
```

```
        //Middle loop executes logn times
        for(j=1; j<=n; j= 2 * j)
                //outer loop execute logn times
                for(k=1; k<=n; k= k*2)
                        count++;
}
```

The complexity of the above function is $O(nlog^2n)$.

Problem-27 Find the complexity of the below program.

```
function( int n ) {
        if(n == 1) return;
        for(int i = 1 ; i <= n ; i + + ) {
                for(int j= 1 ; j <= n ; j + + ) {
                        printf("*" );
                        break;
                }
        }
}
```

Solution: Consider the comments in the following function.

```
function( int n ) {
        //constant time
        if( n == 1 ) return;
        //outer loop execute n times
        for(int i = 1 ; i <= n ; i + + ) {
                // inner loop executes only time due to break statement.
                for(int j= 1 ; j <= n ; j + + ) {
                        printf("*" );
                        break;
                }
        }
}
```

The complexity of the above function is $O(n)$. Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-28 Write a recursive function for the running time $T(n)$ of the function given below. Prove using the iterative method that $T(n) = \Theta(n^3)$.

```
        function( int n ) {
                if( n == 1 ) return;
                for(int i = 1 ; i <= n ; i + + )
                        for(int j = 1 ; j <= n ; j + + )
                                printf("*" ) ;
                function( n-3 );
        }
```

Solution: Consider the comments in below function:

```
function (int n) {
        //constant time
        if( n == 1 ) return;
        //outer loop execute n times
        for(int i = 1 ; i <= n ; i + + )
                //inner loop executes n times
```

```
            for(int j = 1 ; j <= n ; j + + )
                //constant time
                printf("*" ) ;
        function( n-3 );
}
```

The recurrence for this code is clearly $T(n) = T(n - 3) + cn^2$ for some constant $c > 0$ since each call prints out n^2 asterisks and calls itself recursively on n - 3. Using the iterative method we get: $T(n) = T(n - 3) + cn^2$. Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(n^3)$.

Problem-29 Determine Θ bounds for the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$.

Solution: Using Divide and Conquer master theorem, we get $O(nlog^2n)$.

Problem-30 Determine Θ bounds for the recurrence: $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$.

Solution: Substituting in the recurrence equation, we get: $T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \leq k * n$, where k is a constant. This clearly says $\Theta(n)$.

Problem-31 Determine Θ bounds for the recurrence relation: $T(n) = T(\lceil n/2 \rceil) + 7$.

Solution: Using Master Theorem we get $\Theta(logn)$.

Problem-32 Prove that the running time of the code below is $\Omega(logn)$.

```
void Read(int n) {
        int k = 1;
        while( k < n )
                k = 3*k;
}
```

Solution: The *while* loop will terminate once the value of *'k'* is greater than or equal to the value of *'n'*. In each iteration the value of *'k'* is multiplied by 3. If i is the number of iterations, then *'k'* has the value of $3i$ after i iterations. The loop is terminated upon reaching i iterations when $3^i \geq n \leftrightarrow i \geq \log_3 n$, which shows that $i = \Omega(logn)$.

Problem-33 Solve the following recurrence.

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ T(n-1) + n(n-1), & if\ n \geq 2 \end{cases}$$

Solution: By iteration:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$

...

$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$

$$T(n) = T(1) + \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i$$

$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$

$$T(n) = \Theta(n^3)$$

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-34 Consider the following program:

```
Fib[n]
if(n==0) then return 0
else if(n==1) then return 1
```

else return Fib[n-1]+Fib[n-2]

Solution: The recurrence relation for running time of this program is: $T(n) = T(n-1) + T(n-2) + c$. Notice T($n$) has two recurrence calls indicating a binary tree. Each step recursively calls the program for n reduced by 1 and 2, so the depth of the recurrence tree is O(n). The number of leaves at depth n is 2^n since this is a full binary tree, and each leaf takes at least O(1) computation for the constant factor. Running time is clearly exponential in n and it is O(2^n).

Problem-35 Running time of following program?

```
function(n) {
        for(int i = 1; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j+ = i )
                        printf(" * ") ;
}
```

Solution: Consider the comments in below function:

```
function (n) {
        //this loop executes n times
        for(int i = 1;  i <= n ; i + + )
                //this loop executes j times with j increase by the rate of i
                for(int j = 1 ; j <= n ; j+ = i )
                        printf( " * " ) ;
}
```

In the above code, inner loop executes n/i times for each value of i. Its running time is $n \times (\sum_{i=1}^{n} n/i) = O(nlogn)$.

Problem-36 What is the complexity of $\sum_{i=1}^{n} log\, i$?

Solution: Using the logarithmic property, $logxy = logx + logy$, we can see that this problem is equivalent to

$$\sum_{i=1}^{n} logi = log\,1 + log\,2 + \cdots + log\,n = log(1 \times 2 \times \ldots \times n) = log(n!) \leq log(n^n) \leq nlogn$$

This shows that that the time complexity = O($nlogn$).

Problem-37 What is the running time of the following recursive function (specified as a function of the input value n)? First write the recurrence formula and then find its complexity.

```
function(int n) {
        if(n <= 1) return;
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

Solution: Consider the comments in below function:

```
function (int n) {
        //constant time
        if(n <= 1) return;
        //this loop executes with recursive loop of n/3 value
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

We can assume that for asymptotical analysis $k = \lceil k \rceil$ for every integer $k \geq 1$. The recurrence for this code is $T(n) = 3T(\frac{n}{3}) + \Theta(1)$. Using master theorem, we get $T(n) = \Theta(n)$.

Problem-38 What is the running time of the following recursive function (specified as a function of the input value n)? First write a recurrence formula, and show its solution using induction.

```
function(int n) {
```

```
        if(n <= 1) return;
        for (int i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

Solution: Consider the comments in below function:

```
function (int n) {
        //constant time
        if(n <= 1) return;
        //this loop executes 3 times with recursive call of n-1 value
        for (int i=1 ; i <= 3 ; i++ )
                function (n – 1).
}
```

The *if* statement requires constant time [O(1)]. With the *for* loop, we neglect the loop overhead and only count three times that the function is called recursively. This implies a time complexity recurrence:

$$T(n) = c, if\ n \leq 1;$$
$$= c + 3T(n-1), if\ n > 1.$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(3^n)$.

Problem-39 Write a recursion formula for the running time $T(n)$ of the function whose code is below.

```
function (int n) {
        if(n <= 1) return;
        for(int i = 1; i < n; i + +)
                printf(" * ");
        function ( 0.8n ) ;
}
```

Solution: Consider the comments in below function:

```
function (int n) {
        if(n <= 1) return;          //constant time
        // this loop executes n times with constant time loop
        for(int i = 1; i < n; i + +)
                printf(" * ");
        //recursive call with 0.8n
        function ( 0.8n ) ;
}
```

The recurrence for this piece of code is $T(n) = T(.8n) + O(n) = T(4/5n) + O(n) = 4/5\ T(n) + O(n)$. Applying master theorem, we get $T(n) = O(n)$.

Problem-40 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + logn$

Solution: The given recurrence is not in the master theorem form. Let us try to convert this to master theorem format by assuming $n = 2^m$. Applying logarithm on both sides gives, $logn = mlog2 \Rightarrow m = logn$. Now, the given function becomes,

$$T(n) = T(2^m) = 2T\left(\sqrt{2^m}\right) + m = 2T\left(2^{\frac{m}{2}}\right) + m.$$

To make it simple we assume $S(m) = T(2^m) \Rightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Rightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$. Applying the master theorem would result $S(m) = O(mlogm)$. If we substitute $m = logn$ back, $T(n) = S(logn) = O((logn)\ loglogn)$.

Problem-41 Find the complexity of the recurrence: $T(n) = T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40, gives $S(m) = S\left(\frac{m}{2}\right) + 1$. Applying the master theorem would result $S(m) = O(logm)$. Substituting $m = logn$, gives $T(n) = S(logn) = O(loglogn)$.

Problem-42 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40, gives: $S(m) = 2S\left(\frac{m}{2}\right) + 1$. Using the master theorem results $S(m) = O(m^{log_2^2}) = O(m)$. Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the below function.

```
int Function (int n) {
        if(n <= 2) return 1;
        else return (Function (floor(sqrt(n))) + 1);
}
```

Solution: Consider the comments in below function:

```
int Function (int n) {
        //constant time
        if(n <= 2) return 1;
        else    // executes √n + 1 times
                return (Function (floor(sqrt(n))) + 1);
}
```

For the above code, the recurrence function can be given as: $T(n) = T(\sqrt{n}) + 1$. This is same as that of Problem-41.

Problem-44 Analyze the running time of the following recursive psuedocode as a function of n.

```
void function(int n) {
        if( n < 2 ) return;
        else counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for i =1 to n^3 do
                counter = counter + 1;
}
```

Solution: Consider the comments in below psuedocode and call running time of function(n) as $T(n)$.

```
void function(int n) {
        if( n < 2 ) return;    //constant time
        else    counter = 0;
        // this loop executes 8 times with n value half in every call
        for i = 1 to 8 do
                function (n/2);
        // this loop executes n^3 times with constant time loop
        for i =1 to n^3 do
                counter = counter + 1;
}
```

$T(n)$ can be defined as follows:

$$T(n) = 1 \; if \; n < 2,$$
$$= 8T(\frac{n}{2}) + n^3 + 1 \; otherwise.$$

Using the master theorem gives, $T(n) = \Theta(n^{log_2^8} logn) = \Theta(n^3 logn)$.

Problem-45 Find the complexity of the below psuedocode.

```
temp = 1
repeat
        for i = 1 to n
                temp = temp + 1;
```

```
        n = n/2;
    until n <= 1
```

Solution: Consider the comments in below psuedocode:

```
    temp = 1  //const time
    repeat  // this loops executes n times
        for i = 1 to n
            temp = temp + 1;
        //recursive call with n/2 value
        n = n/2;
    until n <= 1
```

The recurrence for this function is $T(n) = T(n/2) + n$. Using master theorem we get, $T(n) = O(n)$.

Problem-46 Running time of following program?

```
function(int n) {
    for(int i = 1 ; i <= n ; i++ )
        for(int j = 1 ; j <= n ; j *= 2 )
            printf( " * " );
}
```

Solution: Consider the comments in below function:

```
function(int n) {
    for(int i = 1 ; i <= n ; i++ ) // this loops executes n times
        // this loops executes logn times from our logarithms guideline
        for(int j = 1 ; j <= n ; j *= 2 )
            printf( " * " );
}
```

Complexity of above program is : $O(nlogn)$.

Problem-47 Running time of following program?

```
function(int n) {
    for(int i = 1 ; i <= n/3 ; i++ )
        for(int j = 1 ; j <= n ; j += 4 )
            printf( " * " );
}
```

Solution: Consider the comments in below function:

```
function(int n) { // this loops executes n/3 times
    for(int i = 1 ; i <= n/3 ; i++ )
        // this loops executes n/4 times
        for(int j = 1 ; j <= n ; j += 4)
            printf( " * " );
}
```

The time complexity of this program is : $O(n^2)$.

Problem-48 Find the complexity of the below function.

```
void function(int n) {
    if(n <= 1) return;
    if(n > 1) {
        printf (" * ");
        function(n/2);
        function(n/2);
    }
```

```
}
```

Solution: Consider the comments in below function:

```
void function(int n) {
        //constant time
        if(n <= 1) return;
        if(n > 1) {
                //constant time
                printf (" * ");
                //recursion with n/2 value
                function( n/2 );
                //recursion with n/2 value
                function( n/2 );
        }
}
```

The recurrence for this function is: $T(n) = 2T\left(\frac{n}{2}\right) + 1$. Using master theorem, we get $T(n) = O(n)$.

Problem-49 Find the complexity of the below function.

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                        j = j/2;
                i=2*i;
        } // i
}
```

Solution:

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                        j = j/2;   //logn code
                i=2*i; //logn times
        } // i
}
```

Time Complexity: $O(logn * logn) = O(log^2 n)$.

Problem-50 $\sum_{1 \le k \le n} O(n)$, where $O(n)$ stands for order n is:

(a) $O(n)$ (b) $O(n^2)$ (c) $O(n^3)$ (d) $O(3n^2)$ (e) $O(1.5n^2)$

Solution: (b). $\sum_{1 \le k \le n} O(n) = O(n) \sum_{1 \le k \le n} 1 = O(n^2)$.

Problem-51 Which of the following three claims are correct

I $(n + k)^m = \Theta(n^m)$, where k and m are constants II $2^{n+1} = O(2^n)$ III $2^{2n+1} = O(2^n)$

(a) I and II (b) I and III (c) II and III (d) I, II and III

Solution: (a). (I) $(n + k)^m = n^k + c1*n^{k-1} + \ldots k^m = \Theta(n^k)$ and (II) $2^{n+1} = 2*2^n = O(2^n)$

Problem-52 Consider the following functions:

$f(n) = 2^n$ $g(n) = n!$ $h(n) = n^{logn}$

Which of the following statements about the asymptotic behaviour of $f(n)$, $g(n)$, and $h(n)$ is true?

(A) f(n) = O(g(n)); g(n) = O(h(n)) (B) f(n) = Ω (g(n)); g(n) = O(h(n))
(C) g(n) = O(f(n)); h(n) = O(f(n)) (D) h(n) = O(f(n)); g(n) = Ω (f(n))

Solution: (D). According to rate of growths: h(n) < f(n) < g(n) (g(n) is asymptotically greater than f(n) and f(n) is asymptotically greater than h(n)). We can easily see above order by taking logarithms of the given 3 functions: $lognlogn < n < log(n!)$. Note that, $log(n!) = O(nlogn)$.

Problem-53 Consider the following segment of C-code:

```
int j=1, n;
while (j <=n)
        j = j*2;
```

The number of comparisons made in the execution of the loop for any $n > 0$ is:
(A) ceil(log_2^n)+ 1 (B) n (C) ceil(log_2^n) (D) floor(log_2^n) + 1

Solution: (a). Let us assume that the loop executes k times. After k^{th} step the value of j is 2^k. Taking logarithms on both sides gives $k = log_2^n$. Since we are doing one more comparison for exiting from loop, the answer is ceil(log_2^n)+ 1.

Problem-54 Consider the following C code segment. Let T(n) denotes the number of times the for loop is executed by the program on input n. Which of the following is TRUE?

```
int IsPrime(int n){
 for(int i=2;i<=sqrt(n);i++)
  if(n%i == 0)
   {       printf("Not Prime\n"); return 0;    }
 return 1;
}
```

(A) T(n) = O($\sqrt{n}$) and T(n) = Ω($\sqrt{n}$) (B) T(n) = O($\sqrt{n}$) and T(n) = Ω(1)
(C) T(n) = O(n) and T(n) = Ω($\sqrt{n}$) (D) None of the above

Solution: (B). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. The for loop in the question is run maximum $\sqrt{n}$ times and minimum 1 time. Therefore, T(n) = O($\sqrt{n}$) and T(n) = Ω(1).

Problem-55 In the following C function, let $n \geq m$. How many recursive calls are made by this function?

```
int gcd(n,m){
        if (n%m ==0) return m;
        n = n%m;
        return gcd(m,n);
}
```

(A) Θ(log_2^n) (B) Ω(n) (C) Θ($log_2 log_2^n$) (D) Θ(n)

Solution: No option is correct. Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. For $m = 2$ and for all $n = 2^i$, running time is O(1) which contradicts every option.

Problem-56 Suppose $T(n) = 2T(n/2) + n$, T(0)=T(1)=1. Which one of the following is FALSE?

(A) $T(n) = O(n^2)$ (B) $T(n) = \Theta(nlogn)$ (C) $T(n) = \Omega(n^2)$ (D) $T(n) = O(nlogn)$

Solution: (C). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. Based on master theorem, we get $T(n) = \Theta(nlogn)$. This indicates that tight lower bound and tight upper bound are same. That means, O($nlogn$) and Ω($nlogn$) are correct for given recurrence. So option (C) is wrong.

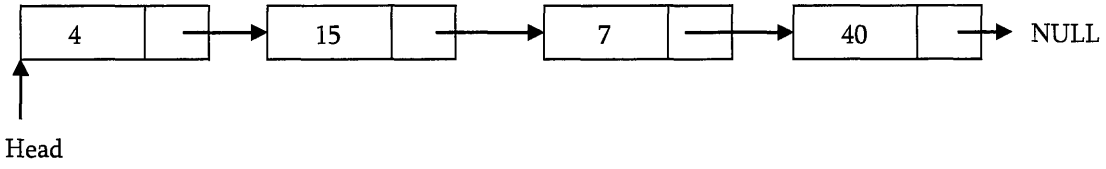
LINKED LISTS

Chapter-3

3.1 What is a Linked List?

Linked list is a data structure used for storing collections of data. Linked list has the following properties.

- Successive elements are connected by pointers
- Last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- It does not waste memory space (but takes some extra memory for pointers)



3.2 Linked Lists ADT

The following operations make linked lists an ADT.

Main Linked Lists Operations

- Insert: inserts an element into the list
- Delete: removes and returns the specified position element from the list

Auxiliary Linked Lists Operations

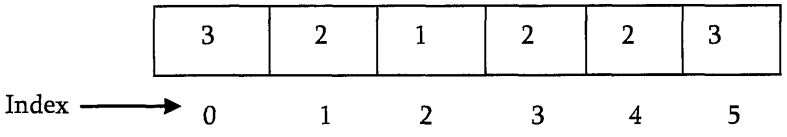
- Delete List: removes all elements of the list (disposes the list)
- Count: returns the number of elements in the list
- Find n^{th} node from the end of the list etc...

3.3 Why Linked Lists?

There are many other data structures which do the same thing as that of linked lists. Before discussing linked lists it is important to understand the difference between linked lists and arrays. Both linked lists and arrays are used to store collections of data. Since both are used for the same purpose, we need to differentiate the usage of them. That means in which cases *arrays* are suitable and in which cases *linked lists* are suitable.

3.4 Arrays Overview

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in a constant time by using the index of the particular element as the subscript.



Why Constant Time for Accessing Array Elements?

To access an array element, address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.

This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

Disadvantages of Arrays

- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation:** To allocate the array at the beginning itself, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion:** To insert an element at a given position then we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning then the shifting operation is more expensive.

Dynamic Arrays

Dynamic array (also called as *growable array*, *resizable array*, *dynamic table*, or *array list*) is a random access, variable-size list data structure that allows elements to be added or removed.

One simple way of implementing dynamic arrays is, initially start with some fixed size array. As soon as that array becomes full, create the new array of size double than the original array. Similarly, reduce the array size to half if the elements in the array are less than half.

Note: We will see the implementation for *dynamic arrays* in *Stacks*, *Queues* and *Hashing* chapters.

Advantages of Linked Lists

Linked lists have advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in constant time. To create an array we must allocate memory for a certain number of elements. To add more elements to the array then we must create a new array and copy the old array into the new array. This can take lot of time.

We can prevent this by allocating lots of space initially but then you might allocate more than you need and wasting memory. With a linked list we can start with space for just one element allocated and *add* on new elements easily without the need to do any copying and reallocating.

Issues with Linked Lists (Disadvantages)

There are a number of issues in linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes O(1) to access any element in the array. Linked lists takes $O(n)$ for access to an element in the list in the worst case. Another advantage of arrays in access time is special *locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the *overhead* with storing and retrieving data can make a big difference. Sometimes linked lists are *hard* to *manipulate*. If the last item is deleted, the last but one must

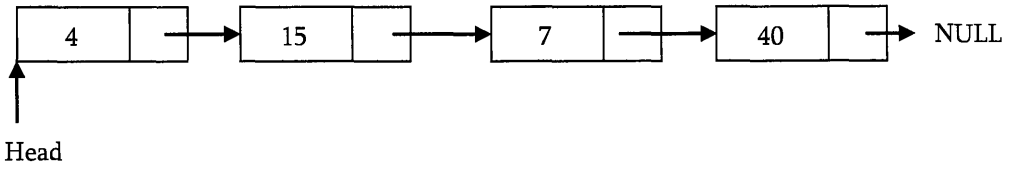
now have its pointer changed to hold a NULL reference. This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference. Finally, linked lists wastes memory in terms of extra reference points.

3.5 Comparison of Linked Lists with Arrays & Dynamic Arrays

Parameter	Linked list	Array	Dynamic array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	-	$O(n)$
Insertion/deletion at ending	$O(n)$	-	$O(1)$, if the array is not full $O(n)$, if the array is full
Insertion/deletion in middle	$O(n)$	-	$O(n)$
Wasted space	$O(n)$	0	$O(n)$

3.6 Singly Linked Lists

Generally "linked list" means a singly linked list. This list consists of a number of nodes in which each node has a *next* pointer to the following element. The link of the last node in the list is NULL which indicates end of the list.



Following is a type declaration for a linked list of integers:

```
struct ListNode {
        int data;
        struct ListNode *next;
};
```

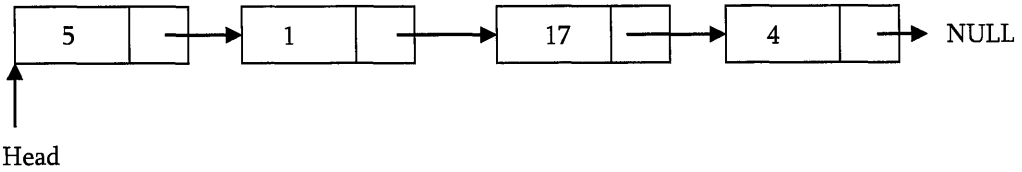
Basic Operations on a List

- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

Traversing the Linked List

Let us assume that the *head* points to the first node of the list. To traverse the list we do the following.

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.



The ListLength() function takes a linked list as input and counts the number of nodes in the list. Below function can be used for printing the list data with extra print function.

```
int ListLength(struct ListNode *head) {
        struct ListNode *current = head;
        int count = 0;
```

```
        while (current != NULL) {
                count++;
                current = current→next;
        }
        return count;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

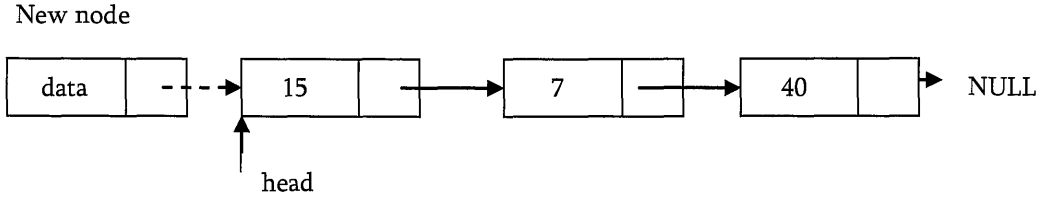
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

Note: To insert an element in the linked list at some position p, assume that after inserting the element the position of this new node is p.

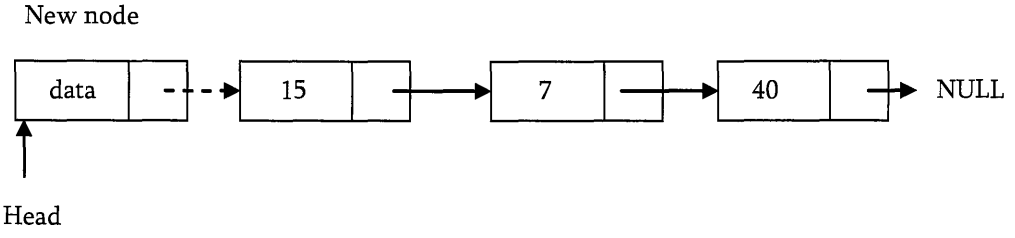
Inserting a Node in Singly Linked List at the Beginning

In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (new node's next pointer) and it can be done in two steps:

- Update the next pointer of new node, to point to the current head.



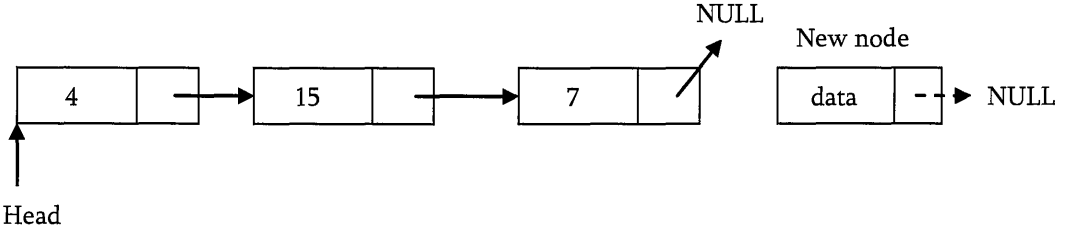
- Update head pointer to point to the new node.



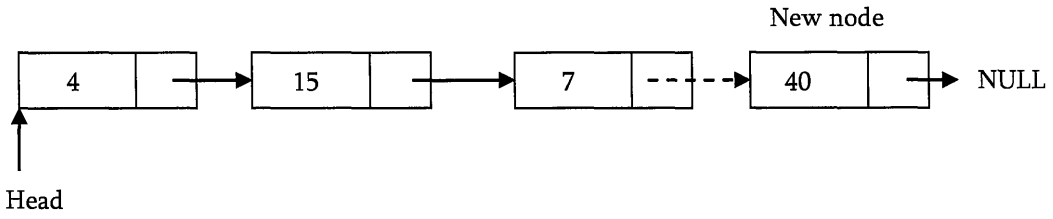
Inserting a Node in Singly Linked List at the Ending

In this case, we need to modify *two next pointers* (last nodes next pointer and new nodes next pointer).

- New nodes next pointer points to NULL.



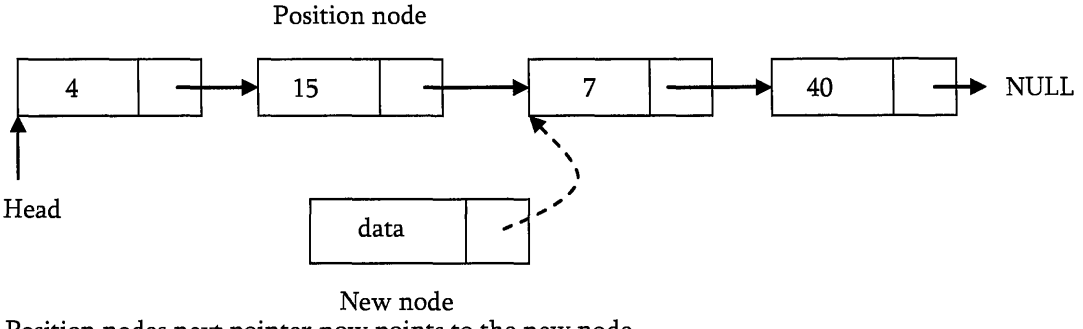
- Last nodes next pointer points to the new node.



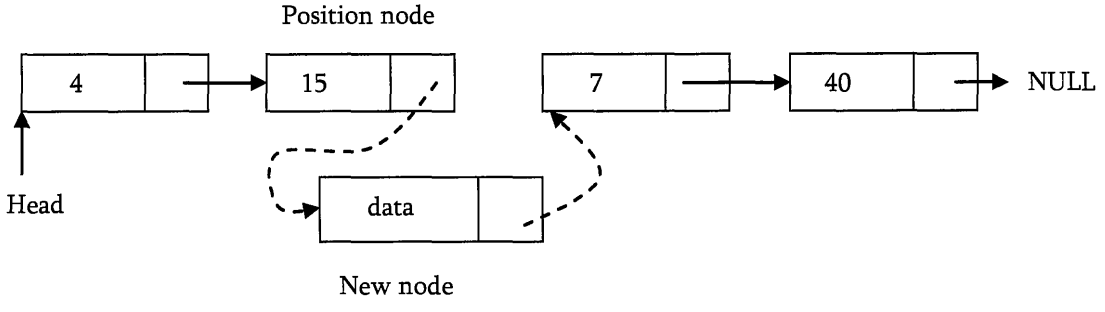
Inserting a Node in Singly Linked List at the Middle

Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- If we want to add an element at position 6 then we stop at position 5. That means we traverse 5 nodes and insert the new node. For simplicity let us assume that fifth is called *position* node. New node points to the next node of the position where we want to add this node.



- Position nodes next pointer now points to the new node.



Let us write the code for all these three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send double pointer. The following code inserts a node in the singly linked list.

```
void InsertInLinkedList (struct ListNode **head, int data, int position) {
        int k = 1;
        struct ListNode *p, *q, *newNode;
        newNode = (ListNode *) malloc(sizeof(struct ListNode));
        if(!newNode) {                                  //Always Check for Memory Errors
                printf ("Memory Error");
                return;
        }
        newNode→data=data;
        p = *head;
        if(position == 1)  {                    // Inserting at the beginning
                newNode→next = p;
                *head = newNode;
        }
```

```
        else {    //Traverse the list until position-1
                  while ((p != NULL) && ( k < position - 1)) {
                            k++;
                            q = p;
                            p = p→next;
                  }
                  if(p == NULL)    {                    //Inserting at the end
                            q→next = newNode;
                            newNode→next = NULL;
                  }
                  else {                                 // In the middle
                            q→next = newNode;
                            newNode→next = p;
                  }
        }
}
```

Note: We can implement the three variations of the *insert* operation separately.

Time Complexity: O(n). Since, in the worst we may need to insert the node at end of the list. Space Complexity: O(1), for creating one temporary variable.

Singly Linked List Deletion

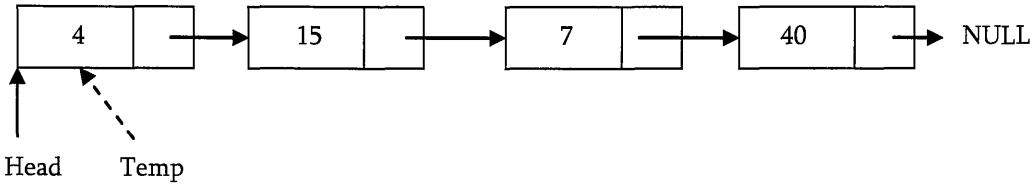
As similar to insertion here also we have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

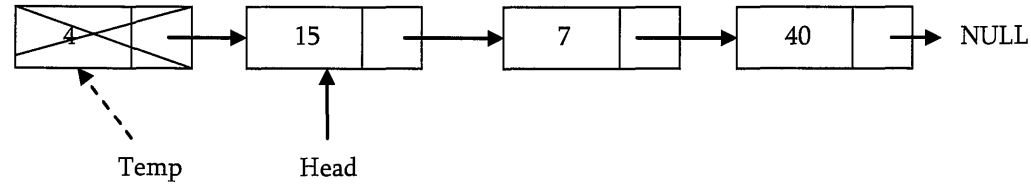
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.



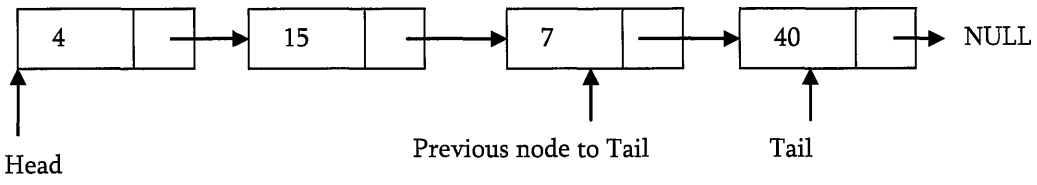
- Now, move the head nodes pointer to the next node and dispose the temporary node.



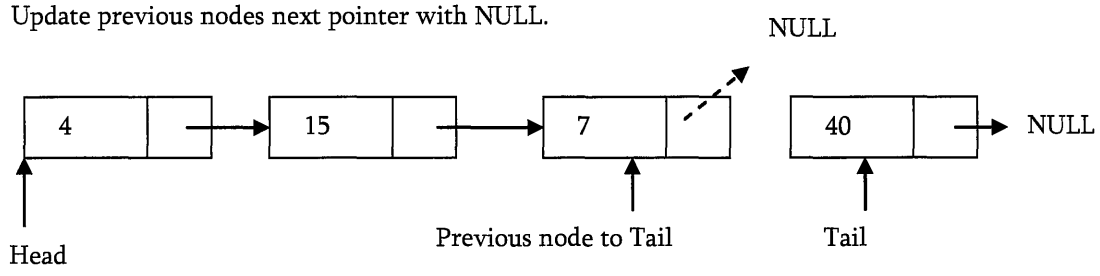
Deleting the last node in Singly Linked List

In this case, last node is removed from the list. This operation is a bit trickier than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps:

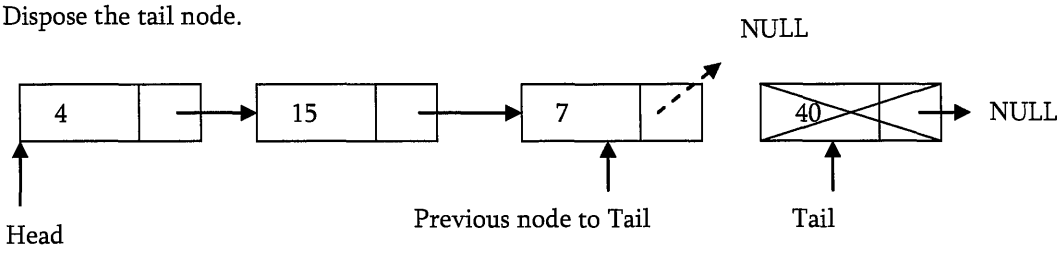
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the *tail* node and other pointing to the node *before* tail node.



- Update previous nodes next pointer with NULL.



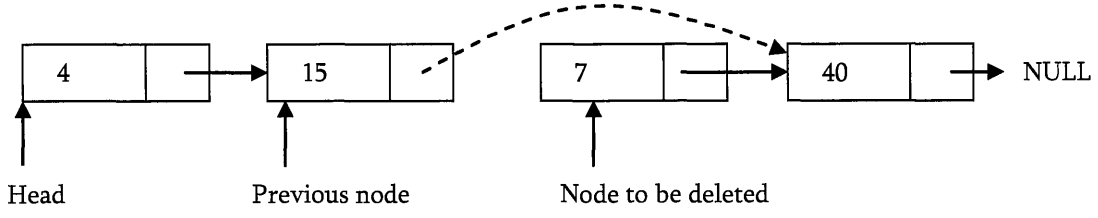
- Dispose the tail node.



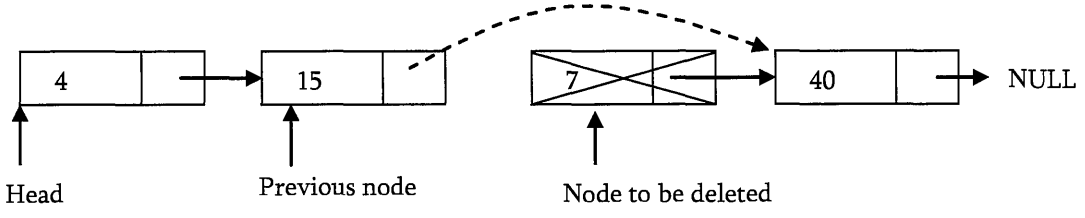
Deleting an Intermediate Node in Singly Linked List

In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- As similar to previous case, maintain previous node while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to next pointer of the node to be deleted.



- Dispose the current node to be deleted.




```
void DeleteNodeFromLinkedList (struct ListNode **head, int position) {
        int k = 1;
        struct ListNode *p, *q;
```

```
        if(*head == NULL) {
                printf ("List Empty");
                return;
        }
        p = *head;
        if(position == 1) {                    /* from the beginning */
                p = *head;
                *head = *head→next;
                free (p); return;
        }
        else {   //Traverse the list until the position from which we want to delete
                while ((p != NULL) && (k < position - 1)) {
                        k++;   q = p;
                        p = p→next;
                }
                if(p == NULL)                          /* At the end */
                        printf ("Position does not exist.");
                else {                                 /* From the middle */
                        q→next = p→next;
                        free(p);
                }
        }
}
```

Time Complexity: O(n). In the worst we may need to delete the node at the end of the list. Space Complexity: O(1). Since, we are creating only one temporary variable.

Deleting Singly Linked List

This works by storing the current node in some temporary variable and freeing the current node. After freeing the current node go to next node with temporary variable and repeat this process for all nodes.

```
void DeleteLinkedList(struct ListNode **head) {
        struct ListNode *auxilaryNode, *iterator;
        iterator = *head;
        while (iterator) {
                auxilaryNode = iterator→next;
                free(iterator);
                iterator = auxilaryNode;
        }
        *head = NULL;                          // to affect the real head back in the caller.
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

3.7 Doubly Linked Lists

The *advantage* of a doubly linked list (also called *two – way linked list*) is given a node in the list, we can navigate in both directions. A node in a singly linked list cannot be removed unless we have the pointer to its predecessor. But in doubly linked list we can delete a node even if we don't have previous nodes address (since, each node has left pointer pointing to previous node and can move backward). The primary *disadvantages* of doubly linked lists are:

- Each node requires an extra pointer, requiring more space.

- The insertion or deletion of a node takes a bit longer (more pointer operations).

As similar to singly linked list, let us implement the operations of doubly linked lists. If you understand the singly linked list operations then doubly linked list operations are very obvious. Following is a type declaration for a doubly linked list of integers:

```
struct DLLNode {
        int data;
        struct DLLNode *next;
        struct DLLNode *prev;
};
```

Doubly Linked List Insertion

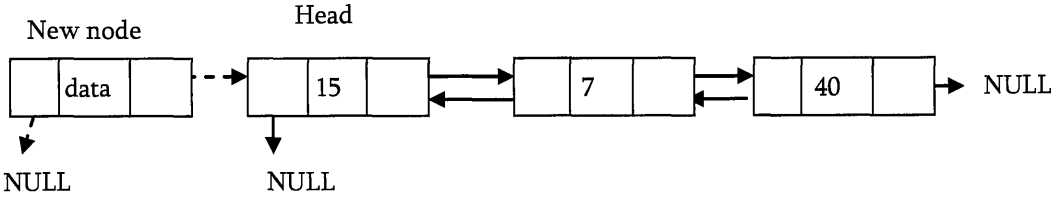
Insertion into a doubly-linked list has three cases (same as singly linked list):

- Inserting a new node before the head.
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node at the middle of the list.

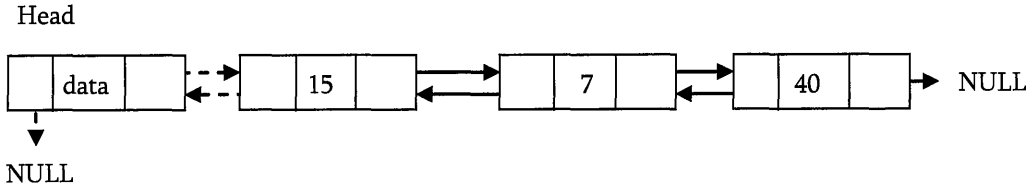
Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



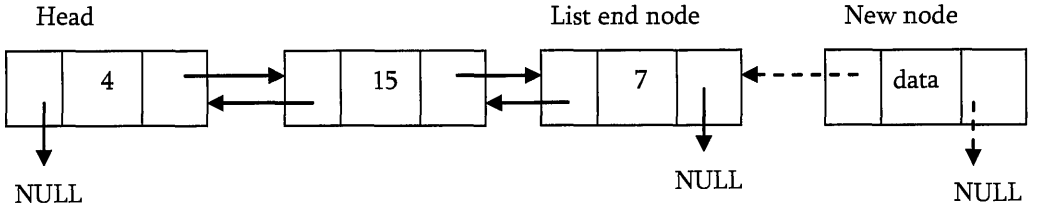
- Update head nodes left pointer to point to the new node and make new node as head.



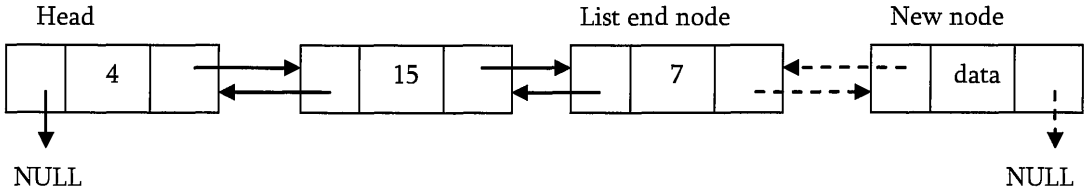
Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.



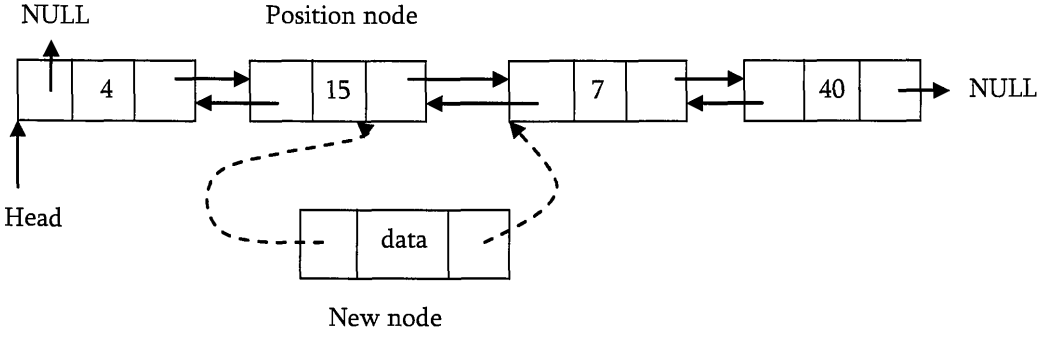
- Update right of pointer of last node to point to new node.



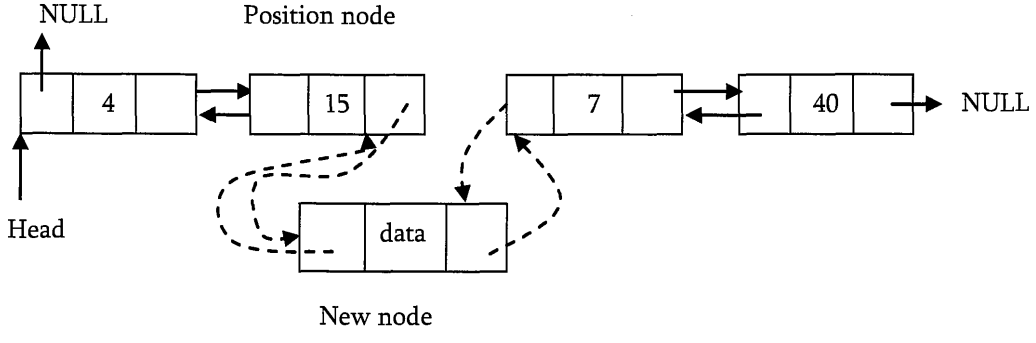
Inserting a Node in Doubly Linked List at the Middle

As discussed in singly linked lists, traverse the list till the position node and insert the new node.

- *New node* right pointer points to the next node of the *position node* where we want to insert the new node. Also, *new node* left pointer points to the *position node*.



- Position node right pointer points to the new node and the *next node* of position nodes left pointer points to new node.



Now, let us write the code for all these three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send double pointer. The following code inserts a node in the doubly linked list.

```
void DLLInsert(struct DLLNode **head, int data, int position) {
        int k = 1;
        struct DLLNode *temp, *newNode;
        newNode = (struct DLLNode *) malloc(sizeof ( struct DLLNode ));
        if(!newNode) {                          //Always check for memory errors
                printf ("Memory Error"); return;
        }
        newNode→data = data;
        if(position == 1) {                     //Inserting a node at the beginning
                newNode→next = *head;
                newNode→prev = NULL;
                *head→prev = newNode;
                *head = newNode;
                return;
```

```
        }
        temp = *head;
        while ( (k < position – 1) && temp→next!=NULL) {
                temp = temp→next;
                k++;
        }
        if( temp→next== NULL) {                    //Insertion at the end
                newNode→next = temp→next;
                newNode→prev = temp;
                temp→next = newNode;
        }
        else {                                     //Insertion in the middle
                newNode→next = temp→next;
                newNode→prev = temp;
                temp→next→prev = newNode;
                temp→next = newNode;
        }
        return;
}
```

Time Complexity: O(n). In the worst we may need to insert the node at the end of the list. Space Complexity: O(1), for creating one temporary variable.

Doubly Linked List Deletion

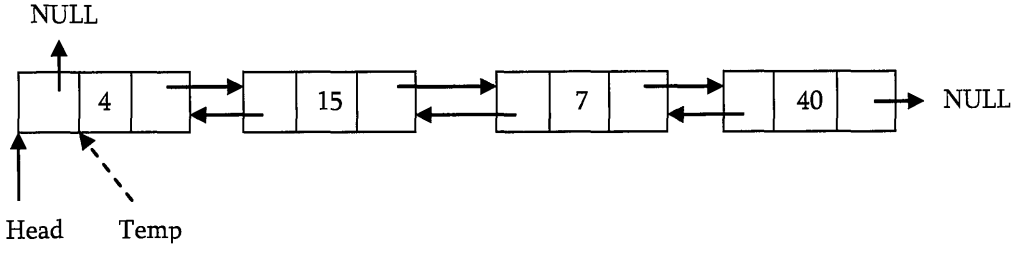
As similar to singly linked list deletion, here also we have three cases:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

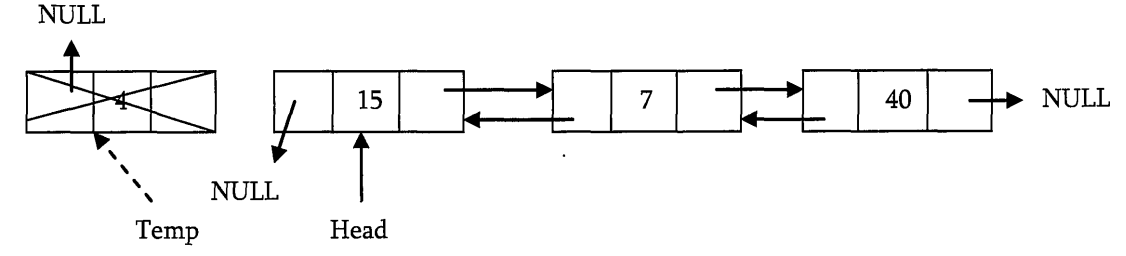
Deleting the First Node in Doubly Linked List

In this case, first node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.



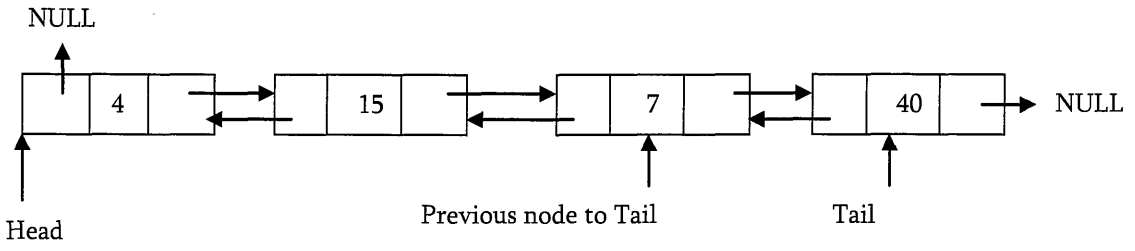
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose the temporary node.



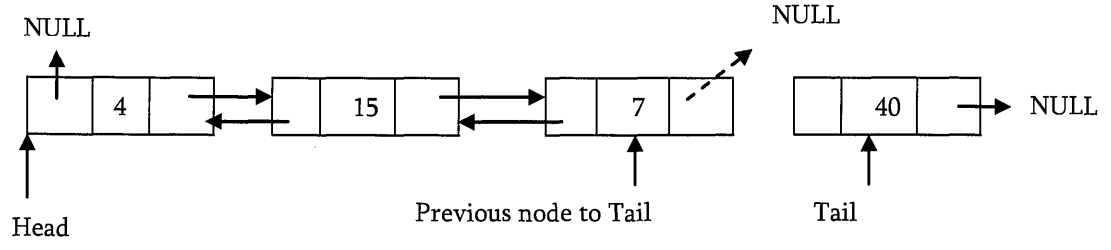
Deleting the Last Node in Doubly Linked List

This operation is a bit trickier, than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps:

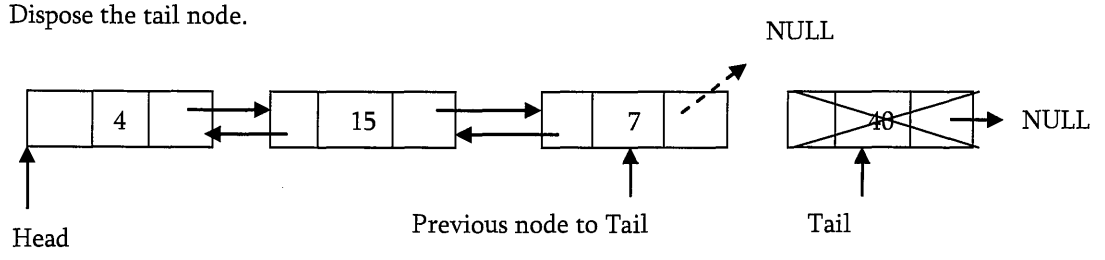
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the NULL (tail) and other pointing to the node before tail node.



- Update tail nodes previous nodes next pointer with NULL.



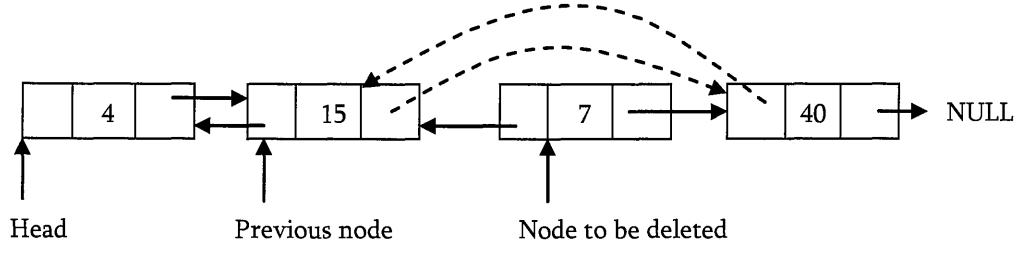
- Dispose the tail node.



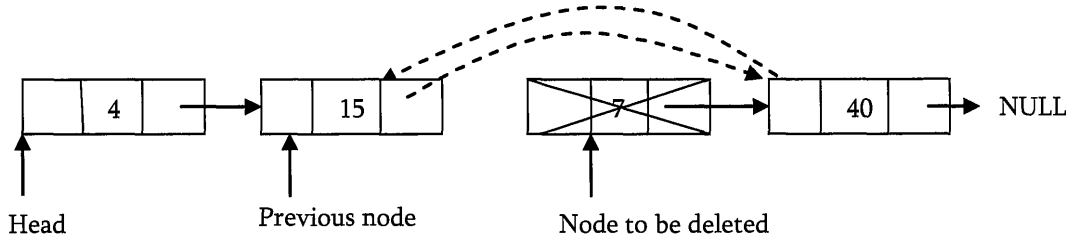
Deleting an Intermediate Node in Doubly Linked List

In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- As similar to previous case, maintain previous node also while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to the next node of the node to be deleted.



- Dispose the current node to be deleted.




```
void DLLDelete(struct DLLNode **head, int position) {
        struct DLLNode *temp, *temp2, temp = *head;
        int k = 1;
        if(*head == NULL) {
                printf("List is empty"); return;
        }
        if(position == 1) {
                *head = *head→next;
                if(*head != NULL)
                        *head→prev = NULL;
                free(temp);
                return;
        }
        while((k < position - 1) && temp→next!=NULL) {
                temp = temp→next;
                k++;
        }
        if( temp→next== NULL) {          //Deletion from end
                temp2 = temp→prev;
                temp2→next = NULL;
                free(temp);
        }
        else {   temp2 = temp→prev;
                temp2→next = temp→next;
                temp→next→prev = temp2;
                free (temp);
        }
        return;
}
```

Time Complexity: $O(n)$, for scanning the complete list of size n. Space Complexity: $O(1)$, for creating one temporary variable.

3.8 Circular Linked Lists

In singly linked lists and doubly linked lists the end of lists are indicated with NULL value. But circular linked lists do not have ends. While traversing the circular linked lists we should be careful otherwise we will be traversing the list infinitely. In circular linked lists each node has a successor. Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list. In some situations, circular linked lists are useful.

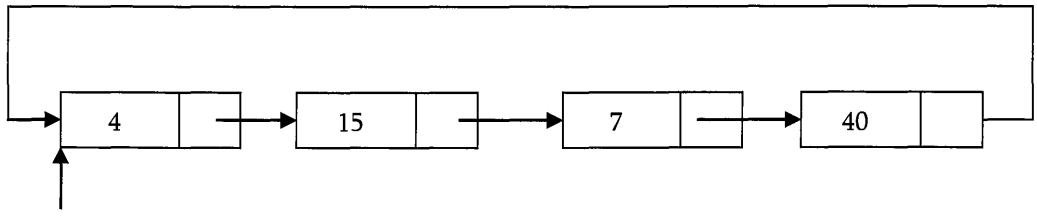
For example, when several processes are using the same computer resource (CPU) for the same amount of time, and we have to assure that no process accesses the resource before all other processes did (round robin algorithm). Following is a type declaration for a circular linked list of integers:

```
        typedef struct CLLNode {
```

```
        int data;
        struct ListNode *next;
};
```

In circular linked list we access the elements using the *head* node (similar to *head* node in singly linked list and doubly linked lists).

Counting Nodes in a Circular List



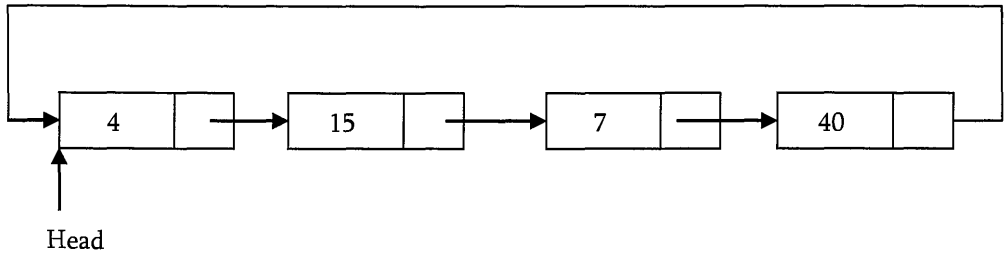
The circular list is accessible through the node marked *head*. To count the nodes, the list has to be traversed from node marked *head*, with the help of a dummy node *current* and stop the counting when *current* reaches the starting node *head*. If the list is empty, *head* will be NULL, and in that case set *count* = 0. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

```
int CircularListLength(struct CLLNode *head) {
        struct CLLNode *current = head;
        int count = 0;
        if(head == NULL) return 0;
        do {    current = current→next;
                count++;
        } while (current != head);
        return count;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

Printing the contents of a circular list

We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node next to the *head* node. Let us assume we want to print the contents of the nodes starting with the *head* node. Print its contents, move to the next node and continue printing till we reach the *head* node again.




```
void PrintCircularListData(struct CLLNode *head) {
        struct CLLNode *current = head;
        if(head == NULL) return;
        do {    printf ("%d", current→data);
                current = current→next;
```

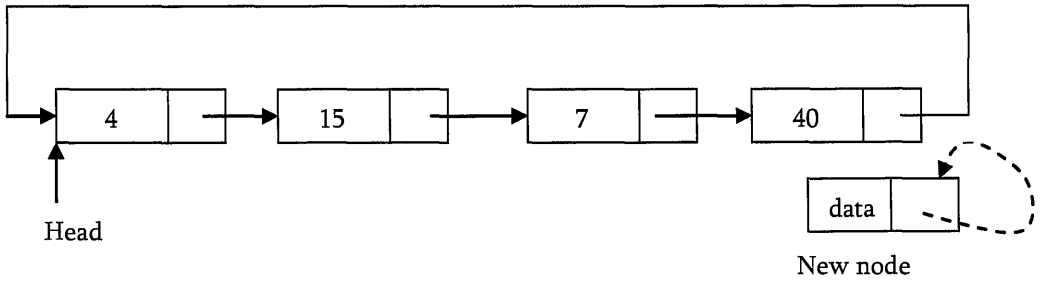
```
        } while (current != head);
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating ne temporary variable.

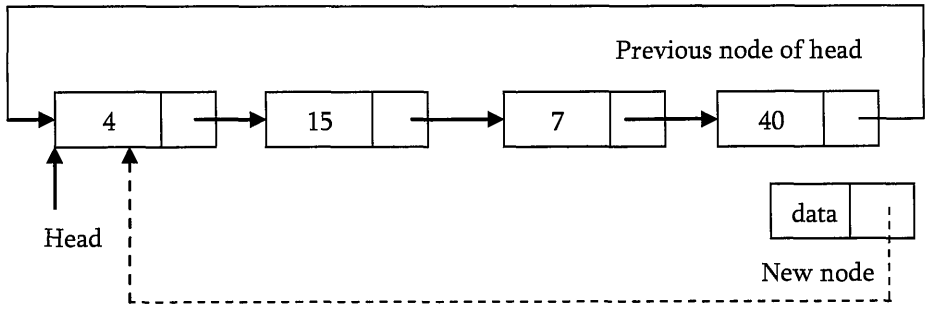
Inserting a Node at the End of a Circular Linked List

Let us add a node containing *data*, at the end of a list (circular list) headed by *head*. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

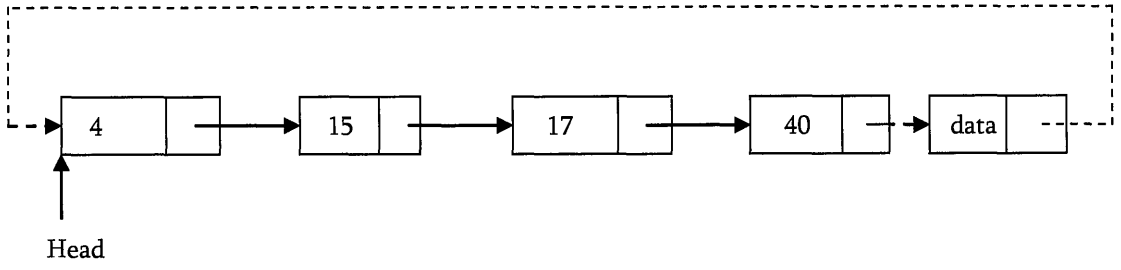
- Create a new node and initially keep its next pointer points to itself.



- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at a node whose next node is head.



- Update the next pointer of previous node to point to new node and we get the list as shown below.




```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
        struct CLLNode current = *head;
        struct CLLNode *newNode = (struct node*) (malloc(sizeof(struct CLLNode)));
        if(!newNode) {
                printf("Memory Error"); return;
        }
        newNode→data = data;
        while (current→next != *head)
                current = current→next;
        newNode→next = newNode;
        if(*head ==NULL)
```

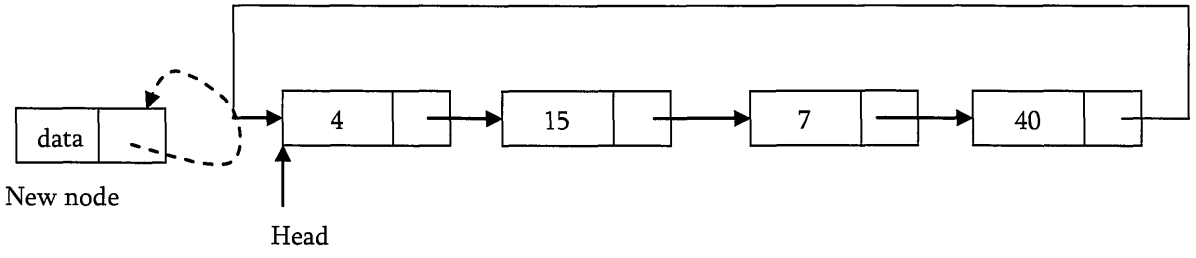
```
            *head = newNode;
        ese {   newNode→next = *head;
            current→next = newNode;
        }
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

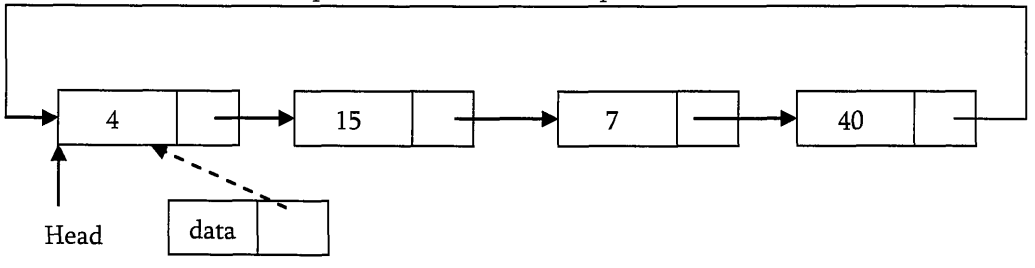
Inserting a Node at Front of a Circular Linked List

The only difference between inserting a node at the beginning and at the ending is that, after inserting the new node we just need to update the pointer. Below are the steps for doing the same.

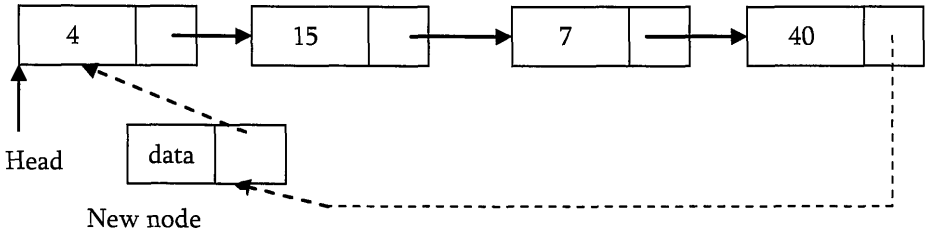
- Create a new node and initially keep its next pointer points to itself.



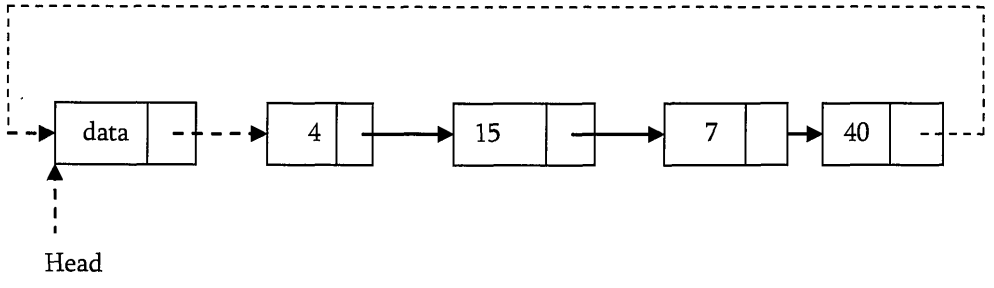
- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at the node which is its previous node in the list.



- Update the previous node of head in the list to point to new node.



- Make new node as head.




```
void InsertAtBeginInCLL (struct CLLNode **head, int data) {
```

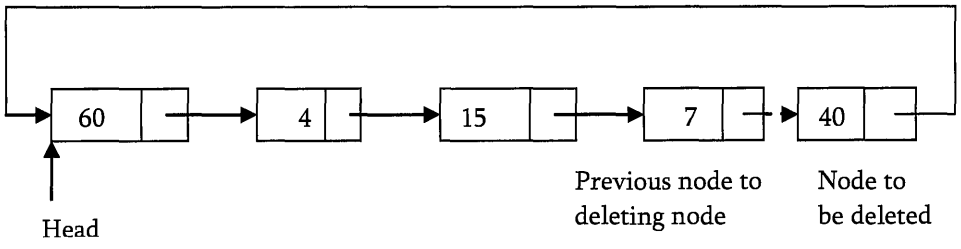
```
        struct CLLNode *current = *head;
        struct CLLNode * newNode = (struct node*) (malloc(sizeof(struct CLLNode)));
        if(!newNode) {
                printf("Memory Error"); return;
        }
        newNode→data = data;
        while (current→next != *head)
                current = current→next;
        newNode→next = newNode;
        if(*head ==NULL)  *head = newNode;
        else {    newNode→next = *head;
                current→next = newNode;
                *head = newNode;
        }
        Return;
}
```

Time Complexity: $O(n)$, for scanning the complete list of size n. Space Complexity: $O(1)$, for creating only one temporary variable.

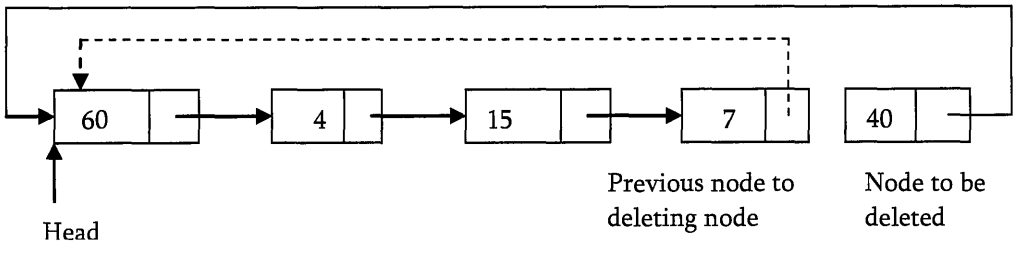
Deleting the Last Node in a Circular List

The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list. To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed *pTail*.

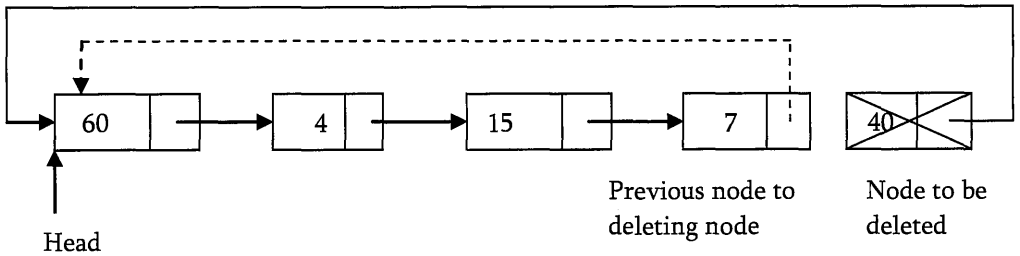
- Traverse the list and find the tail node and its previous node.



- Update the tail nodes previous node next pointer to point to head.



- Dispose the tail node.



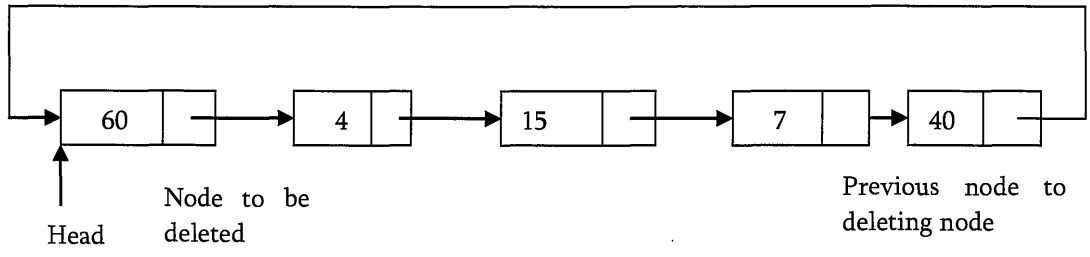
```
void DeleteLastNodeFromCLL (struct CLLNode **head) {
        struct CLLNode *temp = *head;
        struct CLLNode *current = *head;
        if(*head == NULL) {
                printf( "List Empty");
               return;
        }
        while (current→next != *head) {
                temp = current;
                current = current→next;
        }
        free(current);
        return;
}
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for creating one temporary variable.

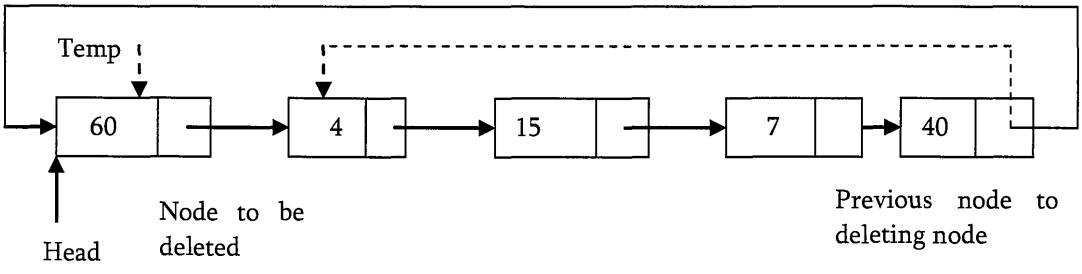
Deleting the First Node in a Circular List

The first node can be deleted by simply replacing the next field of tail node with the next field of the first node.

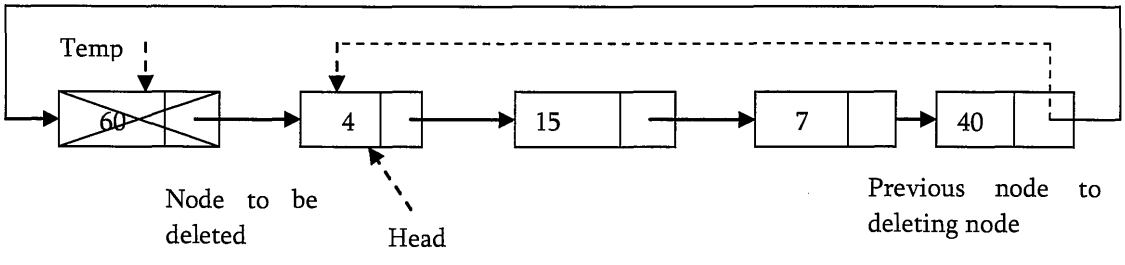
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



```
void DeleteFrontNodeFromCLL (struct CLLNode **head) {
        struct CLLNode *temp = *head;
        struct CLLNode *current = *head;
        if(*head == NULL) {
                printf("List Empty"); return;
        }
        while (current→next != *head)
                current = current→next;
        current→ next = *head→next;
        *head = *head→next;
        free(temp);
        return;
}
```

Time Complexity: $O(n)$, for scanning the complete list of size n. Space Complexity: $O(1)$, for creating one temporary variable.

Applications of Circular List

Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.

3.9 A Memory-Efficient Doubly Linked List

In conventional implementation, we need to keep a forward pointer to the next item on the list and a backward pointer to the previous item. That means, elements in doubly linked list implementations consists of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

Conventional Node Definition

```
typedef struct ListNode {
        int data;
        struct ListNode * prev;
        struct ListNode * next;
};
```

Recently a journal (Sinha) presented an alternative implementation of the doubly linked list ADT, with insertion, traversal and deletion operations. This implementation is based on pointer difference. Each node uses only one pointer field to traverse the list back and forth.

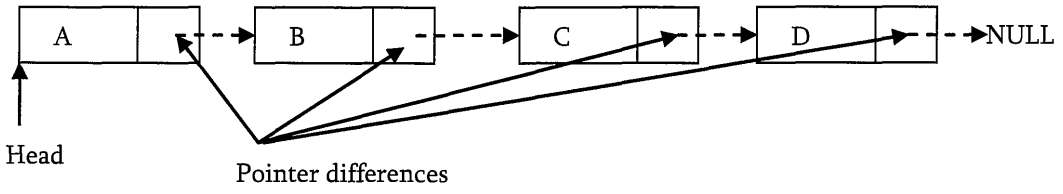
New Node Definition

```
typedef struct ListNode {
        int data;
        struct ListNode * ptrdiff;
};
```

The $ptrdiff$ pointer field contains the difference between the pointer to the next node and the pointer to the previous node. Pointer difference is calculated by using exclusive-or ($\oplus$) operation.

$$ptrdiff = pointer\ to\ previous\ node \oplus pointer\ to\ next\ node.$$

The $ptrdiff$ of the start node (head node) is the $\oplus$ of NULL and *next* node (next node to head). Similarly, the $ptrdiff$ of end node is the $\oplus$ of *previous* node (previous to end node) and NULL. As an example, consider the following linked list.



In the above example,

- The next pointer of A is: NULL ⊕ B
- The next pointer of B is: A ⊕ C
- The next pointer of C is: B ⊕ D
- The next pointer of D is: C ⊕ NULL

Why does it work?

To have answer for this question let us consider the properties of ⊕:

X ⊕ X = 0
X ⊕ 0 = X
X ⊕ Y = Y ⊕ X (symmetric)
(X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z) (transitive)

For the above example, let us assume that we are at C node and want to move to B. We know that Cs *ptrdiff* is defined as B ⊕ D. If we want to move to B, performing ⊕ on Cs *ptrdiff* with D would give B. This is due to fact that,

(B ⊕ D) ⊕ D = B (since, D ⊕ D=0)

Similarly, if we want to move to D, then we have to applying ⊕ to Cs *ptrdiff* with B would give D.

(B ⊕ D) ⊕ B = D (since, B ⊕ B=0)

From the above discussion we can see that just by using single pointer, we are able to move back and forth. A memory-efficient implementation of a doubly linked list is possible to have without compromising much timing efficiency.

3.10 Problems on Linked Lists

Problem-1 Implement Stack using Linked List

Solution: Refer *Stacks* chapter.

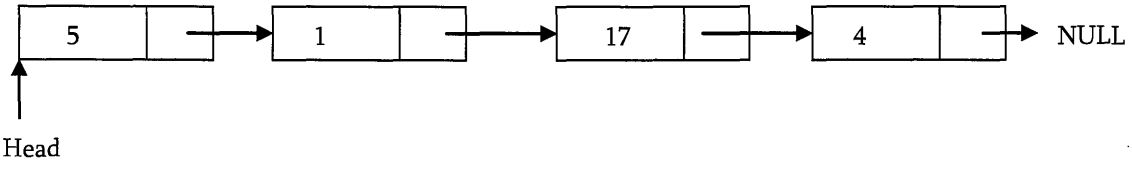
Problem-2 Find n^{th} node from the end of a Linked List.

Solution: Brute-Force Method: Start with the first node and count how many nodes are there after that node. If the number of nodes are $< n-1$ then return saying "fewer number of nodes in the list". If the number of nodes are $> n-1$ then go to next node. Continue this until the numbers of nodes after current node are $n-1$.

Time Complexity: $O(n^2)$, for scanning the remaining list (from current node) for each node. Space Complexity: O(1).

Problem-3 Can we improve the complexity of Problem-2?

Solution: Yes, using hash table. As an example consider the following list.



In this approach, create a hash table whose entries are < *position of node, node address* >. That means, key is the position of the node in the list and value is the address of that node.

Position in List	Address of Node
1	Address of 5 node
2	Address of 1 node
3	Address of 17 node
4	Address of 4 node

By the time we traverse the complete list (for creating hash table), we can find the list length. Let us say, the list length is M. To find n^{th} from end of linked list, we can convert this to M- $n + 1^{th}$ from the beginning. Since we already know the length of the list, it's just a matter of returning M- $n + 1^{th}$ key value from the hash table.

Time Complexity: Time for creating the hash table. Therefore, $T(m) = O(m)$. Space Complexity: $O(m)$. Since, we need to create a hash table of size m.

Problem-4 Can we use Problem-3 approach for solving Problem-2 without creating the hash table?

Solution: Yes. If we observe the Problem-3 solution, what actually we are doing is finding the size of the linked list. That means, we are using hash table to find the size of the linked list. We can find the length of the linked list just by starting at the head node and traversing the list. So, we can find the length of the list without creating the hash table. After finding the length, compute $M - n + 1$ and with one more scan we can get the $M - n + 1^{th}$ node from the beginning. This solution needs two scans: one for finding the length of list and other for finding $M - n + 1^{th}$ node from the beginning.

Time Complexity: Time for finding the length + Time for finding the M- $n + 1^{th}$ node from the beginning. Therefore, $T(n = O(n) + O(n) \approx O(n)$. Space Complexity: $O(1)$. Since, no need of creating the hash table.

Problem-5 Can we solve Problem-2 in one scan?

Solution: Yes. Efficient Approach: Use two pointers *pNthNode* and *pTemp*. Initially, both points to head node of the list. *pNthNode* starts moving only after *pTemp* made n moves. From there both moves forward until *pTemp* reaches end of the list. As a result *pNthNode* points to n^{th} node from end of the linked list.

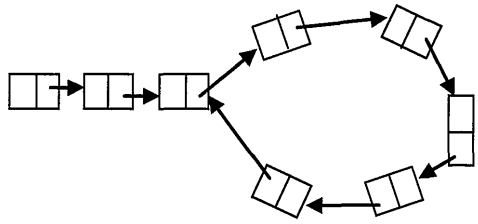
Note: at any point of time both moves one node at time.

```
struct ListNode *NthNodeFromEnd(struct ListNode *head , int NthNode) {
        struct ListNode *pTemp = NULL, *pNthNode = NULL;
        int count = 0;
        for (pTemp = head; pTemp!= NULL;) {
                count++;
                if(NthNode – count == 0 )
                        pNthNode = head;
                else if(NthNode – count > 0)
                        pNthNode = pNthNode→pNext;
                pTemp = pTemp→pNext;
        }
        if(pNthNode) return pNthNode;
        return NULL;
}
```

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

Problem-6 Check whether the given linked list is either NULL-terminated or ends in a cycle (cyclic)

Solution: Brute-Force Approach. As an example consider the following linked list which has a loop in it. The difference between this list and regular list is that, in this list there are two nodes whose next pointers are same. In regular singly linked lists (without loop) each nodes next pointer is unique. That means, the repetition of next pointers indicates the existence of loop.



One simple and brute force way of solving this is, start with the first node and see whether there is any node whose next pointer is current nodes address. If there is a node with same address then that indicates that some other node is pointing to the current node and we can say loops exists. Continue this process for all the nodes of the linked list.

Does this method works? As per the algorithm we are checking for the next pointer addresses, but how do we find the end of the linked list (otherwise we will end up in infinite loop)?

Note: If we start with a node in loop, this method may work depending on the size of the loop.

Problem-7 Can we use hashing technique for solving Problem-6?

Solution: Yes. Using Hash Tables we can solve this problem.
Algorithm:

- Traverse the linked list nodes one by one.
- Check if the nodes address is there in the hash table or not.
- If it is already there in the hash table then that indicates that we are visiting the node which was already visited. This is possible only if the given linked list has a loop in it.
- If the address of the node is not there in the hash table then insert that nodes address into the hash table.
- Continue this process until we reach end of the linked list *or* we find loop.

Time Complexity: $O(n)$ for scanning the linked list. Note that we are doing only scan of the input. Space Complexity: $O(n)$ for hash table.

Problem-8 Can we solve the Problem-6 using sorting technique?

Solution: No. Consider the following algorithm which is based on sorting. And then, we see why this algorithm fails.
Algorithm:

- Traverse the linked list nodes one by one and take all the next pointer values into some array.
- Sort the array which is having next node pointers.
- If there is a loop in the linked list, definitely two nodes next pointers will pointing to the same node.
- After sorting if there is a loop in the list, the nodes whose next pointers are same will come adjacent in the sorted list.
- If there is any such pair exists in the sorted list then we say the linked list has loop in it.

Time Complexity: $O(nlogn)$ for sorting the next pointers array. Space Complexity: $O(n)$ for the next pointers array.

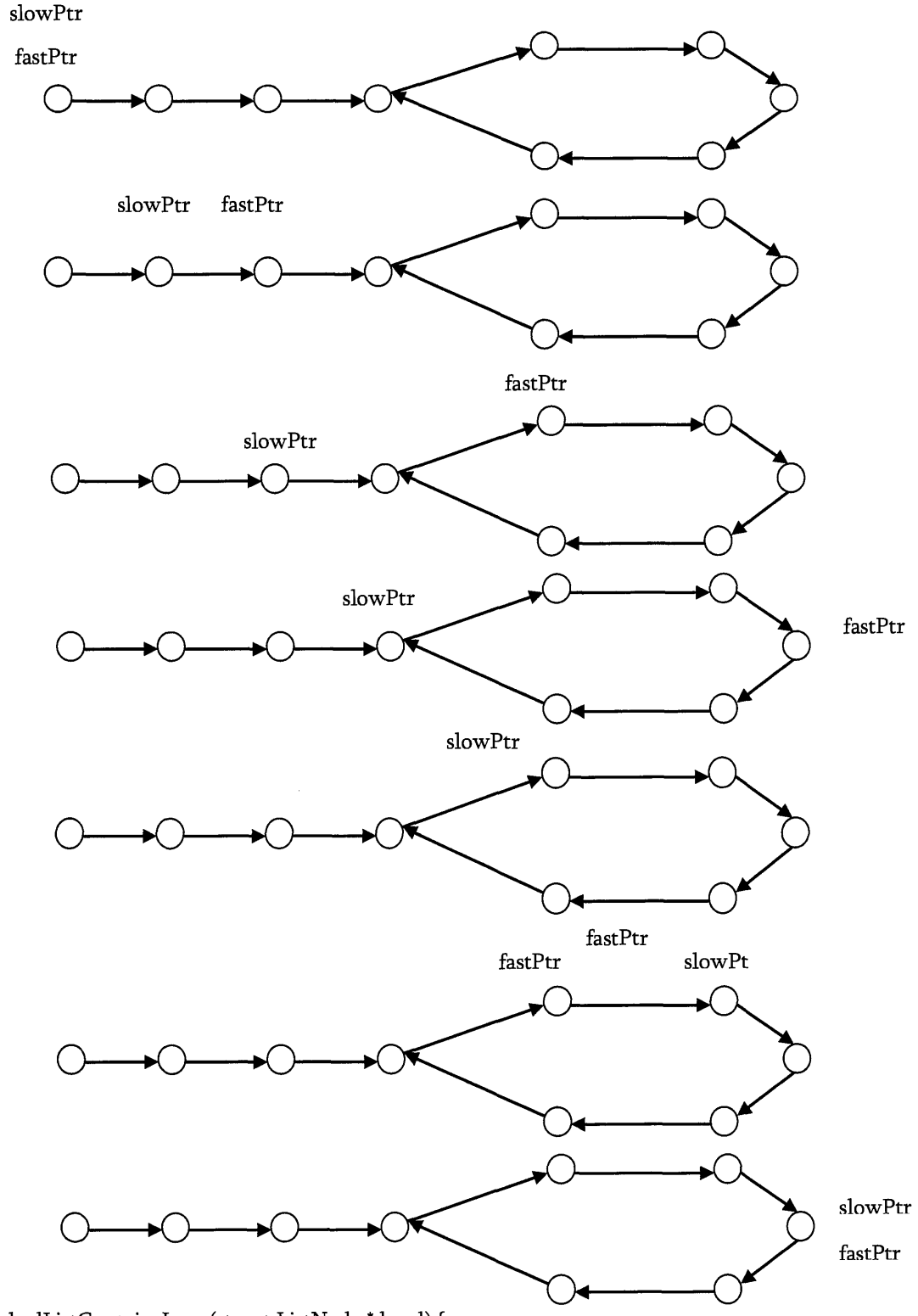
Problem with above algorithm? The above algorithm works only if we can find the length of the list. But if the list is having loop then we may end up in infinite loop. Due to this reason the algorithm fails.

Problem-9 Can we solve the Problem-6 in $O(n)$?

Solution: Yes. Efficient Approach (Memory less Approach): This problem was solved by *Floyd*. The solution is named as Floyd cycle finding algorithm. It uses 2 pointers moving at different speeds to walk the linked list. Once they enter the loop they are expected to meet, which denotes that there is a loop. This works because the only way a faster moving pointer would point to the same location as a slower moving pointer is, if somehow the entire list or a part of it is circular.

Think of a tortoise and a hare running on a track. The faster running hare will catch up with the tortoise if they are running in a loop. As an example, consider the following example and trace out the Floyd algorithm. From the below diagrams we can see that after the final step they are meeting at some point in the loop which may not be the starting of the loop.

Note: *slowPtr* (*tortoise*) moves one pointer at a time and *fastPtr* (*hare*) moves two pointers at a time.




```
int IsLinkedListContainsLoop(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        while(slowPtr  && fastPtr) {
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)
```

```
                return 1;
            if(fastPtr == NULL)
                return 0;
            fastPtr = fastPtr→next;
            if(fastPtr == slowPtr)
                return 1;
            slowPtr = slowPtr→next;
        }
        return 0;
}
```

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

Problem-10 We are given a pointer to the first element of a linked list L. There are two possibilities for L, it either ends (snake) or its last element points back to one of the earlier elements in the list (snail). Give an algorithm that tests whether a given list L is a snake or a snail.

Solution: It is same as Problem-6.

Problem-11 Check whether the given linked list is either NULL-terminated or not. If there is a cycle find the start node of the loop.

Solution: The solution is an extension to the previous solution (Problem-9). After finding the loop in the linked list, we initialize the *slowPtr* to head of the linked list. From that point onwards both *slowPtr* and *fastPtr* moves only one node at a time. The point at which they meet is the start of the loop. Generally we use this method for removing the loops.

```
int FindBeginofLoop(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        int loopExists = 0;
        while(slowPtr  && fastPtr) {
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)  {
                        loopExists = 1;
                        break;
                }
                if(fastPtr == NULL)
                        loopExists = 0;
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr) {
                        loopExists = 1;
                        break;
                }
                slowPtr = slowPtr→next;
        }
        if(loopExists) {
                slowPtr = head;
                while(slowPtr  != fastPtr) {
                        fastPtr = fastPtr→next;
                        slowPtr = slowPtr→next;
                }
                return slowPtr;
        }
        return NULL;
```

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

Problem-12 From the previous discussion and problems we understand that tortoise and hares meeting concludes the existence of loop, but how does moving tortoise to beginning of linked list while keeping the hare at meeting place, followed by moving both one step at a time make them meet at starting point of cycle?

Solution: This problem is the heart of number theory. In Floyd cycle finding algorithm, notice that the tortoise and the hare will meet when they are $n \times L$, where L is the loop length. Furthermore, the tortoise is at the midpoint between the hare and the beginning of the sequence, because of the way they move. Therefore the tortoise is $n \times L$ away from the beginning of the sequence as well.

If we move both one step at a time, from the tortoise position and from the start of the sequence, we know that they will meet as soon as both are in the loop, since they are $n \times L$, a multiple of the loop length, apart. One of them is already in the loop, so we just move the other one in single step until it enters the loop, keeping the other $n \times L$ away from it at all times.

Problem-13 In Floyd cycle finding algorithm, does it work if we use the step 2 and 3 instead of 1 and 2?

Solution: **Yes**, but the complexity might be more. Trace out some example.

Problem-14 Check whether the given linked list is either NULL-terminated or not. If there is a cycle find the length of the loop.

Solution: This solution is also an extension to the basic cycle detection problem. After finding the loop in the linked list, initialize the slowPtr to *fastPtr*. *slowPtr* keeps on moving until it again comes back to *fastPtr*. While moving *slowPtr*, use a counter variable which increments at the rate of 1.

```
int FindLoopLength(struct ListNode * head) {
        struct ListNode *slowPtr = head, *fastPtr = head;
        int loopExists = 0, counter = 0;
        while(slowPtr  && fastPtr) {
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr)
                        loopExists = 1;
                if(fastPtr == NULL)
                        loopExists = 0;
                fastPtr = fastPtr→next;
                if(fastPtr == slowPtr) loopExists = 1;
                slowPtr = slowPtr→next;
        }
        if(loopExists) {
                fastPtr = fastPtr→next;
                while(slowPtr  != fastPtr) {
                        fastPtr = fastPtr→next;
                        counter++;
                }
                return counter;
        }
        return 0;                                   //If no loops exists
}
```

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

Problem-15 Insert a node in a sorted linked list

Solution: Traverse the list and find a position for the element and insert it.

```
struct ListNode *InsertInSortedList(struct ListNode * head, struct ListNode * newNode) {
        struct ListNode *current = head, temp;
        if(!head) return newNode;
        // traverse the list until you find item bigger the new node value
        while (current != NULL && current→data < newNode→data){
                temp = current;
                current = current→next;
        }
        //insert the new node before the big item
        newNode→next = current;
        temp→next = newNode;
        return head;
}
```

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

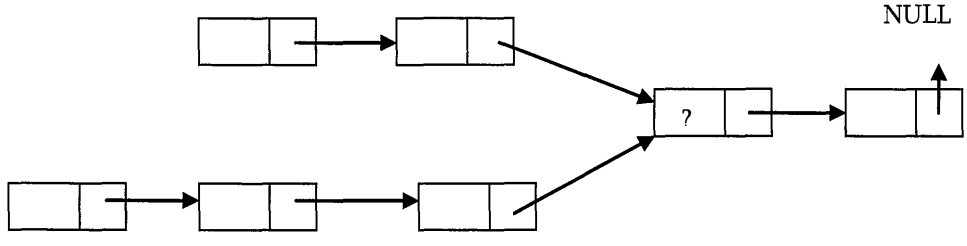
Problem-16 Reverse a singly linked list

Solution: // iterative version

```
struct ListNode *ReverseList(struct ListNode *head) {
        struct ListNode *temp = NULL, *nextNode = NULL;
        while (head) {
                nextNode = head→next;
                head→next = temp;
                temp = head;
                head = nextNode;
        }
        return temp;
}
```

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

Problem-17 Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the list before they intersect are unknown and both list may have it different. $List1$ may have n nodes before it reaches intersection point and $List2$ might have m nodes before it reaches intersection point where m and n may be $m = n, m < n$ or $m > n$. Give an algorithm for fidning the merging point.



Solution: Brute-Force Approach: One easy solution is to compare every node pointer in the first list with every other node pointer in the second list by which the matching node pointers will lead us to the intersecting node. But, the time complexity in this case will O(mn) which will be high.

Time Complexity: O(mn). Space Complexity: O(1).

Problem-18 Can we solve Problem-17 using sorting technique?

Solution: No. Consider the following algorithm which is based on sorting and see why this algorithm fails.

Algorithm

- Take first list node pointers and keep in some array and sort them.

- Take second list node pointers and keep in some array and sort them.
- After sorting, use two indexes: one for first sorted array and other for second sorted array.
- Start comparing values at the indexes and increment the index whichever is having lower value (increment only if the values are not equal).
- At any point, if we were able to find two indexes whose values are same then that indicates that those two nodes are pointing to the same node and we return that node.

Time Complexity: Time for sorting lists + Time for scanning (for comparing)=O($mlogm$) +O($nlogn$) +O($m + n$). We need to consider the one which gives the maximum value. Space Complexity: O(1).

Problem with the above algorithm? Yes. In the algorithm, we are storing all the node pointers of both the lists and sorting. But we are forgetting the fact that, there can be many repeated elements. This is because after the merging point all node pointers are same for both the lists. The algorithm works fine only in one case and it is when both lists have ending node at their merge point.

Problem-19 Can we solve Problem-17 using hash tables?

Solution: Yes.
Algorithm:

- Select a list which is having less number of nodes (If we do not know the lengths beforehand then select one list randomly).
- Now, traverse the other list and for each node pointer of this list check whether the same node pointer exists in the hash table or not.
- If there a merge point for the give lists then we will definitely encounter the node pointer in the hash table.

Time Complexity: Time for creating the hash table + Time for scanning the second list = O(m) + O(n) (or O(n) + O(m), depends on which list we select for creating the hash table). But in both cases the time complexity is same. Space Complexity: O(n) or O(m).

Problem-20 Can we use stacks for solving the Problem-17?

Solution: Yes.
Algorithm:

- Create two stacks: one for the first list and one for the second list.
- Traverse the first list and push all the node address on to the first stack.
- Traverse the second list and push all the node address on to the second stack.
- Now both stacks contain the node address of the corresponding lists.
- Now, compare the top node address of both stacks.
- If they are same, then pop the top elements from both the stacks and keep in some temporary variable (since both node addresses are node, it is enough if we use one temporary variable).
- Continue this process until top node addresses of the stacks are not same.
- This point is the one where the lists merge into single list.
- Return the value of the temporary variable.

Time Complexity: O($m + n$), for scanning both the lists. Space Complexity: O($m + n$), for creating two stacks for both the lists.

Problem-21 Is there any other way of solving the Problem-17?

Solution: Yes. Using "finding the first repeating number" approach in an array (for algorithm refer *Searching* chapter).

Algorithm:

- Create an array A and keep all the next pointers of both the lists in the array.
- In the array find the first repeating element in the array [Refer *Searching* chapter for algorithm].
- The first repeating number indicates the merging point of the both lists.

Time Complexity: $O(m + n)$. Space Complexity: $O(m + n)$.

Problem-22 Can we still think for finding alternative solution for the Problem-17?

Solution: Yes. By combining sorting and search techniques we can reduce the complexity.

Algorithm:

- Create an array A and keep all the next pointers of the first list in the array.
- Sort these array elements.
- Then, for each of the second list element, search in the sorted array (let us assume that we are using binary search which gives $O(logn)$).
- Since we are scanning the second list one by one, the first repeating element which appears in the array is nothing but the merging point.

Time Complexity: Time for sorting + Time for searching = $O(Max(mlogm, nlogn))$. Space Complexity: $O(Max(m, n))$.

Problem-23 Can we improve the complexity for the Problem-17?

Solution: Yes.

Efficient Approach:

- Find lengths (L1 and L2) of both list -- $O(n) + O(m) = O(max(m, n))$.
- Take the difference d of the lengths -- $O(1)$.
- Make d steps in longer list -- $O(d)$.
- Step in both lists in parallel until links to next node match -- $O(min(m, n))$.
- Total time complexity = $O(max(m, n))$.
- Space Complexity = $O(1)$.

```
struct ListNode* FindIntersectingNode(struct ListNode* list1, struct ListNode* list2) {
    int L1=0, L2=0, diff=0;
    struct ListNode *head1 = list1, *head2 = list2;
    while(head1!= NULL) {
            L1++;
            head1 = head1→next;
    }
    while(head2!= NULL) {
            L2++;
            head2 = head2→next;
    }
    diff = L1 - L2;
    if(L1 < L2) {
            head1 = list2;
            head2 = list1;
            diff = L2 - L1;
    }
    for(int i = 0; i < diff; i++)
            head1 = head1→next;
    while(head1 != NULL && head2 != NULL) {
            if(head1 == head2)
                    return head1→data;
            head1= head1→next;
            head2= head2→next;
    }
    return NULL;
```

```
}
```

Problem-24 How will you find the middle of the linked list?

Solution: Brute-Force Approach: For each of the node count how many nodes are there in the list and see whether it is the middle.

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

Problem-25 Can we improve the complexity of Problem-24?

Solution: Yes.
Algorithm:

- Traverse the list and find the length of the list.
- After finding the length, again scan the list and locate $n/2$ node from the beginning.

Time Complexity: Time for finding the length of the list + Time for locating middle node = $O(n) + O(n) \approx O(n)$. Space Complexity: $O(1)$.

Problem-26 Can we use hash table for solving Problem-24?

Solution: Yes. The reasoning is same as that of Problem-3.

Time Complexity: Time for creating the hash table. Therefore, $T(n) = O(n)$. Space Complexity: $O(n)$. Since, we need to create a hash table of size n.

Problem-27 Can we solve Problem-24 just in one scan?

Solution: Efficient Approach: Use two pointers. Move one pointer at twice the speed of the second. When the first pointer reaches end of the list, the second pointer will be pointing to the middle node.

Note: If the list has even number of nodes, the middle node will be of $\lfloor n/2 \rfloor$.

```
struct ListNode * FindMiddle(struct ListNode *head) {
        struct ListNode *ptr1x, *ptr2x;
        ptr1x = ptr2x = head;
        int i=0;
        // keep looping until we reach the tail (next will be NULL for the last node)
        while(ptr1x→next != NULL) {
                if(i == 0) {
                        ptr1x = ptr1x→next; //increment only the 1st pointer
                        i=1;
                }
                else if( i == 1) {
                        ptr1x = ptr1x→next; //increment both pointers
                        ptr2x = ptr2x→next;
                        i = 0;
                }
        }
        return ptr2x;       //now return the ptr2 which points to the middle node
}
```

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

Problem-28 How will you display a linked list from the end?

Solution: Traverse recursively till end of the linked list. While coming back, start printing the elements.

```
//This Function will print the linked list from end
void PrintListFromEnd(struct ListNode *head) {
```

```
        if(!head)
            return;
        PrintListFromEnd(head→next);
        printf("%d ",head→data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$→ for Stack.

Problem-29 Check whether the given Linked List length is even or odd?

Solution: Use $2x$ pointer. Take a pointer which moves at $2x$ [two nodes at a time]. At the end, if the length is even then pointer will be NULL otherwise it will point to last node.

```
int IsLinkedListLengthEven(struct ListNode * listHead) {
        while(listHead && listHead→next)
            listHead = listHead→next→next;
        if(!listHead)
                return 0;
        return 1;
}
```

Time Complexity: $O(\lfloor n/2 \rfloor) \approx O(n)$. Space Complexity: $O(1)$.

Problem-30 If the head of a linked list is pointing to kth element, then how will you get the elements before kth element?

Solution: Use Memory Efficient Linked Lists [XOR Linked Lists].

Problem-31 Given two sorted Linked Lists, we need to merge them into the third list in sorted order.

Solution:

```
struct ListNode *MergeList(struct ListNode *a, struct ListNode *b) {
    struct ListNode *result = NULL;
    if(a == NULL) return b;
    if(b == NULL) return a;
    if(a→data <= b→data) {
        result =a;
        result→next = MergeList(a→next, b);
    }
    else {
        result =b;
        result→next = MergeList(b→next,a);
    }
    return result;
}
```

Time Complexity – $O(n + m)$, where n and m are lengths of two lists.

Problem-32 Reverse the linked list in pairs. If you have a linked list that holds $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$, then after the function has been called the linked list would hold $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$.

Solution: //Recursive Version

```
        void ReversePairRecursive(struct ListNode *head) {
                struct ListNode *temp;
                if(head ==NULL || head→next ==NULL)
                        return;                         //base case for empty or 1 element list
                else {  //Reverse first pair
                        temp = head→next;
```

```
                head→next = temp→next;
                temp→next = head;
                //Call the method recursively for the rest of the list
                ReversePairRecursive(head→next);
        }
}
/*Iterative version*/
void ReversePairIterative(struct ListNode *head) {
        struct ListNode *temp, *temp2, *current = head;
        while(current != NULL && current→next != NULL) {
                //Swap the pair
                temp = current→next;
                temp2 = temp→next;
                temp→next = current;
                current→next = temp2;
                //Advance the current pointer
                if(current)
                    current = current→next;
        }
}
```

Time Complexity – O(n). Space Complexity - O(1).

Problem-33 Given a binary tree convert it to doubly linked list.

Solution: Refer *Trees* chapter.

Problem-34 How do we sort the Linked Lists?

Solution: Refer *Sorting* chapter.

Problem-35 If we want to concatenate two linked lists which of the following gives O(1) complexity?
1) Singly linked lists 2) Doubly linked lists 3) Circular doubly linked lists

Solution: Circular Doubly Linked Lists. This is because for singly and doubly linked lists, we need to traverse the first list till the end and append the second list. But in case of circular doubly linked lists we don't have to traverse the lists.

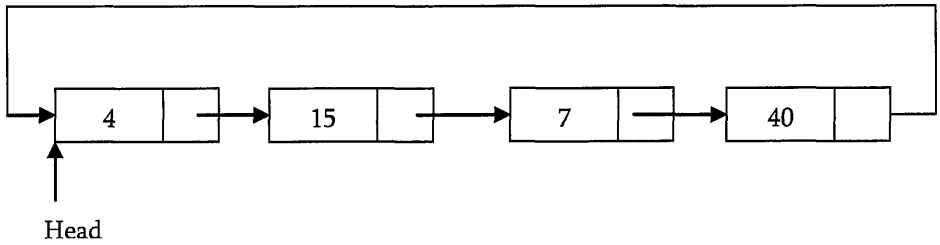
Problem-36 Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

Solution:

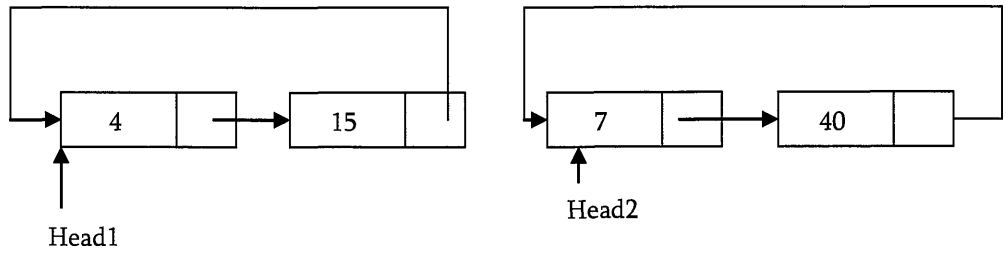
Algorithm

- Store the mid and last pointers of the circular linked list using Floyd cycle finding algorithm.
- Make the second half circular.
- Make the first half circular.
- Set head pointers of the two linked lists.

As an example, consider the following circular list.



After the split, the above list will look like:




```
/* structure for a node */
struct ListNode {
        int data;
        struct ListNode *next;
};
void SplitList(struct ListNode *head, struct ListNode **head1, struct ListNode **head2) {
        struct ListNode *slowPtr = head;
        struct ListNode *fastPtr = head;
        if(head == NULL) return;
        /* If there are odd nodes in the circular list then fastPtr→next becomes
          head and for even nodes fastPtr→next→next becomes head */
        while(fastPtr→next != head && fastPtr→next→next != head) {
                fastPtr = fastPtr→next→next;
                slowPtr = slowPtr→next;
        }
        /* If there are even elements in list then move fastPtr */
        if(fastPtr→next→next == head)
                fastPtr = fastPtr→next;
        /* Set the head pointer of first half */
        *head1 = head;
        /* Set the head pointer of second half */
        if(head→next != head)
                *head2 = slowPtr→next;
        /* Make second half circular */
        fastPtr→next = slowPtr→next;
        /* Make first half circular */
        slowPtr→next = head;
}
```

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

Problem-37 How will you check if the linked list is palindrome or not?

Solution:

Algorithm

1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and second half.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half.

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

Problem-38 For a given K value ($K > 0$) reverse blocks of K nodes in a list.

Example: Input: 1 2 3 4 5 6 7 8 9 10. Output for different K values:

For $K = 2$: 2 1 4 3 6 5 8 7 10 9 For $K = 3$: 3 2 1 6 5 4 9 8 7 10 For $K = 4$: 4 3 2 1 8 7 6 5 9 10

Solution:

Algorithm: This is an extension of swapping nodes in a linked list.

1) Check if remaining list has K nodes.
    a. If yes get the pointer of $K + 1^{th}$ node.
    b. Else return.
2) Reverse first K nodes.
3) Set next of last node (after reversal) to $K + 1^{th}$ node.
4) Move to $K + 1^{th}$ node.
5) Go to step 1.
6) $K - 1^{th}$ node of first K nodes becomes the new head if available. Otherwise, we can return the head.

```
struct ListNode * GetKPlusOneThNode(int K, struct ListNode *head) {
        struct ListNode *Kth;
        int i = 0;
        if(!head)
                return head;
        for (i=0, Kth=head; Kth && (i < K); i++, Kth=Kth→next);
        if(i==K && Kth!=NULL)
                return Kth;
        return head→next;
}
int HasKnodes(struct ListNode *head, int K) {
        int i =0;
        for(i=0; head && (i < K); i++, head=head→next);
        if(i == K)
                return 1;
        return 0;
}
struct ListNode *ReverseBlockOfK-nodesInLinkedList(struct ListNode *head, int K) {
        struct ListNode *cur=head, *temp, *next, newHead;
        int i;
        if(K==0 || K==1)
                return head;
        if(HasKnodes(cur, K-1))
           newHead = GetKPlusOneThNode(K-1, cur);
        else newHead = head;
        while(cur && HasKnodes(cur, K)) {
                temp = GetKPlusOneThNode(K, cur);
                i=0;
                while(i < K) {
                        next = cur→next;
                        cur→next=temp;
                        temp = cur;
                        cur = next;
                        i++;
                }
        }
        return newHead;
}
```

Problem-39 Is it possible to get O(1) access time for Linked Lists?

Solution: **Yes.** Create a linked list at the same time keep it in a hash table. For n elements we have to keep all the elements into hash table which gives preprocessing time of $O(n)$. To read any element we require only constant time $O(1)$ and to read n elements we require $n * 1$ unit of time $= n$ units. Hence by using amortized analysis we can say that element access can be performed within $O(1)$ time.

Time Complexity – $O(1)$ [Amortized]. Space Complexity - $O(n)$ for Hash.

Problem-40 **JosephusCircle:** N people have decided to elect a leader by arranging themselves in a circle and eliminating every M^{th} person around the circle, closing ranks as each person drops out. Find which person will be the last one remaining (with rank 1).

Solution: Assume the input is a circular linked list with N nodes and each node has a number (range 1 to N) associated with it. The head node has number 1 as data.

```
struct ListNode *GetJosephusPosition(){
        struct ListNode *p, *q;
        printf("Enter N (number of players): "); scanf("%d", &N);
        printf("Enter M (every M-th payer gets eliminated): "); scanf("%d", &M);
        // Create circular linked list containing all the players:
        p = q = malloc(sizeof(struct node));
        p→data = 1;
        for (int i = 2; i <= N; ++i) {
                p→next = malloc(sizeof(struct node));
                p = p→next;
                p→data = i;
        }
        p→next = q; // Close the circular linked list by having the last node point to the first.
        // Eliminate every M-th player as long as more than one player remains:
        for (int count = N; count > 1; --count) {
                for (int i = 0; i < M - 1; ++i)
                        p = p→next;
                p→next = p→next→next; // Remove the eiminated player from the circular linked list.
        }
        printf("Last player left standing (Josephus Position) is %d\n.", p→data);
}
```

Problem-41 Given a linked list consists of data, next pointer and also a random pointer which points to a random node of the list. Give an algorithm for cloning the list.

Solution: We can use the hash table to associate newly created nodes with the instances of node in the given list.
Algorithm:

- Scan the original list and for each node X, create a new node Y with data of X, then store the pair (X, Y) in hash table using X as a key. Note that during this scan we set $Y->next$ and $Y->random$ to $NULL$ and we will fix them in the next scan
- Now for each node X in the original list we have a copy Y stored in our hash table. We scan again the original list and set the pointers buildings the new list

```
struct ListNode *Clone(struct ListNode *head){
        struct ListNode *X, *Y;
        struct HashTable *HT = CreateHashTable();
        X = head;
        while (X != NULL) {
                Y = (struct ListNode *)malloc(sizeof(struct ListNode *));
                Y→data = X→data;
```

```
                Y→next = NULL;
                Y→random = NULL;
                HT.insert(X, Y);
                X = X→next;
        }
        X = head;
        while (X != NULL) {
                // get the node Y corresponding to X from the hash table
                Y = HT.get(X);
                Y→next = HT.get(X→next);
                Y.setRandom = HT.get(X→random);
                X = X→next;
        }
        // Return the head of the new list, that is the Node Y
        return HT.get(head);
}
```

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

Problem-42 Can we solve Problem-41 without any extra space?

Solution: Yes. Follow the comments in below code and traceout.

```
void Clone(struct ListNode *head){
        struct ListNode *temp, *temp2;
        //Step1: put temp→random in temp2→next, so that we can reuse the temp→random field to point to temp2.
        temp = head;
        while (temp != NULL) {
                temp2 = (struct ListNode *)malloc(sizeof(struct ListNode *));
                temp2→data = temp→data;
                temp2→next = temp→random;
                temp→random = temp2;
                temp = temp→next;
        }
        //Step2: Setting temp2→random. temp2→next is the old copy of the node that
        // temp2→random should point to, so temp2→next→random is the new copy.
        temp = head;
        while (temp != NULL) {
                temp2 = temp→random;
                temp2→random = temp2→next→random;
                temp = temp→next;
        }
        //Step3: Repair damage to old list and fill in next pointer in new list.
        temp = head;
        while (temp != NULL) {
                temp2 = temp→random;
                temp→random = temp2→next;
                temp2→next = temp→next→random;
                temp = temp→next;
        }
}
```

Time Complexity: $O(3n) \approx O(n)$. Space Complexity: $O(1)$.

STACKS

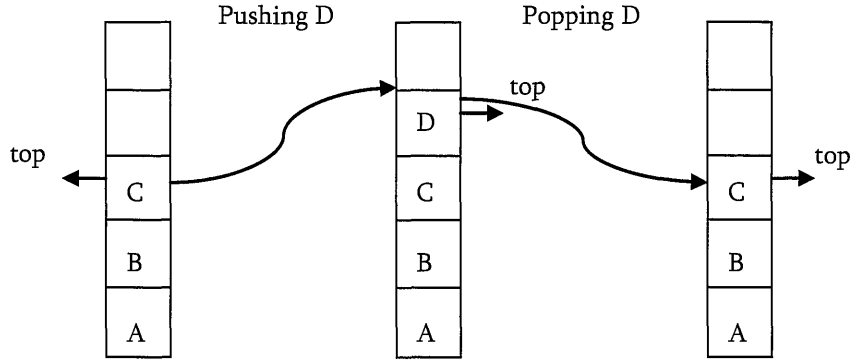
Chapter-4

4.1 What is a Stack?

A stack is a simple data structure used for storing data (similar to Linked Lists). In stack, the order in which the data arrives is important. The pile of plates of a cafeteria is a good example of stack. The plates are added to the stack as they are cleaned. They are placed on the top. When a plate is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

Definition: A *stack* is an ordered list in which insertion and deletion are done at one end, where the end is called as *top*. The last element inserted is the first one to be deleted. Hence, it is called Last in First out (LIFO) or First in Last out (FILO) list.

Special names are given to the two changes that can be made to a stack. When an element is inserted in a stack, the concept is called as *push*, and when an element is removed from the stack, the concept is called as *pop*. Trying to pop out an empty stack is called as *underflow* and trying to push an element in a full stack is called as *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.



4.2 How Stacks are used?

Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task, which is more important. The developer places the long-term project aside and begins work on the new task. The phone then rings, this is the highest priority, as it must be answered immediately. The developer pushes the present task into the pending tray and answers the phone. When the call is complete the task abandoned top answer the phone is retrieved from the pending tray and work progresses. If another call comes in, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

4.3 Stack ADT

The following operations make a stack an ADT. For simplicity assume the data is of integer type.

Main stack operations

- Push (int data): Inserts *data* onto stack.

- int Pop(): Removes and returns the last inserted element from the stack.

Auxiliary stack operations

- int Top(): Returns the last inserted element without removing it.
- int Size(): Returns the number of elements stored in stack.
- int IsEmptyStack(): Indicates whether any elements are stored in stack or not.
- int IsFullStack(): Indicates whether the stack is full or not.

Exceptions

Attempting the execution of an operation may sometimes cause an error condition, called an exception. Exceptions are said to be "thrown" by an operation that cannot be executed. In the Stack ADT, operations pop and top cannot be performed if the stack is empty. Attempting the execution of pop (top) on an empty stack throws an exception. Trying to push an element in a full stack throws an exception.

4.4 Applications

Following are the some of the applications in which stacks plays an important role.

Direct applications

- Balancing of symbols
- Infix-to-postfix conversion
- Evaluation of postfix expression
- Implementing function calls (including recursion)
- Finding of spans (finding spans in stock markets, refer *Problems* section)
- Page-visited history in a Web browser [Back Buttons]
- Undo sequence in a text editor
- Matching Tags in HTML and XML

Indirect applications

- Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)
- Component of other data structures (Example: Simulating queues, refer *Queues* chapter)

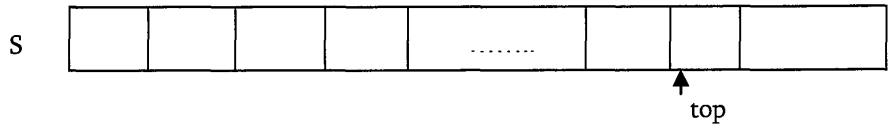
4.5 Implementation

There are many ways of implementing stack ADT and below are the commonly used methods.

- Simple array based implementation
- Dynamic array based implementation
- Linked lists implementation

Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.



The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from empty stack then it will throw *stack empty exception*.

```
struct ArrayStack {
        int top;
        int capacity;
        int *array;
};
struct ArrayStack *CreateStack() {
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        if(!S) return NULL;
        S→capacity = 1;
        S→top = -1;
        S→array= malloc(S→capacity * sizeof(int));
        if(!S→array) return NULL;
        return S;
}
int IsEmptyStack(struct ArrayStack *S) {
        return (S→top == -1);      // if the condition is true then 1 is returned else 0 is returned
}
int IsFullStack(struct ArrayStack *S){
        //if the condition is true then 1 is returned else 0 is returned
        return (S→top == S→capacity - 1);
}
void Push(struct ArrayStack *S, int data){
        /* S→top == capacity -1 indicates that the stack is full*/
        if(IsFullStack(S)) printf( "Stack Overflow");
        else      /*Increasing the 'top' by 1 and storing the value at 'top' position*/
                S→ array[++S→top]= data;
}
int Pop(struct ArrayStack *S){
        if(IsEmptyStack(S)){      /* S→top == - 1 indicates empty stack*/
                printf("Stack is Empty");
                return 0;
        }
        else /* Removing element from 'top' of the array and reducing 'top' by 1*/
                return (S→ array[S→top--]);
}
void DeleteStack(struct DynArrayStack *S){
        if(S) {   if(S→array) free(S→array);
                free(S);
        }
}
```

Performance & Limitations

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$

Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

Limitations

The maximum size of the stack must be defined in prior and cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

Dynamic Array Implementation

First, let's consider how we implemented a simple array based stack. We took one index variable *top* which points to the index of the most recently inserted element in the stack. To insert (or push) an element, we increment *top* index and then place the new element at that index. Similarly, to delete (or pop) an element we take the element at *top* index and then decrement the *top* index. We represent empty queue with *top* value equal to -1. The issue still need to be resolved is that what we do when all the slots in fixed size array stack are occupied?

First try: What if we increment the size of the array by 1 every time the stack is full?

- Push(): increase size of S[] by 1
- Pop(): decrease size of S[] by 1

Problems with this approach?

This way of incrementing the array size is too expensive. Let us a see the reason for this. For example, at $n = 1$, to push an element create a new array of size 2 and copy all the old array elements to new array and at the end add the new element. At $n = 2$, to push an element create a new array of size 3 and copy all the old array elements to new array and at the end add the new element.

Similarly, at $n = n - 1$, if we want to push an element create a new array of size n and copy all the old array elements to new array and at the end add the new element. After n push operations the total time $T(n)$ (number of copy operations) is proportional to $1 + 2 + \ldots + n \approx O(n^2)$.

Alternative Approach: Repeated Doubling

Let us improve the complexity by using array *doubling* technique. If the array is full, create a new array of twice the size, and copy items. With this approach, pushing n items takes time proportional to n (not n^2).

For simplicity, let us assume that initially we started with $n = 1$ and moved till $n = 32$. That means, we do the doubling at $1, 2, 4, 8, 16$. The other way of analyzing the same is, at $n = 1$, if we want to add (push) an element then double the current size of array and copy all the elements of old array to new array.

At, $n = 1$, we do 1 copy operation, at $n = 2$, we do 2 copy operations, and n = 4, we do 4 copy operations and so on. By the time we reach $n = 32$, the total number of copy operations is $1 + 2 + 4 + 8 + 16 = 31$ which is approximately equal to $2n$ value (32). If we observe carefully, we are doing the doubling operation $logn$ times.

Now, let us generalize the discussion. For n push operations we double the array size $logn$ times. That means, we will have $logn$ terms in below expression. The total time $T(n)$ of a series of n push operations is proportional to

$$1 + 2 + 4 + 8 \ldots + \frac{n}{4} + \frac{n}{2} + n = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} \ldots + 4 + 2 + 1$$
$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}\right)$$
$$= n(2) \approx 2n = O(n)$$

$T(n)$ is $O(n)$ and the amortized time of a push operation is $O(1)$.

```
struct DynArrayStack {
        int top;
        int capacity;
        int *array;
};
struct DynArrayStack *CreateStack(){
        struct DynArrayStack *S = malloc(sizeof(struct DynArrayStack));
        if(!S) return NULL;
        S→capacity = 1;
        S→top = -1;
        S→array = malloc(S→capacity * sizeof(int));          // allocate an array of size 1 initially
        if(!S→array) return NULL;
        return S;
}
int IsFullStack(struct DynArrayStack *S){
        return (S→top == S→capacity-1);
}
void DoubleStack(struct DynArrayStack *S){
        S→capacity *= 2;
        S→array = realloc(S→array, S→capacity);
}
void Push(struct DynArrayStack *S, int x){
        // No overflow in this implementation
         if(IsFullStack(S))
                DoubleStack(S);
        S→array[++S→top] = x;
}
int IsEmptyStack(struct DynArrayStack *S){
         return S→top == -1;
}
int Top(struct DynArrayStack *S){
         if(IsEmptyStack(S))
                return INT_MIN;
        return S→array[S→top];
}
int Pop(struct DynArrayStack *S){
        if(IsEmptyStack(S))
                return INT_MIN;
        return S→array[S→top--];
}
void DeleteStack(struct DynArrayStack *S){
        if(S) {
                if(S→array) free(S→array);
                free(S);
        }
}
```

Performance

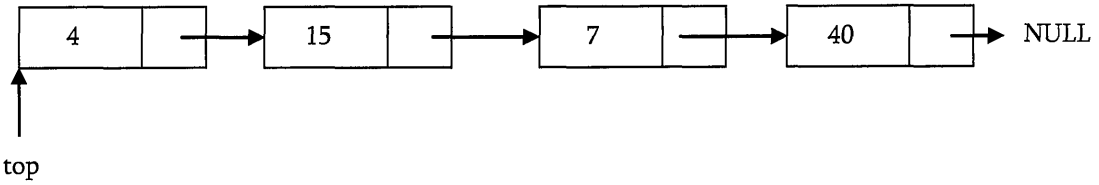
Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1))
Time Complexity of IsFullStack()	O(1)
Time Complexity of DeleteStack()	O(1)

Note: Too many doublings may cause memory overflow exception.

Linked List Implementation

The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).




```
struct ListNode{
        int data;
        struct ListNode *next;
};
struct Stack *CreateStack(){
        return NULL;
}
void Push(struct Stack **top, int data){
         struct Stack *temp;
         temp = malloc(sizeof(struct Stack));
        if(!temp) return NULL;
        temp→data = data;
        temp→next = *top;
        *top = temp;
}
int IsEmptyStack(struct Stack *top){
        return top == NULL;
}
int Pop(struct Stack **top){
        int data;
        struct Stack *temp;
        if(IsEmptyStack(top))
                return INT_MIN;
        temp = *top;
        *top = *top→next;
        data = temp→data;
        free(temp);
        return data;
```

```
}
int Top(struct Stack * top){
        if(IsEmptyStack(top))  return INT_MIN;
        return top→next→data;
}
void DeleteStack(struct Stack **top){
        struct Stack *temp, *p;
        p = *top;
        while( p→next) {
                temp = p→next;
                p→next = temp→next;
                free(temp);
        }
        free(p);
}
```

Performance

Let n be the number of elements in the stack. Let n be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for n push operations)	$O(n)$
Time Complexity of CreateStack()	$O(1)$
Time Complexity of Push()	$O(1)$ (Average)
Time Complexity of Pop()	$O(1)$
Time Complexity of Top()	$O(1)$
Time Complexity of IsEmptyStack()	$O(1)$
Time Complexity of DeleteStack()	$O(n)$

4.6 Comparison of Implementations

Comparing Incremental Strategy and Doubling Strategy

We compare the incremental strategy and doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations. We start with an empty stack represented by an array of size 1. We call *amortized* time of a push operation is the average time taken by a push over the series of operations, i.e., $T(n)/n$.

Incremental Strategy: The amortized time (average time per operation) of a push operation is $O(n)$ $[O(n^2)/ n]$.

Doubling Strategy: In this method, the amortized time of a push operation is $O(1)$ $[O(n) / n]$.

Note: For reasoning, refer implementation section.

Comparing Array Implementation and Linked List Implementation

Array Implementation

- Operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of n operations (starting from empty stack) -- "amortized" bound takes time proportional to n.

Linked list Implementation

- Grows and shrinks gracefully.
- Every operation takes constant time $O(1)$.

- Every operation uses extra space and time to deal with references.

4.7 Problems on Stacks

Problem-1 Discuss how stacks can be used for checking balancing of symbols?

Solution: Stacks can be used to check whether the given expression has balanced symbols or not. This algorithm is very much useful in compilers. Each time parser reads one character at a time. If the character is an opening delimiter like (, {, or [- then it is written to the stack. When a closing delimiter is encountered like),}, or]- is encountered the stack is popped. The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line. A linear-time $O(n)$ algorithm based on stack can be given as:

Algorithm

a) Create a stack.
b) while (end of input is not reached) {
    1) If the character read is not a symbol to be balanced, ignore it.
    2) If the character is an opening symbol like (, [, {, push it onto the stack
    3) If it is a closing symbol like),],}, then if the stack is empty report an error. Otherwise pop the stack.
    4) If the symbol popped is not the corresponding opening symbol, report an error.
   }
c) At end of input, if the stack is not empty report an error

Examples:

Example	Valid?	Description
(A+B)+(C-D)	Yes	The expression is having balanced symbol
((A+B)+(C-D)	No	One closing brace is missing
((A+B)+[C-D])	Yes	Opening and immediate closing braces correspond
((A+B)+[C-D]}	No	The last closing brace does not correspond with the first opening parenthesis

For tracing the algorithm let us assume that the input is: () (() [()])

Input Symbol, A[i]	Operation	Stack	Output
(	Push (	(	
)	Pop (Test if (and A[i] match? YES		
(	Push (	(	
(	Push (	((	
)	Pop (Test if (and A[i] match? YES	(	
[	Push [	([	
(	Push (	([(	
)	Pop (Test if(and A[i] match? YES	([	
]	Pop [Test if [and A[i] match? YES	(	

)	Pop (Test if(and A[i] match? YES		
	Test if stack is Empty? YES		TRUE

Time Complexity: $O(n)$. Since, we are scanning the input only once. Space Complexity: $O(n)$ [for stack].

Problem-2 Discuss infix to postfix conversion algorithm using stack?

Solution: Before discussing the algorithm, first let us see the definitions of infix, prefix and postfix expressions.

Infix: An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another Infix string.

A
A+B
(A+B)+ (C-D)

Prefix: A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.

A
+AB
++AB-CD

Postfix: A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.

A
AB+
AB+CD-+

Prefix and postfix notions are methods of writing mathematical expressions without parenthesis. Time to evaluate a postfix and prefix expression is $O(n)$, were n is the number of elements in the array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+ABC	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

Now, let us concentrate on the algorithm. In infix expressions, the operator precedence is implicit unless we use parentheses. Therefore, for the infix to postfix conversion algorithm we have to define the operator precedence (or priority) inside the algorithm. The table shows the precedence and their associatively (order of evaluation) among operators.

Token	Operator	Precedence	Associatively
( ) [ ] → .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^=	assignment	2	right-to-left
,	Comma	1	left-to-right

Important Properties

- Let us consider the infix expression 2 + 3 * 4 and its postfix equivalent 2 3 4 * +. Notice that between infix and postfix the order of the numbers (or operands) is unchanged. It is 2 3 4 in both cases. But the order of the operators * and + is affected in the two expressions.
- Only one stack is enough to convert an infix expression to postfix expression. The stack that we use in the algorithm will be used to change the order of operators from infix to postfix. The stack we use will only contain operators and the open parentheses symbol '('. Postfix expressions do not contain parentheses. We shall not output the parentheses in the postfix output.

Algorithm

a) Create a stack
b) for each character t in the input stream{
    if(t is an operand)
        output t
    else if(t is a right parenthesis){
        Pop and output tokens until a left parenthesis is popped (but not output)
    }
    else // t is an operator or left parenthesis{
        pop and output tokens until one of lower priority than t is encountered or a left parenthesis is encountered or the stack is empty
        Push t
    }
}
c) pop and output tokens until the stack is empty

For better understanding let us trace out some example: A * B- (C + D) + E

Input Character	Operation on Stack	Stack	Postfix Expression
A		Empty	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C

D			AB*CD
)	Pop and append to postfix till '('	-	AB*CD+
+	Check and Push	+	AB*CD+-
E		+	AB*CD+-E
End of input	Pop till empty		AB*CD+-E+

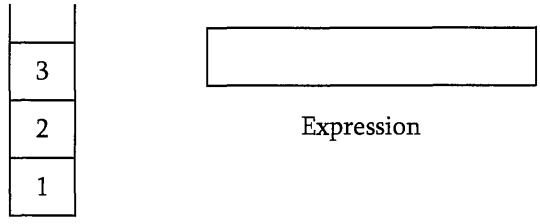
Problem-3 Discuss postfix evaluation using stacks?

Solution:

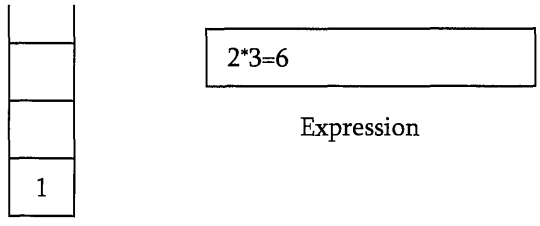
Algorithm

1 Scan the Postfix string from left to right.
2 Initialize an empty stack.
3 Repeat the below steps 4 and 5 till all the characters are scanned.
4 If the scanned character is an operand, push it onto the stack.
5 If the scanned character is an operator, and if the operator is unary operator then pop an element from the stack. If the operator is binary operator then pop two elements from the stack. After popping the elements, apply the operator to those popped elements. Let the result of this operation be retVal onto the stack.
6 After all characters are scanned, we will have only one element in the stack.
7 Return top of the stack as result.

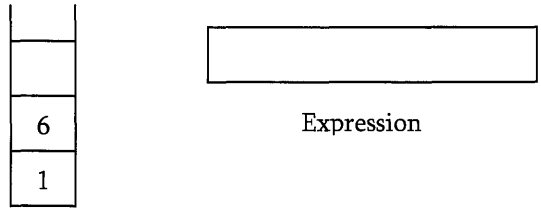
Example: Let us see how the above algorithm works using an example. Assume that the postfix string is 123*+5-. Initially the stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. They will be pushed into the stack in that order.



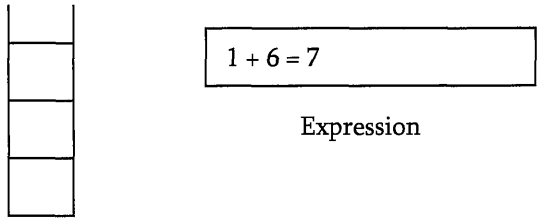
Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.



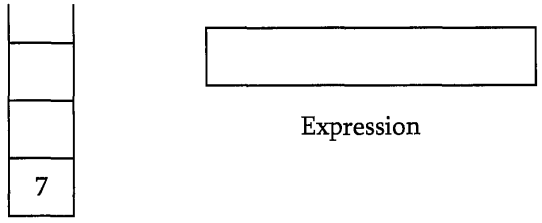
The value of the expression (2*3) that has been evaluated (6) is pushed into the stack.



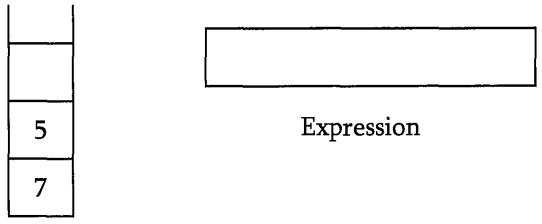
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



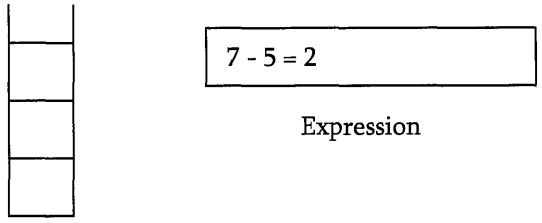
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



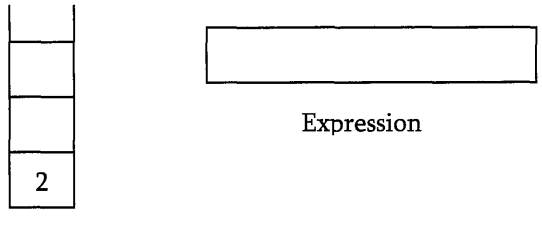
Next character scanned is "5", which is added to the stack.



Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-5) that has been evaluated(23) is pushed into the stack.



Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned. End result:

- Postfix String : 123*+5-
- Result : 2

Problem-4 Can we evaluate the infix expression with stacks in one pass?

Solution: Using 2 stacks we can evaluate an infix expression in 1 pass without converting to postfix.

Algorithm

1) Create an empty operator stack
2) Create an empty operand stack
3) For each token in the input string
    a. Get the next token in the infix string
    b. If next token is an operand, place it on the operand stack
    c. If next token is an operator
        i. Evaluate the operator (next op)
4) While operator stack is not empty, pop operator and operands (left and right), evaluate left operator right and push result onto operand stack
5) Pop result from operator stack

Problem-5 How to design a stack such that GetMinimum( ) should be O(1)?

Solution: Take an auxiliary stack which maintains the minimum of all values in the stack. Also, assume that, each element of the stack is less than its below elements. For simplicity let us call the auxiliary stack as *min stack*.

When we *pop* the main stack, *pop* the min stack too. When we push the main stack, push either the new element or the current minimum, whichever is lower. At any point, if we want to get the minimum then we just need to return the top element from the min stack. Let us take some example and trace out. Initially let us assume that we have pushed 2, 6, 4, 1 and 5. Based on above algorithm the *min stack* will look like:

Main stack	Min stack
5 → top	1 → top
1	1
4	2
6	2
2	2

After popping twice we get:

Main stack	Min stack
4 → top	2 → top
6	2
2	2

Based on the above discussion, now let us code the push, pop and GetMinimum() operations.

```
struct AdvancedStack{
        struct Stack elementStack;
        struct Stack minStack;
};
void Push(struct AdvancedStack *S, int data ){
        Push (S→elementStack, data);
        if(IsEmptyStack(S→minStack) || Top(S→minStack) >= data)
                Push (S→minStack, data);
        else    Push (S→minStack, Top(S→minStack));
}
int Pop(struct AdvancedStack *S ){
```

```
        int temp;
        if(IsEmptyStack(S→elementStack))
                return -1;
        temp = Pop (S→elementStack);
        Pop (S→minStack);
        return temp;
}
int GetMinimum(struct AdvancedStack *S){
        return Top(S→minStack);
}
struct AdvancedStack *CreateAdvancedStack(){
        struct AdvancedStack *S = (struct AdvancedStack *)malloc(sizeof(struct AdvancedStack));
        if(!S)
                return NULL;
        S→elementStack = CreateStack();
        S→minStack = CreateStack();
        return S;
}
```

Time complexity: O(1). Space complexity: O(n) [for Min stack]. This algorithm has much better space usage if we rarely get a "new minimum or equal".

Problem-6 For the Problem-5 is it possible to improve the space complexity?

Solution: Yes. The main problem of previous approach is, for each push operation we are pushing the element on to *min stack* also (either the new element or existing minimum element). That means, we are pushing the duplicate minimum elements on to the stack.

Now, let us change the above algorithm to improve the space complexity. We still have the min stack, but we only pop from it when the value we pop from the main stack is equal to the one on the min stack. We only *push* to the min stack when the value being pushed onto the main stack is less than *or equa* to the current min value. In this modified algorithm also, if we want to get the minimum then we just need to return the top element from the min stack. For example, taking the original version and pushing 1 again, we'd get:

Main stack	Min stack
1 → top	
5	
1	
4	1 → top
6	1
2	2

Popping from the above pops from both stacks because 1 == 1, leaving:

Main stack	Min stack
5 → top	
1	
4	
6	1 → top
2	2

Popping again *only* pops from the main stack, because 5 > 1:

Main stack	Min stack
1 → top	
4	
6	1 → top

2	2

Popping again pops both stacks because 1 == 1:

Main stack	Min stack
4 → top	
6	
2	2 → top

Note: The difference is only in push & pop operations.

```
struct AdvancedStack {
        struct Stack elementStack;
        struct Stack minStack;
};
void Push(struct AdvancedStack *S, int data){
        Push (S→elementStack, data);
        if(IsEmptyStack(S→minStack) || Top(S→minStack) >= data)
                Push (S→minStack, data);
}
int Pop(struct AdvancedStack *S ){
        int temp;
        if(IsEmptyStack(S→elementStack)) return -1;
        temp = Top (S→elementStack);
        if(Top(S→ minStack) == Pop(S→elementStack))
                Pop (S→ minStack);
        return temp;
}
int GetMinimum(struct AdvancedStack *S){
        return Top(S→minStack);
}
Struct AdvancedStack * AdvancedStack(){
        struct AdvancedStack *S = (struct AdvancedStack) malloc (sizeof (struct AdvancedStack));
        if(!S) return NULL;
        S→elementStack = CreateStack();
        S→minStack = CreateStack();
        return S;
}
```

Time complexity: O(1). Space complexity: O(n) [for Min stack]. But this algorithm has much better space usage if we rarely get a "new minimum or equal".

Problem-7 For a given array with n symbols how many stack permutations are possible?

Solution: The number of stack permutations with n symbols is represented by *Catalan number* and we will discuss this in *Dynamic Programming* chapter.

Problem-8 Given an array of characters formed with a's and b's. The string is marked with special character X which represents the middle of the list (for example: ababa...ababXbabab.....baaa). Check whether the string is palindrome or not?

Solution: This is one of the simplest algorithms. What we do is, start two indexes one at the beginning of the string and other at the ending of the string. Each time compare whether the values at both the indexes are same or not. If the values are not same then we say that the given string is a palindrome. If the values are same then increment the left index and decrement the right index. Continue this process until both the indexes meet at the middle (at X) or if the string is not palindrome.

```
int IsPalindrome(char *A){
        int i=0, j = strlen(A)-1;
        while(i < j && A[i] == A[j]) {
                i++;
                j--;
        }
        if(i < j ) {
                printf("Not a Palindrome");
                return 0;
        }
        else {  printf("Palindrome");
                return 1;
        }
}
```

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

Problem-9 For the Problem-8, if the input is in singly linked list then how do we check whether the list elements form a palindrome or not? (That means, moving backward is not possible).

Solution: Refer *Linked Lists* chapter.

Problem-10 Can we solve Problem-8 using stacks?

Solution: Yes.

Algorithm

- Traverse the list till we encounter X as input element.
- During the traversal push all the elements (until X) on to the stack.
- For the second half of the list, compare each elements content with top of the stack. If they are same then pop the stack and go to the next element in the input list.
- If they are not same then the given string is not a palindrome.
- Continue this process until the stack is empty or the string is not a palindrome.

```
int IsPalindrome(char *A){
        int i=0;
        struct Stack S= CreateStack();
        while(A[i] != 'X') {
                Push(S, A[i]);
                i++;
        }
        i++;
        while(A[i]) {
                if(IsEmptyStack(S) ||A[i] != Pop(S)) {
                        printf("Not a Palindrome");
                        return 0;
                }
                i++;
        }
        return IsEmptyStack(S);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n/2) \approx O(n)$.

Problem-11 Given a stack, how to reverse the elements of stack by using only stack operations (push & pop)?

Solution: Algorithm

- First pop all the elements of the stack till it becomes empty.
- For each upward step in recursion, insert the element at the bottom of stack.

```
void ReverseStack(struct Stack *S){
        int data;
        if(IsEmptyStack(S)) return;
        data = Pop(S);
        ReverseStack(S);
        InsertAtBottom(S, data);
}
void InsertAtBottom(struct Stack *S, int data){
        int temp;
        if(IsEmptyStack(S)) {
                Push(S, data);
                return;
        }
        temp = Pop(S);
        InsertAtBottom(S, data);
        Push(S, temp);
}
```

Time Complexity: $O(n^2)$. Space Complexity: $O(n)$, for recursive stack.

Problem-12 Show how to implement one queue efficiently using two stacks. Analyze the running time of the queue operations.

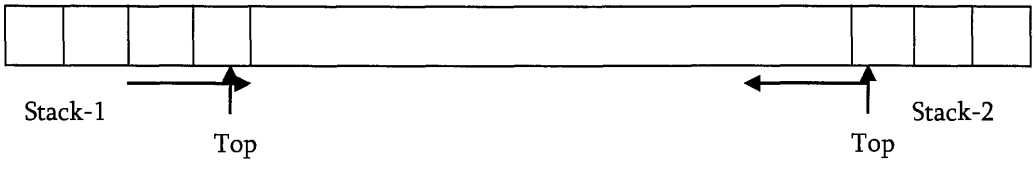
Solution: Refer *Queues* chapter.

Problem-13 Show how to implement one stack efficiently using two queues. Analyze the running time of the stack operations.

Solution: Refer *Queues* chapter.

Problem-14 How do we implement 2 stacks using only one array? Our stack routines should not indicate an exception unless every slot in the array is used?

Solution:



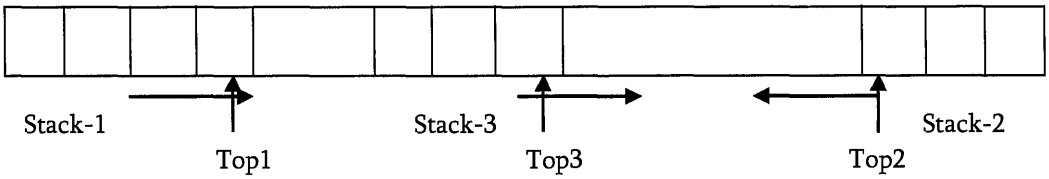
Algorithm:

- Start two indexes one at the left end and other at the right end.
- The left index simulates the first stack and the right index simulates the second stack.
- If we want to push an element into the first stack then put the element at left index.
- Similarly, if we want to push an element into the second stack then put the element at right index.
- First stack gets grows towards right, second stack grows towards left.

Time Complexity of push and pop for both stacks is O(1). Space Complexity is O(1).

Problem-15 3 stacks in one array: How to implement 3 stacks in one array?

Solution: For this problem, there could be other way of solving it. Below is one such possibility and it works as long as there is an empty space in the array.



To implement 3 stacks we keep the following information.

- The index of the first stack (Top1): this indicates the size of the first stack.
- The index of the second stack (Top2): this indicates the size of the second stack.
- Starting index of the third stack (base address of third stack).
- Top index of the third stack.

Now, let us define the push and pop operations for this implementation.

Pushing:

- For pushing on to the first stack, we need to see if adding a new element causes it to bump into the third stack. If so, try to shift the third stack upwards. Insert the new element at (start1 + Top1).
- For pushing to the second stack, we need to see if adding a new element causes it to bump into the third stack. If so, try to shift the third stack downward. Insert the new element at (start2 – Top2).
- When pushing to the third stack, see if it bumps the second stack. If so, try to shift the third stack downward and try pushing again. Insert the new element at (start3 + Top3).

Time Complexity: $O(n)$. Since, we may need to adjust the third stack. Space Complexity: O(1).

Popping: For popping, we don't need to shift, just decrement the size of the appropriate stack.

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

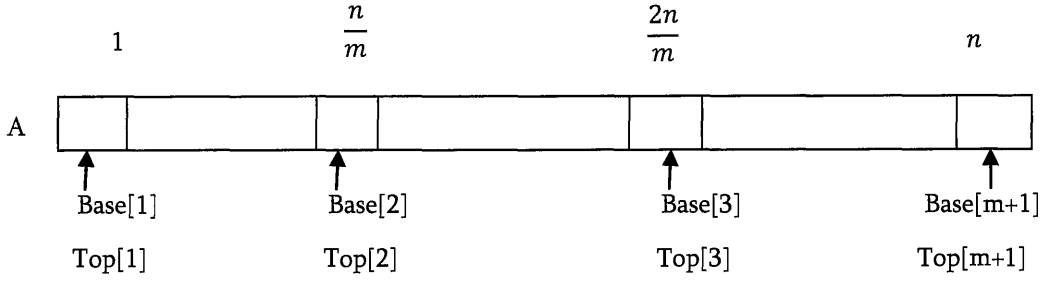
Problem-16 For Problem-15, is there any other way implementing middle stack?

Solution: Yes. When either the left stack (which grows to the right) or the right stack (which grows to the left) bumps into the middle stack, we need to shift the entire middle stack to make room. The same thing happens if a push on the middle stack causes it to bump into the right stack. To solve the above problem (number of shifts) what we can do is, alternating pushes could be added at alternating sides of the middle list (For example, even elements are pushed to the left, odd elements are pushed to the right).This would keep the middle stack balanced in the center of the array but it would still need to be shifted when it bumps into the left or right stack, whether by growing on its own or by the growth of a neighboring stack.

We can optimize the initial locations of the three stacks if they grow/shrink at different rates and if they have different average sizes. For example, suppose one stack doesn't change much. If you put it at the left then the middle stack will eventually get pushed against it and leave a gap between the middle and right stacks, which grow toward each other. If they collide, then it's likely you've run out of space in the array. There is no change in the time complexity but the average number of shifts will get reduced.

Problem-17 Multiple (m) stacks in one array: As similar to Problem-15, what if we want to implement m stacks in one array?

Solution: Let us assume that array indexes are from 1 to n. As similar to the discussion of Problem-15, to implement m stacks in one array, we divide the array into m parts (as shown below). The size of each part is $\frac{n}{m}$.



From the above representation we can see that, first stack is starting at index 1 (starting index is stored in Base[1]), second stack is starting at index $\frac{n}{m}$ (starting index is stored in Base[2]), third stack is starting at index $\frac{2n}{m}$ (starting index is stored in Base[3]) and so on. Similar to *Base* array, let us assume that *Top* array stores the top indexes for each of the stack. Consider the following terminology for the discussion.

- Top[i], for $1 \leq i \leq m$ will point to the topmost element of the stack i.
- If Base[i] == Top[i], then we can say the stack i is empty.
- If Top[i] == Base[i+1], then we can say the stack i is full.
  Initially Base[i] = Top[i] = $\frac{n}{m}(i-1)$, for $1 \leq i \leq m$.
- The i^{th}stack grows from Base[i]+1 to Base[i+1].

Pushing on to i^{th} stack:

1) For pushing on to the i^{th} stack, we check whether top of i^{th} stack is pointing to Base[i+1] (this case defines that i^{th} stack is full). That means, we need to see if adding a new element causes it to bump into the $i + 1^{th}$ stack. If so, try to shift the stacks from $i + 1^{th}$ stack to m^{th} stack towards right. Insert the new element at (Base[i] + Top[i]).
2) If right shifting is not possible then try shifting the stacks from 1 to $i - 1^{th}$ stack towards left.
3) If both of them are not possible then we can say that all stacks are full.

```
void Push(int StackID, int data) {
        if(Top[i] == Base[i+1])
                Print i^th Stack is full and does the necessary action (shifting);
        Top[i] = Top[i]+1;
        A[Top[i]] = data;
}
```

Time Complexity: O(n). Since, we may need to adjust the stacks. Space Complexity: O(1).

Popping from i^{th} stack: For popping, we don't need to shift, just decrement the size of the appropriate stack. The only case to check is stack empty case.

```
int Pop(int StackID) {
        if(Top[i] == Base[i])
                Print i^th Stack is empty;
        return  A[Top[i]--];
}
```

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

Problem-18 Consider an empty stack of integers. Let the numbers 1, 2, 3, 4, 5, 6 be pushed on to this stack only in the order they appeared from left to right. Let S indicates a push and X indicates a pop operation. Can they be permuted in to the order 325641(output) and order 154623? (If a permutation is possible give the order string of operations.

Solution: SSSXXSSXSXXX outputs 325641. 154623 cannot be output as 2 is pushed much before 3 so can appear only after 3 is output.

Problem-19 Earlier of this chapter, we have seen that, for dynamic array implementation of stack, we have used repeated doubling approach. For the same problem what is the complexity if we create a new array whose size is $n + K$ instead of doubling?

Solution: Let us assume that the initial stack size is 0. For simplicity let us assume that $K = 10$. For inserting the element we create a new array whose size is $0 + 10 = 10$. Similarly, after 10 elements we again create a new array whose size is $10 + 10 = 20$ and this process continues at values: 30, 40 ... That means, for a given n value, we are creating the new arrays at: $\frac{n}{10}, \frac{n}{20}, \frac{n}{30}, \frac{n}{40}$... The total number of copy operations are:

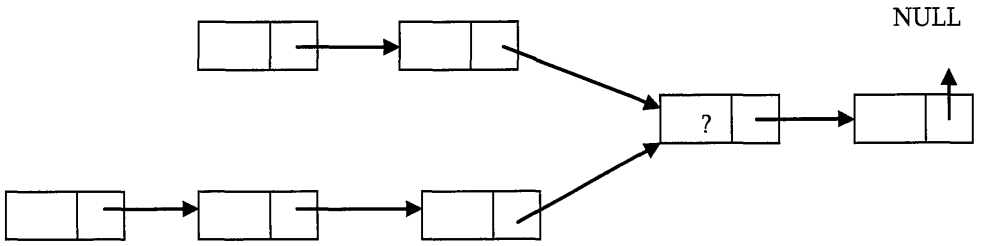
$$= \frac{n}{10} + \frac{n}{20} + \frac{n}{30} + \cdots 1 = \frac{n}{10}\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots \frac{1}{n}\right) = \frac{n}{10} logn \approx O(nlogn)$$

If we are performing n push operations, the cost of per operation is $O(logn)$.

Problem-20 Given a string containing $n\ S's$ and $n\ X's$ where S indicates a push operation and X indicates a pop operation, and with the stack initially empty, Formulate a rule to check whether a given string S of operations is admissible or not?

Solution: Given a string of length $2n$, we wish to check whether the given string of operations is permissible or not with respect to its functioning on a stack. The only restricted operation is pop whose prior requirement is that the stack should not be empty. So while traversing the string from left to right, prior to any pop the stack shouldn't be empty which means the no of $S's$ is always greater than or equal to that of $X's$. Hence the condition is at any stage on processing of the string, number of push operations (S) should be greater than number of pop operations (X).

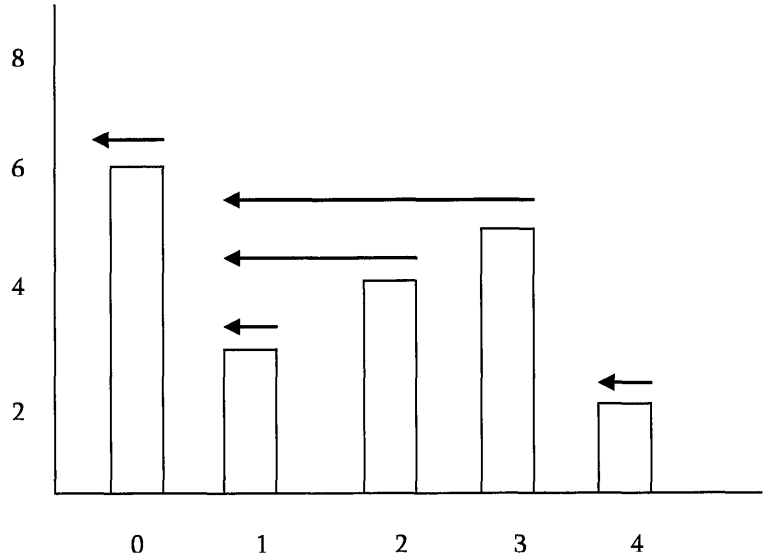
Problem-21 Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the list before they intersect are unknown and both list may have it different. $List1$ may have n nodes before it reaches intersection point and $List2$ might have m nodes before it reaches intersection point where m and n may be $m = n, m < n$ or $m > n$. Can we find the merging point using stacks?



Solution: Yes. For algorithm refer *Linked Lists* chapter.

Problem-22 **Finding Spans:** Given an array A the span $S[i]$ of $A[i]$ is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ and such that $A[j] \leq A[i]$?

Solution:



Day: Index i	Input Array A[i]	S[i]: Span of A[i]
0	6	1
1	3	1
2	4	2
3	5	3
4	2	1

This is a very common problem in stock markets to find the peaks. Spans have applications to financial analysis (E.g., stock at 52-week high). The span of a stocks price on a certain day, i, is the maximum number of consecutive days (up to the current day) the price of the stock has been less than or equal to its price on i. As an example, let us consider the following table and the corresponding spans diagram. In the figure the arrows indicates the length of the spans. Now, let us concentrate on the algorithm for finding the spans. One simple way is, each day, check how many contiguous days are with less stock price than current price.

```
Algorithm: FindingSpans(int A[],int n) {
                //Input: array A of n integers, Output: array S of spans of A
                int i, j, S[n]; //new array of n integers;
                for (i = 0; i < n; i++) {                          //Executes n times
                        j = 1;                                          n
                        while j <= i && A[i] > A[i-j]                   1 + 2 + ...+ (n – 1)
                                j = j + 1;                              1 + 2 + ...+ (n – 1)
                        S[i] = j;                                       n
                }
                return S;                                               1
        }
```

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

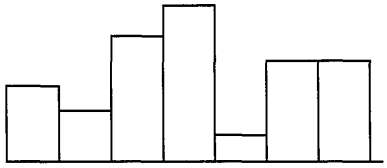
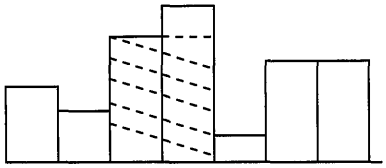
Problem-23 Can we improve the complexity of Problem-22?

Solution: From the above example, we can see that the span $S[i]$ on day i can be easily calculated if we know the closest day preceding i, such that the price is greater than on that day than the price on day i. Let us call such a day as P. If such a day exists then the span is now defined as $S[i] = i - P$.

```
Algorithm: FindingSpans(int A[], int n) {
                struct stack *D = CreateStack();
                int P;
                for (int i = 0 i< n; i++) {
                        while (!IsEmptyStack(D)) {
                                if(A[i] > A[Top(D)])
                                        Pop(D);
                        }
                        if(IsEmptyStack(D))
                                P = -1;
                        else    P = Top(D);
                        S[i] = i-P;
                        Push(D, i);
                }
                return S;
        }
```

Time Complexity: Each index of the array is pushed into the stack exactly one and also popped from the stack at most once. The statements in the while loop are executed at most n times. Even though the algorithm has nested loops, the complexity is $O(n)$ as the inner loop is executing only n times during the course of algorithm (trace out an example and see how many times the inner loop is becoming success). Space Complexity: $O(n)$ [for stack].

Problem-24 **Largest rectangle under histogram:** A histogram is a polygon composed of a sequence of rectangles aligned at a common base line. For simplicity, assume that the rectangles are having equal widths but may have different heights. For example, the figure on the left shows the histogram that consists of rectangles with the heights $3, 2, 5, 6, 1, 4, 4$, measured in units where 1 is the width of the rectangles. Here our problem is: given an array with heights of rectangles (assuming width is 1), we need to find the largest rectabgle possible. For the given example the largest rectangle is the shared part.

Solution: A straightforward answer is to go for each bar in the histogram and find the maximum possible area in histogram for it. Finally, find the maximum of these values. This will require $O(n^2)$.

Problem-25 For Problem-24, can we improve the time complexity?

Solution: Linear search using a stack of incomplete subproblems: There are many ways of solving this problem. *Judge* has given a nice algorithm for this problem which is based on stack. Process the elements in left-to-right order and maintain a stack of information about started but yet unfinished sub histograms.

If the stack is empty, open a new subproblem by pushing the element onto the stack. Otherwise compare it to the element on top of the stack. If the new one is greater we again push it. If the new one is equal we skip it. In all these cases, we continue with the next new element. If the new one is less, we finish the topmost subproblem by updating the maximum area with respect to the element at the top of the stack. Then, we discard the element at the top, and repeat the procedure keeping the current new element. This way, all subproblems are finished until the stack becomes empty, or its top element is less than or equal to the new element, leading to the actions described above. If all elements have been processed, and the stack is not yet empty, we finish the remaining subproblems by updating the maximum area with respect to to the elements at the top.

```
struct StackItem {
        int height;
        int index;
};
int MaxRectangleArea(int A[], int n) {
        int i, maxArea=-1, top = -1, left, currentArea;
        struct StackItem *S = (struct StackItem *) malloc(sizeof(struct StackItem) * n);
        for(i=0; i<=n; i++) {
                while(top >= 0 && (i==n || S[top]→data > A[i])) {
                        if(top > 0)
                                left = S[top-1]→index;
                        else    left = -1;
                        currentArea = (i – left-1) * S[top]→data;
                        --top;
                        if(currentArea > maxArea)
                                maxArea = currentArea;
                }
                if(i<n) { ++top;
                        S[top]→data = A[i];
                        S[top]→index = i;
                }
        }
        return maxArea;
}
```

In first impression, this solution seems to be having $O(n^2)$ complexity. But if we look carefully, every element is pushed and popped at most once and in every step of the function at least one element is pushed or popped. Since the amount of work for the decisions and the update is constant, the complexity of the algorithm is $O(n)$ by amortized analysis. Space Complexity: $O(n)$ [for stack].

QUEUES

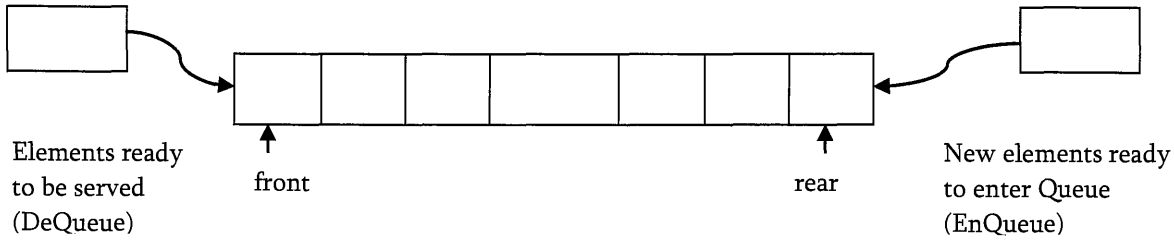
Chapter-5

5.1 What is a Queue?

A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which the data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

Definition: A *queue* is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called as First in First out (FIFO) or Last in Last out (LILO) list.

Similar to *Stacks*, special names are given to the two changes that can be made to a queue. When an element is inserted in a queue, the concept is called as *EnQueue*, and when an element is removed from the queue, the concept is called as *DeQueue*. Trying to *DeQueue* an empty queue is called as *underflow* and trying to *EnQueue* an element in a full queue is called as *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.



5.2 How is Queues Used?

Line at reservation counter explains the concept of a queue. When we enter the line we put ourselves at the end of the line and the person who is at the front of the line is the next who will be served. The person will exit the queue and will be served.

In the meanwhile the queue is served and next person at head of the line will exit the queue and will be served. While the queue is served, we will move towards the head of the line since each person that is served will be removed from the head of the queue. Finally we will reach head of the line and we will exit the queue and be served. This behavior is very useful in any cases where there is need to maintain the order of arrival.

5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in queue must follow the FIFO scheme. For simplicity we assume the elements are integers.

Main Queue Operations

- EnQueue(int data): Inserts an element at the end of the queue
- int DeQueue(): Removes and returns the element at the front of the queue

Auxiliary Queue Operations

- int Front(): Returns the element at the front without removing it

- int QueueSize(): Returns the number of elements stored
- int IsEmptyQueue(): Indicates whether no elements are stored

5.4 Exceptions

As similar to other ADTs attempting execution of *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and attempting execution of *EnQueue* on an full queue throws an "*Full Queue Exception*".

5.5 Applications

Following are the some of the applications in which queues are being used.

Direct Applications

- Operating systems schedule jobs (with equal priority) in the order of arrival (e.g., a print queue).
- Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario requires a queue.
- Multiprogramming.
- Asynchronous data transfer (file IO, pipes, sockets).
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

Indirect Applications

- Auxiliary data structure for algorithms
- Component of other data structures

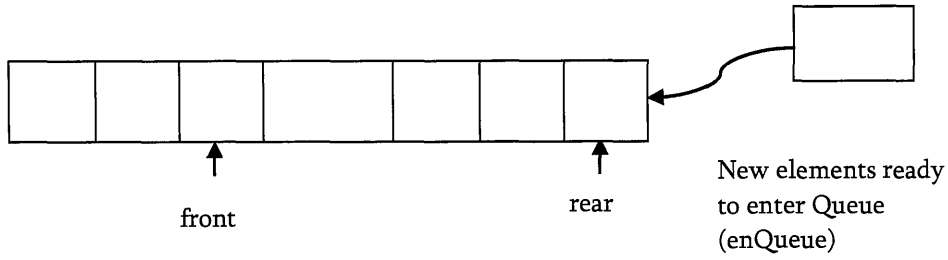
5.6 Implementation

There are many ways (similar to Stacks) of implementing queue operations and below are commonly used methods.

- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked lists implementation

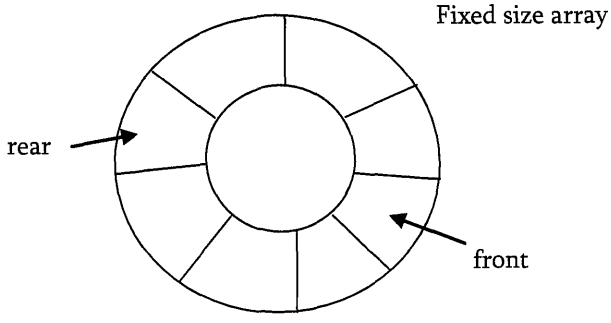
Why Circular Arrays?

First, let us see whether we can use simple arrays for implementing queues which we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at other end. After some insertions and deletions it is easy to get the situation as shown below. It can be seen clearly that, the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat last element and first array elements are contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.



Note: The simple circular array and dynamic circular array implementations are very much similar to stack array implementations. Refer *Stacks* chapter for analysis of these implementations.

Simple Circular Array Implementation



This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of start element and end element. Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue. The array storing the queue elements may become full. An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from empty queue then it will throw *empty queue exception*.

Note: Initially, both front and rear points to -1 which indicates that the queue is empty.

```
struct ArrayQueue {
        int front, rear;
        int capacity;
        int *array;
};
struct ArrayQueue *Queue(int size) {
        struct ArrayQueue *Q = malloc(sizeof(struct ArrayQueue));
        if(!Q) return NULL;
        Q→capacity = size;
        Q→front = Q→rear = -1;
        Q→array= malloc(Q→capacity * sizeof(int));
        if(!Q→array)
              return NULL;
        return Q;
}
int IsEmptyQueue(struct ArrayQueue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == -1);
}
int IsFullQueue(struct ArrayQueue *Q) {
        //if the condition is true then 1 is returned else 0 is returned
        return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
        return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct ArrayQueue *Q, int data) {
        if(IsFullQueue(Q))
                printf("Queue Overflow");
        else {  Q→rear = (Q→rear+1) % Q→capacity;
                Q→ array[Q→rear]= data;
                if(Q→front == -1)
```

```
                Q→front = Q→rear;
        }
}
int DeQueue(struct ArrayQueue *Q) {
        int data = 0;//or element which does not exist in Queue
        if(IsEmptyQueue(Q)) {
                printf("Queue is Empty");
                return 0;
        }
        else {  data = Q→array[Q→front];
                if(Q→front == Q→rear)
                        Q→front = Q→rear = -1;
                else    Q→front = (Q→front+1) % Q→capacity;
        }
        return data;
}
void DeleteQueue(struct ArrayQueue *Q) {
        if(Q) {
             if(Q→array)
                        free(Q→array);
             free(Q);
        }
}
```

Performance & Limitations

Performance: Let n be the number of elements in the queue:

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Limitations: The maximum size of the queue must be defined a prior and cannot be changed. Trying to *EnQueue* a new element into a full queue causes an implementation-specific exception.

Dynamic Circular Array Implementation

```
struct DynArrayQueue {
        int front, rear;
        int capacity;
        int *array;
};
struct DynArrayQueue *CreateDynQueue() {
        struct DynArrayQueue *Q = malloc(sizeof(struct DynArrayQueue));
        if(!Q) return NULL;
        Q→capacity = 1;
        Q→front = Q→rear = -1;
        Q→array= malloc(Q→capacity * sizeof(int));
        if(!Q→array)
```

```
                return NULL;
        return Q;
}
int IsEmptyQueue(struct DynArrayQueue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == -1);
}
int IsFullQueue(struct DynArrayQueue *Q) {
        //if the condition is true then 1 is returned else 0 is returned
        return ((Q→rear +1) % Q→capacity == Q→front);
}
int QueueSize() {
        return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
void EnQueue(struct DynArrayQueue *Q, int data) {
        if(IsFullQueue(Q))
                ResizeQueue(Q);
        Q→rear = (Q→rear+1)% Q→capacity;
        Q→ array[Q→rear]= data;
        if(Q→front == -1)
                Q→front = Q→rear;
}
void ResizeQueue(struct DynArrayQueue *Q) {
        int size = Q→capacity;
        Q→capacity = Q→capacity*2;
        Q→array = realloc (Q→array, Q→capacity);
        if(!Q→array) {
                printf("Memory Error");
                return;
        }
        if(Q→front > Q→rear ) {
                for(int i=0; i < Q→front; i++) {
                        Q→array[i+size] =Q→array[i];
                }
                Q→rear = Q→rear + size;
        }
}
int DeQueue(struct DynArrayQueue *Q) {
        int data = 0;//or element which does not exist in Queue
        if(IsEmptyQueue(Q)) {
                printf("Queue is Empty");
                return 0;
        }
        else {  data = Q→array[Q→front];
                if(Q→front== Q→rear)
                        Q→front= Q→rear = -1;
                else
                        Q→front = (Q→front+1) % Q→capacity;
        }
        return data;
```

```
}
void DeleteQueue(struct DynArrayQueue *Q) {
        if(Q) {
                if(Q→array)
                        free(Q→array);
                free(Q→array);
        }
}
```

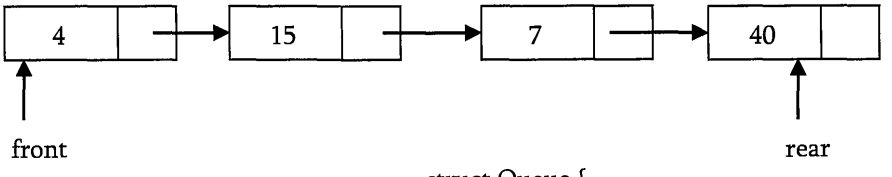
Performance

Let n be the number of elements in the queue.

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of IsFullQueue()	O(1)
Time Complexity of QueueSize()	O(1)
Time Complexity of DeleteQueue()	O(1)

Linked List Implementation

The other way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting element at the ending of the list. *DeQueue* operation is implemented by deleting an element from the beginning of the list.




```
struct ListNode {
   int data;
   struct ListNode *next;
};
```

```
struct Queue {
   struct ListNode *front;
   struct ListNode *rear;
};
```

```
struct Queue *CreateQueue() {
        struct Queue *Q;
        struct ListNode *temp;
        Q = malloc(sizeof(struct Queue));
        if(!Q) return NULL;
        temp = malloc(sizeof(struct ListNode));
        Q→front = Q→rear = NULL;
        return Q;
}
int IsEmptyQueue(struct Queue *Q) {
        // if the condition is true then 1 is returned else 0 is returned
        return (Q→front == NULL);
}
void EnQueue(struct Queue *Q, int data) {
        struct ListNode *newNode;
        newNode = malloc(sizeof(struct ListNode));
```

```
        if(!newNode)
                return NULL;
        newNode→data = data;
        newNode→next = NULL;
        Q→rear→next = newNode;
        Q→rear = newNode;
        if(Q→front == NULL)
                Q→front = Q→rear;
}
int DeQueue(struct Queue *Q) {
        int data = 0;     //or element which does not exist in Queue
        struct ListNode *temp;
        if(IsEmptyQueue(Q)) {
                printf("Queue is empty");
                return 0;
        }
        else {    temp = Q→front;
                data = Q→front→data;
                Q→front== Q→front→next;
                free(temp);
        }
        return data;
}
void DeleteQueue(struct Queue *Q) {
        struct ListNode *temp;
        while(Q) {
                temp = Q;
                Q = Q→next;
                free(temp);
        }
        free(Q);
}
```

Performance

Let n be the number of elements in the queue, then

Space Complexity (for n EnQueue operations)	O(n)
Time Complexity of EnQueue()	O(1) (Average)
Time Complexity of DeQueue()	O(1)
Time Complexity of IsEmptyQueue()	O(1)
Time Complexity of DeleteQueue()	O(1)

Comparison of Implementations

Note: Comparison is very much similar to stack implementations and *Stacks* chapter.

5.7 Problems on Queues

Problem-1 Give an algorithm for reversing a queue Q. To access the queue, we are only allowed to use the methods of queue ADT.

Solution:

```
void ReverseQueue(struct Queue *Q) {
        struct Stack *S = CreateStack();
        while (!IsEmptyQueue(Q))
                Push(S, DeQueue(Q))
        while (!IsEmptyStack(S))
                EnQueue(Q, Pop(S));
}
```

Time Complexity: $O(n)$.

Problem-2 How to implement a queue using two stacks?

Solution: Let S1 and S2 be the two stacks to be used in the implementation of queue. All we have to do is to define the EnQueue and DeQueue operations for the queue.

```
struct Queue {
        struct Stack *S1; // for EnQueue
        struct Stack *S2; // for DeQueue
}
```

EnQueue Algorithm

- Just push on to stack S1

```
void EnQueue(struct Queue *Q, int data) {
        Push(Q→S1, data);
}
```

Time Complexity: O(1).

DeQueue Algorithm

- If stack S2 is not empty then pop from S2 and return that element.
- If stack is empty, then transfer all elements from S1 to S2 and pop the top element from S2 and return that popped element [we can optimize the code little by transferring only $n-1$ elements from S1 to S2 and pop the n^{th} element from S1 and return that popped element].
- If stack S1 is also empty then throw error.

```
int DeQueue(struct Queue *Q) {
        if(!IsEmptyStack(Q→S2))
                return Pop(Q→S2);
        else {  while(!IsEmptyStack(Q→S1))
                        Push(Q→S2, Pop(Q→S1));
                return Pop(Q→S2);
        }
}
```

Time Complexity: From the algorithm, if the stack S2 is not empty then the complexity is O(1). If the stack S2 is empty then, we need to transfer the elements from S1 to S2. But if we carefully observe, the number of transferred elements and the number of popped elements from S2 are equal. Due to this the average complexity of pop operation in this case is O(1). Amortized complexity of pop operation is O(1).

Problem-3 Show how to efficiently implement one stack using two queues. Analyze the running time of the stack operations.

Solution: Let Q1 and Q2 be the two queues to be used in the implementation of stack. All we have to do is to define the push and pop operations for the stack.

```
struct Stack {
        struct Queue *Q1;
```

```
        struct Queue *Q2;
}
```

In below algorithms, we make sure that one queue is empty always.

Push Operation Algorithm: Whichever is queue is not empty, push the element into it.

- Check whether queue Q1 is empty or not. If Q1 is empty then Enqueue the element into Q2.
- Otherwise EnQueue the element into Q1.

```
Push(struct Stack *S, int data) {
        if(IsEmptyQueue(S→Q1))
                EnQueue(S→Q2, data);
        else    EnQueue(S→Q1, data);
}
```

Time Complexity: O(1).

Pop Operation Algorithm: Transfer $n-1$ elements to other queue and delete last from queue for performing pop operation.

- If queue Q1 is not empty then transfer $n-1$ elements from Q1 to Q2 and then, DeQueue the last element of Q1 and return it.
- If queue Q2 is not empty then transfer $n-1$ elements from Q2 to Q1 and then, DeQueue the last element of Q2 and return it.

```
int Pop(struct Stack *S) {
        int i, size;
        if(IsEmptyQueue(S→Q2)) {
                size = size(S→Q1);
                i = 0;
                while(i < size-1) {
                        EnQueue(S→Q2, DeQueue(S→Q1));
                        i++;
                }
                return DeQueue(S→Q1);
        }
        else {  size =  size(S→Q2);
                while(i < size-1) {
                        EnQueue(S→Q1, DeQueue(S→Q2));
                        i++;
                }
                return DeQueue(S→Q2);
        }
}
```

Time Complexity: Running time of pop operation is $O(n)$ as each time pop is called, we are transferring all the elements from one queue to other.

Problem-4 **Maximum sum in sliding window:** Given array A[] with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5

1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width w. **Output**: An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]. **Requirement**: Find a good optimal way to get B[i]

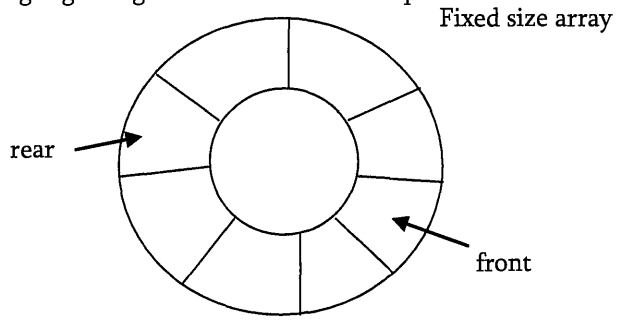
Solution: This problem can be solved with doubly ended queue (which support insertion and deletions at both ends). Refer *Priority Queues* chapter for algorithms.

Problem-5 Given a queue Q containing n elements, transfer these items on to a stack S (initially empty) so that front element of Q appears at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue and push and pop operations for the stack, outline an efficient O(n) algorithm to accomplish the above task, using only a constant amount of additional storage.

Solution: Assume the elements of queue Q are $a_1, a_2 \ldots a_n$. Dequeuing all elements and pushing them onto the stack will result in a stack with an at the top and a_1 at the bottom. This is done in O(n) time as dequeue and push each require constant time per operation. The queue is now empty. By popping all elements and pushing them on the the queue we will get a_1 at the top of the stack. This is done again in O(n) time. As in big-oh arithmetic we can ignore constant factors, the process is carried out in O(n) time. The amount of additional storage needed here has to be big enough to temporarily hold one item.

Problem-6 A queue is set up in a circular array A[0..n - 1] with front and rear defined as usual. Assume that $n - 1$ locations in the array are available for storing the elements (with the other element being used to detect full/empty condition). Give a formula for the number of elements in the queue in terms of $rear$, $front$, and n.

Solution: Consider the following fugre to get clear idea about the queue.



- Rear of the queue is somewhere clockwise from the front
- To enqueue an element, we move rear one position clockwise and write the element in that position
- To dequeue, we simply move front one position clockwise
- Queue migrates in a clockwise direction as we enqueue and dequeue
- Emptiness and fullness to be checked carefully.
- Analyze the possible situations (make some drawings to see where $front$ and $rear$ are when the queue is empty, and partially and totally filled). We will get this:

$$Number\ Of\ Elements = \begin{cases} rear - front + 1 & \text{if rear} == \text{front} \\ rear - front + n & \text{otherwise} \end{cases}$$

TREES

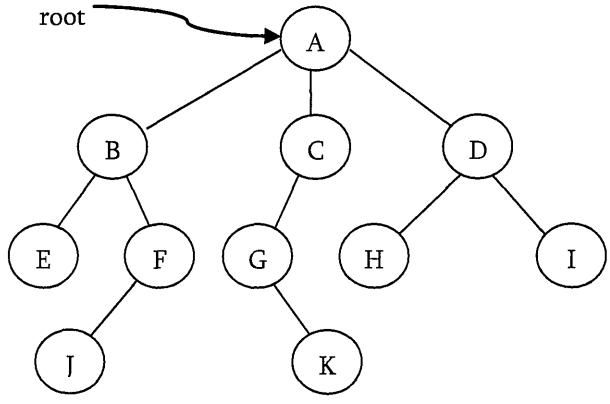
Chapter-6

6.1 What is a Tree?

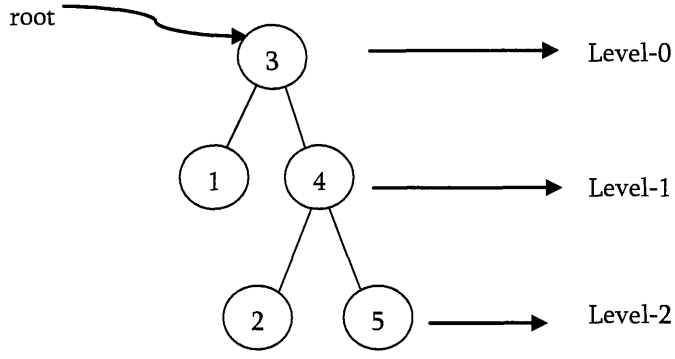
A *tree* is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes. Tree is an example of non-linear data structures. A *tree* structure is a way of representing the hierarchical nature of a structure in a graphical form.

In trees ADT (Abstract Data Type), order of the elements is not important. If we need ordering information linear data structures like linked lists, stacks, queues, etc. can be used.

6.2 Glossary

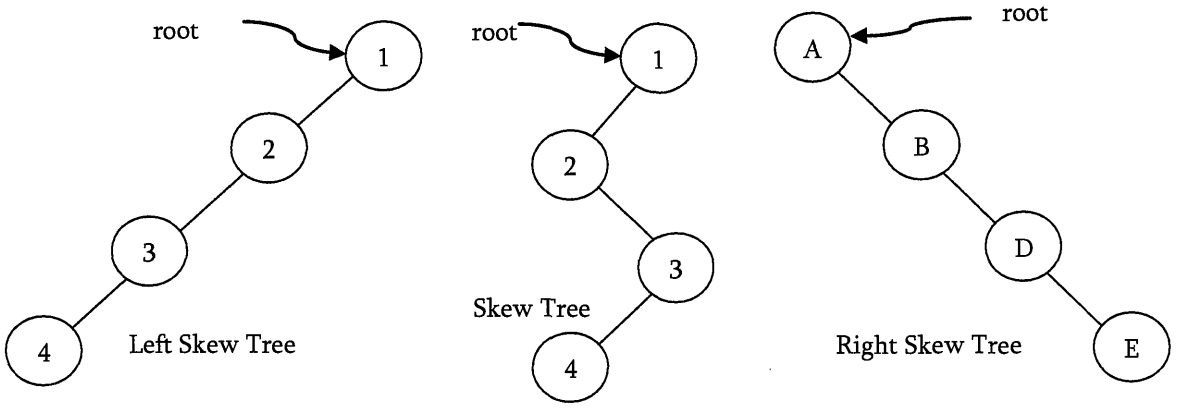


- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).
- An *edge* refers to the link from parent to child (all links in the figure).
- A node with no children is called *leaf* node (E, J, K, H and I).
- Children of same parent are called *siblings* (B, C, D are siblings of A and E, F are the siblings of B).
- A node p is an *ancestor* of a node q if there exists a path from root to q and p appears on the path. The node q is called a *descendant* of p. For example, A, C and G are the ancestors for K.
- Set of all nodes at a given depth is called *level* of the tree (B, C and D are same level). The root node is at level zero.



- The *depth* of a node is the length of the path from the root to the node (depth of G is 2, $A - C - G$).

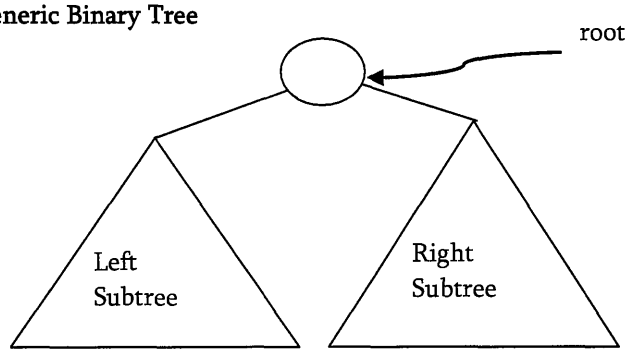
- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, height of B is 2 ($B - F - J$).
- *Height of the tree* is the maximum height among all the nodes in the tree and *depth of the tree* is the maximum depth among all the nodes in the tree. For a given tree depth and height returns the same value. But for individual nodes we may get different results.
- Size of a node is the number of descendants it has including itself (size of the subtree C is 3).
- If every node in a tree has only one child (except leaf nodes) then we call such trees as *skew trees*. If every node has only left child then we call them as *left skew trees*. Similarly, if every node has only right child then we call them as *right skew trees*.



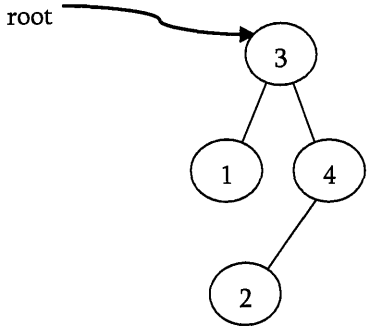
6.3 Binary Trees

A tree is called *binary tree* if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

Generic Binary Tree

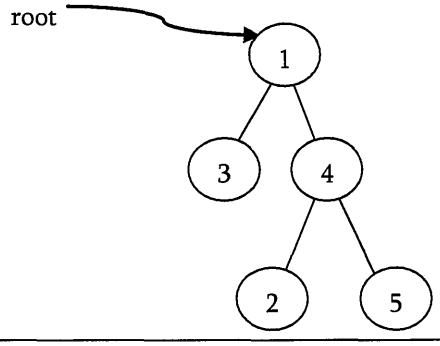


Example

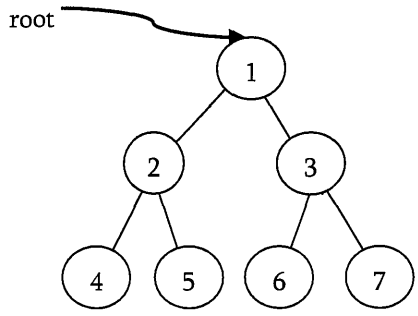


6.4 Types of Binary Trees

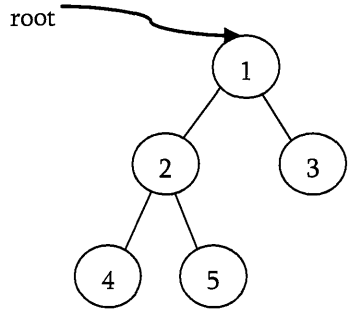
Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



Full Binary Tree: A binary tree is called *full binary tree* if each node has exactly two children and all leaf nodes are at same level.

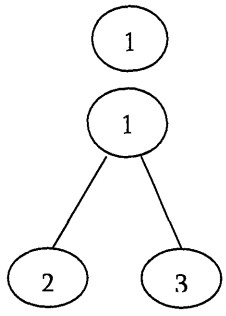
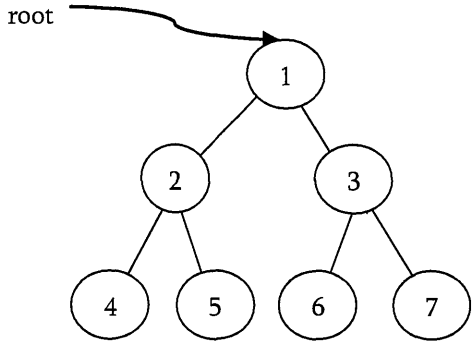


Complete Binary Tree: Before defining the *complete binary tree*, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at root (let us say the root node has 1) then we get a complete sequence from 1 to number of nodes in the tree. While traversing we should give numbering for NULL pointers also. A binary tree is called complete binary tree if all leaf nodes are at height h or $h - 1$ and also without any missing number in the sequence.



6.5 Properties of Binary Trees

For the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height zero.

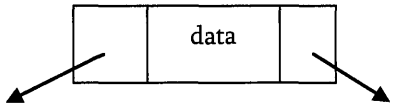
	Height	Number of nodes at level h
	$h = 0$	$2^0 = 1$
	$h = 1$	$2^1 = 2$
	$h = 2$	$2^2 = 4$

From the diagram we can infer the following properties:

- The number of nodes n in a full binary tree is $2^{h+1} - 1$. Since, there are h levels we need to add all nodes at each level $[2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1]$.
- The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum). For more information on this, refer *Priority Queues* chapter.
- The number of leaf nodes in a full binary tree are 2^h.
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes are $n + 1$.

Structure of Binary Trees

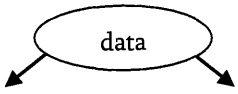
Now let us define structure of the binary tree. For simplicity, assume that the data of the nodes are integers. One way to represent a node (which contains the data) is to have two links which points to left and right children along with data fields as shown below:

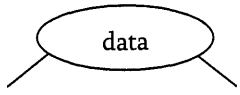


Or


```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *left;
   struct BinaryTreeNode *right;
};
```

Note: In trees, the default flow is from parent to children and showing directed branches is not compulsory. For our discussion, we assume both the below representations are same.





Operations on Binary Trees

Basic Operations

- Inserting an element in to a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding size of the tree
- Finding the height of the tree
- Finding the level which has maximum sum
- Finding least common ancestor (LCA) for a given pair of nodes and many more.

Applications of Binary Trees

Following are the some of the applications where *binary trees* play important role:

- Expression trees are used in compilers.
- Huffman coding trees which are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in $O(logn)$ (average).
- Priority Queues (PQ), which supports search and deletion of minimum(or maximum) on a collection of items in logarithmic time (in worst case),

6.6 Binary Tree Traversals

In order to process trees, we need a mechanism for traversing them and that forms the subject of this section. The process of visiting all nodes of a tree is called *tree traversal*. Each of the nodes is processed only once but they may be visited more than once. As we have already seen that in linear data structures (like linked lists, stacks, queues, etc...), the elements are visited in sequential order. But, in tree structures there are many different ways.

Tree traversal is like searching the tree except that in traversal the goal is to move through the tree in some particular order. In addition, all nodes are processed in the traversal but searching stops when the required node is found.

Traversal Possibilities

Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node and denotes with "*D*"), traversing to the left child node (denotes with "*L*"), and traversing to the right child node (denotes with "*R*"). This process can be easily described through recursion. Based on the above definition there are 6 possibilities:

1. *LDR*: Process left subtree, process the current node data and then process right subtree
2. *LRD*: Process left subtree, process right subtree and then process the current node data
3. *DLR*: Process the current node data, process left subtree and then process right subtree
4. *DRL*: Process the current node data, process right subtree and then process left subtree
5. *RDL*: Process right subtree, process the current node data and then process left subtree
6. *RLD*: Process right subtree, process left subtree and then process the current node data

Classifying the Traversals

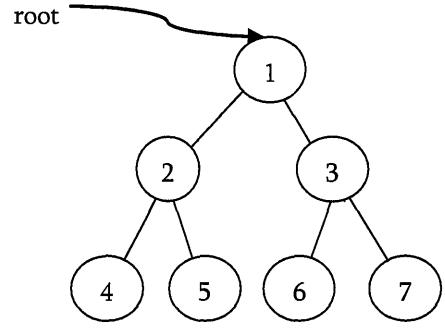
The sequence in which these entities processed defines a particular traversal method. The classification based on the order in which current node is processed. That means, if we are classifying based on current node (*D*) and if *D* comes in the middle then it does not matter whether *L* on left side of *D* or *R* is on left side of *D*. Similarly, it does not matter whether *L* is on right side of *D* or *R* is on right side of *D*. Due to this, the total 6 possibilities were reduced to 3 and they are:

- Preorder (*DLR*) Traversal
- Inorder (*LDR*) Traversal
- Postorder (*LRD*) Traversal

There is another traversal method which does not depend on above orders and it is:

- Level Order Traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

Let us use the below diagram for remaining discussion.



PreOrder Traversal

In pre-order traversal, each node is processed before (pre) either of its sub-trees. This is the simplest traversal to understand. However, even though each node is processed before the subtrees, it still requires that some information

must be maintained while moving down the tree. In the example above, the 1 is processed first, then the left sub-tree followed by the right subtree. Therefore, processing must return to the right sub-tree after finishing the processing of the left subtree. To move to right subtree after processing left subtree, we must maintain the root information. The obvious ADT for such information is a stack. Because of its LIFO structure, it is possible to get the information about the right subtrees back in the reverse order.

Preorder traversal is defined as follows:

- Visit the root.
- Traverse the left subtree in Preorder.
- Traverse the right subtree in Preorder.

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

```
void PreOrder(struct BinaryTreeNode *root){
        if(root) {
                printf("%d",root→data);
                PreOrder(root→left);
                PreOrder (root→right);
        }
}
```

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

Non-Recursive Preorder Traversal

In recursive version a stack is required as we need to remember the current node so that after completing the left subtree we can go to right subtree. To simulate the same, first we process the current node and before going to left subtree, we store the current node on stack. After completing the left subtree processing, *pop* the element and go to its right subtree. Continue this process until stack is nonempty.

```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = CreateStack();
        while(1) {
                while(root) {
                        //Process current node
                        printf("%d",root→data);
                        Push(S,root);
                        //If left subtree exists, add to stack
                        root = root→left;
                }
                if(IsEmptyStack(S))
                        break;
                root = Pop(S);
                //Indicates completion of left subtree and current node, now go to right subtree
                root = root→right;
        }
        DeleteStack(S);
}
```

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

InOrder Traversal

In Inorder traversal the root is visited between the subtrees. Inorder traversal is defined as follows:

- Traverse the left subtree in Inorder.
- Visit the root.

- Traverse the right subtree in Inorder.

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

```
void InOrder(struct BinaryTreeNode *root){
        if(root) {
                InOrder(root→left);
                printf("%d",root→data);
                InOrder(root→right);
        }
}
```

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

Non-Recursive Inorder Traversal

Non-recursive version of Inorder traversal is very much similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which indicates after completion of left subtree processing).

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = CreateStack();
        while(1) {
                while(root) {
                        Push(S,root);
                        //Got left subtree and keep on adding to stack
                        root = root→left;
                }
                if(IsEmptyStack(S))
                        break;
                root = Pop(S);
                printf("%d", root→data);          //After popping, process the current node
                //Indicates completion of left subtree and current node, now go to right subtree
                root = root→right;
        }
        DeleteStack(S);
}
```

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

PostOrder Traversal

In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as follows:

- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.

The nodes of tree would be visited in the order: 4 5 2 6 7 3 1

```
void PostOrder(struct BinaryTreeNode *root){
        if(root) {
                PostOrder(root→left);
                PostOrder(root→right);
                printf("%d",root→data);
        }
}
```

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

Non-Recursive Postorder Traversal

In preorder and inorder traversals, after poping the stack element we do not need to visit the same vertex again. But in postorder traversal, each node is visited twice. That means, after processing left subtree we will be visiting the current node and also after processing the right subtree we will be visiting the same current node. But we should be processing the node during the second visit. Here the problem is how to differentiate whether we are returning from left subtree or right subtree?

Trick for this problem is: after popping an element from stack, check whether that element and right of top of the stack are same or not. If they are same then we are done with processing of left subtree and right subtree. In this case we just need to pop the stack one more time and print its data.

```
void PostOrderNonRecursive(struct BinaryTreeNode *root){
        struct Stack *S = CreateStack();
        while (1) {
                if (root) {
                        Push(S,root);
                        root=root→left;
                }
                else {  if(IsEmptyStack(S)) {
                                printf("Stack is Empty");
                                return;
                        }
                        else    if(Top(S)→right == NULL) {
                                        root = Pop(S);
                                        printf("%d",root→data);
                                        if(root == Top(S)→right) {
                                                printf("%d",Top(S)→data);
                                                Pop(S);
                                        }
                                }
                        if(!IsEmptyStack(S))
                                root=Top(S)→right;
                        else    root=NULL;
                }
        }
        DeleteStack(S);
}
```

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

Level Order Traversal

Level order traversal is defined as follows:

- Visit the root.
- While traversing level l, keep all the elements at level $l + 1$ in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

The nodes of tree would be visited in the order: 1 2 3 4 5 6 7

```
void LevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
```

```
        struct Queue *Q = CreateQueue();
        if(!root)
                return;
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                //Process current node
                printf("%d", temp→data);
                if(temp→left)
                        EnQueue(Q, temp→left);
                if(temp→right)
                        EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$. Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Problems on Binary Trees

Problem-1 Give an algorithm for finding maximum element in binary tree.

Solution: One simple way of solving this problem is: find the maximum element in left subtree, find maximum element in right sub tree, compare them with root data and select the one which is giving the maximum value. This approach can be easily implemented with recursion.

```
int FindMax(struct BinaryTreeNode  *root) {
        int root_val, left, right, max = INT_MIN;
        if(root !=NULL) {
                root_val = root→data;
                left = FindMax(root→left);
                right = FindMax(root→right);

                // Find the largest of the three values.
                if(left > right)
                        max = left;
                else    max = right;
                if(root_val > max)
                        max = root_val;
        }
        return max;
}
```

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

Problem-2 Give an algorithm for finding maximum element in binary tree without recursion.

Solution: Using level order traversal: just observe the elements data while deleting.

```
int FindMaxUsingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        int max = INT_MIN;
        struct Queue *Q = CreateQueue();
        EnQueue(Q,root);
```

```
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                // largest of the three values
                if(max < temp→data)
                        max = temp→data;
                if(temp→left)
                        EnQueue (Q, temp→left);
                if(temp→right)
                        EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return max;
}
```

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

Problem-3 Give an algorithm for searching an element in binary tree.

Solution: Given a binary tree, return true if a node with the data is found in the tree. Recurse down the tree, choose the left or right branch by comparing the data with each nodes data.

```
int FindInBinaryTreeUsingRecursion(struct BinaryTreeNode *root, int data) {
        int temp;
        // Base case == empty tree, in that case, the data is not found so return false
        if(root == NULL)
                return 0;
        else {  //see if found here
                if(data == root→data)
                        return 1;
                else {  // otherwise recur down the correct subtree
                        temp = FindInBinaryTreeUsingRecursion (root→left, data)
                        if(temp != 0)
                                return temp;
                        else    return(FindInBinaryTreeUsingRecursion(root→right, data));
                }
        }
        return 0;
}
```

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

Problem-4 Give an algorithm for searching an element in binary tree without recursion.

Solution: We can use level order traversal for solving this problem. The only change required in level order traversal is, instead of printing the data we just need to check whether the root data is equal to the element we want to search.

```
int SearchUsingLevelOrder(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        if(!root)
                return -1;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
```

```
            //see if found here
            if(data == root→data)
                    return 1;
            if(temp→left)
                    EnQueue (Q, temp→left);
            if(temp→right)
                    EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return 0;
}
```

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

Problem-5 Give an algorithm for inserting an element into binary tree.

Solution: Since the given tree is a binary tree, we can insert the element wherever we want. To insert an element, we can use the level order traversal and insert the element wherever we found the node whose left or right child is NULL.

```
void InsertInBinaryTree(struct BinaryTreeNode *root, int data){
        struct Queue *Q;
        struct BinaryTreeNode *temp;
        struct BinaryTreeNode *newNode;
        newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
        newNode→left = newNode→right = NULL;
        if(!newNode) {
                printf("Memory Error");
                return;
        }
        if(!root) {
                root = newNode;
                return;
        }
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→left)
                        EnQueue(Q, temp→left);
                else {  temp→left=newNode;
                        DeleteQueue(Q);
                        return;
                }
                if(temp→right)
                        EnQueue(Q, temp→right);
                else {  temp→right=newNode;
                        DeleteQueue(Q);
                        return;
                }
        }
        DeleteQueue(Q);
}
```

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

Problem-6 Give an algorithm for finding the size of binary tree.

Solution: Calculate the size of left and right subtrees recursively, add 1 (current node) and return to its parent.

```
// Compute the number of nodes in a tree.
int SizeOfBinaryTree(struct BinaryTreeNode *root) {
        if(root==NULL)
                return 0;
        else    return(SizeOfBinaryTree(root→left) + 1 + SizeOfBinaryTree(root→right));
}
```

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

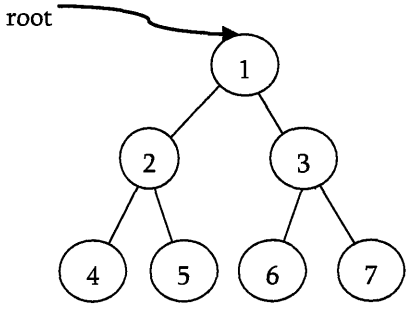
Problem-7 Can we solve the Problem-6 without recursion?

Solution: Yes, using level order traversal.

```
int SizeofBTUsingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        int count = 0;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                count++;
                if(temp→left)
                        EnQueue (Q, temp→left);
                if(temp→right)
                        EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return count;
}
```

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

Problem-8 Give an algorithm for printing the level order data in reserve order. For example, the output for the below tree should be: 4 5 6 7 2 3 1



Solution:

```
void LevelOrderTraversalInReverse(struct BinaryTreeNode *root){
        struct Queue *Q;
        struct Stack *s = CreateStack();
        struct BinaryTreeNode *temp;
        if(!root) return;
        Q = CreateQueue();
        EnQueue(Q, root);
```

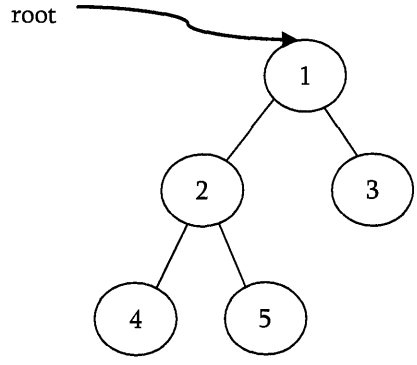
```
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→right)
                        EnQueue(Q, temp→right);
                if(temp→left)
                        EnQueue (Q, temp→left);
                Push(s, temp);
        }
        while(!IsEmptyStack(s))
                printf("%d",Pop(s)→data);
}
```

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

Problem-9 Give an algorithm for deleting the tree.

Solution: To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use Inorder, Preorder, Postorder or Level order Traversal?

Before deleting the parent node we should delete its children nodes first. We can use postorder traversal as it does the work without storing anything. We can delete tree with other traversals also with extra space complexity. For the following tree nodes are deleted in order – 4, 5, 2, 3, 1.




```
void DeleteBinaryTree(struct BinaryTreeNode *root){
        if(root == NULL)
                return;
        /* first delete both subtrees */
        DeleteBinaryTree(root→left);
        DeleteBinaryTree(root→right);
        //Delete current node only after deleting subtrees
        free(root);
}
```

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

Problem-10 Give an algorithm for finding the height (or depth) of the binary tree.

Solution: Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. This is similar to *PreOrder* tree traversal (and *DFS* of Graph algorithms).

```
int HeightOfBinaryTree(struct BinaryTreeNode  *root){
        int leftheight, rightheight;
        if(root == NULL)
                return 0;
        else {    /* compute the depth of each subtree */
```

```
                leftheight = HeightOfBinaryTree(root→left);
                rightheight = HeightOfBinaryTree(root→right);
                if(leftheight > rightheight)
                        return(leftheight + 1);
                else    return(rightheight + 1);
        }
}
```

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

Problem-11 Can we solve the Problem-10 without recursion?

Solution: Yes. Using level order traversal. This is similar to *BFS* of Graph algorithms. End of level is identified with NULL.

```
int FindHeightofBinaryTree(struct BinaryTreeNode *root){
        int level=1;
        struct Queue *Q;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        // End of first level
        EnQueue(Q,NULL);
        while(!IsEmptyQueue(Q)) {
                root=DeQueue(Q);
                // Completion of current level.
                if(root==NULL) {
                        //Put another marker for next level.
                        if(!IsEmptyQueue(Q))
                                EnQueue(Q,NULL);
                        level++;
                }
                else {  if(root→left)
                                EnQueue(Q, root→left);
                        if(root→right)
                                EnQueue(Q, root→right);
                }
        }
        return level;
}
```

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

Problem-12 Give an algorithm for finding the deepest node of the binary tree.

Solution:

```
struct BinaryTreeNode *DeepestNodeinBinaryTree(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        if(!root) return NULL;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                if(temp→left)
```

```
                EnQueue(Q, temp→left);
            if(temp→right)
                EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
        return temp;
}
```

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

Problem-13 Give an algorithm for deleting an element from binary tree.

Solution: The deletion of a node in binary tree can be implemented as

- Find the node which we want to delete.
- Find the deepest node in the tree.
- Replace the deepest nodes data with node to be deleted.
- Then delete the deepest node.

Problem-14 Give an algorithm for finding the number of leaves in the binary tree without using recursion.

Solution: The set of all nodes whose both left and either right are NULL are called leaf nodes.

```
void NumberOfLeavesInBTusingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        int count = 0;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            if(!temp→left && !temp→right)
                count++;
            else {  if(temp→left)
                        EnQueue(Q, temp→left);
                    if(temp→right)
                        EnQueue(Q, temp→right);
            }
        }
        DeleteQueue(Q);
        return count;
}
```

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

Problem-15 Give an algorithm for finding the number of full nodes in the binary tree without using recursion.

Solution: The set of all nodes with both left and right children are called full nodes.

```
void NumberOfFullNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        int count = 0;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
```

```
            temp = DeQueue(Q);
            if(temp→left && temp→right)
                count++;
            if(temp→left)
                EnQueue (Q, temp→left);
            if(temp→right)
                EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return count;
}
```

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

Problem-16 Give an algorithm for finding the number of half nodes (nodes with only one child) in the binary tree without using recursion.

Solution: The set of all nodes with either left or either right child (but not both) are called half nodes.

```
void NumberOfHalfNodesInBTusingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        int count = 0;
        if(!root)
            return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
            temp = DeQueue(Q);
            //we can use this condition also instead of two temp→left ^ temp→right
            if(!temp→left && temp→right || temp→left && !temp→right)
                count++;
            if(temp→left)
                EnQueue (Q, temp→left);
            if(temp→right)
                EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return count;
}
```

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

Problem-17 Given two binary trees, return true if they are structurally identical.

Solution:

Algorithm:

- If both trees are NULL then return true.
- If both trees are not NULL, then compare data and recursively check left and right subtree structures.

```
//Return true if they are structurally identical.
int AreStructurullySameTrees(struct BinaryTreeNode *root1, struct BinaryTreeNode *root2) {
        // both empty→1
        if(root1==NULL && root2==NULL)
            return 1;
        if(root1==NULL || root2==NULL)
```

```
        return 0;
    // both non-empty→compare them
    return(root1→data == root2→data && AreStructurullySameTrees(root1→left, root2→left) &&
        AreStructurullySameTrees(root1→right, root2→right));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for recursive stack.

Problem-18 Give an algorithm for finding the diameter of the binary tree. The diameter of a tree (sometimes called the *width*) is the number of nodes on the longest path between two leaves in the tree.

Solution: To find the diameter of a tree, first calculate the diameter of left subtree and right sub trees recursively. Among these two values, we need to send maximum along with current level (+1).

```
int DiameterOfTree(struct BinaryTreeNode *root, int *ptr){
    int left, right;
    if(!root)
        return 0;
    left = DiameterOfTree(root→left, ptr);
    right = DiameterOfTree(root→right, ptr);
    if(left + right > *ptr)
        *ptr = left + right;
    return Max(left, right)+1;
}
```

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

Problem-19 Give an algorithm for finding the level which is having maximum sum in the binary tree.

Solution: The logic is very much similar to finding number of levels. The only change is, we need to keep track of sums as well.

```
int FindLevelwithMaxSum(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    int level=0, maxLevel=0;
    struct Queue *Q;
    int currentSum = 0, maxSum = 0;
    if(!root) return 0;
    Q=CreateQueue();
    EnQueue(Q,root);
    EnQueue(Q,NULL);                    //End of first level.
    while(!IsEmptyQueue(Q)) {
        temp =DeQueue(Q);
        // If the current level is completed then compare sums
        if(temp == NULL) {
            if(currentSum> maxSum) {
                maxSum = currentSum;
                maxLevel = level;
            }
            currentSum = 0;
            //place the indicator for end of next level at the end of queue
            if(!IsEmptyQueue(Q))
                EnQueue(Q,NULL);
            level++;
        }
```

```
                else {  currentSum  += temp→data;
                        if(temp→left)
                                EnQueue(temp, temp→left);
                        if(root→right)
                                EnQueue(temp, temp→right);
                }
        }
        return maxLevel;
}
```

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

Problem-20 Given a binary tree, print out all of its root-to-leaf paths.

Solution: Refer comments in functions.

```
void PrintPathsRecur(struct BinaryTreeNode *root, int path[], int pathLen) {
        if(root ==NULL)
                return;
        // append this node to the path array
        path[pathLen] = root→data;
        pathLen++;
        // it's a leaf, so print the path that led to here
        if(root→left==NULL && root→right==NULL)
                PrintArray(path, pathLen);
        else {  // otherwise try both subtrees
                PrintPathsRecur(root→left, path, pathLen);
                PrintPathsRecur(root→right, path, pathLen);
        }
}
// Function that prints out an array on a line.
void PrintArray(int ints[], int len) {
        for (int i=0; i<len; i++)
                printf("%d",ints[i]);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for recursive stack.

Problem-21 Give an algorithm for checking the existence of path with given sum. That means, given a sum check whether there exists a path from root to any of the nodes.

Solution: For this problem, the strategy is: subtract the node value from the sum before calling its children recursively, and check to see if the sum is 0 when we run out of tree.

```
int HasPathSum(struct BinaryTreeNode  * root, int sum) {
        // return true if we run out of tree and sum==0
        if(root == NULL) return(sum == 0);
         else {      // otherwise check both subtrees
                int remainingSum = sum - root→data;
                if((root→left && root→right)||(!root→left && !root→right))
                      return(HasPathSum(root→left, remainingSum) ||  HasPathSum(root→right, remainingSum));
                else if(root→left)
                        return HasPathSum(root→left, remainingSum);
                else    return HasPathSum(root→right, remainingSum);
        }
```

```
}
```
Time Complexity: O(n). Space Complexity: O(n).

Problem-22 Give an algorithm for finding the sum of all elements in binary tree.

Solution: Recursively, call left subtree sum, right subtree sum and add their values to current nodes data.

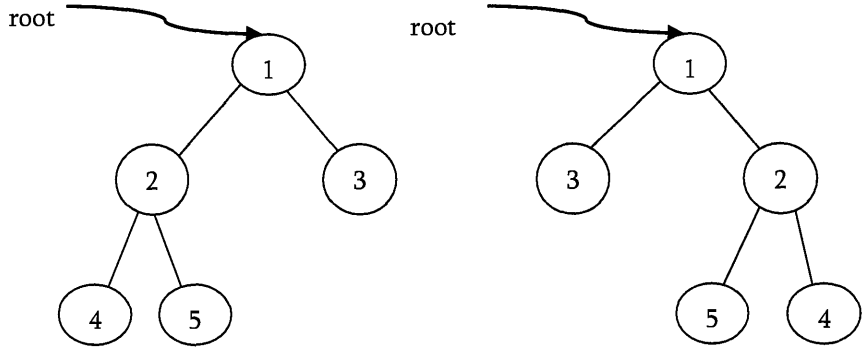
```
int Add(struct BinaryTreeNode *root) {
        if(root == NULL) return 0;
        else return (root→data + Add(root→left) + Add(root→right));
}
```
Time Complexity: O(n). Space Complexity: O(n).

Problem-23 Can we solve Problem-22 without recursion?

Solution: We can use level order traversal with simple change. Every time after deleting an element from queue, add the nodes data value to *sum* variable.

```
int SumofBTusingLevelOrder(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        struct Queue *Q;
        int sum = 0;
        if(!root) return 0;
        Q = CreateQueue();
        EnQueue(Q,root);
        while(!IsEmptyQueue(Q)) {
                temp = DeQueue(Q);
                sum += temp→data;
                if(temp→left)
                        EnQueue (Q, temp→left);
                if(temp→right)
                        EnQueue (Q, temp→right);
        }
        DeleteQueue(Q);
        return sum;
}
```
Time Complexity: O(n). Space Complexity: O(n).

Problem-24 Give an algorithm for converting a tree to its mirror. Mirror of a tree is another tree with left and right children of all non-leaf nodes interchanged.



Solution:

```
struct BinaryTreeNode *MirrorOfBinaryTree(struct BinaryTreeNode *root){
        struct BinaryTreeNode * temp;
        if(root) {
```

```
            MirrorOfBinaryTree(root→left);
            MirrorOfBinaryTree(root→right);
            /* swap the pointers in this node */
            temp  = root→left;
            root→left  = root→right;
            root→right = temp;
        }
        return root;
}
```

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

Problem-25 Given two trees, give an algorithm for checking whether they are mirrors of each other.

Solution:

```
int AreMirrors(struct BinaryTreeNode * root1, struct BinaryTreeNode * root2) {
        if(root1 == NULL && root2 == NULL)       return 1;
        if(root1 == NULL || root2 == NULL)  return 0;
        if(root1→data != root2→data)       return 0;
        else return AreMirrors(root1→left, root2→right) && AreMirrors(root1→right, root2→left);
}
```

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

Problem-26 Give an algorithm for finding LCA (Least Common Ancestor) of two nodes in a Binary Tree.

Solution:

```
struct BinaryTreeNode *LCA(struct BinaryTreeNode *root, struct BinaryTreeNode *α, struct BinaryTreeNode *β){
        struct BinaryTreeNode *left, *right;
        if(root == NULL) return root;
        if(root == α || root == β) return root;
        left = LCA (root→left, α, β );
        right = LCA (root→right, α, β );
        if(left && right)
                return root;
        else    return (left? left: right)
}
```

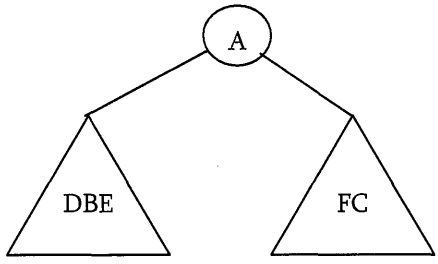
Time Complexity: O(n). Space Complexity: O(n) for recursion.

Problem-27 Give an algorithm for constructing binary tree from given Inorder and Preorder traversals.

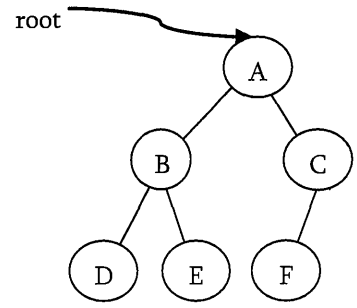
Solution: Let us consider the below traversals:

Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

In a Preorder sequence, leftmost element denotes the root of the tree. So we know '*A*' is root for given sequences. By searching '*A*' in Inorder sequence we can find out all elements on left side of '*A*' which come under left subtree and elements right side of '*A*' which come under right subtree. So we get the below structure.



We recursively follow above steps and get the following tree.



Algorithm: BuildTree()

1 Select an element from Preorder. Increment a Preorder index variable (preIndex in below code) to pick next element in next recursive call.
2 Create a new tree node (newNode) with the data as selected element.
3 Find the selected elements index in Inorder. Let the index be inIndex.
4 Call BuildBinaryTree for elements before inIndex and make the built tree as left subtree of newNode.
5 Call BuildBinaryTree for elements after inIndex and make the built tree as right subtree of newNode.
6 return newNode.

```
struct BinaryTreeNode * BuildBinaryTree(int inOrder[], int preOrder[], int inStrt, int inEnd){
        static int preIndex = 0;
        struct BinaryTreeNode *newNode
        if(inStrt > inEnd) return NULL;
        newNode = (struct BinaryTreeNode *) malloc  (sizeof(struct BinaryTreeNode));
        if(!newNode) {
                printf("Memory Error");
                return;
        }
        // Select current node from Preorder traversal using preIndex
        newNode→data = preOrder[preIndex];
        preIndex++;
        if(inStrt == inEnd)                          /* if this node has no children then return */
                return newNode;
        /* else find the index of this node in Inorder traversal */
        int inIndex = Search(inOrder, inStrt, inEnd, newNode→data);

        /* Using index in Inorder traversal, construct left and right subtress */
        newNode→left = BuildBinaryTree(inOrder, preOrder, inStrt, inIndex-1);
        newNode→right = BuildBinaryTree(inOrder, preOrder, inIndex+1, inEnd);
        return newNode;
}
```

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

Problem-28 If we are given two traversal sequences, can we construct the binary tree uniquely?

Solution: It depends on what traversals are given. If one of the traversal methods is *Inorder* then the tree can be constructed uniquely, otherwise not.

Therefore, following combination can uniquely identify a tree:

- Inorder and Preorder
- Inorder and Postorder
- Inorder and Level-order

The following combinations do not uniquely identify a tree.

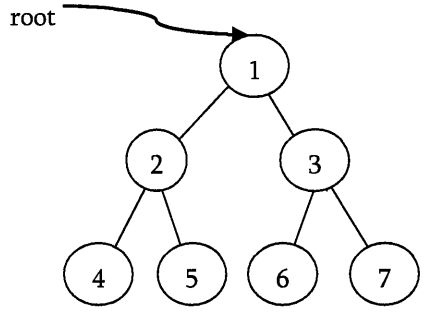
- Postorder and Preorder
- Preorder and Level-order
- Postorder and Level-order

For example, Preorder, Level-order and Postorder traversals are same for above trees:

Preorder Traversal = AB Postorder Traversal = BA Level-order Traversal = AB

So, even if three of them (PreOrder, Level-Order and PostOrder) are given, tree cannot be constructed uniquely.

Problem-29 Give an algorithm for printing all the ancestors of a node in a Binary tree. For the below tree, for 7 the ancestors are 1 3 7.

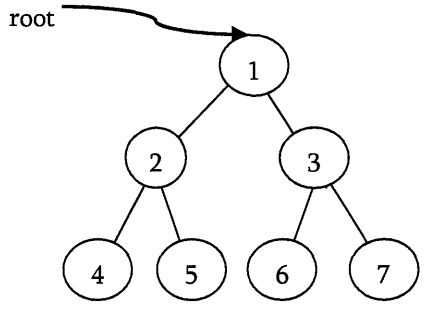


Solution: Apart from the Depth First Search of this tree, we can use the following recursive way to print the ancestors.

```
int PrintAllAncestors(struct BinaryTreeNode *root, struct BinaryTreeNode *node){
        if(root == NULL) return 0;
        if(root→left == node || root→right == node || PrintAllAncestors(root→left, node) ||
                        PrintAllAncestors(root→right, node)) {
            printf("%d",root→data);
            return 1;
        }
        return 0;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$ for recursion.

Problem-30 **Zigzag Tree Traversal:** Give an algorithm to traverse a binary tree in Zigzag order. For example, the output for the below tree should be: 1 3 2 4 5 6 7



Solution: This problem can be solved easily using two stacks. Assume the two stacks are: *currentLevel* and *nextLevel*. We would also need a variable to keep track of the current level order (whether it is left to right or right or left).

We pop from *currentLevel* stack and print the nodes value. Whenever the current level order is from left to right, push the nodes left child, then its right child to stack *nextLevel*. Since a stack is a Last In First OUT (*LIFO*) structure, next time when nodes are popped off nextLevel, it will be in the reverse order. On the other hand, when the current level order is from right to left, we would push the nodes right child first, then its left child. Finally, don't forget to swap those two stacks at the end of each level (*i. e.*, when *currentLevel* is empty).

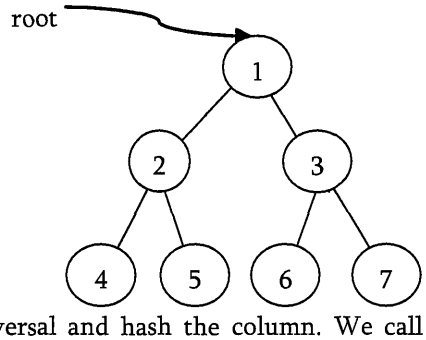
```
void ZigZagTraversal(struct BinaryTreeNode *root){
        struct BinaryTreeNode *temp;
        int leftToRight = 1;
        if(!root) return;
        struct Stack *currentLevel = CreateStack(), *nextLevel = CreateStack();
        Push(currentLevel, root);
        while(!IsEmptyStack(currentLevel)) {
                temp = Pop(currentLevel);
                if(temp) {
                        printf("%d",temp→data);
                        if(leftToRight) {
                                if(temp→left)  Push(nextLevel, temp→left);
                                if(temp→right) Push(nextLevel, temp→right);
                        }
                        else {  if(temp→right) Push(nextLevel, temp→right);
                                if(temp→left)  Push(nextLevel, temp→left);
                        }
                }
                if(IsEmptyStack(currentLevel)) {
                        leftToRight = 1-leftToRight;
                        swap(currentLevel, nextLevel);
                }
        }
}
```

Time Complexity: $O(n)$. Space Complexity: Space for two stacks = $O(n) + O(n) = O(n)$.

Problem-31 Give an algorithm for finding the vertical sum of a binary tree. For example,

The tree has 5 vertical lines

Vertical-1: nodes-4 => vertical sum is 4
Vertical-2: nodes-2 => vertical sum is 2
Vertical-3: nodes-1,5,6 => vertical sum is 1 + 5 + 6 = 12
Vertical-4: nodes-3 => vertical sum is 3
Vertical-5: nodes-7 => vertical sum is 7
We need to output: 4 2 12 3 7

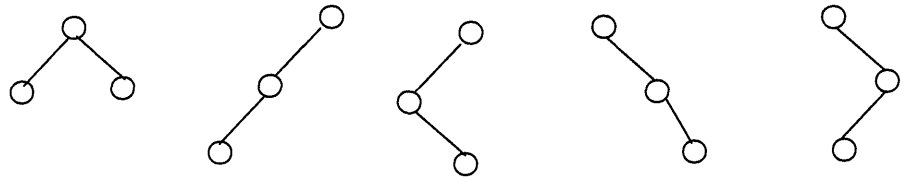


Solution: We can do an inorder traversal and hash the column. We call VerticalSumInBinaryTree(root, 0) which means the root is at column 0. While doing the traversal, hash the column and increase its value by *root → data*.

```
void VerticalSumInBinaryTree (struct BinaryTreeNode *root, int column){
        if(root==NULL) return;
        VerticalSumInBinaryTree(root→left, column-1);
        //Refer Hashing chapter for implementation of hash table
        Hash[column] += root→data;
        VerticalSumInBinaryTree(root→right, column+1);
}
VerticalSumInBinaryTree(root, 0);
Print Hash;
```

Problem-32 How many different binary trees are possible with n nodes?

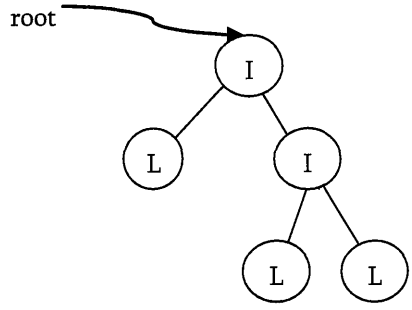
Solution: For example, consider a tree with 3 nodes ($n = 3$), it will have the maximum combination of 5 different (i.e., $2^3 - 3 = 5$) trees.



In general, if there are n nodes, there exist $2^n - n$ different trees.

Problem-33 Given a tree with a special property where leaves are represented with 'L' and internal node with 'I'. Also, assume that each node has either 0 or 2 children. Given preorder traversal of this tree, construct the tree [1].

Example: Given preorder string => ILILL



Solution: First, we should see how preorder traversal is arranged. Pre-order traversal means first put root node, then pre-order traversal of left subtree and then pre-order traversal of right subtree. In normal scenario, it's not possible to detect where left subtree ends and right subtree starts using only pre-order traversal. Since every node has either 2 children or no child, we can surely say that if a node exists then its sibling also exists. So every time we are computing a subtree, we need to compute its sibling subtree as well.

Secondly, whenever we get 'L' in the input string, that is a leaf and we can stop for a particular subtree at that point. After this 'L' node (left child of its parent 'L'), its sibling starts. If 'L' node is right child of its parent, then we need to go up in the hierarchy to find next subtree to compute. Keeping above invariant in mind, we can easily determine when a subtree ends and next start. It means that we can give any start node to our method and it can easily complete the subtree it generates going outside of its nodes. We just need to take care of passing correct start nodes to different sub-trees.

```
struct BinaryTreeNode *BuildTreeFromPreOrder(char* A, int *i){
        struct BinaryTreeNode *newNode;
        newNode = (struct BinaryTreeNode *) malloc(sizeof(struct BinaryTreeNode));
        newNode→data = A[*i];
        newNode→left = newNode→right = NULL;
        if(A == NULL){                          //Boundary Condition
                free(newNode);
```

```
        return NULL;
    }
    if(A[*i] == 'L')                    //On reaching leaf node, return
        return newNode;
    *i = *i + 1;                        //Populate left sub tree
    newNode→left = BuildTreeFromPreOrder(A, i);
    *i = *i + 1;                        //Populate right sub tree
    newNode→right = BuildTreeFromPreOrder(A, i);
    return newNode;
}
```

Time Complexity: $O(n)$.

Problem-34 Given a binary tree with three pointers (left, right and nextSibling), give an algorithm for filling the *nextSibling* pointers assuming they are NULL initially.

Solution: We can use simple queue (similar to the solution of Problem-11). Let us assume that the structure of binary tree is:

```
struct BinaryTreeNode {
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
    struct BinaryTreeNode* nextSibling;
};
int FillNextSiblings(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root) return 0;
    Q = CreateQueue();
    EnQueue(Q,root);
    EnQueue(Q,NULL);
    while(!IsEmptyQueue(Q)) {
        temp =DeQueue(Q);
        // Completion of current level.
        if(temp ==NULL) { //Put another marker for next level.
            if(!IsEmptyQueue(Q))
                EnQueue(Q,NULL);
        }
        else { temp→nextSibling = QueueFront(Q);
            if(root→left) EnQueue(Q, temp→left);
            if(root→right) EnQueue(Q, temp→right);
        }
    }
}
```

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

Problem-35 For Problem-34, is there any otherway of solving?

Solution: The trick is to re-use the populated *nextSibling* pointers. As mentioned earlier, we just need one more step for it to work. Before we passed the *left* and *right* to the recursion function itself, we connect the right childs *nextSibling* to the current nodes nextSibling left child. In order for this to work, the current node *nextSibling* pointer must be populated, which is true in this case.

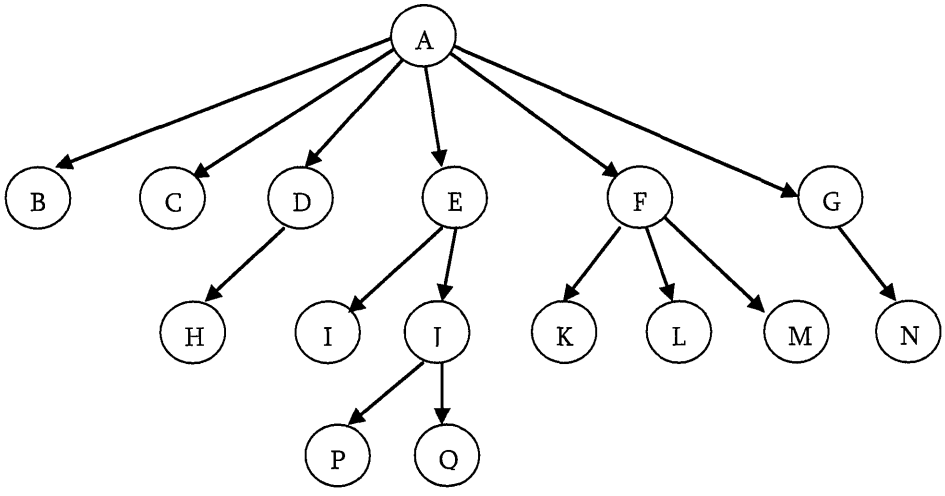
```
void FillNextSiblings(struct BinaryTreeNode* root) {
    if (!root) return;
```

```
        if (root→left) root→left→nextSibling = root→right;
        if (root→right)
                root→right→nextSibling = (root→nextSibling) ? root→nextSibling→left : NULL;
        FillNextSiblings(root→left);
        FillNextSiblings(root→right);
}
```

Time Complexity: O(n).

6.7 Generic Trees (N-ary Trees)

In the previous section we have discussed binary trees where each node can have maximum of two children only and represented them easily with two pointers. But suppose if we have a tree with many children at every node and also if we do not know how many children a node can have, how do we represent them? For example, consider the tree shown above.



For a tree like this, how do we represent the tree?

In the above tree, there are nodes with 6 children, with 3 children, 2 children, with 1 child, and with zero children (leaves). To present this tree we have to consider the worst case (6 children) and allocate those many child pointers for each node. Based on this, the node representation can be given as:

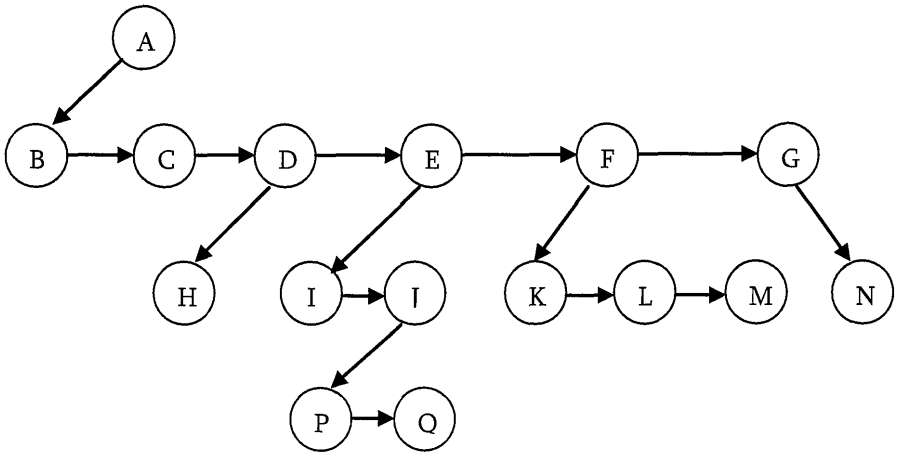
```
        struct TreeNode{
                int data;
                struct TreeNode *firstChild;
                struct TreeNode *secondChild;
                struct TreeNode *thirdChild;
                struct TreeNode *fourthChild;
                struct TreeNode *fifthChild;
                struct TreeNode *sixthChild;
        };
```

Since we are not using all the pointers in all the cases there is a lot of memory wastage. Also, another problem is that, in advance we do not know the number of children for each node. In order to solve this problem we need a representation that minimizes the wastage and also accept nodes with any number of children.

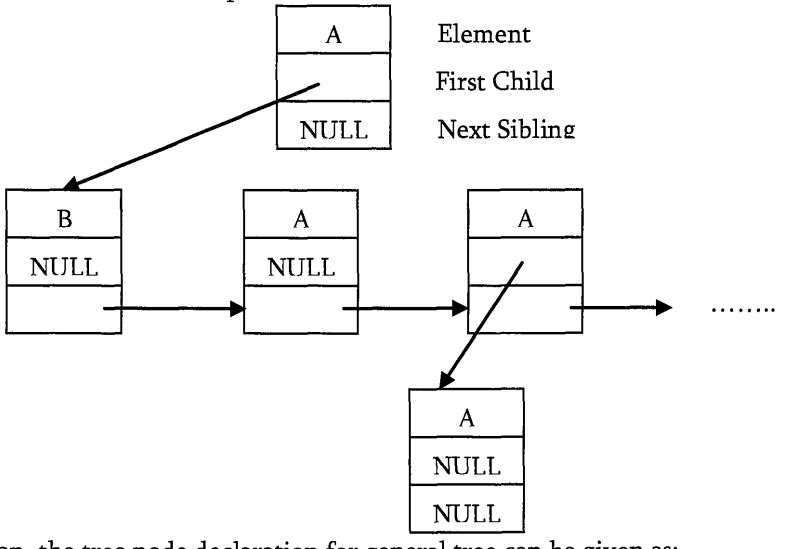
Representation of Generic Trees

Since our objective is to reach all nodes of the tree, a possible solution to this is as follows:

- At each node link children of same parent (siblings) from left to right.
- Remove the links from parent to all children except the first child.



What these above statements say is if we have a link between childrens then we do not need extra links from parent to all children. This is because we can traverse all the elements by starting at the first child of the parent. So if we have link between parent and first child and also links between all children of same parent then it solves our problem. This representation is sometimes called first child/next sibling representation. First child/next sibling representation of the generic tree is shown above. The actual representation for this tree is:



Based on this discussion, the tree node declaration for general tree can be given as:

```
struct TreeNode {
        int data;
        struct TreeNode *firstChild;
        struct TreeNode *nextSibling;
};
```

Note: Since we are able to represent any generic tree with binary representation, in practice we use only binary tree.

Problems on Generic Trees

Problem-36 Given a tree, give an algorithm for finding the sum of all the elements of the tree.

Solution: The solution is similar to what we have done for simple binary trees. That means, traverse the complete list and keep on adding the values. We can either use level order traversal or simple recursion.

```
int FindSum(struct TreeNode *root){
        if(!root) return 0;
        return root→data + FindSum(root→firstChild) + FindSum(root→sibling);
}
```

Time Complexity: O(n). Space Complexity: O(1) (if we do not consider stack space), otherwise O(n).

Note: All problems which we have discussed for binary trees are applicable for generic trees also. Instead of left and right pointers we just need to use firstChild and nextSibling.

Problem-37 For a 4-ary tree (each node can contain maximum of 4 children), what is the maximum possible height with 100 nodes? Assume height of a single node is 0.

Solution: In 4-ary tree each node can contain 0 to 4 children and to get maximum height, we need to keep only one child for each parent. With 100 nodes the maximum possible height we can get is 99. If we have a restriction that at least one node is having 4 children, then we keep one node with 4 children and remaining all nodes with 1 child. In this case, the maximum possible height is 96. Similarly, with n nodes the maximum possible height is $n - 4$.

Problem-38 For a 4-ary tree (each node can contain maximum of 4 children), what is the minimum possible height with n nodes?

Solution: Similar to above discussion, if we want to get minimum height, then we need to fill all nodes with maximum children (in this case 4). Now let's see the following table, which indicates the maximum number of nodes for a given height.

Height, h	Maximum Nodes at height, $h = 4^h$	Total Nodes height $h = \frac{4^{h+1}-1}{3}$
0	1	1
1	4	1+4
2	4 × 4	1+ 4 × 4
3	4 × 4 × 4	1+ 4 × 4 + 4 × 4 × 4

For a given height h the maximum possible nodes are: $\frac{4^{h+1}-1}{3}$. To get minimum height, take logarithm on both sides:

$$n = \frac{4^{h+1}-1}{3} \Rightarrow 4^{h+1} = 3n + 1 \Rightarrow (h+1)log4 = \log(3n+1) \Rightarrow h + 1 = \log_4(3n+1) \Rightarrow h = \log_4(3n+1) - 1$$

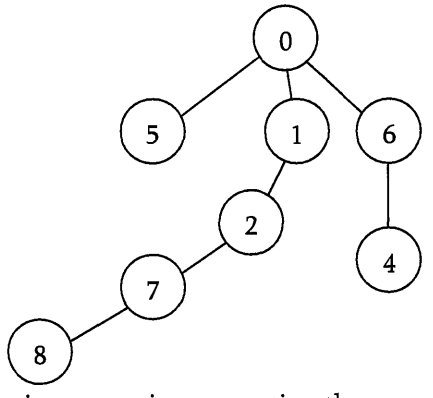
Problem-39 Given a parent array P, where $P[i]$ indicates the parent of i^{th} node in the tree (assume parent of root node is indicated with -1). Give an algorithm for finding the height or depth of the tree.

Solution:

For example: if the P is

-1	0	1	6	6	0	0	2	7
0	1	2	3	4	5	6	7	8

Its corresponding tree is:



From the problem definition, the given array is representing the parent array. That means, we need to consider the tree for that array and find the depth of the tree. The depth of this given tree is 4. If we carefully observe, we just need to start at every node and keep going to its parent until we reach -1 and also keep track of the maximum depth among all nodes.

```
int FindDepthInGenericTree(int P[], int n){
        int maxDepth =-1, currentDepth =-1, j;
        for (int i = 0; i < n; i++)    {
```

```
                currentDepth = 0; j = i;
                while(P[j] != -1) {
                        currentDepth++; j = P[j];
                }
                if(currentDepth > maxDepth)
                        maxDepth = currentDepth;
        }
        return maxDepth;
}
```

Time Complexity: $O(n^2)$. For skew trees we will be re-calculating the same values. Space Complexity: $O(1)$.

Note: We can optimize the code by storing the previous calculated nodes depth in some hash table or other array. This reduces the time complexity but uses extra space.

Problem-40 Given a node in the generic tree, give an algorithm for counting the number of siblings for that node.

Solution: Since tree is represented with first child/next sibling method, the tree structure can be given as:

```
struct TreeNode{
        int data;
        struct TreeNode *firstChild;
        struct TreeNode *nextSibling;
};
```

For a given node in the tree, we just need to traverse all its nextsiblings.

```
int SiblingsCount(struct TreeNode *current){
        int count = 0;
        while(current) {
                count++;
                current = current→nextSibling;
        }
        reutrn count;
}
```

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

Problem-41 Given a node in the generic tree, give an algorithm for counting the number of children for that node.

Solution: Since the tree is represented as first child/next sibling method, the tree structure can be given as:

```
struct TreeNode{
        int data;
        struct TreeNode *firstChild;
        struct TreeNode *nextSibling;
};
```

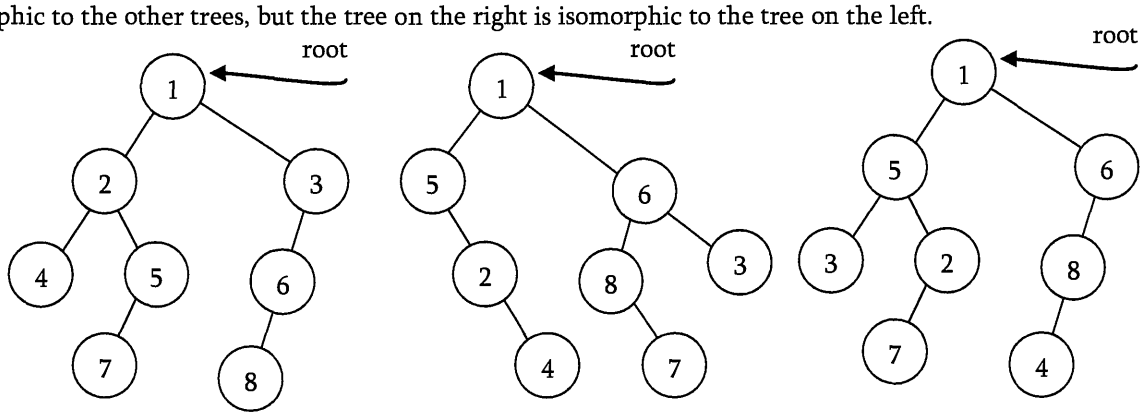
For a given node in the tree, we just need to point to its first child and keep traversing all its nextsiblings.

```
int ChildCount(struct TreeNode *current){
        int count = 0;
        current = current→firstChild;
        while(current)  {
                count++;
                current = current→nextSibling;
        }
        reutrn count;
}
```

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

Problem-42 Given two trees how do we check whether the trees are isomorphic to each other or not?

Solution: Two binary trees *root*1 and *root*2 are isomorphic if they have the same structure. The values of the nodes does not affect whether two trees are isomorphic or not. In the diagram below, the tree in the middle is not isomorphic to the other trees, but the tree on the right is isomorphic to the tree on the left.

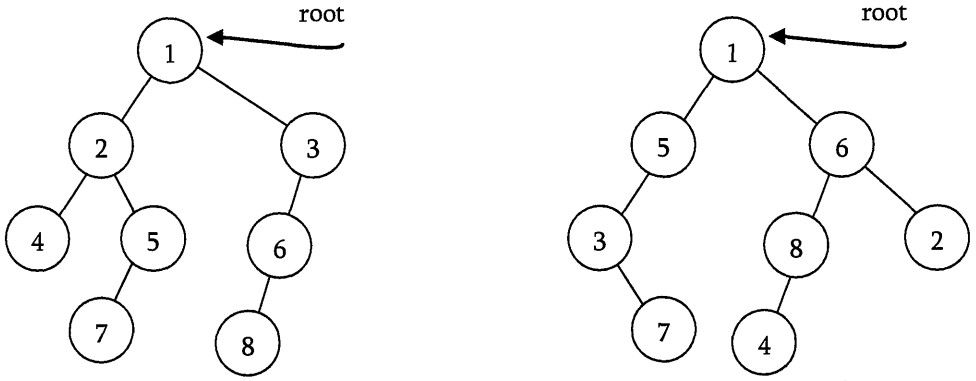



```
int IsIsIsomorphic(struct TreeNode *root1, struct TreeNode *root2){
        if(!root1 && !root2) return 1;
        if((!root1 && root2) || (root1 && !root2))
                return 0;
        return (IsIsomorphic(root1→left, root2→left) && IsIsomorphic(root1→right, root2→right));
}
```

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

Problem-43 Given two trees how do we check whether they are quasi-isomorphic to each other or not?

Solution:



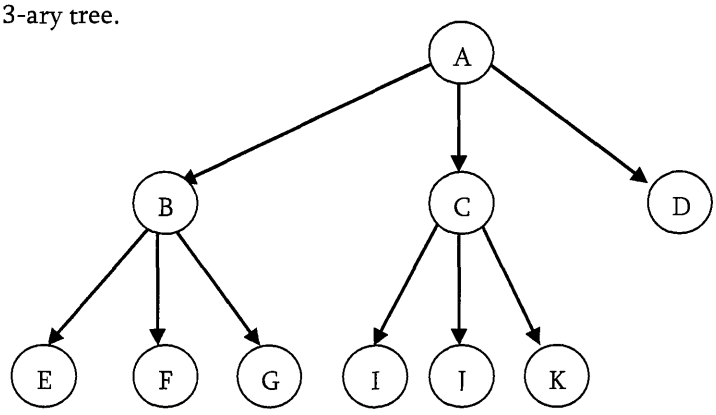
Two trees *root*1 and *root*2 are quasi-isomorphic if *root*1 can be transformed into *root*2 by swapping left and right children of some of the nodes of *root*1. The data in the nodes are not important in determining quasi-isomorphism, only the shape is important. The trees below are quasi-isomorphic because if the children of the nodes on the left are swapped, the tree on the right is obtained.

```
int QuasiIsomorphic(struct TreeNode *root1, struct TreeNode *root2){
        if(!root1 && !root2) return 1;
        if((!root1 && root2) || (root1 && !root2))
                return 0;
        return (QuasiIsomorphic(root1→left, root2→left) && QuasiIsomorphic(root1→right, root2→right)
                || QuasiIsomorphic(root1→right, root2→left) && QuasiIsomorphic(root1→left, root2→right));
}
```

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

Problem-44 A full k −ary tree is a tree where each node has either 0 or k children. Given an array which contains the preorder traversal of full k −ary tree, give an algorithm for constructing the full k −ary tree.

Solution: In k −ary tree, for a node at i^{th} position its children will be at $k * i + 1$ to $k * i + k$. For example, the below is an example for full 3-ary tree.



As we have seen, in preorder traversal first left subtree is processed then followed by root node and right subtree. Because of this, to construct a full k-ary, we just need to keep on creating the nodes without bothering about the previous constructed nodes. We can use this trick to build the tree recursively by using one global index. Declaration for k-ary tree can be given as:

```
struct K-aryTreeNode{
        char data;
        struct K-aryTreeNode *child[];
};
int *Ind = 0;
struct K-aryTreeNode *BuildK-aryTree(char A[], int n, int k){
        if(n<=0)
                return NULL;
        struct K-aryTreeNode *newNode = (struct K-aryTreeNode*) malloc(sizeof(struct K-aryTreeNode));
        if(!newNode) {
                printf("Memory Error");
                return;
        }
        newNode→child = (struct K-aryTreeNode*) malloc( k * sizeof(struct K-aryTreeNode));
        if(!newNode→child) {
                printf("Memory Error");
                return;
        }
        newNode→data = A[Ind];
        for (int i = 0; i<k; i++) {
                if(k * Ind + i <n) {
                        Ind++;
                        newNode→child[i] = BuildK-aryTree(A, n, k,Ind );
                }
                else newNode→child[i] =NULL;
        }
        return newNode;
}
```

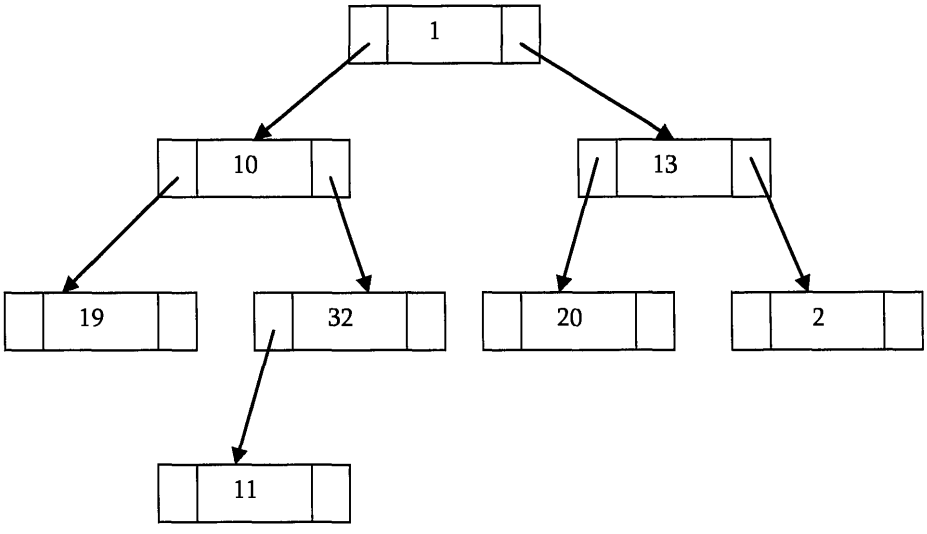
Time Complexity: $O(n)$, where n is the size of the pre-order array. This is because we are moving sequentially and not visiting the already constructed nodes.

6.8 Threaded Binary Tree Traversals [Stack or Queue less Traversals]

In earlier sections we have seen that, *preorder, inorder and postorder* binary tree traversals used stacks and *level order* traversal used queues as an auxiliary data structure. In this section we will discuss new traversal algorithms which do not need both stacks and queues and such traversal algorithms are called *threaded binary tree traversals* or *stack/queue less traversals*.

Issues with Regular Binary Tree Traversals

- The storage space required for the stack and queue is large.
- The majority of pointers in any binary tree are NULL. For example, a binary tree with n nodes has $n + 1$ NULL pointers and these were wasted.



- It is difficult to find successor node (preorder, inorder and postorder successors) for a given node.

Motivation for Threaded Binary Trees

To solve these problems, one idea is to store some useful information in NULL pointers. If we observe previous traversals carefully, stack/queue is required because we have to record the current position in order to move to right subtree after processing the left subtree. If we store the useful information in NULL pointers, then we don't have to store such information in stack/queue. The binary trees which store such information in NULL pointers are called *threaded binary trees*. From the above discussion, let us assume that we have decided to store some useful information in NULL pointers. The next question is what to store?

The common convention is put predecessor/successor information. That means, if we are dealing with preorder traversals then for a given node, NULL left pointer will contain preorder predecessor information and NULL right pointer will contain preorder successor information. These special pointers are called *threads*.

Classifying Threaded Binary Trees

The classification is based on whether we are storing useful information in both NULL pointers or only in one of them.

- If we store predecessor information in NULL left pointers only then we call such binary trees as *left threaded binary trees*.
- If we store successor information in NULL right pointers only then we call such binary trees as *right threaded binary trees*.
- If we store predecessor information in NULL left pointers only then we call such binary trees as *fully threaded binary trees* or simply *threaded binary trees*.

Note: For the remaining discussion we consider only *(fully) threaded binary trees.*

Types of Threaded Binary Trees

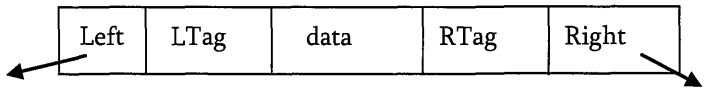
Cased on above discussion we get three representations for threaded binary trees.

- *Preorder Threaded Binary Trees*: NULL left pointer will contain PreOrder predecessor information and NULL right pointer will contain PreOrder successor information
- *Inorder Threaded Binary Trees*: NULL left pointer will contain InOrder predecessor information and NULL right pointer will contain InOrder successor information
- *Postorder Threaded Binary Trees*: NULL left pointer will contain PostOrder predecessor information and NULL right pointer will contain PostOrder successor information

Note: As the representations are similar, for the remaining discussion, we will use InOrder threaded binary trees.

Threaded Binary Tree structure

Any program examining the tree must be able to differentiate between a regular *left/right* pointer and a *thread*. To do this, we use two additional fields into each node giving us, for threaded trees, nodes of the following form:



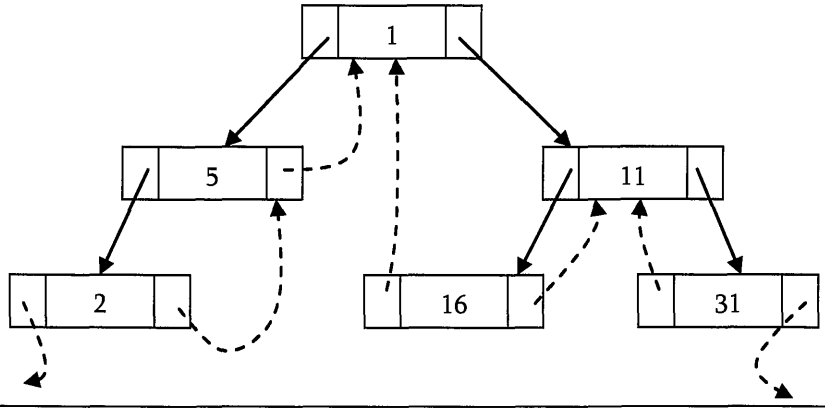

```
struct ThreadedBinaryTreeNode{
        struct ThreadedBinaryTreeNode *left;
        int LTag;
        int data;
        int RTag;
        struct ThreadedBinaryTreeNode *right;
};
```

Difference between Binary Tree and Threaded Binary Tree Structures

	Regular Binary Trees	Threaded Binary Trees
if LTag == 0	NULL	left points to the in-order predecessor
if LTag == 1	left points to the left child	left points to left child
if RTag == 0	NULL	right points to the in-order successor
if RTag == 1	right points to the right child	right points to the right child

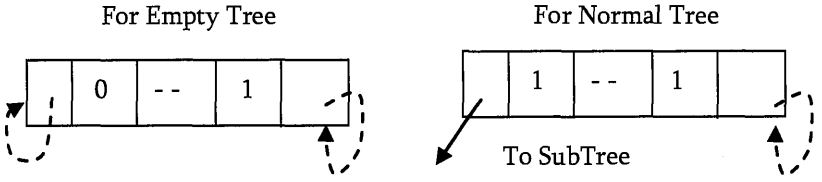
Note: Similarly, we can define for preorder/postorder differences as well.

As an example, let us try representing a tree in inorder threaded binary tree form. The below tree shows how an inorder threaded binary tree will look like. The dotted arrows indicate the threads. If we observe, the left pointer of left most node (2) and right pointer of right most node (31) are hanging.

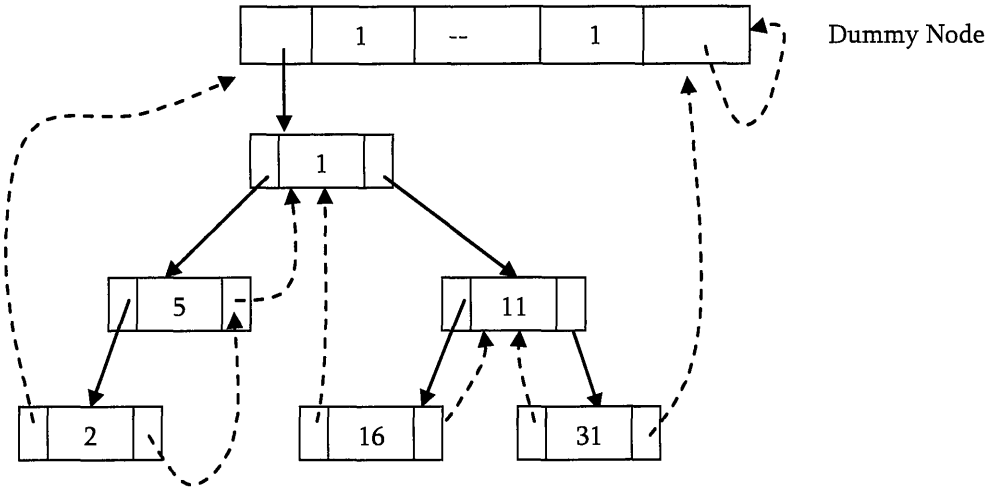


What should leftmost and rightmost pointers point to?

In the representation of a threaded binary tree, it is convenient to use a special node *Dummy* which is always present even for an empty tree. Note that, right tag of dummy node is 1 and its right child points to itself.



With this convention the above tree can be represented as:



Finding Inorder Successor in Inorder Threaded Binary Tree

To find inorder successor of a given node without using a stack, assume that the node for which we want to find the inorder successor is *P*.

Strategy: If *P* has a no right subtree, then return the right child of *P*. If *P* has right subtree, then return the left of the nearest node whose left subtree contains *P*.

```
struct ThreadedBinaryTreeNode* InorderSuccessor(struct ThreadedBinaryTreeNode *P){
        struct ThreadedBinaryTreeNode *Position;
        if(P→RTag == 0) return P→right;
        else {  Position = P→right;
                while(Position→LTag == 1)
                        Position = Position→left;
                return Position;
        }
}
```

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

Inorder Traversal in Inorder Threaded Binary Tree

We can start with *dummy* node and call InorderSuccessor() to visit each node until we reach *dummy* node.

```
void InorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P = InorderSuccessor(root);
        while(P != root) {
                P = InorderSuccessor(P);
                printf("%d",P→data);
```

```
        }
}
```

Other way of coding:

```
void InorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P = root;
        while(1) {  P = InorderSuccessor(P);
                    if(P == root) return;
                    printf("%d",P→data);
        }
}
```

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

Finding PreOrder Successor in InOrder Threaded Binary Tree

Strategy: If P has a left subtree, then return the left child of P. If P has no left subtree, then return the right child of the nearest node whose right subtree contains P.

```
struct ThreadedBinaryTreeNode* PreorderSuccessor(struct ThreadedBinaryTreeNode *P){
        struct ThreadedBinaryTreeNode *Position;
        if(P→LTag == 1) return P→left;
        else {    Position = P;
                  while(Position→RTag == 0)
                          Position = Position→right;
                  return Position→right;
        }
}
```

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

PreOrder Traversal of InOrder Threaded Binary Tree

As similar to inorder traversal, start with *dummy* node and call PreorderSuccessor() to visit each node until we get *dummy* node again.

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root){
        struct ThreadedBinaryTreeNode *P;
        P = PreorderSuccessor(root);
        while(P != root) {
                  P = PreorderSuccessor(P);
                  printf("%d",P→data);
        }
}
```

Other way of coding:

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root) {
        struct ThreadedBinaryTreeNode *P = root;
        while(1){P = PreorderSuccessor(P);
                  if(P == root) return;
                  printf("%d",P→data);
        }
}
```

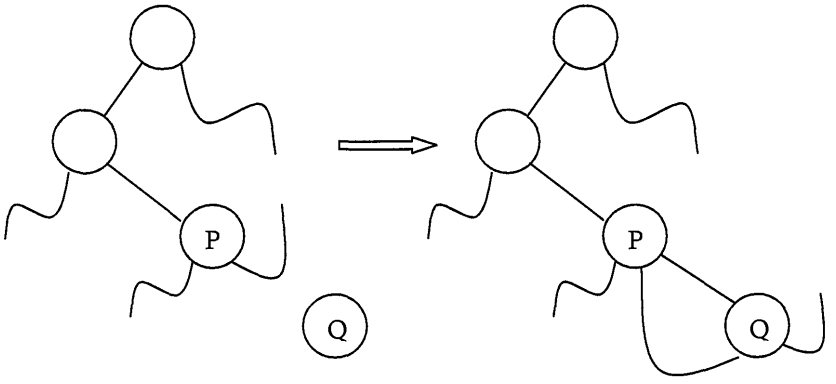
Time Complexity: O(n). Space Complexity: O(1).

Note: From the above discussion, it should be clear that inorder and preorder successor finding is easy with threaded binary trees. But finding postorder successor is very difficult if we do not use stack.

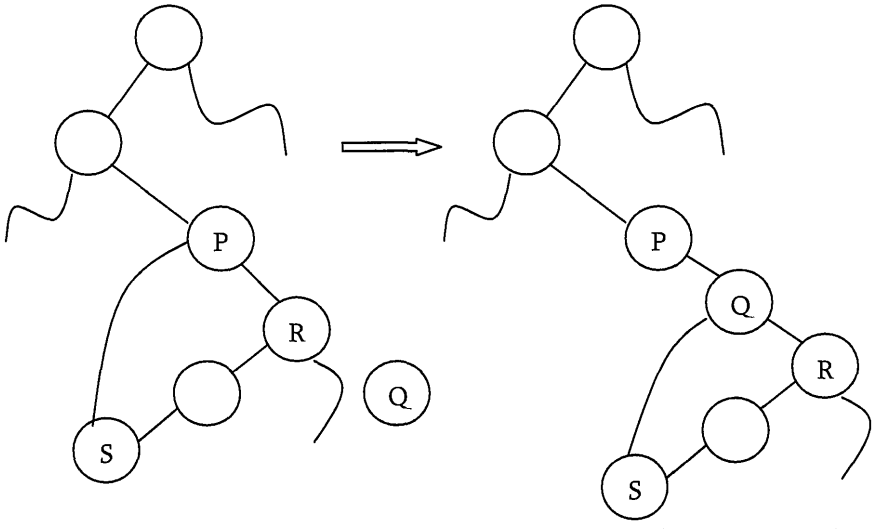
Insertion of Nodes in InOrder Threaded Binary Trees

For simplicity, let us assume that there are two nodes P and Q and we want to attach Q to right of P. For this we will have two cases.

- Node P does not has right child: In this case we just need to attach Q to P and change its left and right pointers.



- Node P has right child (say, R): In this case we need to traverse $R's$ left subtree and find the left most node and then update the left and right pointer of that node (as shown below).




```
void InsertRightInInorderTBT(struct ThreadedBinaryTreeNode *P, struct ThreadedBinaryTreeNode *Q){
        struct ThreadedBinaryTreeNode *Temp;
        Q→right = P→right;
        Q→RTag = P→RTag;
        Q→left = P;
        Q→LTag = 0;
        P→right = Q;
        P→RTag = 1;
        if(Q→RTag == 1) {                              //Case-2
                Temp = Q→right;
                while(Temp→LTag)
                        Temp = Temp→left;
                Temp→left = Q;
        }
}
```

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

Problems on Threaded binary Trees

Problem-45 For a given binary tree (not threaded) how do we find the preorder successor?

Solution: For solving this problem, we need to use an auxiliary stack *S*. On the first call, the parameter node is a pointer to the head of the tree, thereafter its value is NULL. Since we are simply asking for the successor of the node we got last time we called the function. It is necessary that the contents of the stack *S* and the pointer *P* to the last node "visited" are preserved from one call of the function to the next, they are defined as static variables.

```
// pre-order successor for an unthreaded binary tree
struct BinaryTreeNode *PreorderSuccssor(struct BinaryTreeNode *node){
        static struct BinaryTreeNode *P;
        static Stack *S = CreateStack();
        if(node != NULL)
                P = node;
        if(P→left != NULL) {
                Push(S,P);
                P = P→left;
        }
        else {  while (P→right = NULL)
                        P = Pop(S);
                P = P→right;
        }
        return P;
}
```

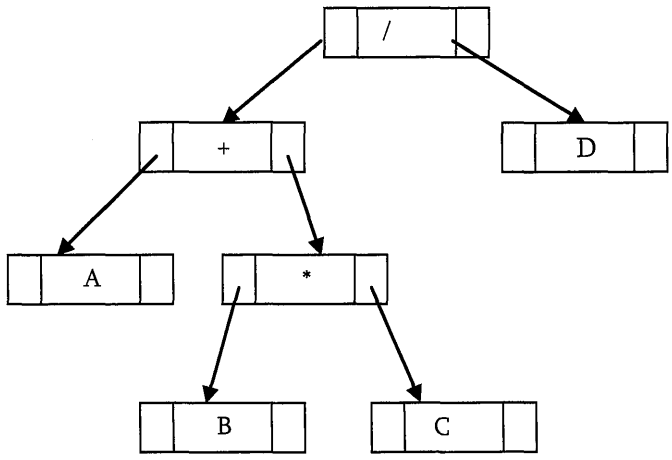
Problem-46 For a given binary tree (not threaded) how do we find the inorder successor?

Solution: Similar to above discussion, we can find the inorder successor of a node as:

```
// In-order successor for an unthreaded binary tree
struct BinaryTreeNode *InorderSuccssor(struct BinaryTreeNode *node){
        static struct BinaryTreeNode *P;
        static Stack *S = CreateStack();
        if(node != NULL)
                P = node;
        if(P→right == NULL)
                P = Pop(S);
        else {  P = P→right;
                while (P→left != NULL)
                        Push(S, P);
                P = P→left;
        }
        return P;
}
```

6.9 Expression Trees

A tree representing an expression is called as an expression tree. In expression trees leaf nodes are operands and non-leaf nodes are operators. That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands. Expression tree consists of binary expression. But for a unary operator, one subtree will be empty. Below figure shows a simple expression tree for (A + B * C) / D.

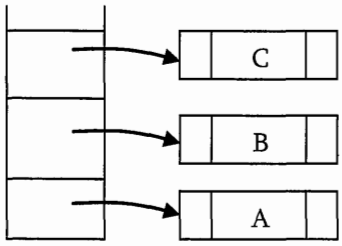


Algorithm for Building Expression Tree from Postfix Expression

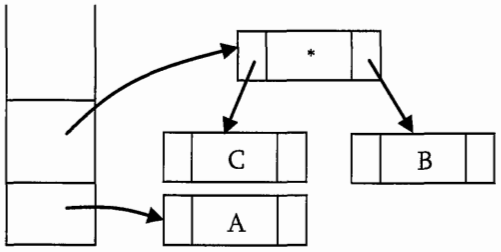
```
struct BinaryTreeNode *BuildExprTree(char postfixExpr[], int size){
        struct Stack *S = Stack(size);
        for (int i = 0; i< size; i++) {
                if(postfixExpr[i] is an operand) {
                        struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
                                malloc( sizeof (struct BinaryTreeNode));
                        if(!newNode) {
                                printf("Memory Error");
                                return;
                        }
                        newNode→data =postfixExpr[i];
                        newNode→left = newNode→right = NULL;
                        Push(S, newNode);
                }
                else {
                        struct BinaryTreeNode *T2 = Pop(S), *T1 = Pop(S);
                        struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
                                malloc(sizeof(struct  BinaryTreeNode));
                        if(!newNode) {
                                printf("Memory Error"); return;
                        }
                        newNode→data = postfixExpr[i];
                        newNode→left = T1; newNode→right = T2;
                        Push(S, newNode);
                }
        }
        return S;
}
```

Example: Assume that one symbol is read at a time. If the symbol is an operand, we create a tree node and push a pointer to it onto a stack. If the symbol is an operator, pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

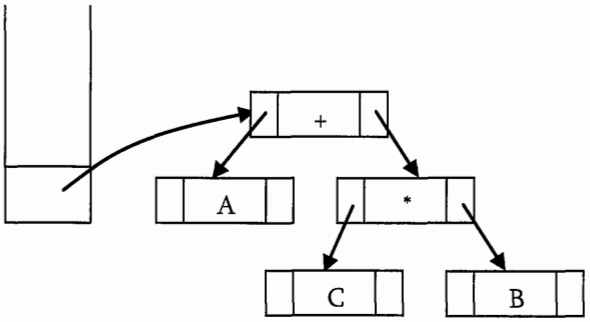
As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



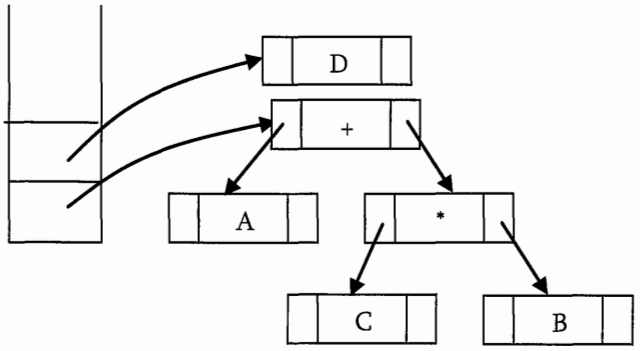
Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



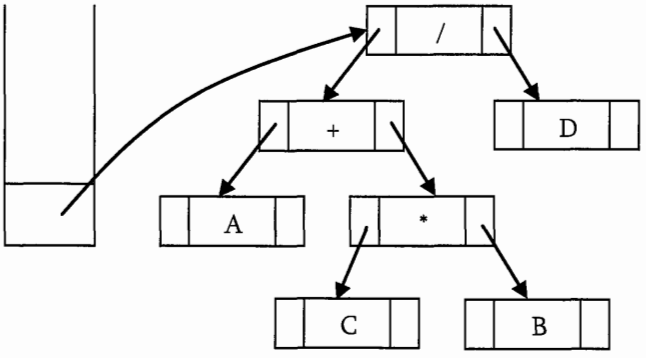
Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



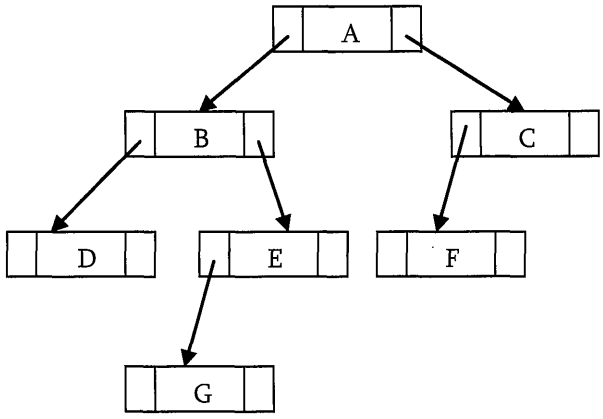
Next, an operand 'D' is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Finally, the last symbol ('/') is read, two trees are merged and a pointer to the final tree is left on the stack.



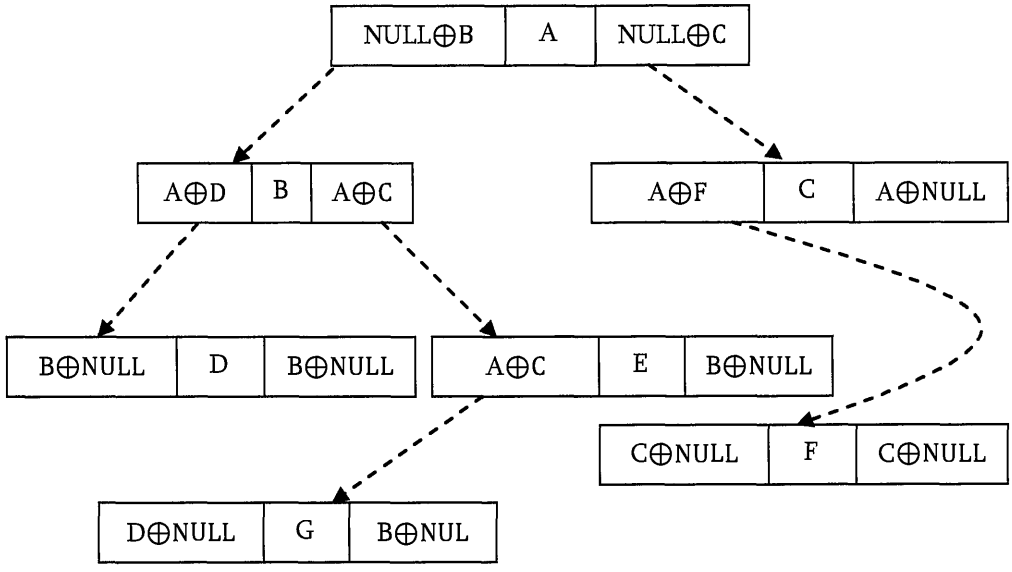
6.10 XOR Trees



This concept is very much similar to *memory efficient doubly linked lists* of *Linked Lists* chapter. Also, like threaded binary trees this representation does not need stacks or queues for traversing the trees. This representation is used for traversing back (to parent) and forth (to children) using ⊕ operation. To represent the same in XOR trees, for each node below are the rules used for representation:

- Each nodes left will have the ⊕ of its parent and its left children.
- Each nodes right will have the ⊕ of its parent and its right children.
- The root nodes parent is NULL and also leaf nodes children are NULL nodes.

Based on the above rules and discussion the tree can be represented as:



The major objective of this presentation is ability to move to parent as well to children. Now, let us see how to use this representation for traversing the tree. For example, if we are at node B and want to move to its parent node A, then we just need to perform ⊕ on its left content with its left child address (we can use right child also for going to parent node).

Similarly, if we want to move to its child (say, left child D) then we have to perform ⊕ on its left content with its parent node address. One important point that we need to understand about this representation is: When we are at node B how do we know the address of its children D? Since the traversal starts at node root node, we can apply ⊕ on roots left content with NULL. As a result we get its left child, B. When we are at B, we can apply ⊕ on its left content with A address.

6.11 Binary Search Trees (BSTs)

Why Binary Search Trees?

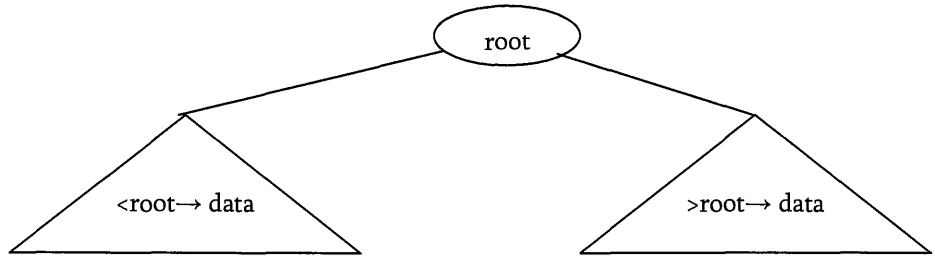
In previous sections we have discussed different tree representations and in all of them we did not impose any restriction on the nodes data. As a result, to search for an element we need to check both in left subtree and also right subtree. Due to this, the worst case complexity of search operation is $O(n)$.

In this section, we will discuss another variant of binary trees: Binary Search Trees (BSTs). As the name suggests, the main use of this representation is for *searching*. In this representation we impose restriction on the kind of data a node can contain. As a result, it reduces the worst case average search operation to $O(logn)$.

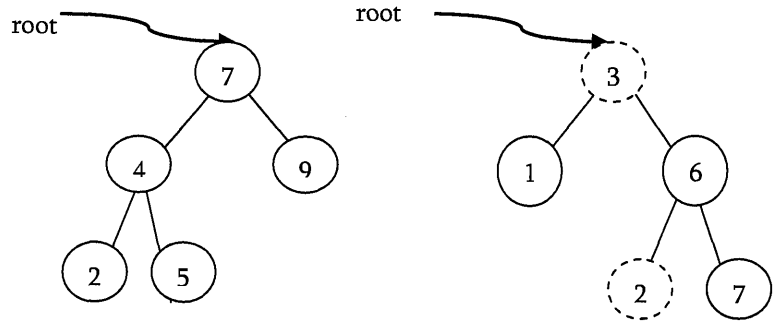
Binary Search Tree Property

In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys less than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.



Example: The left tree is a binary search tree and right tree is not binary search tree (at node 6 it's not satisfying the binary search tree property).



Binary Search Tree Declaration

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

```
struct BinarySearchTreeNode{
        int data;
        struct BinarySearchTreeNode *left;
        struct BinarySearchTreeNode *right;
};
```

Operations on Binary Search Trees

Main operations: The main operations that were supported by binary search trees are:

- Find/ Find Minimum / Find Maximum element in binary search trees
- Inserting an element in binary search trees
- Deleting an element from binary search trees

Auxiliary operations: Checking whether the given tree is a binary search tree or not

- Finding k^{th}-smallest element in tree
- Sorting the elements of binary search tree and many more

Important Notes on Binary Search Trees

- Since root data is always in between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, most of the time, first we process left subtree, process root data and then process right subtree. That means, depending on the problem only the intermediate step (processing root data) changes and we will not touch first and third steps.
- If we are searching for an element and if the left subtree roots data is less than the element we want to search then skip it. Same is the case with right subtree as well. Because of this binary search trees takes less time for searching an element than regular binary trees. In other words, the binary search trees consider only either left or right subtrees for searching an element but not both.

Finding an Element in Binary Search Trees

Find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the data we are searching is same as nodes data then we return current node. If the data we are searching is less than nodes data then search left subtree of current node otherwise search in right subtree of current node. If the data is not present, we end up in a NULL link.

```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
        if( root == NULL )
                return NULL;
        if( data < root→data )
                return Find(root→left, data);
        else if( data > root→data )
                return( Find( root→right, data );
        return root;
}
```

Time Complexity: O(n), in worst case (when BST is a skew tree). Space Complexity: O(n), for recursive stack.

Non recursive version of the above algorithm can be given as:

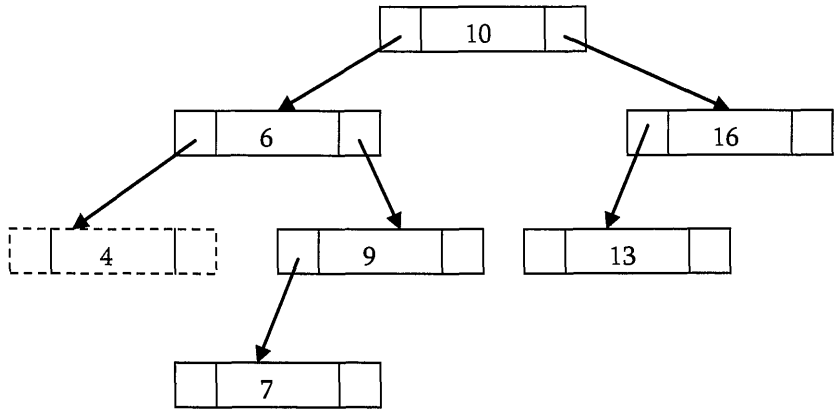
```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
        if( root == NULL ) return NULL;
        while (root) {
                if(data == root→data)
                        return root;
                else if(data > root→data)
                        root = root→right;
                else    root = root→left;
        }
        return NULL;
}
```

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

Finding Minimum Element in Binary Search Trees

In BSTs, the minimum element is the left most node which does not has left child. In the below BST, the minimum element is 4.

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode *root){
        if(root == NULL) return NULL;
        else    if( root→left == NULL )
                        return root;
                else    return FindMin( root→left );
}
```



Time Complexity: O(n), in worst case (when BST is a *left skew* tree). Space Complexity: O(n), for recursive stack.

Non recursive version of the above algorithm can be given as:

```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode * root ) {
        if( root == NULL )
                return NULL;
        while( root→left != NULL )
                root = root→left;
        return root;
}
```

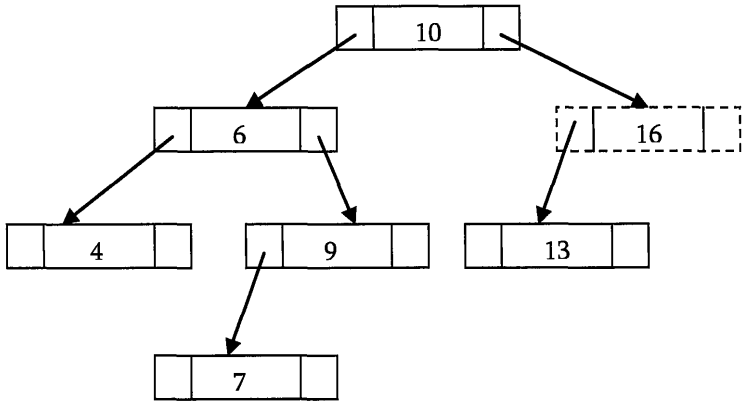
Time Complexity: O(n). Space Complexity: O(1).

Finding Maximum Element in Binary Search Trees

In BSTs, the maximum element is the right most node which does not has right child. In the below BST, the maximum element is 16.

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode *root) {
        if(root == NULL)
                return NULL;
        else    if( root→right == NULL )
                        return root;
                else    return FindMax( root→right );
}
```

Time Complexity: O(n), in worst case (when BST is a *right skew* tree). Space Complexity: O(n), for recursive stack.



Non recursive version of the above algorithm can be given as:

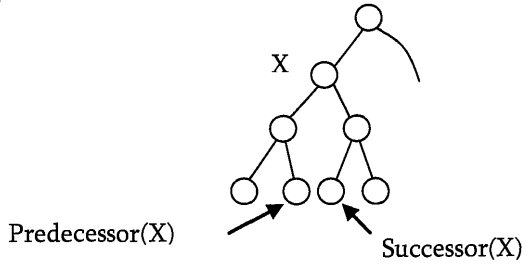
```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode * root ) {
        if( root == NULL )
                return NULL;
        while( root→right != NULL )
                root = root→right;
        return root;
}
```

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

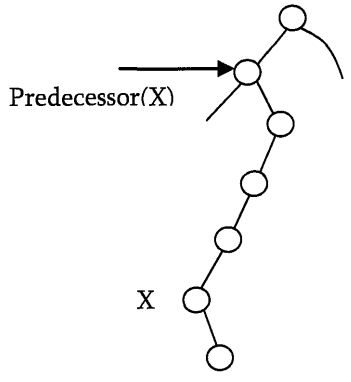
Where is Inorder Predecessor and Successor?

Where is the inorder predecessor and successor of a node X in a binary search tree assuming all keys are distinct?

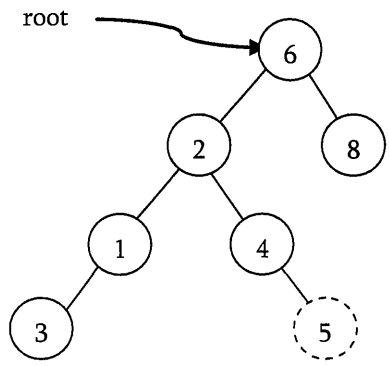
If X has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor the minimum value in its right subtree.



If it does not have a left child a nodes inorder predecessor is its first left ancestor.



Inserting an Element from Binary Search Tree



To insert *data* into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism as that of *find* operation. While finding the location if the *data* is already there then we can simply neglect and come out. Otherwise, insert *data* at the last location on the path traversed. As an example let us consider the following tree. The dotted node indicates the element (5) to be inserted. To insert 5, traverse the tree as using *find* function. At node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.

```
struct BinarySearchTreeNode *Insert(struct BinarySearchTreeNode *root, int data) {
        if( root == NULL ) {
                root = (struct BinarySearchTreeNode *) malloc(sizeof(struct  BinarySearchTreeNode));
                if( root == NULL ) {
                        printf("Memory Error");
                        return;
                }
                else {
                        root→data = data;
                        root→left = root→right = NULL;
                }
        }
        else {  if( data < root→data )
                        root→left = Insert(root→left,  data);
                else if( data > root→data )
                        root→right = Insert(root→right,  data);
        }
        return root;
}
```

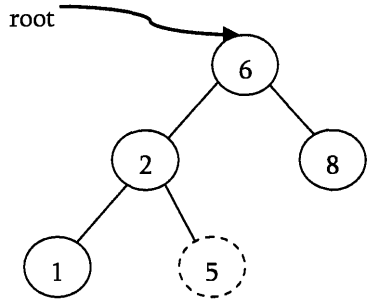
Note: In the above code, after inserting an element in subtrees the tree is returned to its parent. As a result, the complete tree will get updated.

Time Complexity:O(n). Space Complexity:O(n), for recursive stack. For iterative version, space complexity is O(1).

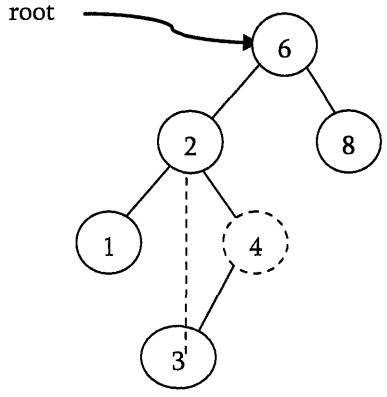
Deleting an Element from Binary Search Tree

The delete operation is little complicated than other operations. This is because the element to be deleted may not be the leaf node. In this operation also, first we need to find the location of the element which we want to delete. Once we have found the node to be deleted, consider the following cases:

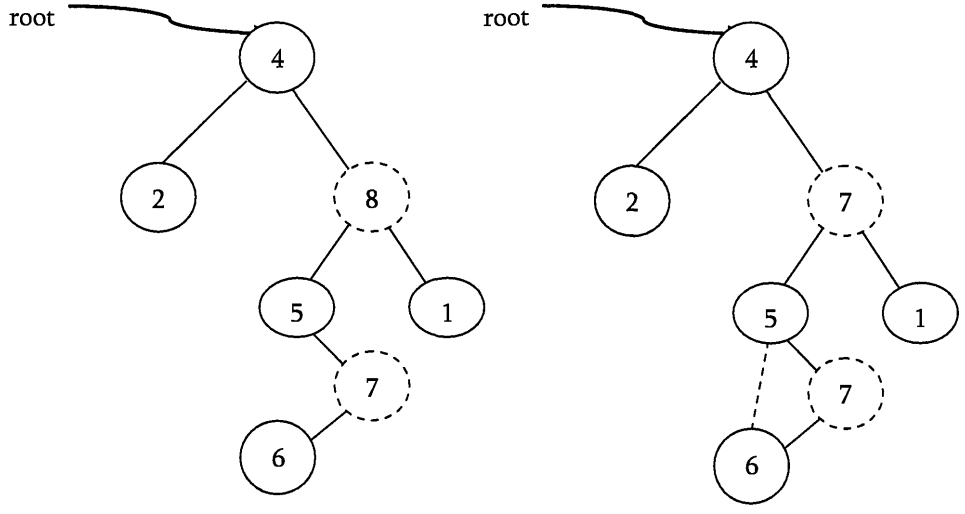
- If the element to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointer NULL. In the below tree to delete 5, set NULL to its parent node 2.



- If the element to be deleted has one child: In this case we just need to send the current nodes child to its parent. In the below tree, to delete 4, 4 left subtree is set to its parent node 2.



- If the element to be deleted has both children: The general strategy is to replace the key of this node with the largest element of the left subtree and recursively delete that node (which is now empty). The largest node in the left subtree cannot have a right child, the second *delete* is an easy one. As an example, let us consider the following tree. In the below tree, to delete 8, it is the right child of root. The key value is 8. It is replaced with the largest key in its left subtree (7), and then that node is deleted as before (second case).



Note: We can replace with minimum element in right subtree also.

```
struct BinarySearchTreeNode *Delete(struct BinarySearchTreeNode *root, int data) {
        struct BinarySearchTreeNode *temp;
        if( root == NULL )
                printf("Element not there in tree");
        else if(data < root→element )
                root→left = Delete(root→left, data);
```

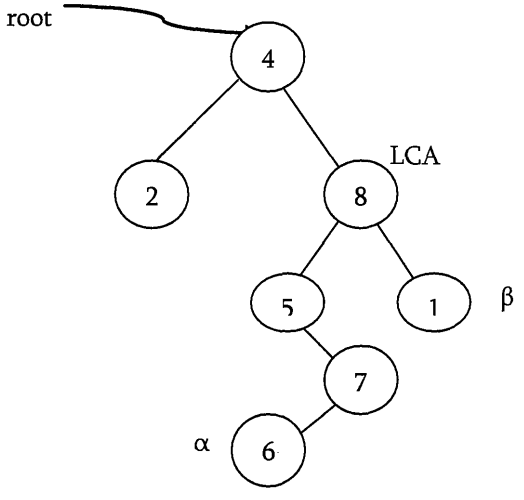
```
        else if(data > root→element )
                root→right = Delete(root→right, data);
        else {  //Found element
                if( root→left && root→right ) {
                        /* Replace with largest in left subtree */
                        temp = FindMax( root→left );
                        root→data = temp→element;
                        root→left = Delete(root→left, root→data);
                }
                else {  /* One child */
                        temp = root;
                        if( root→left == NULL )
                                root = root→right;
                        if( root→right == NULL )
                                root = root→left;
                        free( temp );
                }
        }
        return root;
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$ for recursive stack. For iterative version, space complexity is O(1).

Problems on Binary Search Trees

Problem-46 Given pointers to two nodes in a binary search tree, find lowest common ancestor (*LCA*). Assume that both values already exist in the tree.

Solution:



The main idea of the solution is: while traversing BST from root to bottom, the first node we encounter with value between α and β, i.e., $\alpha < node \rightarrow data < \beta$ is the Least Common Ancestor(LCA) of α and β (where $\alpha < \beta$). So just traverse the BST in pre-order, if we find a node with value in between α and β then that node is the LCA. If its value is greater than both α and β then LCA lies on left side of the node and if its value is smaller than both α and β then LCA lies on right side.

```
struct BinarySearchTreeNode *FindLCA(struct BinarySearchTreeNode *root,
                                     struct BinarySearchTreeNode *α, struct BinarySearchTreeNode * β) {
        while(1) {
                if((α→data < root→data && β→data > root→data) ||
```

```
                    (α→data > root→data && β→data < root→data))
                return root;
            if(α→data < root→data)
                root = root→left;
            else    root = root→right;
        }
}
```

Time complexity: $O(n)$. Space complexity: $O(n)$, for skew trees.

Problem-47 Give an algorithm for finding the shortest path between two nodes in a BST.

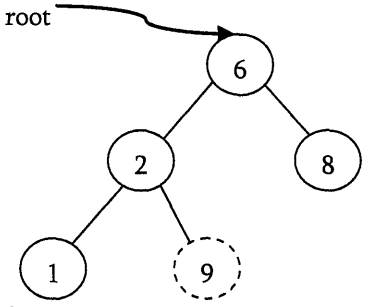
Solution: It's nothing but finding the LCA of two nodes in BST.

Problem-48 Give an algorithm for counting the number of BSTs possible with n nodes.

Solution: This is a DP problem and refer *Dynamic Programming* chapter for algorithm.

Problem-49 Give an algorithm to check whether the given binary tree is a BST or not.

Solution: Consider the following simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node. This approach is wrong as this will return true for below binary tree. Checking only at current node is not enough.




```
int IsBST(struct BinaryTreeNode* root) {
        if(root == NULL) return 1;
        /* false if left is > than root */
        if(root→left != NULL && root→left→data > root→data)
                return 0;
        /* false if right is < than root */
        if(root→right != NULL && root→right→data < root→data)
                return 0;
        /* false if, recursively, the left or right is not a BST */
        if(!IsBST(root→left) || !IsBST(root→right))
                return 0;
        /* passing all that, it's a BST */
        return 1;
}
```

Problem-50 Can we think of getting the correct algorithm?

Solution: For each node, check if max value in left subtree is smaller than the current node data and min value in right subtree greater than the node data. It is assumed that we have helper functions *FindMin*() and *FindMax*() that return the min or max integer value from a non-empty tree.

```
/* Returns true if a binary tree is a binary search tree */
int IsBST(struct BinaryTreeNode* root) {
        if(root == NULL) return 1;
```

```
        /* false if the max of the left is > than root */
        if(root→left != NULL && FindMax(root→left) > root→data)
                return 0;
        /* false if the min of the right is <= than root */
        if(root→right != NULL && FindMin(root→right) < root→data)
                return 0;
        /* false if, recursively, the left or right is not a BST */
        if(!IsBST(root→left) || !IsBST(root→right)) return 0;
        /* passing all that, it's a BST */
        return 1;
}
```

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

Problem-51 Can we improve the complexity of Problem-50?

Solution: Yes. A better solution looks at each node only once. The trick is to write a utility helper function IsBSTUtil(struct BinaryTreeNode* root, int min, int max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX — they narrow from there.

Initial call: IsBST(root, INT_MIN, INT_MAX);

```
int IsBST(struct BinaryTreeNode *root, int min, int max) {
        if(!root)   return 1;
        return (root→data >min && root→data < max &&
                IsBSTUtil(root→left, min, root→data) && IsBSTUtil(root→right, root→data, max));
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-52 Can we further improve the complexity of Problem-50?

Solution: Yes, using inorder traversal. The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, at each node check the condition that its key value should be greater than the key value of its previous visited node. Also, we need to initialize the prev with possible minimum integer value (say, INT_MIN).

```
int prev = INT_MIN;
int IsBST(struct BinaryTreeNode *root, int *prev) {
        if(!root) return 1;
        if(!IsBST(root→left, prev))
                return 0;
        if(root→data < *prev)
                return 0;
        *prev = root→data;
        return IsBST(root→right, prev);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-53 Give an algorithm for converting BST to circular DLL with space complexity O(1).

Solution: Convert left and right subtrees to DLLs and maintain end of those lists. Then, adjust the pointers.

```
struct BinarySearchTreeNode *BST2DLL(struct BinarySearchTreeNode *root,  struct BinarySearchTreeNode **Ltail) {
        struct BinarySearchTreeNode *left, *ltail, *right, *rtail;
        if(!root) {
                * ltail = NULL;
```

```
            return NULL;
        }
        left = BST2DLL(root→left, &ltail);
        right = BST2DLL(root→right, &rtail);
        root→left = ltail;
        root→right = right;
        if(!right)
            * ltail = root;
        else {  right→left = root;
            * ltail = rtail;
        }
        if(!left)
            return root;
        else {  ltail→right = root;
            return left;
        }
}
```

Time Complexity: O(n).

Problem-54 For Problem-53, is there any other way of solving?

Solution: Yes. There is an alternative solution based on divide and conquer method which is quite neat.

```
struct BinarySearchTreeNode *Append(struct BinarySearchTreeNode *a, struct BinarySearchTreeNode *b) {
  struct BinarySearchTreeNode  *aLast, *bLast;
        if (a==NULL)
            return b;
        if (b==NULL)
            return a;
        aLast = a->left;
        bLast = b->left;
        aLast->right = b;
        b->left = aLast;
        bLast->right = a;
        a->left = bLast;
        return a;
}
struct BinarySearchTreeNode* TreeToList(struct BinarySearchTreeNode *root) {
        struct BinarySearchTreeNode *aList, *bList;
        if (root==NULL)
            return NULL;
        aList = TreeToList(root->left);
        bList = TreeToList(root->right);
        root->left = root;
        root->right = root;
        aList = Append(aList, root);
        aList = Append(aList, bList);
        return(aList);
}
```

Time Complexity: O(n).

Problem-55 Given a sorted doubly linked list, give an algorithm for converting it to balanced binary search tree.

Solution: Find the middle node and adjust the pointers.

```
struct DLLNode * DLLtoBalancedBST(struct DLLNode *head) {
        struct DLLNode *temp, *p, *q;
        if( !head || !head→next)
                return head;
        temp = FindMiddleNode(head);
        p = head;
        while(p→next != temp)
                p = p→next;
        p→next = NULL;
        q = temp→next;
        temp→next = NULL;
        temp→prev = DLLtoBalancedBST(head);
        temp→next = DLLtoBalancedBST(q);
        return temp;
}
```

Time Complexity: $2T(n/2) + O(n)$ [for finding the middle node] = $O(nlogn)$.

Note: For *FindMiddleNode* function refer *Linked Lists* chapter.

Problem-56 Given a sorted array, give an algorithm for converting the array to BST.

Solution: If we have to choose an array element to be the root of a balanced BST, which element we should pick? The root of a balanced BST should be the middle element from the sorted array. We would pick the middle element from the sorted array in each iteration. We then create a node in the tree initialized with this element. After the element is chosen, what is left? Could you identify the sub-problems within the problem?

There are two arrays left — The one on its left and the one on its right. These two arrays are the sub-problems of the original problem, since both of them are sorted. Furthermore, they are subtrees of the current node's left and right child.

The code below creates a balanced BST from the sorted array in $O(n)$ time (n is the number of elements in the array). Compare how similar the code is to a binary search algorithm. Both are using the divide and conquer methodology.

```
struct BinaryTreeNode *BuildBST(int A[], int left, int right) {
        struct BinaryTreeNode *newNode;
        int mid;
        if(left > right)
                return NULL;
        newNode = (struct BinaryTreeNode *)malloc(sizeof(struct BinaryTreeNode));
        if(!newNode) {
                printf("Memory Error");
                return;
        }
        if(left == right) {
                newNode→data = A[left];
                newNode→left = newNode→right = NULL;
        }
        else {  mid = left + (right-left)/ 2;
                newNode→data = A[mid];
                newNode→left = BuildBST(A, left, mid - 1);
                newNode→right = BuildBST(A, mid + 1, right);
        }
        return newNode;
```

}
Time Complexity: O(n). Space Complexity: O(n), for stack space.

Problem-57 Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Solution: A naive way is to apply the Problem-55 solution directly. In each recursive call, we would have to traverse half of the list's length to find the middle element. The run time complexity is clearly O($nlogn$), where n is the total number of elements in the list. This is because each level of recursive call requires a total of $n/2$ traversal steps in the list, and there are a total of $logn$ number of levels (ie, the height of the balanced tree).

Problem-58 For Problem-57, can we improve the complexity?

Solution: Hint: How about inserting nodes following the list's order? If we can achieve this, we no longer need to find the middle element, as we are able to traverse the list while inserting nodes to the tree.

Best Solution: As usual, the best solution requires us to think from another perspective. In other words, we no longer create nodes in the tree using the top-down approach. Create nodes bottom-up, and assign them to its parents. The bottom-up approach enables us to access the list in its order while creating nodes [42].

Isn't the bottom-up approach neat? Each time we are stucked with the top-down approach, give bottom-up a try. Although bottom-up approach is not the most natural way we think, it is extremely helpful in some cases. However, we should prefer top-down instead of bottom-up in general, since the latter is more difficult to verify in correctness.

Below is the code for converting a singly linked list to a balanced BST. Please note that the algorithm requires the list's length to be passed in as the function's parameters. The list's length could be found in O(n) time by traversing the entire list's once. The recursive calls traverse the list and create tree's nodes by the list's order, which also takes O(n) time. Therefore, the overall run time complexity is still O(n).

```
struct BinaryTreeNode* SortedListToBST(struct ListNode *& list, int start, int end) {
        if(start > end)
                return NULL;
        // same as (start+end)/2, avoids overflow
        int mid = start + (end - start) / 2;
        struct BinaryTreeNode *leftChild = SortedListToBST(list, start, mid-1);
        struct BinaryTreeNode  * parent;
        parent = (struct BinaryTreeNode  *)malloc(sizeof(struct BinaryTreeNode));
        if(!parent) {
                printf("Memory Error");
                return;
        }
        parent→data=list→data;
        parent→left = leftChild;
        list = list→next;
        parent→right = SortedListToBST(list, mid+1, end);
        return parent;
}
struct BinaryTreeNode * SortedListToBST(struct ListNode *head, int n) {
        return SortedListToBST(head, 0, n-1);
}
```

Problem-59 Give an algorithm for finding the k^{th} smallest element in BST.

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the number of elements visited.

```
struct BinarySearchTreeNode *kthSmallestInBST(struct BinarySearchTreeNode *root, int k, int *count){
        if(!root) return NULL;
        struct BinarySearchTreeNode *left = kthSmallestInBST(root→left, k, count);
        if( left ) return left;
        if(++count == k)
                return root;
        return kthSmallestInBST(root→right, k, count);
}
```

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

Problem-60 **Floor and ceiling:** If a given key is less than the key at the root of a BST then floor of key (the largest key in the BST less than or equal to key) must be in the left subtree. If key is greater than the key at the root then floor of key could be in the right subtree, but only if there is a key smaller than or equal to key in the right subtree; if not (or if key is equal to the key at the root) then the key at the root is the floor of key. Finding the ceiling is similar with interchanging right and left.For example, if the sorted with input array is {1, 2, 8, 10, 10, 12, 19}, then

For $x = 0$: floor doesn't exist in array, ceil = 1, For $x = 1$: floor = 1, ceil = 1

For $x = 5$: floor = 2, ceil = 8, For $x = 20$: floor = 19, ceil doesn't exist in array

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the values being visited. If the roots data is greater than the given value then return the previous value which we have maintained during traversal. If the roots data is equal to the given data then return root data.

```
struct BinaryTreeNode *FloorInBST(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *prev=NULL;
        return FloorInBSTUtil(root, prev, data);
}
struct BinaryTreeNode  *FloorInBSTUtil(struct BinaryTreeNode *root, struct BinaryTreeNode *prev, int data){
        if(!root)
                return NULL;
        if(!FloorInBSTUtil(root→left, prev, data))
                return 0;
        if(root→data == data)
                return root;
        if(root→data > data)
                return prev;
        prev = root;
        return FloorInBSTUtil(root→right, prev, data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

For ceiling, we just need to call the right subtree first and then followed by left subtree.

```
struct BinaryTreeNode *CeilingInBST(struct BinaryTreeNode *root, int data){
        struct BinaryTreeNode *prev=NULL;
        return CeilingInBSTUtil(root, prev, data);
}
struct BinaryTreeNode  *CeilingInBSTUtil(struct BinaryTreeNode *root, struct BinaryTreeNode *prev, int data){
        if(!root)
                return NULL;
        if(!CeilingInBSTUtil(root→right, prev, data))
                return 0;
        if(root→data == data) return root;
```

```
        if(root→data < data)  return prev;
        prev = root;
        return CeilingInBSTUtil(root→left, prev, data);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-61 Give an algorithm for finding the union and intersection of BSTs. Assume parent pointers are available (say threaded binary trees). Also, assume the lengths of two BSTs are m and n respectively.

Solution: If parent pointers are available then the problem is same as merging of two sorted lists. This is because if we call inorder successor each time we get the next highest element. It's just a matter of which InorderSuccessor to call.

Time Complexity: $O(m + n)$. Space complexity: $O(1)$.

Problem-62 For Problem-61, what if parent pointers are not available?

Solution: If parent pointers are not available then, one possibility is converting the BSTs to linked lists and then merging.

1 Convert both the BSTs into sorted doubly linked lists in $O(n + m)$ time. This produces 2 sorted lists.
2 Merge the two double linked lists into one and also maintain the count of total elements in $O(n + m)$ time.
3 Convert the sorted doubly linked list into height balanced tree in $O(n + m)$ time.

Problem-63 For Problem-61, is there any alternative way of solving the problem?

Solution: Yes, using inorder traversal.

- Perform inorder traversal on one of the BST.
- While performing the traversal store them in table (hash table).
- After completion of the traversal of first BST, start traversal of the second BST and compare them with hash table contents.

Time Complexity: $O(m + n)$. Space Complexity: $O(Max(m, n))$.

Problem-64 Given a BST and two numbers $K1$ and $K2$, give an algorithm for printing all the elements of BST in the range $K1$ and $K2$.

Solution:

```
void RangePrinter(struct BinarySearchTreeNode *root, int  K1, int K2) {
        if(root == NULL)
                return;
        if(root→data >= K1)
                RangePrinter(root→left, K1, K2);
        if(root→data >= K1 && root→data <= K2)
                printf("%d", root→data);
        if(root→data <= K2)
                RangePrinter(root→right, K1, K2);
}
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for stack space.

Problem-65 For Problem-64, is there any alternative way of solving the problem?

Solution: We can use level order traversal: while adding the elements to queue check for the range.

```
void  RangeSeachLevelOrder(struct BinarySearchTreeNode *root, int  K1, int K2){
        struct BinarySearchTreeNode *temp;
        struct Queue *Q = CreateQueue();
        if(!root)
                return NULL;
        Q = EnQueue(Q, root);
```

```
        while(!IsEmptyQueue(Q)) {
                temp=DeQueue(Q);
                if(temp→data >= K1 && temp→data <= K2)
                        printf("%d",temp→data);
                if(temp→left && temp→data >= K1)
                        EnQueue(Q, temp→left);
                if(temp→right && temp→data <= K2)
                        EnQueue(Q, temp→right);
        }
        DeleteQueue(Q);
        return NULL;
}
```

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

Problem-66 For Problem-64, can we still think of alternative way for solving the problem?

Solution: First locate $K1$ with normal binary search and after that use InOrder successor until we encounter $K2$. For algorithm, refer problems section of threaded binary trees.

Problem-67 Given root of a Binary Search tree, trim the tree, so that all elements in the new tree returned are between the inputs A and B.

Solution: It's just another way of asking the Problem-64.

Problem-68 Given two BSTs, check whether the elements of them are same or not. For example: two BSTs with data 10 5 20 15 30 and 10 20 15 30 5 should return true and the dataset with 10 5 20 15 30 and 10 15 30 20 5 should return false. **Note:** BSTs data can be in any order.

Solution: One simple way is performing a traversal on first tree and storing its data in hash table. As a second step perform traversal on second tree and check whether that data is already there in hash table or not. During the traversal of second tree if we find any mismatch return false.

Time Complexity: $O(max(m,n))$, where m and n are the number of elements in first and second BST. Space Complexity: $O(max(m,n))$. This depends on the size of the first tree.

Problem-69 For Problem-68, can we reduce the time complexity?

Solution: Instead of performing the traversals one after the other, we can perform $in-order$ traversal of both the trees in parallel. Since the $in-order$ traversal gives the sorted list, we can check whether both the trees are generating the same sequence or not.

Time Complexity: $O(max(m,n))$. Space Complexity: $O(1)$. This depends on the size of the first tree.

Problem-70 For the key values $1 \ldots n$, how many structurally unique BSTs are possible that store those keys.

Solution: Strategy: consider that each value could be the root. Recursively find the size of the left and right subtrees.

```
int CountTrees(int n) {
        if (n <= 1)  return 1;
        else {  // there will be one value at the root, with whatever remains on the left and right
                // each forming their own subtrees.Iterate through all the values that could be the root...
                int sum = 0;
                int left, right, root;
                for (root=1; root<=n; root++) {
                        left = CountTrees(root - 1);
                        right = CountTrees(numKeys - root);
                        // number of possible trees with this root == left*right
                        sum += left*right;
```

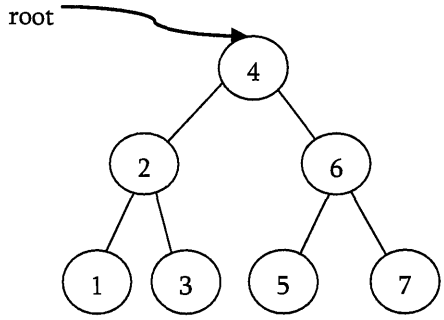
```
            }
            return(sum);
        }
}
```

6.12 Balanced Binary Search Trees

In earlier sections we have seen different trees whose worst case complexity is $O(n)$, where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to $O(logn)$ by imposing restrictions on the heights. In general, the height balanced trees are represented with $HB(k)$, where k is the difference between left subtree height and right subtree height. Sometimes k is called balance factor.

Complete Balanced Binary Search Trees

In $HB(k)$, if $k = 0$ (if balance factor is zero), then we call such binary search trees as $full$ balanced binary search trees. That means, in $HB(0)$ binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example,



Note: For constructing $HB(0)$ tree refer problems section.

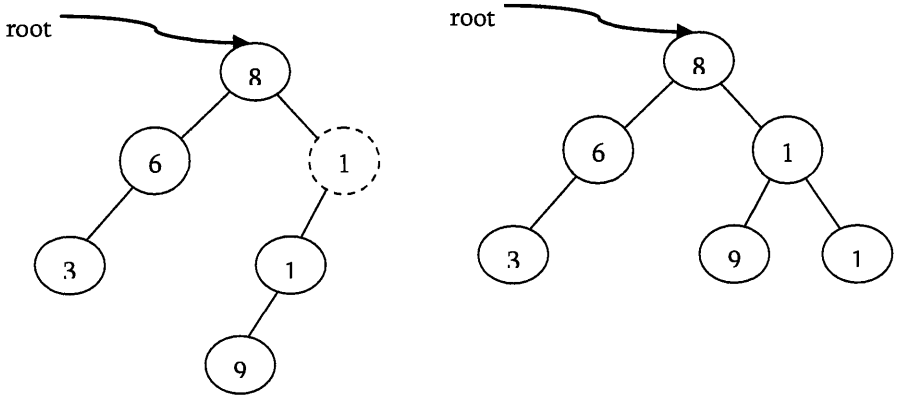
6.13 AVL (Adelson-Velskii and Landis) Trees

In $HB(k)$, if $k = 1$ (if balance factor is one), such binary search tree is called an AVL tree. That means an AVL tree is a binary search tree with a *balance* condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

A binary tree is said to be an AVL tree, if:

- It is a binary search tree, and
- For any node X, the height of left subtree of X and height of right subtree of X differ by at most 1.



As an example among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree

For simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get minimum number of nodes with height h, we should fill the tree with as minimum nodes as possible. That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$. As a result, the minimum number of nodes with height h is:

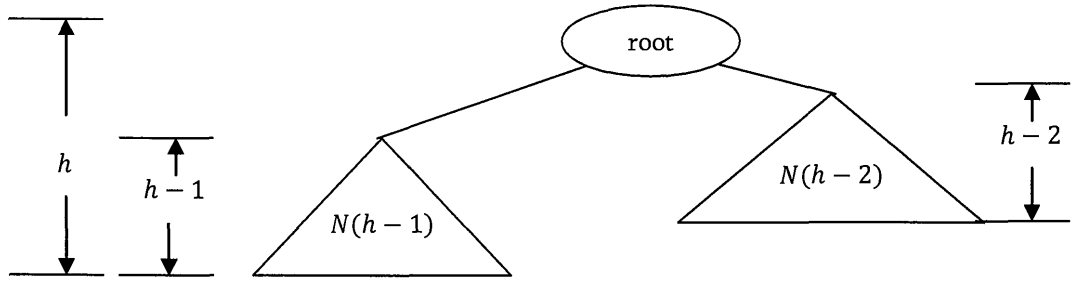
$$N(h) = N(h-1) + N(h-2) + 1$$

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h-2$.
- In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Rightarrow h = 1.44 logn \approx O(logn)$$



Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is $O(logn)$. Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h-1$. As a result, we get

$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = O(2^h) \Rightarrow h = logn \approx O(logn)$$

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is $O(logn)$.

AVL Tree Declaration

Since AVL tree is a BST, the declaration of AVL is similar to that of BST. But just to simplify the operations, we include the height also as part of declaration.

```
struct AVLTreeNode{
        struct AVLTreeNode *left;
        int data;
        struct AVLTreeNode *right;
        int height;
};
```

Finding Height of an AVL tree

```
int Height(struct AVLTreeNode *root ){
        if( !root)  return -1;
        else return root→height;
}
```

Time Complexity: O(1).

Rotations

When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using single rotations or double rotations. Since an insertion/deletion involves

adding/deleting a single node, this can only increase/decrease the height of some subtree by 1. So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. That means, we need to apply the rotations for the node X.

Observation: One important observation is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered because only those nodes have their subtrees altered. To restore the AVL tree property, we start at the insertion point and keep going to root of the tree. While moving to root, we need to consider the first node whichever is not satisfying the AVL property. From that node onwards every node on the path to root will have the issue. Also, if we fix the issue for that first node, then all other nodes on the path to root will automatically satisfy the AVL tree property. That means we always need to care for the first node whichever is not satisfying the AVL property on the path from insertion point to root and fix it.

Types of Violations

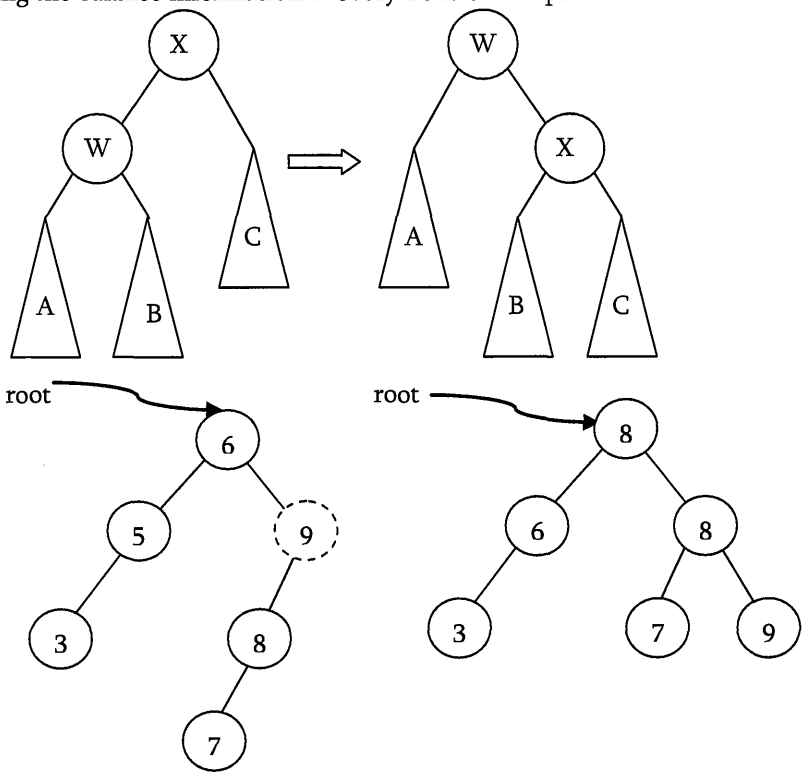
Let us assume the node that must be rebalanced is X. Since any node has at most two children, and a height imbalance requires that $X's$ two subtrees' heights differ by two. We can easily observe that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of X.
2. An insertion into the right subtree of the left child of X.
3. An insertion into the left subtree of the right child of X.
4. An insertion into the right subtree of the right child of X.

Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations (needs two single rotations).

Single Rotations

Left Left Rotation (LL Rotation) [Case-1]: In the below case, at node X, the AVL tree property is not satisfying. As discussed earlier, rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

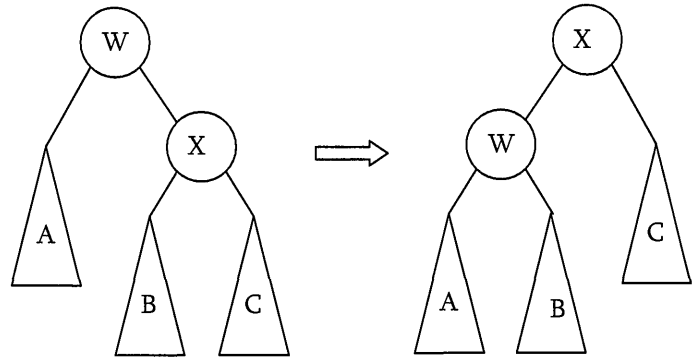


For example, in above figure, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

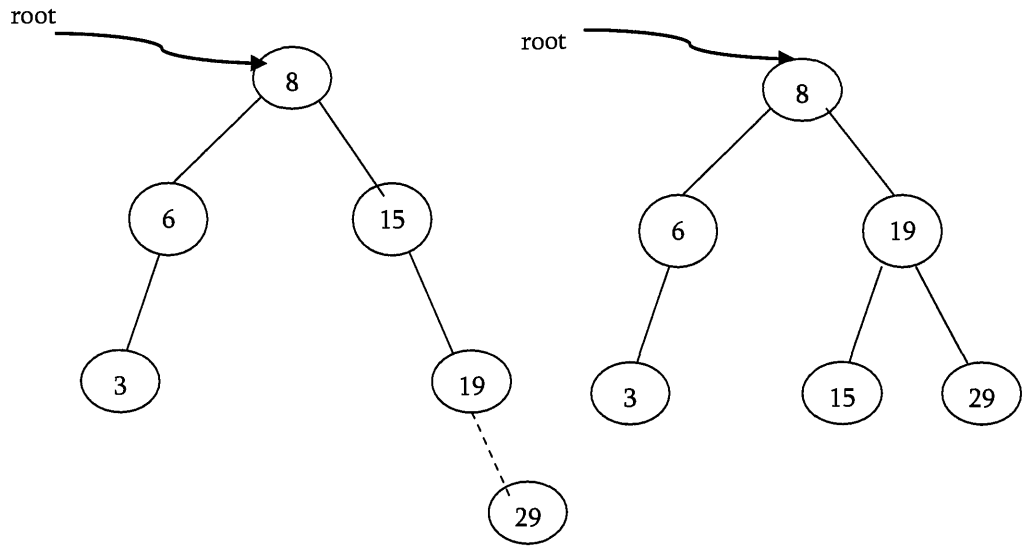
```
struct AVLTreeNode *SingleRotateLeft(struct AVLTreeNode *X ){
        struct AVLTreeNode *W = X→left;
        X→left = W→right;
        W→right = X;
        X→height = max( Height(X→left), Height(X→right) ) + 1;
        W→height = max( Height(W→left), X→height ) + 1;
        return W; /* New root */
}
```

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

Right Right Rotation (RR Rotation) [Case-4]: In this case, the node *X* is not satisfying the AVL tree property.



For example, in above figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

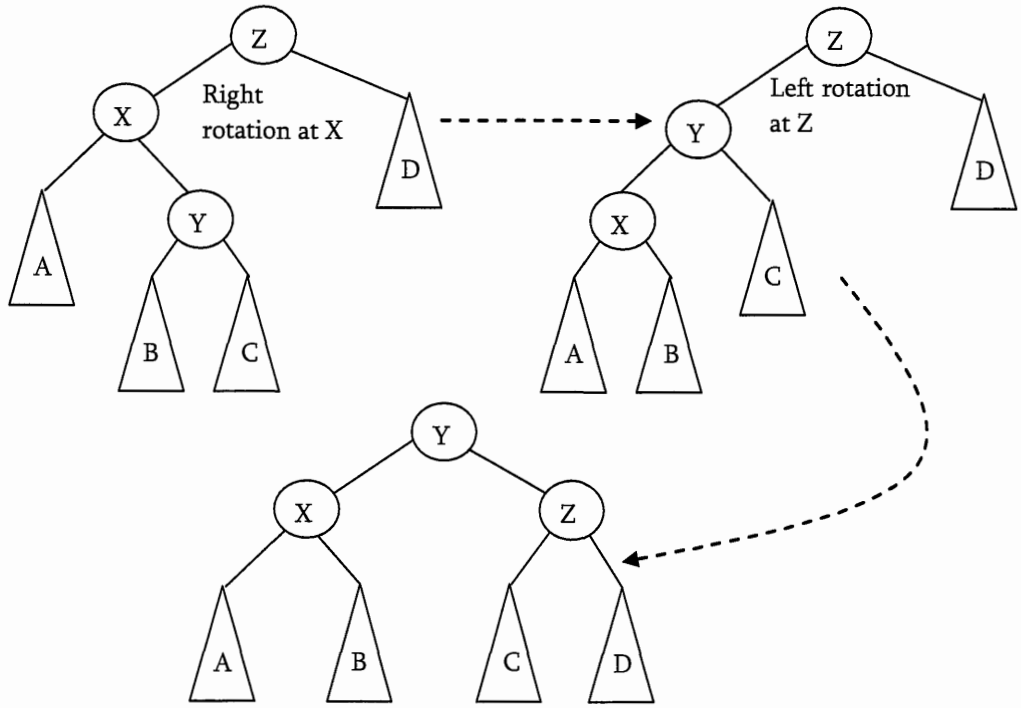



```
struct AVLTreeNode *SingleRotateRight(struct AVLTreeNode *W ) {
        struct AVLTreeNode *X  = W→right;
        W→right = X→left;
        X→left = W;
        W→height = max( Height(W→right), Height(W→left) ) + 1;
        X→height = max( Height(X→right), W→height) + 1;
        return X;
```

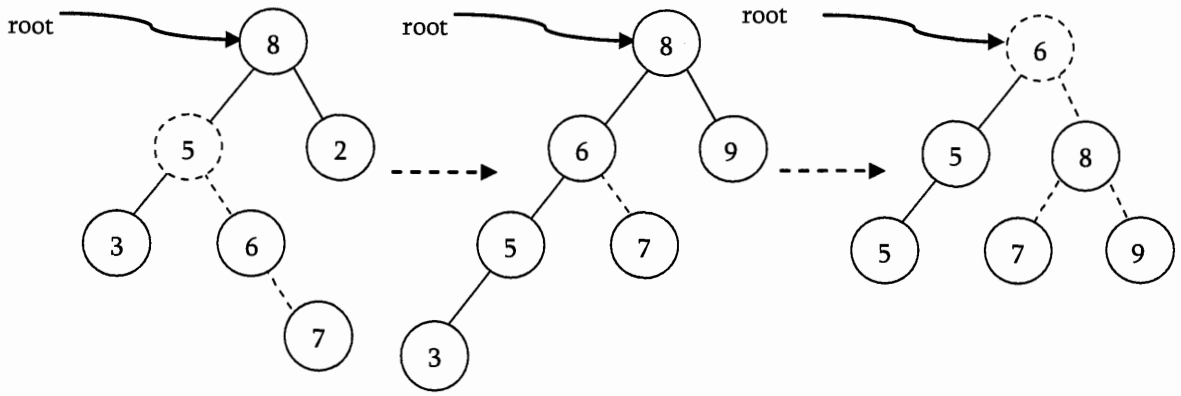
}
Time Complexity: O(1). Space Complexity: O(1).

Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



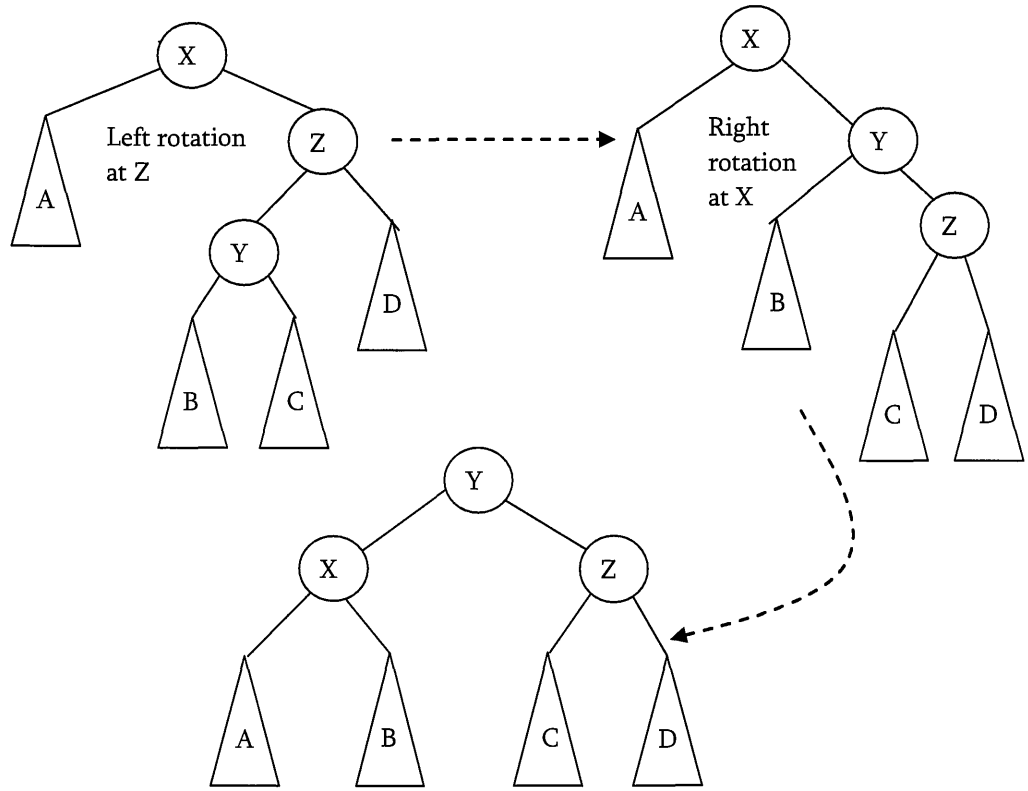
As an example, let us consider the following tree: Insertion of 7 is creating the case-2 scenario and right side tree is the one after double rotation.



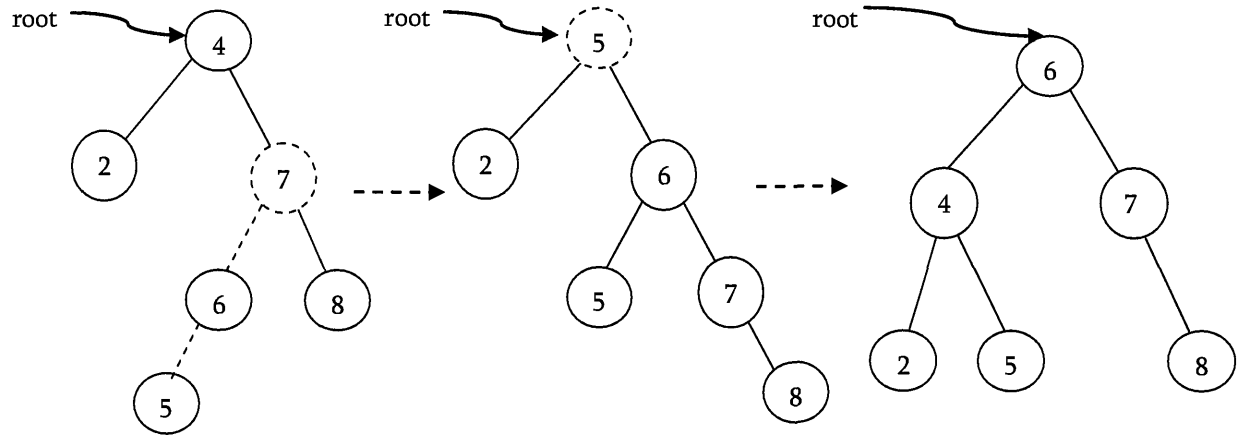
Code for left-right double rotation can be given as:

```
struct AVLTreeNode *DoubleRotatewithLeft( struct AVLTreeNode *Z ){
        Z→left = SingleRotateRight( Z→left );
        return SingleRotateLeft(Z);
}
```

Right Left Rotation (RL Rotation) [Case-3]: As similar to case-2, we need to perform two rotations for fixing this scenario.



As an example, let us consider the following tree: Insertion of 6 is creating the case-3 scenario and right side tree is the one after double rotation.



Insertion into an AVL tree

Insertion in AVL tree is very much similar to BST insertion. After inserting the element, we just need to check whether there is any height imbalance. If there is any imbalance, call the appropriate rotation functions.

```
struct AVLTreeNode *Insert( struct AVLTreeNode *root, struct AVLTreeNode *parent, int data){
        if( !root) {
                root = (struct AVLTreeNode*) malloc(sizeof (struct AVLTreeNode*) );
                if(!root) { printf("Memory Error");
                        return;
                }
                else {  root→data = data;
                        root→height = 0;
                        root→left = root→right = NULL;
```

```
        }
    }
    else if( data < root→data ) {
        root→left = Insert( root→left, root, data );
        if( ( Height( root→left ) - Height( root→right ) ) == 2 ) {
            if( data < root→left→data )
                root = SingleRotateLeft( root );
            else    root = DoubleRotateLeft( root );
        }
    }
    else if( data > root→data ) {
        root→right = Insert( root→right, root, data );
        if( ( Height( root→right ) - Height( root→left ) ) == 2 ) {
            if( data < root→right→data )
                root = SingleRotateRight( root );
            else    root = DoubleRotateRight( root );
        }
    }
    /* Else data is in the tree already. We'll do nothing */
    root→height = max( Height(root→left), Height(root→right) ) + 1;
    return root;
}
```

Time Complexity: O(n). Space Complexity: O($logn$).

Problems on AVL Trees

Problem-71 Given a height h, give an algorithm for generating the $HB(0)$.

Solution: As we have discussed, $HB(0)$ is nothing but generating full binary tree. In full binary tree the number of nodes with height h are: $2^{h+1} - 1$ (let us assume that the height of a tree with one node is 0). As a result the nodes can be numbered as: 1 to $2^{h+1} - 1$.

```
struct BinarySearchTreeNode *BuildHB0(int h){
    struct BinarySearchTreeNode *temp;
    if(h == 0)
        return NULL;
    temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
    temp→left = BuildHB0 (h-1);
    temp→data = count++;                    //assume count is a global variable
    temp→right = BuildHB0 (h-1);
    return temp;
}
```

Time Complexity: O(n). Space Complexity: O($logn$), where $logn$ indicates the maximum stack size which is equal to height of tree.

Problem-72 Is there any alternative way of solving Problem-71?

Solution: Yes, we can solve following Mergesort logic. That means, instead of working with height, we can take the range. With this approach we do not need any global counter to be maintained.

```
Struct BinarySearchTreeNode *BuildHB0(int l, int r){
    struct BinarySearchTreeNode *temp;
    int mid = l + (r-l)/2;
```

```
    if( l > r)
        return NULL;
    temp = (struct BinarySearchTreeNode *) malloc (sizeof(struct BinarySearchTreeNode));
    temp→data = mid;
    temp→left = BuildHB0(l, mid-1);
    temp→right = BuildHB0(mid+1, r);
    return temp;
}
```

The initial call to $BuildHB0$ function could be: $BuildHB0(1, 1 \ll h)$. $1 \ll h$ does the shift operation for calculating the $2^{h+1} - 1$.

Time Complexity: $O(n)$. Space Complexity: $O(logn)$. Where $logn$ indicates maximum stack size which is equal to height of the tree.

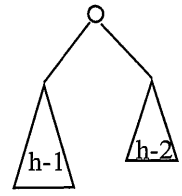
Problem-73 Construct minimal AVL trees of height 0, 1, 2, 3, 4, and 5. What is the number of nodes in a minimal AVL tree of height 6?

Solution Let $N(h)$ be the number of nodes in a minimal AVL tree with height h.

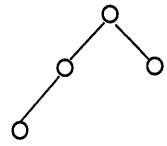
$N(0) = 1$

$N(1) = 2$

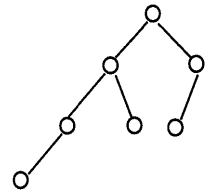
$N(h) = 1 + N(h-1) + N(h-2)$



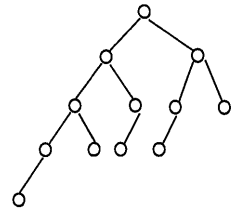
$N(2) = 1 + N(1) + N(0)$
$= 1 + 2 + 1 = 4$



$N(3) = 1 + N(2) + N(1)$
$= 1 + 4 + 2 = 7$



$N(4) = 1 + N(3) + N(2)$
$= 1 + 7 + 4 = 12$



$N(5) = 1 + N(4) + N(3)$
$= 1 + 12 + 7 = 20$

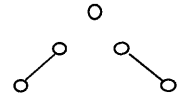
...

Problem-74 For the Problem-71 how many different shapes of a minimal AVL tree of height h can have?

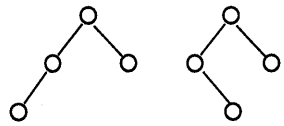
Solution: Let $NS(h)$ be the number of different shapes of a minimal AVL tree of height h.

$NS(0) = 1$

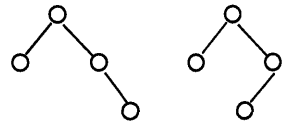
$NS(1) = 2$



$$NS(2) = 2 * NS(1) * NS(0)$$
$$= 2 * 2 * 1 = 4$$

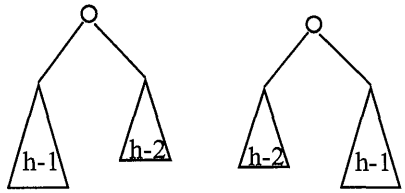


$$NS(3) = 2 * NS(2) * NS(1)$$
$$= 2 * 4 * 1 = 8$$



...

$$NS(h) = 2 * NS(h-1) * NS(h-2)$$



Problem-75 Given a binary search tree check whether the tree is an AVL tree or not?

Solution: Let us assume that *IsAVL* is the function which checks whether the given binary search tree is an AVL tree or not. *IsAVL* returns -1 if the tree is not an AVL tree. During the checks each node sends height of it to their parent.

```
int IsAVL(struct BinarySearchTreeNode *root){
        int left, right;
        if(!root)
            return 0;
        left = IsAVL(root→left);
        if(left == -1)
                return left;
        right = IsAVL(root→right);
        if(right == -1)
                return right;
        if(abs(left-right)>1)
                return -1;
        return Max(left, right)+1;
}
```

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

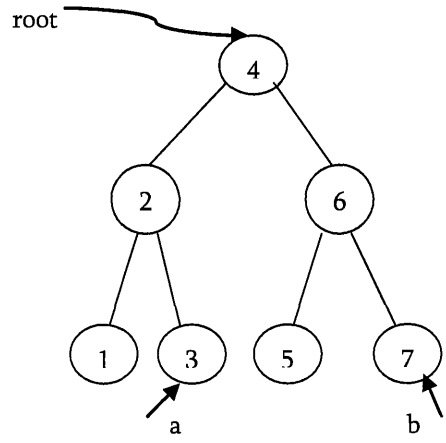
Problem-76 Given a height h, give an algorithm to generate an AVL tree with min number of nodes.

Solution: To get minimum number of nodes, fill one level with $h-1$ and other with $h-2$.

```
struct AVLTreeNode *GenerateAVLTree(int h){
        struct AVLTreeNode *temp;
        if(h == 0)
                return NULL;
        temp = (struct AVLTreeNode *)malloc (sizeof(struct AVLTreeNode));
        temp→left = GenerateAVLTree(h-1);
        temp→data = count++; //assume count is a global variable
        temp→right = GenerateAVLTree(h-2);
        temp→height = temp→left→height+1; // or temp→height = h;
        return temp;
}
```

Problem-77 Given an AVL tree with n integer items and two integers a and b, where a and b can be any integers with $a <= b$. Implement an algorithm to count the number of nodes in the range $[a, b]$.

Solution:



The idea is to make use of the recursive property of binary search trees. There are three cases to consider, whether the current node is in the range $[a, b]$, on the left side of the range $[a, b]$ or on the right side of the range $[a, b]$. Only subtrees that possibly contain the nodes will be processed under each of the three cases.

```
int RangeCount(struct AVLNode *root, int a, int b) {
        if(root == NULL)
                return 0;
        else if(root→data > b)
                return RangeCount(curr→left, a, b);
        else if(root→data < a)
                return RangeCount(root→right, a, b);
        else if(root→data >= a && root→data <= b)
                return RangeCount(root→left, a, b) + RangeCount(root→right, a, b) + 1;
}
```

The complexity is similar to $in - order$ traversal of the tree but skipping left or right sub-trees when they do not contain any answers. So in the worst case, if the range covers all the nodes in the tree, we need to traverse all the n nodes to get the answer. The worst time complexity is therefore $O(n)$.

If the range is small, which only covers few elements in a small subtree at the bottom of the tree, the time complexity will be $O(h) = O(logn)$, where h is the height of the tree. This is because only a single path is traversed to reach the small subtree at the bottom and many higher level subtrees have been pruned along the way.

Note: Refer similar problem in BST.

6.14 Other Variations in Trees

In this section, let us enumerate the other possible representations of trees. In the earlier sections, we have seen AVL trees which is a binary search tree (BST) with balancing property. Now, let us see few more balanced binary search trees: Red-Black Trees and Splay Trees.

Red-Black Trees

In red-black trees each node is associated with extra attribute: the color, which is either red or black. To get logarithmic complexity we impose the following restrictions.

Definition: A red-black tree is a binary search tree that satisfies the following properties:

- Root Property: the root is black
- External Property: every leaf is black
- Internal Property: the children of a red node are black
- Depth Property: all the leaves have the same black

As similar to AVL trees, if the Red-black tree becomes imbalanced then we perform rotations to reinforce the balancing property. With Red-black trees, we can perform the following operations in $O(logn)$ in worst case, where n is the number of nodes in the trees.

- Insertion, Deletion
- Finding predecessor, successor
- Finding minimum, maximum

Splay Trees

Splay-trees are BSTs with self-adjusting property. Another interesting property of splay-trees is: starting with empty tree, any sequence of K operations with maximum of n nodes takes $O(Klogn)$ time complexity in worst case.

Splay trees are easier to program and also ensures faster access to recently accessed items. As similar to AVL and Red-Black trees, at any point if the splay tree becomes imbalanced then we perform rotations to reinforce the balancing property.

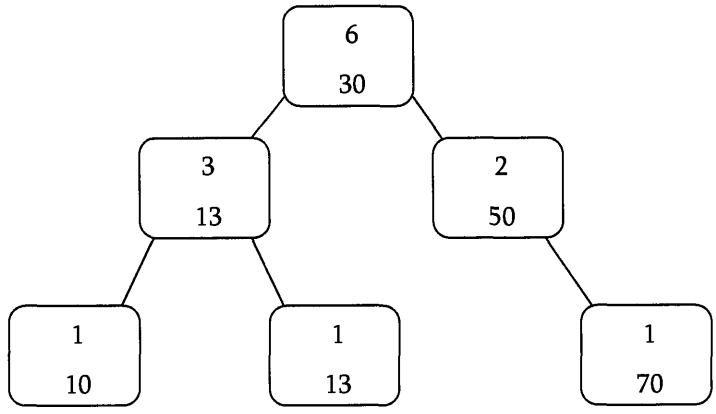
Splay-trees cannot guarantee the $O(logn)$ complexity in worst case. But it gives amortized $O(logn)$ complexity. Even though individual operations can be expensive, any sequence of operations gets the complexity of logarithmic behavior. One operation may take more time (a single operation may take $O(n)$ time) but the subsequent operations may not take worst case complexity and on the average *per operation* complexity is $O(logn)$.

Augmented Trees

In earlier sections, we have seen the problems like finding K^{th} –smallest element in the tree and many other similar problems. For all those problems the worst complexity is $O(n)$, where n is the number of nodes in the tree. To perform such operations in $O(logn)$ augmented trees are useful. In these trees, extra information is added to each node and that extra data depends on the problem we are trying to solve. For example, to find K^{th} –smallest in binary search tree, let us see how augmented trees solves the problem. Let us assume that we are using Red-Black trees as balanced BST (or any balanced BST) and augment the size information in the nodes data. For a given node X in Red-Black tree with a field $size(X)$ equal to the number of nodes in the subtree and can be calculated as:

$$size(X) = size(X \rightarrow left) + size(X \rightarrow right)) + 1$$

Example: With the extra size information, the augmented tree will look like:



Kth-smallest operation can be defined as:

```
struct BinarySearcTreeNode *KthSmallest (struct BinarySearcTreeNode *X, int K) {
        int r = size(X→left) + 1;
        if(K == r)
                return X;
        if(K < r)
                return KthSmallest (X→left, K);
        if(K > r)
```

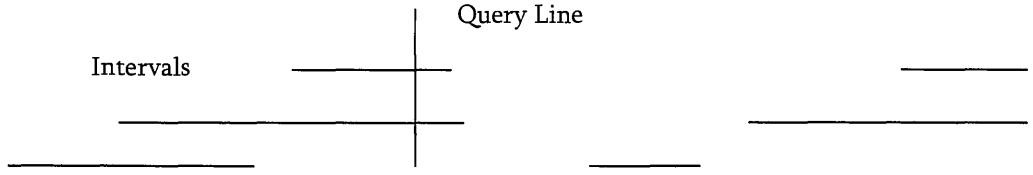
```
        return KthSmallest (X→right, K-r);
}
```

Time Complexity: O($logn$). Space Complexity: O($logn$).

Interval Trees

Interval trees are also binary search trees and stores interval information in the node structure. That means, we maintain a set of n intervals $[i_1, i_2]$ such that one of the intervals containing a query point Q (if any) can be found efficiently. Interval trees are used for performing range queries efficiently.

Example: Given a set of intervals: S = {[2-5], [6-7], [6-10], [8-9], [12-15], [15-23], [25-30]}. A query with $Q = 9$ returns $[6, 10]$ or $[8, 9]$ (assume these are the intervals which contains 9 among all the intervals). A query with $Q = 23$ returns $[15, 23]$.

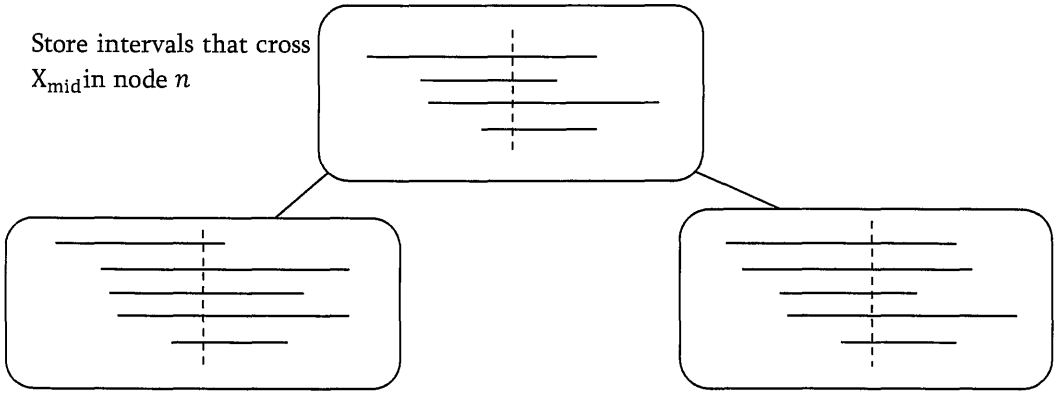


Construction of Interval Trees: Let us assume that we are given a set S of n intervals (also called segments). These n intervals will have $2n$ endpoints. Now, let us see how to construct the interval tree.

Algorithm:

Recursively build tree on interval set S as follows:

- Sort the $2n$ endpoints
- Let X_{mid} be the median point



Time Complexity for building interval trees: O($nlogn$). Since we are choosing the median, Interval Trees will be approximately balanced. This ensures that, we split the set of end points up in half each time. The depth of the tree is O($logn$). To simplify the search process, generally X_{mid} is stored with each node.

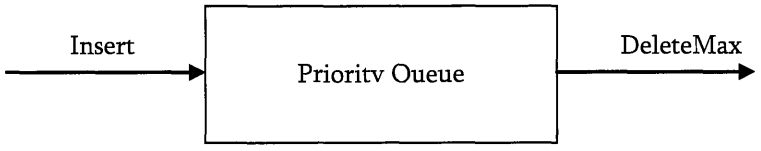
PRIORITY QUEUE AND HEAPS

Chapter-7

7.1 What is a Priority Queue?

In some situations we may need to find minimum/maximum element among a collection of elements. Priority Queue ADT is the one which support these kinds of operations. A priority queue ADT is a data structure that supports the operations *Insert* and *DeleteMin* (which returns and removes the minimum element) or *DeleteMax* (which returns and removes the maximum element).

These operations are equivalent to *EnQueue* and *DeQueue* operations of a queue. The difference is that, in priority queues, the order in which the elements enter the queue may not be same in which they were processed. An example application of a priority queue is job scheduling, which is prioritized instead of serving in first come first serve.



A priority queue is called an *ascending − priority* queue, if the item with smallest key has the highest priority (that means, delete smallest element always). Similarly, a priority queue is said to be a *descending − priority* queue if the item with largest key has the highest priority (delete maximum element always). Since these two types are symmetric we will be concentrating on one of them, say, ascending-priority queue.

7.2 Priority Queue ADT

The following operations make priority queues an ADT.

Main Priority Queues Operations

A priority queue is a container of elements, each having an associated key.

- Insert(key, data): Inserts data with *key* to the priority queue. Elements are ordered based on key.
- DeleteMin/DeleteMax: Remove and return the element with the smallest/largest key.
- GetMinimum/GetMaximum: Return the element with the smallest/largest key without deleting it.

Auxiliary Priority Queues Operations

- k^{th} −Smallest/k^{th} −Largest: Returns the k^{th} −Smallest/k^{th} −Largest key in priority queue.
- Size: Returns number of elements in priority queue.
- Heap Sort: Sorts the elements in the priority queue based on priority (key).

7.3 Priority Queue Applications

Priority queues have many applications and below are few of them:

- Data compression: Huffman Coding algorithm
- Shortest path algorithms: Dijkstra's algorithm
- Minimum spanning tree algorithms: Prim's algorithm
- Event-driven simulation: customers in a line

- Selection problem: Finding k^{th}-smallest element

7.4 Priority Queue Implementations

Before discussing the actual implementation, let us enumerate the possible options.

Unordered Array Implementation

Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are performed by searching the key and then followed by deletion.

Insertions complexity: O(1). DeleteMin complexity: O(n).

Unordered List Implementation

It is very much similar to array implementation, but instead of using arrays linked lists are used.

Insertions complexity: O(1). DeleteMin complexity: O(n).

Ordered Array Implementation

Elements are inserted into the array in sorted order based on key field. Deletions are performed at only one end.

Insertions complexity: O(n). DeleteMin complexity: O(1).

Ordered List Implementation

Elements are inserted into the list in sorted order based on key field. Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.

Insertions complexity: O(n). DeleteMin complexity: O(1).

Binary Search Trees Implementation

Both insertions and deletions take O($logn$) on average if insertions are random (refer *Trees* chapter).

Balanced Binary Search Trees Implementation

Both insertions and deletion take O($logn$) in the worst case (refer *Trees* chapter).

Binary Heap Implementation

In subsequent sections we will discuss this in full detail. For now assume that binary heap implementation gives O($logn$) complexity for search, insertions and deletions and O(1) for finding the maximum or minimum element.

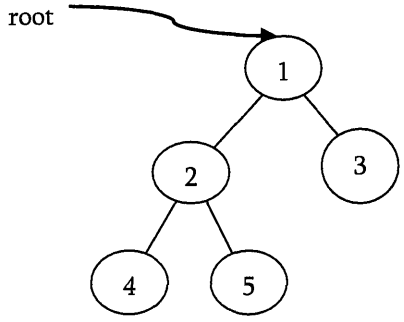
Comparing Implementations

Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	$logn$ (average)	$logn$ (average)	$logn$ (average)
Balanced Binary Search Trees	$logn$	$logn$	$logn$
Binary Heaps	$logn$	$logn$	1

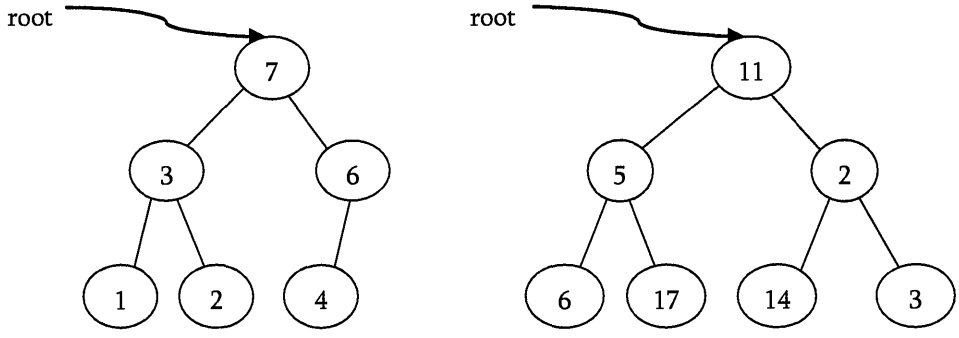
7.5 Heaps and Binary Heap

What is a Heap?

A heap is a tree with some special properties. The basic requirement of a heap is that the value of a node must be $\geq$ (or $\leq$) to the values of its children. This is called *heap property*. A heap also has the additional property that all leaves should be at h or $h - 1$ levels (where h is the height of the tree) for some $h > 0$ (*complete binary trees*). That means heap should form a *complete binary tree* (as shown below).



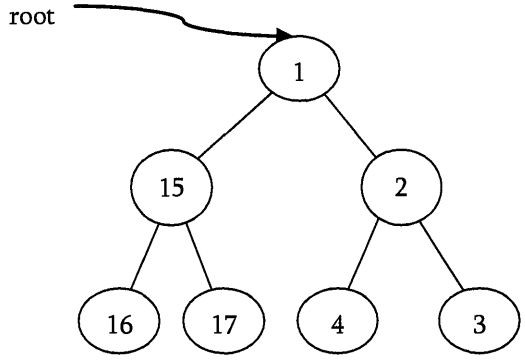
In the below examples, the left tree is a heap (each element is greater than its children) and right tree is not a heap (since, 11 is greater than 2).



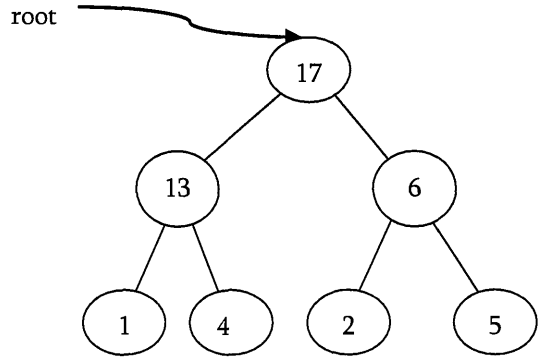
Types of Heaps?

Based on the heap property we can classify the heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children



- **Max heap:** The value of a node must be greater than or equal to the values of its children



7.6 Binary Heaps

In binary heap each node may have up to two children. In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for remaining discussion.

Representing Heaps: Before looking at heap operations, let us see how to represent heaps. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the below discussion let us assume that elements are stored in arrays which starts at index 0. The previous max heap can be represented as:

17	13	6	1	4	2	5
0	1	2	3	4	5	6

Note: For the remaining discussion let us assume that we are doing manipulations in max heap.

Declaration of Heap

```
struct Heap {
        int *array;
        int count;              // Number of elements in Heap
        int capacity;           // Size of the heap
        int heap_type;          // Min Heap or Max Heap
};
```

Creating Heap

```
struct Heap * CreateHeap(int capacity, int heap_type) {
        struct Heap * h = (struct Heap *)malloc(sizeof(struct Heap));
        if(h == NULL) {  printf("Memory Error");
                         return;
        }
        h→heap_type = heap_type;
        h→count = 0;
        h→capacity = capacity;
        h→array = (int *) malloc(sizeof(int) * h→capacity);
        if(h→array == NULL)     {
                printf("Memory Error");
                return;
        }
        return h;
}
```

Time Complexity: O(1).

Parent of a Node

For a node at i^{th} location, its parent is at $\frac{i-1}{2}$ location. For the previous example, the element 6 is at second location and its parent is at 0^{th} location.

```
int Parent (struct Heap * h, int i) {
        if(i <= 0 || i >= h→count)
                return -1;
        return i-1/2;
}
```

Time Complexity: O(1).

Children of a Node

Similar to above discussion for a node at i^{th} location, its children are at $2 * i + 1$ and $2 * i + 2$ locations. For example, in the above tree the element 6 is at second location and its children 2 and 5 are at 5 ($2 * i + 1 = 2 * 2 + 1$) and 6 ($2 * i + 2 = 2 * 2 + 2$) locations.

```
int LeftChild(struct Heap *h, int i) {
        int left = 2 * i + 1;
        if(left >= h→count) return -1;
        return left;
}
```

Time Complexity: O(1).

```
int RightChild(struct Heap *h, int i) {
        int right = 2 * i + 2;
        if(right >= h→count) return -1;
        return right;
}
```

Time Complexity: O(1).

Getting the Maximum Element

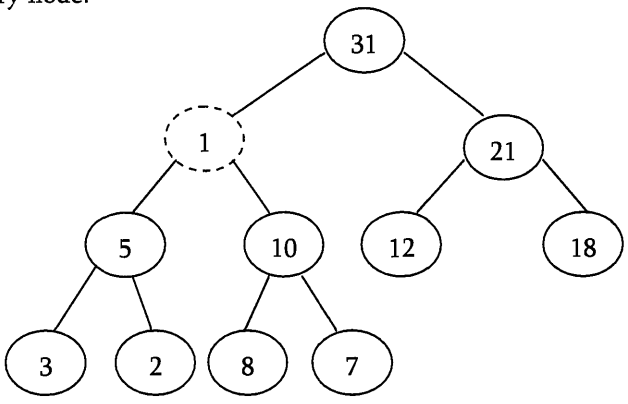
Since the maximum element in max heap is always at root, it will be stored at h→array[0].

```
int GetMaximum(Heap * h) {
        if(h→count == 0) return -1;
        return h→array[0];
}
```

Time Complexity: O(1).

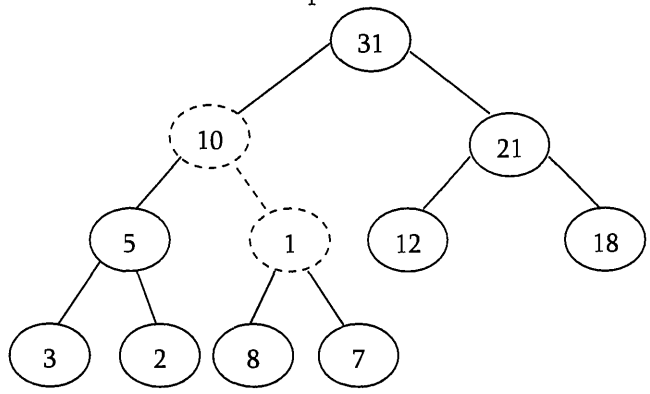
Heapifying an Element

After inserting an element into heap, it may not satisfy the heap property. In that case we need to adjust the locations of the heap to make it heap again. This process is called *heapifying*. In max-heap, to heapify an element, we have to find the maximum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node.

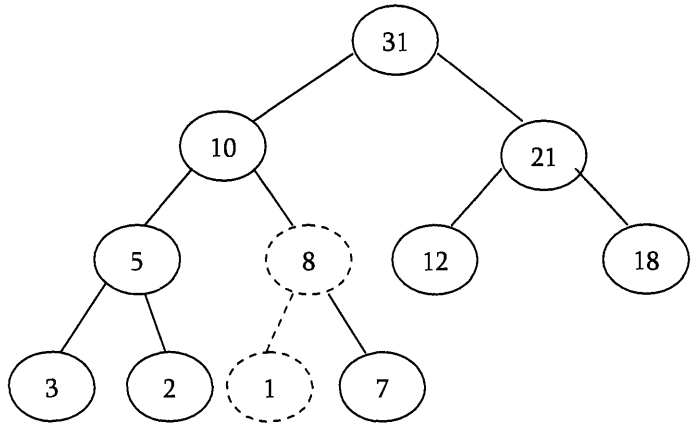


Observation: One important property of heap is that, if an element is not satisfying the heap property then all the elements from that element to root will also have the same problem. In below example, element 1 is not satisfying the heap property and its parent 31 is also having the issue. Similarly, if we heapify an element then all the elements from that element to root will also satisfy the heap property automatically. Let us go through an example. In the above heap, the element 1 is not satisfying the heap property and let us try heapifying this element.

To heapify 1, find maximum of its children and swap with that.



We need to continue this process until the element satisfies the heap properties. Now, swap 1 with 8.



Now the tree is satisfying the heap property. In the above heapifying process, since we are moving from top to bottom, this process is sometimes called as *percolate down*.

```
//Heapifying the element at location i.
void PercolateDown(struct Heap *h, int i) {
        int l, r, max, temp;
        l = LeftChild(h, i);
        r = RightChild(h, i);
        if(l != -1 && h→array[l] > h→array[i])
                max = l;
        else
                max = i;
        if(r != -1&& h→array[r] > h→array[max])
                  max = r;
        if(max != i) {
                  //Swap h→array[i] and  h→array[max];
                  temp = h→array[i];
                  h→array[i] = h→array[max];
                  h→array[i] = temp;
        }
```

```
        PercolateDown(h, max);
}
```

Time Complexity: O(*logn*). Heap is a complete binary tree and in the worst we start at root and coming down till the leaf. This is equal to the height of the complete binary tree. Space Complexity: O(1).

Deleting an Element

To delete an element from heap, we just need to delete the element from root. This is the only operation (maximum element) supported by standard heap. After deleting the root element, copy the last element of the heap (tree) and delete that last element. After replacing the last element the tree may not satisfy the heap property. To make it heap again, call *PercolateDown* function.

- Copy the first element into some variable
- Copy the last element into first element location
- *PercolateDown* the first element

```
int DeleteMax(struct Heap *h) {
        int data;
        if(h→count == 0)
                return -1;
        data = h→array[0];
        h→array[0] = h→array[h→count-1];
        h→count--; //reducing the heap size
        PercolateDown(h, 0);
        return data;
}
```

Note: Deleting an element uses *percolate down.*

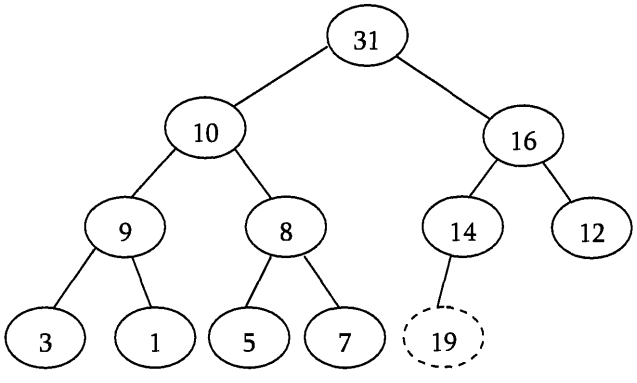
Time Complexity: same as Heapify function and it is O(*logn*).

Inserting an Element

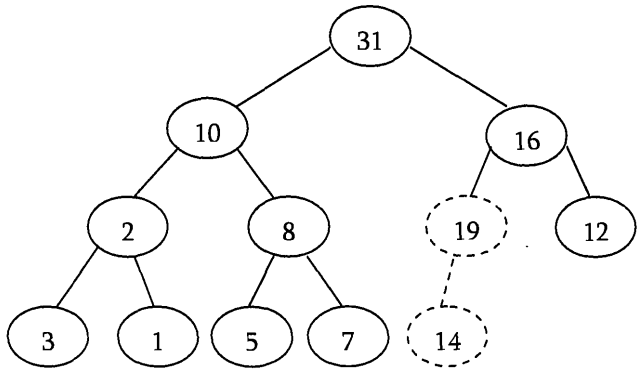
Insertion of an element is very much similar to heapify and deletion process.

- Increase the heap size
- Keep the new element at the end of the heap (tree)
- Heapify the element from bottom to top (root)

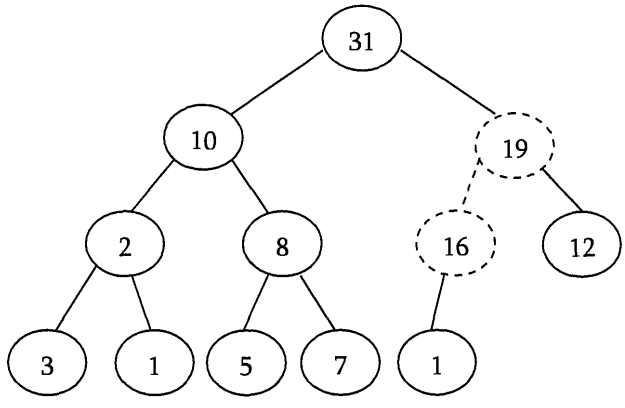
Before going through code, let us take an example. We have inserted the element 19 at the end of the heap and it's not satisfying the heap property.



In-order to heapify this element (19), we need to compare it with its parent and adjust them. Swapping 19 and 14 gives:



Again, swap 19 and16:



Now the tree is satisfying the heap property. Since we are following the bottom-up approach we call this process is sometimes called as *percolate up*.

```
int Insert(struct Heap *h, int data) {
        int i;
        if(h→count == h→capacity)
                ResizeHeap(h);
        h→count++;              //increasing the heap size to hold this new item
        i = h→count-1;
        while(i>=0 && data > h→array[(i-1)/2]) {
                h→array[i] = h→array[(i-1)/2];
                i = i-1/2;
        }
        h→array[i] = data;
}
void ResizeHeap(struct Heap * h) {
        int *array_old = h→array;
        h→array = (int *) malloc(sizeof(int) * h→capacity * 2);
        if(h→array == NULL) {
                printf("Memory Error");
                return;
        }
        for (int i = 0; i < h→capacity; i ++)
                h→array[i] = array_old[i];
        h→capacity *= 2;
        free(array_old);
}
```

Time Complexity: O(*logn*). The explanation is same as that of Heapify function.

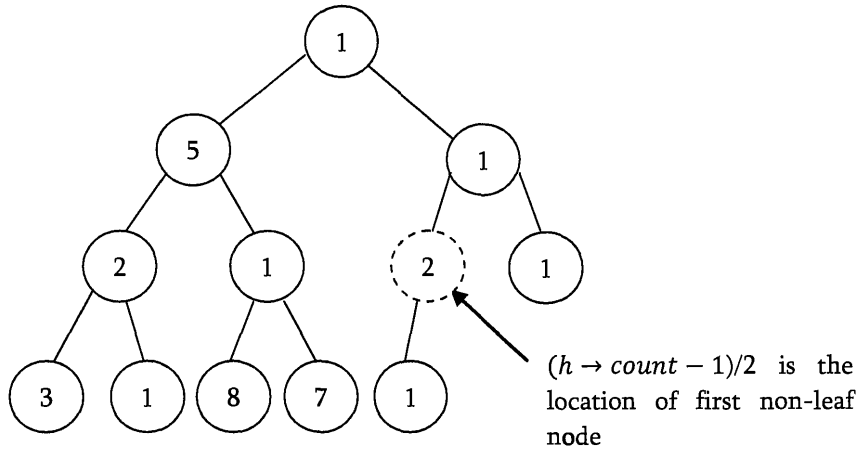
Destroying Heap

```
void DestroyHeap (struct Heap *h) {
        if(h == NULL)
                return;
        free(h→array);
        free(h);
        h = NULL;
}
```

Heapifying the Array

One simple approach for building the heap is, take n input items and place them into an empty heap. This can be done with n successive inserts and takes $O(nlogn)$ in worst case. This is due to the fact that each insert operation takes $O(logn)$.

Observation: Leaf nodes always satisfy the heap property and do not need to care for them. The leaf elements are always at the ending and to heapify the given array it should be enough if we heapify the non-leaf nodes. Now let us concentrate on finding the first non leaf node. The last element of the heap is at location $h \rightarrow count - 1$, and to find the first non-leaf node it is enough to find the parent of last element.




```
void BuildHeap(struct Heap *h, int A[], int n) {
        if(h == NULL)
                return;
        while (n > h→capacity)
                ResizeHeap(h);
        for (int i = 0; i < n; i ++)
                h→array[i] = A[i];
        h→count = n;
        for (int i = (n-1)/2; i >=0; i --)
                PercolateDown(h, i);
}
```

Time Complexity: The linear time bound of building heap, can be shown by computing the sum of the heights of all the nodes. For a complete binary tree of height h containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $n - h - 1 = n - logn - 1$ (for proof refer *Problems Section*). That means, building heap operation can be done in linear time ($O(n)$) by applying a *PercolateDown* function to nodes in reverse level order.

Heapsort

One main application of heap ADT is sorting (heap sort). Heap sort algorithm inserts all elements (from an unsorted array) into a heap, then remove them from the root of a heap until the heap is empty. Note that heap sort can be done in place with the array to be sorted. Instead of deleting an element, exchange the first element (maximum) with last element and reduce the heap size (array size). Then, we heapify the first element. Continue this process until the number of remaining elements is one.

```
void Heapsort(int A[], in n) {
        struct Heap *h = CreateHeap(n);
        int old_size, i, temp;
        BuildHeap(h, A, n);
        old_size = h→count;
        for(i = n-1; i > 0; i--) {
                //h→array [0] is the largest element
                temp = h→array[0]; h→array[0] = h→array[h→count-1]; h→array[0] = temp;
                h→count--;
                PercolateDown(h, i);
        }
        h→count = old_size;
}
```

Time complexity: As we remove the elements from the heap, the values become sorted (since maximum elements are always *root* only). Since the time complexities of both the insertion algorithm and deletion algorithms is $O(logn)$ (where n is the number of items in the heap), the time complexity of the heap sort algorithm is $O(nlogn)$.

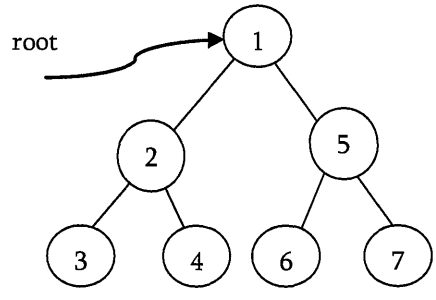
7.7 Problems on Priority Queues [Heaps]

Problem-1 What are the minimum and maximum number of elements in a heap of height h?

Solution: Since heap is a complete binary tree (all levels contain full nodes except possibly the lowest level), it has at most $2^{h+1} - 1$ elements (if it is complete). This is because, to get maximum nodes, we need to fill all the h levels completely and the maximum number of nodes is nothing but sum of all nodes at all h levels. To get minimum nodes, we should fill the $h - 1$ levels fully and last level with only one element. As a result, the minimum number of nodes is nothing but sum of all nodes from $h - 1$ levels plus 1 (for last level) and we get $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and all the other levels are complete).

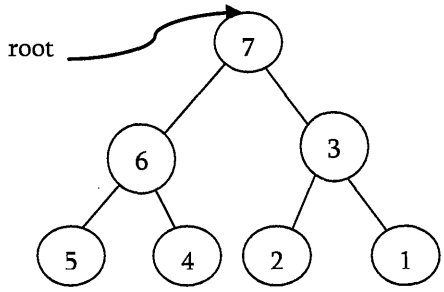
Problem-2 Is there a min-heap with seven distinct elements so that, the preorder traversal of it gives the elements in sorted order?

Solution: Yes. For the below tree, preorder traversal produces ascending order.



Problem-3 Is there a max-heap with seven distinct elements so that, the preorder traversal of it gives the elements in sorted order?

Solution: Yes. For the below tree, preorder traversal produces descending order.

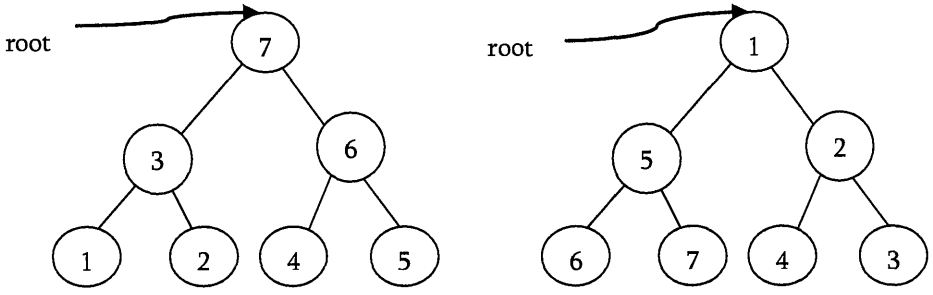


Problem-4 Is there a min-heap/max-heap with seven distinct elements so that, the inorder traversal of it gives the elements in sorted order?

Solution: No, since a heap must be either a min-heap or a max-heap, the root will hold the smallest element or the largest. An inorder traversal will visit the root of tree as its second step, which is not the appropriate place if trees root contains the smallest or largest element.

Problem-5 Is there a min-heap/max-heap with seven distinct elements so that, the postorder traversal of it gives the elements in sorted order?

Solution: Yes, if tree is a max-heap and we want descending order (below left), or if tree is a min-heap and we want ascending order (below right).

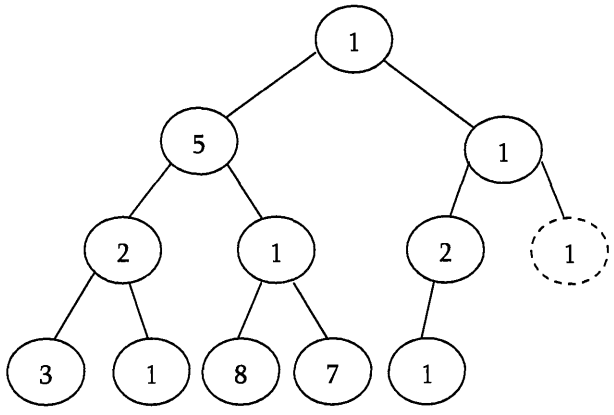


Problem-6 Show that the height of a heap with n elements is $logn$?

Solution: A heap is a complete binary tree. All the levels, except the lowest, are completely full. A heap has at least 2^h element and atmost elements. $2^h \leq n \leq 2^{h+1} - 1$. This implies, $h \leq logn \leq h + 1$. Since h is integer, $h = logn$.

Problem-7 Given a min-heap, give an algorithm for finding the maximum element.

Solution: For a given min heap the maximum element will always be at leaf only. Now, the next question is how to find the leaf nodes in tree?



If we carefully observe, the next node of last elements parent is the first leaf node. Since the last element is always at $h \rightarrow count - 1^{th}$ location, the next node of its parent (parent at location $\frac{h \rightarrow count-1}{2}$) can be calculated as:

$$\frac{h \rightarrow count - 1}{2} + 1 \approx \frac{h \rightarrow count + 1}{2}$$

Now, the only step remaining is scanning the leaf nodes and finding the maximum among them.

```
int FindMaxInMinHeap(struct Heap *h) {
        int Max = -1;
        for(int i = (h→count+1)/2; i < h→count; i++)
                if(h→array[i] > Max)
                        Max = h→array[i];
}
```

Time Complexity: $O(\frac{n}{2}) \approx O(n)$.

Problem-8 Give an algorithm for deleting an arbitrary element from min heap.

Solution: To delete an element, first we need to search for an element. Let us assume that we are using level order traversal for finding the element. After finding the element we need to follow the DeleteMin process.

Time Complexity = Time for finding the element + Time for deleting an element
$=O(n) + O(logn) \approx O(n)$. //Time for searching is dominated.

Problem-9 Give an algorithm for deleting the i^{th} indexed element in a given min-heap.

Solution:

```
Int Delete(struct Heap *h, int i) {
        int key;
        if(n < i) {
                printf("Wrong position");
                return;
        }
        key = h→array[i];
        h→array[i]= h→array[h→count-1];
        h→count--;
        PercolateDown(h, i);
        return key;
}
```

Time Complexity = $O(logn)$.

Problem-10 Prove that, for a complete binary tree of height h the sum of the heights of all nodes is $O(n - h)$.

Solution: A complete binary tree has 2^i nodes on level i. Also, a node on level i has depth i and height $h - i$. Let us assume that S denotes the sum of the heights of all these nodes and S can be calculated as:

$$S = \sum_{i=0}^{h} 2^i(h - i)$$

$$S = h + 2(h - 1) + 4(h - 2) + \cdots + 2^{h-1}(1)$$

Multiplying with 2 on both sides gives: $2S = 2h + 4(h - 1) + 8(h - 2) + \cdots + 2^h(1)$
Now, subtract S from $2S$: $2S - S = -h + 2 + 4 + \cdots + 2^h \Rightarrow S = (2^{h+1} - 1) - (h - 1)$

But, we already know that the total number of nodes n in a complete binary tree with height h is $n = 2^{h+1} - 1$. This gives us: $h = \log(n + 1)$. Finally, replacing $2^{h+1} - 1$ with n, gives: $S = n - (h - 1) = O(n - logn) = O(n - h)$.

Problem-11 Give an algorithm to find all elements less than some value k in a binary heap.

Solution: Start from the root of the heap. If the value of the root is smaller than k then print its value and call recursively once for its left child and once for its right child. If the value of a node is greater or equal than k then the function stops without printing that value.

The complexity of this algorithm is $O(n)$, where n is the total number of nodes in the heap. This bound takes place in the worst case, where the value of every node in the heap will be smaller than k, so the function has to call each node of the heap.

Problem-12 Give an algorithm for merging two binary max-heaps. Let us assume that the size of first heap is $m + n$ and size of second heap is n.

Solution: One simple way of solving this problem is:

- Assume that elements of first array (with size $m + n$) are at the beginning. That means, first m cells are filled and remaining n cells are empty.
- Without changing the first heap, just append the second heap and heapify the array.
- Since the total number of elements in the new array are $m + n$, each heapify operation takes $O(log(m + n))$.

The complexity of this algorithm is : $O((m + n)log(m + n))$.

Problem-13 Can we improve the complexity of Problem-12*?*

Solution: Instead of heapifying all the elements of the $m + n$ array, we can use technique of "building heap with an array of elements (heapifying array)". We can start at non leaf nodes and heapify them. The algorithm can be given as:

- Assume that elements of first array (with size $m + n$) are at the beginning. That means, first m cells are filled and remaining n cells are empty.
- Without changing the first heap, just append the second heap.
- Now, find the first non leaf node and start heapifying from that element.

In theory section, we have already seen that, building a heap with n elements takes $O(n)$ complexity. The complexity of merging with this technique is: $O(m + n)$.

Problem-14 Is there an efficient algorithm for merging 2 max-heaps (stored as an array)? Assume both arrays are having n elements.

Solution: The alternative solution for this problem depends on the type of the heap is. If it's a standard heap where every node has up to two children and which gets filled up that the leaves are on a maximum of two different rows, we cannot get better than $O(n)$ for merge.

There is an $O(logm \times logn)$ algorithm for merging two binary heaps with sizes m and n. For $m = n$, this algorithm takes $O(log^2 n)$ time complexity. We will be skipping it due to its difficulty and scope.

For better merging performance, we can use another variant of binary heap like a *Fibonacci-Heap* which can merge in O(1) on average (amortized).

Problem-15 Give an algorithm for finding the k^{th} smallest element in max-heap.

Solution: One simple solution to this problem is: perform deletion k times from max-heap.

```
int FindKthLargestEle(struct Heap *h, int k) {
        //Just delete first k-1 elements and return the kth element.
        for(int i=0;i<k-1;i++)
                DeleteMin(h);

        return DeleteMin(h);
}
```

Time Complexity: $O(klogn)$. Since we are performing deletion operation k times and each deletion takes $O(logn)$.

Problem-16 For the Problem-15, can we improve the time complexity?

Solution: Assume that the original min-heap is called $HOrig$ and the auxiliary min-heap is named $HAux$. Initially, the element at the top of $HOrig$, the minimum one, is inserted into the $HAux$. Here we don't do the operation of DeleteMin with $HOrig$.

```
Heap HOrig;
Heap HAux;
int FindKthLargestEle( int k ) {
        int heapElement;//Assuming heap data is of integers
        int count=1;
        HAux.Insert(HOrig.Min());
        while( true ) {
                //return the minimum element and delete it from the HA heap
                heapElement = HAux.DeleteMin();
                if(++count == k ) {
                        return heapElement;
                }
                else {  //insert the left and right children in HO into the HA
                        HAux.Insert(heapElement.LeftChild());
                        HAux.Insert(heapElement.RightChild());
                }
        }
}
```

Every while-loop iteration gives the k^{th} smallest element and we need k loops to get the k^{th} smallest elements. Because the size of the auxiliary heap is always less than k, every while-loop iteration the size of the auxiliary heap increases by one, and the original heap $HOrig$ has no operation during the finding, the running time is $O(klogk)$.

Problem-17 Find k max elements from max heap.

Solution: One simple solution to this problem is: build max-heap and perform deletion k times.

$$T(n) = \text{DeleteMin from heap } k \text{ times} = \Theta(klogn).$$

Problem-18 For Problem-17, is there any alternative solution?

Solution: We can use the Problem-16 solution. At the end the auxiliary heap contains the k-largest elements. Without deleting the elements we should keep on adding elements to $HAux$.

Problem-19 How do we implement stack using heap.

Solution: To implement a stack using a priority queue PQ (using min heap), let us assume that we are using one extra integer variable c. Also, assume that c is initialized equal to any known value (e.g. 0). The implementation of the stack ADT is given below. Here c is used as the priority while inserting/deleting the elements from PQ.

```
void Push(int element) {
        PQ.Insert(c, element);
        c--;
}
int Pop() {
        return PQ.DeleteMin();
}
int Top() {
        return PQ.Min();
}
int Size() {
        return PQ.Size();
}
int IsEmpty() {
        return PQ.IsEmpty();
```

```
}
```

We could also increment c back when popping.

Observation: We could use the negative of the current system time instead of c (to avoid overflow). The implementation based on this can be given as:

```
void Push(int element) {
        PQ.insert(-gettime(),element);
}
```

Problem-20 How do we implement Queue using heap?

Solution: To implement a queue using a priority queue PQ (using min heap), as similar to stacks simulation, let us assume that we are using one extra integer variable, c. Also, assume that c is initialized equal to any known value (e.g. 0). The implemention of the queue ADT is given below. Here the c, is used as the priority while inserting/deleting the elements from PQ.

```
void Push(int element) {
        PQ.Insert(c, element);
        c++;
}
int Pop() {
        return PQ.DeleteMin();
}
int Top() {
        return PQ.Min();
}
int Size() {
        return PQ.Size();
}
int IsEmpty() {
        return PQ.IsEmpty();
}
```

Note: We could also decrement c back when popping.

Observation: We could use just the negative of the current system time instead of c (to avoid overflow). The implementation based on this can be given as:

```
void Push(int element) {
        PQ.insert(gettime(),element);
}
```

Note: The only change is that we need to take positive c value instead of negative.

Problem-21 Given a big file containing billions of numbers. How to find maximum 10 numbers from those file?

Solution: Always remember that when we are asked about these types of questions where we need to find max n elements, best data structure to use is priority queues.

One solution for this problem is to divide the data in some sets of 1000 elements (let's say 1000), make a heap of them, and take 10 elements from each heap one by one. Finally heap sort all the sets of 10 elements and take top 10 among those. But the problem in this approach is where to store 10 elements from each heap. That may require a large amount of memory as we have billions of numbers.

Reuse top 10 elements from earlier heap in subsequent elements can solve this problem. That means to take first block of 1000 elements and subsequent blocks of 990 elements each. Initially Heapsort first set of 1000 numbers, took max

10 elements and mix them with 990 elements of 2^{nd} set. Again Heapsort these 1000 numbers (10 from first set and 990 from 2^{nd} set), take 10 max element and mix those with 990 elements of 3^{nd} set. Repeat the same till last set of 990 (or less) elements and take max 10 elements from final heap. Those 10 elements will be your answer.

Time Complexity: $O(n) = n/1000 \times$(complexity of Heapsort 1000 elements) Since complexity of heap sorting 1000 elements will be a constant so the $O(n) = n$ i.e. linear complexity.

Problem-22 **Merge k sorted lists with total of n elements:** We are given k sorted lists with total n inputs in all the lists. Give an algorithm to merge them into one single sorted.

Solution: Since there are k equal size lists with a total of n elements, size of each list is $\frac{n}{k}$. One simple way of solving this problem is:

- Take the first list and merge it with second list. Since the size of each list is $\frac{n}{k}$, this step produces a sorted list with size $\frac{2n}{k}$. This is very much similar to merge sort logic. Time complexity of this step is: $\frac{2n}{k}$. This is because we need to scan all the elements of both the lists.
- Then, merge the second list output with third list. As a result this step produces the sorted list with size $\frac{3n}{k}$. Time complexity of this step is: $\frac{3n}{k}$. This is because we need to scan all the elements of both the lists (one with size $\frac{2n}{k}$ and other with size $\frac{n}{k}$).
- Continue this process until all the lists are merged to one list.

Total time complexity: $= \frac{2n}{k} + \frac{3n}{k} + \frac{4n}{k} + \cdots \cdot \frac{kn}{k} = \sum_{i=2}^{n} \frac{in}{k} = \frac{n}{k}\sum_{i=2}^{n} i \approx \frac{n(k^2)}{k} \approx O(nk)$. Space Complexity: $O(1)$.

Problem-23 For the Problem-22, Can we improve the time complexity?

Solution:

1 Divide the lists into pairs and merge them. That means, first take two lists at a time and merge them so that the total elements parsed for all lists is $O(n)$. This operation gives $k/2$ lists.

2 Repeat step-1 until the number of lists becomes one.

Time complexity: Step-1 executes $logk$ times and each operation parses all n elements in all the lists for making $k/2$ lists. For example, if we have 8 lists then first pass would make 4 lists by parsing all n elements. Second pass would make 2 lists by parsing again n elements and third pass would give 1 list again by parsing n elements. As a result the total time complexity is $O(nlogn)$. Space Complexity: $O(n)$.

Problem-24 For the Problem-23, can we improve the space complexity?

Solution: Let us use heaps for reducing the space complexity.

1. Build the max-heap with all first elements from each list in $O(k)$.
2. In each step extract the maximum element of the heap and add it at the end of the output.
3. Add the next element from the list of the one extracted. That means, we need to select the next element of the list which contains the extracted element of the previous step.
4. Repeat step−2 and step-3 until all the elements are completed from all the lists.

Time Complexity = $O(n\ logk\ )$. At a time we have k elements max heap and for all n elements we have to read just the heap in $logk$ time so total time = $O(nlogk)$. Space Complexity: $O(k)$ [for Max-heap].

Problem-25 Given 2 arrays A and B each with n elements. Give an algorithm for finding largest n pairs $(A[i], B[j])$.

Solution:

Algorithm:

- Heapify A and B. This step takes $O(2n) \approx O(n)$.
- Then keep on deleting the elements from both the heaps. Each of this step takes $O(2logn) \approx O(logn)$.

Total Time complexity: $O(nlogn)$.

Problem-26 **Min-Max heap:** Give an algorithm that supports min and max in O(1) time, insert, delete min, and delete max in O($logn$) time. That means, design a data structure which supports the following operations:

Operation	Complexity
Init	O(n)
Insert	O($logn$)
FindMin	O(1)
FindMax	O(1)
DeleteMin	O($logn$)
DeleteMax	O($logn$)

Solution: This problem can be solved using two heaps. Let us say two heaps are: Minimum-Heap H_{min} and Maximum-Heap H_{max}. Also, assume that elements in both the arrays are having mutual pointers. That means, an element in H_{min} will have a pointer to the same element in H_{max} and an element in H_{max} will have a pointer to the same element in H_{min}.

Init	Build H_{min} in O(n) and H_{max} in O(n)
Insert(x)	Insert x to H_{min} in O($logn$). Insert x to H_{max} in O($logn$). Update the pointers in O(1)
FindMin()	Return root(H_{min}) in O(1)
FindMax	Return root(H_{max}) in O(1)
DeleteMin	Delete the minimum from H_{min} in O($logn$). Delete the same element from H_{max} by using the mutual pointer in O($logn$)
DeleteMax	Delete the maximum from H_{max} in O($logn$). Delete the same element from H_{min} by using the mutual pointer in O($logn$)

Problem-27 Dynamic median finding. Design a heap data structure that supports finding the median.

Solution: In a set of n elements, median is the middle element, such that the number of elements smaller than the median is equal to the number of elements larger than the median. If n is odd, we can find the median by sorting the set and taking the middle element. If n is even, the median is usually defined as the average of the two middle elements. This algorithm work even when some of the elements in the list are equal. For example, the median of the multiset {1, 1, 2, 3, 5} is 2, and the median of the multiset {1, 1, 2, 3, 5, 8} is 2.5.

"*Median heaps*" are the variant of heaps that give access to the median element. A median heap can be implemented using two heaps, each containing half the elements. One is a max-heap, containing the smallest elements, the other is a min-heap, containing the largest elements. The size of the max-heap may be equal to the size of the min-heap, if the total number of elements is even. In this case, the median is the average of the maximum element of the max-heap and the minimum element of the min-heap. If there are an odd number of elements, the max-heap will contain one more element than the min-heap. The median in this case is simply the maximum element of the max-heap.

Problem-28 **Maximum sum in sliding window:** Given array A[] with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width w. **Output:** An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]

Requirement: Find a good optimal way to get B[i]

Solution: Brute force solution is, every time the window is moved, we can search for a total of w elements in the window.

Time complexity: $O(nw)$.

Problem-29 For Problem-28, can we reduce the complexity?

Solution: Yes, we can use heap data structure. This reduces the time complexity to $O(nlogw)$. Insert operation takes $O(logw)$ time, where w is the size of the heap. However, getting the maximum value is cheap, it merely takes constant time as the maximum value is always kept in the root (head) of the heap. As the window slides to the right, some elements in the heap might not be valid anymore (range is outside of the current window). How should we remove them? We would need to be somewhat careful here. Since we only remove elements that are out of the window's range, we would need to keep track of the elements' indices too.

Problem-30 For Problem-28, can we further reduce the complexity?

Solution: Yes, The double-ended queue is the perfect data structure for this problem. It supports insertion/deletion from the front and back. The trick is to find a way such that the largest element in the window would always appear in the front of the queue. How would you maintain this requirement as you push and pop elements in and out of the queue?

Besides, you might notice that there are some redundant elements in the queue that we shouldn't even consider about. For example, if the current queue has the elements: [10 5 3], and a new element in the window has the element 11. Now, we could have emptied the queue without considering elements 10, 5, and 3, and insert only element 11 into the queue.

A natural way most people would think is to try to maintain the queue size the same as the window's size. Try to break away from this thought and try to think outside of the box. Removing redundant elements and storing only elements that need to be considered in the queue is the key to achieve the efficient $O(n)$ solution below. This is because each element in the list is being inserted and then removed at most once. Therefore, the total number of insert + delete operations is $2n$.

```
void MaxSlidingWindow(int A[], int n, int w, int B[]) {
        struct DoubleEndQueue *Q = CreateDoubleEndQueue();
        for (int i = 0; i < w; i++) {
                while (!IsEmptyQueue(Q) && A[i] >= A[QBack(Q)])
                        PopBack(Q);
                PushBack(Q, i);
        }
        for (int i = w; i < n; i++) {
                B[i-w] = A[QFront(Q)];
                while (!IsEmptyQueue(Q) && A[i] >= A[QBack(Q)])
                        PopBack(Q);
                while (!IsEmptyQueue(Q) && QFront(Q) <= i-w)
                        PopFront(Q);
                PushBack(Q, i);
        }
        B[n-w] = A[QFront(Q)];
}
```

By Adaptability

Few sorting algorithms complexity changes based on presortedness [quick sort]: presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

10.4 Other Classifications

Other way of classifying the sorting algorithms are:

- Internal Sort
- External Sort

Internal Sort

Sort algorithms which use main memory exclusively during the sort are called *internal* sorting algorithms. This kind of algorithms assumes high-speed random access to all memory.

External Sort

Sorting algorithms which uses external memory, such as tape or disk, during the sort comes under this category.

10.5 Bubble sort

Bubble sort is the simplest sorting algorithm. It works by iterating the input array from the first element to last, comparing each pair of elements and swapping them if needed. Bubble sort continues its iterations until no swaps are needed. The algorithm got its name from the way smaller elements "*bubble*" to the top of the list. Generally, insertion sort has better performance than bubble sort. Some researchers suggest that we should not teach bubble sort because of its simplicity and bad complexity.

The only significant advantage that bubble sort has over other implementations is that it can detect whether the input list is already sorted or not.

Implementation

```
void BubbleSort(int A[], int n) {
        for (int pass = n - 1; pass >= 0; pass--)       {
                for (int i = 0; i < pass - 1 ; i++)     {
                        if(A[i] > A[i+1]) {
                                // swap elements
                                int temp = x[i];
                                A[i] = A[i+1];
                                A[i+1] = temp;
                        }
                }
        }
}
```

Algorithm takes $O(n^2)$ (even in best case). We can improve it by using one extra flag. When there are no more swaps, indicates the completion of sorting. If the list is already sorted, by using this flag we can skip the remaining passes.

```
void BubbleSortImproved(int A[], int n) {
        int pass, i, temp, swapped = 1;
        for (pass = n - 1; pass >= 0 && swapped; pass--) {
                swapped = 0;
```

```
                for (i = 0; i < pass - 1 ; i++) {
                        if(A[i] > A[i+1]) {
                                // swap elements
                                temp = A[i];
                                A[i] = A[i+1];
                                A[i+1] =  temp;
                                swapped = 1;
                        }
                }
        }
}
```

This modified version improves the best case of bubble sort to $O(n)$.

Performance

Worst case complexity : $O(n^2)$
Best case complexity (Improved version) : $O(n)$
Average case complexity (Basic version) : $O(n^2)$
Worst case space complexity : $O(1)$ auxiliary

10.6 Selection Sort

Selection sort is an in-place sorting algorithm. Selection sort works well for small files. It is used for sorting the files with very large values and small keys. This is because of the fact that selection is made based on keys and swaps are made only when required.

Advantages:

- Easy to implement
- In-place sort (requires no additional storage space)

Disadvantages:

- Doesn't scale well: $O(n^2)$

Algorithm

1. Find the minimum value in the list
2. Swap it with the value in the current position
3. Repeat this process for all the elements until the entire array is sorted

This algorithm is called *selection sort* since it repeatedly *selects* the smallest element.

Implementation

```
void Selection(int A [], int n) {
        int i, j, min, temp;
        for (i = 0; i < n - 1; i++) {
                min = i;
                for (j = i+1; j < n; j++) {
                        if(A [j] < A [min])
                                min = j;
                }
                // swap elements
                temp = A[min];
                A[min] = A[i];
```

```
            A[i] = temp;
        }
}
```

Performance

Worst case complexity : $O(n^2)$
Best case complexity : $O(n)$
Average case complexity : $O(n^2)$
Worst case space complexity: O(1) auxiliary

10.7 Insertion sort

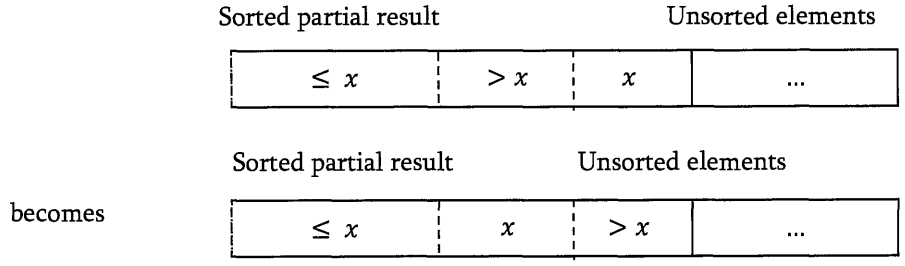
Insertion sort is a simple and efficient comparison sort. In this algorithm each iteration removes an element from the input data and inserts it into the correct position in the list being sorted. The choice of the element being removed from the input is random and this process is repeated until all input elements have been gone through.

Advantages

- Simple implementation
- Efficient for small data
- Adaptive: If the input list is presorted [may not be completely] then insertions sort takes $O(n + d)$, where d is the number of inversions
- Practically more efficient than selection and bubble sorts even though all of them have $O(n^2)$ worst case complexity
- Stable: Maintains relative order of input data if the keys are same
- In-place: It requires only a constant amount O(1) of additional memory space
- Online: Insertion sort can sort the list as it receives it

Algorithm

Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already-sorted list until no input elements remain. Sorting is typically done in-place. The resulting array after k iterations has the property where the first $k + 1$ entries are sorted.



Each element greater than x copied to the right as it is compared against x.

Implementation

```
void InsertionSort(int a[], int n) {
        int i, j; int v;
        for (i = 2; i <= n - 1; i++) {
                v = A[i];
                j = i;
                while (A[j-1] > v && j >= 1) {
                        A[j] = A[j-1];
```

```
                j--;
            }
            A[j] = v;
        }
}
```

Example

Given an array: 6 8 1 4 5 3 7 2 and the goal is to put them in ascending order.

6 8 1 4 5 3 7 2 (Consider index 0)
6 8 1 4 5 3 7 2 (Consider indices 0 - 1)
1 6 8 4 5 3 7 2 (Consider indices 0 - 2: insertion places 1 in front of 6 and 8)
1 4 6 8 5 3 7 2 (Process same as above is repeated until array is sorted)
1 4 5 6 8 3 7 2
1 3 4 5 6 7 8 2
1 2 3 4 5 6 7 8 (The array is sorted!)

Analysis

Worst case analysis

Worst case occurs when for every i the inner loop has to move all elements $A[1], \ldots, A[i-1]$ (which happens when $A[i]$ = key is smaller than all of them), that takes $\Theta(i-1)$ time.

$$T(n) = \Theta(1) + \Theta(2) + \Theta(2) + \ldots\ldots + \Theta(n-1)$$
$$= \Theta(1 + 2 + 3 + \ldots\ldots + n - 1) = \Theta(\frac{n(n-1)}{2}) \approx \Theta(n^2)$$

Average case analysis

For the average case, the inner loop will insert $A[i]$ in the middle of $A[1], \ldots, A[i-1]$. This takes $\Theta(i/2)$ time.

$$T(n) = \sum_{i=1}^{n} \Theta(i/2) \approx \Theta(n^2)$$

Performance

Worst case complexity : $O(n^2)$
Best case complexity : $O(n^2)$
Average case complexity : $O(n^2)$
Worst case space complexity: $O(n^2)$ total, O(1) auxiliary

Comparisons to Other Sorting Algorithms

Insertion sort is one of the elementary sorting algorithms with $O(n^2)$ worst-case time. Insertion sort is used when the data is nearly sorted (due to its adaptiveness) or when the input size is small (due to its low overhead). For these reasons and due to its stability, insertion sort is used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Note:

- Bubble sort takes $\frac{n^2}{2}$ comparisons and $\frac{n^2}{2}$ swaps (inversions) in both average case and in worst case.
- Selection sort takes $\frac{n^2}{2}$ comparisons and n swaps.
- Insertion sort takes $\frac{n^2}{4}$ comparisons and $\frac{n^2}{8}$ swaps in average case and in the worst case they are double.
- Insertion sort is almost linear for partially sorted input.
- Selection sort is best suits for elements with bigger values and small keys.

10.8 Shell sort

Shell sort (also called *diminishing increment sort*) is invented by *Donald Shell*. This sorting algorithm is a generalization of insertion sort. Insertion sort works efficiently on input that is already almost sorted.

In insertion sort, comparison happens between the adjacent elements. At most 1 inversion is eliminated for each comparison done with insertion sort. The variation used in shell sort is to avoid comparing adjacent elements until the last step of the algorithm. So, the last step of shell sort is effectively the insertion sort algorithm. It improves insertion sort by allowing the comparison and exchange of elements that are far away. This is the first algorithm which got less than quadratic complexity among comparison sort algorithms.

Shellsort uses a sequence $h1, h2, ..., ht$ called the *increment sequence*. Any increment sequence is fine as long as $h1 = 1$ and some choices are better than others. Shellsort makes multiple passes through input list and sorts a number of equally sized sets using the insertion sort. Shellsort improves on the efficiency of insertion sort by *quickly* shifting values to their destination.

Implementation

```
void ShellSort(int A[], int array_size) {
        int i, j, h, v;
        for (h = 1; h = array_size/9; h = 3*h+1);

        for ( ; h > 0; h = h/3) {
                for (i = h+1; i = array_size; i += 1) {
                        v = A[i];
                        j = i;
                        while (j > h && A[j-h] > v) {
                                A[j] = A[j-h];
                                j -= h;
                        }
                        A[j] = v;
                }
        }
}
```

Note that when $h == 1$, the algorithm makes a pass over the entire list, comparing adjacent elements, but doing very few element exchanges. For $h == 1$, shell sort works just like insertion sort, except the number of inversions that have to be eliminated is greatly reduced by the previous steps of the algorithm with $h > 1$.

Analysis

Shell sort is efficient for medium size lists. For bigger lists, the algorithm is not the best choice. Fastest of all $O(n^2)$ sorting algorithms.

Disadvantage of Shell sort is that: it is a complex algorithm and not nearly as efficient as the merge, heap, and quick sorts. Shell sort is still significantly slower than the merge, heap, and quick sorts, but it is relatively simple algorithm which makes it a good choice for sorting lists of less than 5000 items unless speed important. It is also a good choice for repetitive sorting of smaller lists.

The best case in the Shell sort is when the array is already sorted in the right order. The number of comparisons is less. Running time of Shell sort depends on the choice of increment sequence.

Performance

Worst case complexity depends on gap sequence. Best known: $O(nlog^2n)$
Best case complexity: $O(n)$

Average case complexity depends on gap sequence
Worst case space complexity: O(n)

10.9 Merge sort

Merge sort is an example of the divide and conquer.

Important Notes

- *Merging* is the process of combining two sorted files to make one bigger sorted file.
- *Selection* is the process of dividing a file into two parts: k smallest elements and $n - k$ largest elements.
- Selection and merging are opposite operations
    - selection splits a list into two lists
    - merging joins two files to make one file
- Merge sort is Quick sorts complement
- Merge sort accesses the data in a sequential manner
- This algorithm is used for sorting a linked list
- Merge sort is insensitive to the initial order of its input
- In Quick sort most of the work is done before the recursive calls. Quick sort starts with the largest subfile and finishes up with the small ones and as a result it needs stack and also this algorithm is not stable. Whereas Merge sort divides the list into two parts, then each part is conquered individually. Merge sort starts with the small subfiles and finishes up with the largest one and as a result it doesn't need stack and this algorithm is stable.

Implementation

```
void Mergesort(int A[], int temp[], int left, int right) {
        int mid;
        if(right > left) {
                mid = (right + left) / 2;
                Mergesort(A, temp, left, mid);
                Mergesort(A, temp, mid+1, right);
                Merge(A, temp, left, mid+1, right);
        }
}
void Merge(int A[], int temp[], int left, int mid, int right) {
        int i, left_end, size, temp_pos;
        left_end = mid - 1;
        temp_pos = left;
        size = right - left + 1;
        while ((left <= left_end) && (mid <= right)) {
                if(A[left] <= A[mid]) {
                        temp[temp_pos] = A[left];
                        temp_pos = temp_pos + 1;
                        left = left +1;
                }
                else {  temp[temp_pos] = A[mid];
                        temp_pos = temp_pos + 1;
                        mid = mid + 1;
                }
        }
```

```
        while (left <= left_end) {
                temp[temp_pos] = A[left];
                left = left + 1;
                temp_pos = temp_pos + 1;
        }
        while (mid <= right) {
                temp[temp_pos] = A[mid];
                mid = mid + 1;
                temp_pos = temp_pos + 1;
        }
        for (i = 0; i <= size; i++) {
                A[right] = temp[right];
                right = right - 1;
        }
}
```

Analysis

In Merge sort the input list is divided into two parts and solve them recursively. After solving the sub problems merge them by scanning the resultant sub problems. Let us assume $T(n)$ is the complexity of Merge sort with n elements. The recurrence for the Merge Sort can be defined as:

Recurrence for Mergesort is $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.

Using Master theorem, we get, $T(n) = \Theta(nlogn)$.

Note: For more details, refer *Divide and Conquer* chapter.

Performance

Worst case complexity : $\Theta(nlogn)$
Best case complexity : $\Theta(nlogn)$
Average case complexity : $\Theta(nlogn)$
Worst case space complexity: $\Theta(n)$ auxiliary

10.10 Heapsort

Heapsort is a comparison-based sorting algorithm and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quick sort, it has the advantage of a more favorable worst-case $\Theta(nlogn)$ runtime. Heapsort is an in-place algorithm but is not a stable sort.

Performance

Worst case performance: $\Theta(nlogn)$
Best case performance: $\Theta(nlogn)$
Average case performance: $\Theta(nlogn)$
Worst case space complexity: $\Theta(n)$ total, $\Theta(1)$ auxiliary

For other details on Heapsort refer *Priority Queues* chapter.

10.11 Quicksort

The quick sort is an example for divide-and-conquer algorithmic technique. It is also called *partition exchange sort*. It uses recursive calls for sorting the elements. It is one of famous algorithm among comparison based sorting algorithms.

Divide: The array $A[low \dots high]$ is partitioned into two non-empty sub arrays $A[low \dots q]$ and $A[q+1 \dots high]$, such that each element of $A[low \dots high]$ is less than or equal to each element of $A[q+1 \dots high]$. The index q is computed as part of this partitioning procedure.

Conquer: The two sub arrays $A[low \dots q]$ and $A[q+1 \dots high]$ are sorted by recursive calls to Quick sort.

Algorithm

The recursive algorithm consists of four steps:

1) If there is one or no elements in the array to be sorted, return.
2) Pick an element in the array to serve as "*pivot*" point. (Usually the left-most element in the array is used.)
3) Split the array into two parts - one with elements larger than the pivot and the other with elements smaller than the pivot.
4) Recursively repeat the algorithm for both halves of the original array.

Implementation

```
Quicksort( int A[], int low, int high ) {
        int pivot;
        /* Termination condition! */
        if( high > low ) {
                pivot = Partition( A, low, high );
                Quicksort( A, low, pivot-1 );
                Quicksort( A, pivot+1, high );
        }
}
int Partition( int A, int low, int high ) {
        int left, right, pivot_item = A[low];
        left = low;
        right = high;
        while ( left < right ) {
                /* Move left while item < pivot */
                while( A[left] <= pivot_item )
                        left++;
                /* Move right while item > pivot */
                while( A[right] > pivot_item )
                        right--;
                if( left < right )
                        swap(A,left,right);
        }
        /* right is final position for the pivot */
        A[low] = A[right];
        A[right] = pivot_item;
        return right;
}
```

Analysis

Let us assume that T(n) be the complexity of Quick sort and also assume that all elements are distinct. Recurrence for $T(n)$ depends on two subproblem sizes which depend on partition element. If pivot is i^{th} smallest element then exactly $(i-1)$ items will be in left part and $(n-i)$ in right part. Let us call it as i –split. Since each element has equal probability of selecting it as pivot the probability of selecting i^{th} element is $\frac{1}{n}$.

Best Case: Each partition splits array in halves and gives

$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n)$, [using *Divide and Conquer* master theorem]

Worst Case: Each partition gives unbalanced splits and we get

$T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$[*using Subtraction and Conquer master theorem*]

The worst-case occurs when the list is already sorted and last element chosen as pivot.

Average Case: In the average case of Quick sort, we do not know where the split happens. For this reason, we take all possible values of split locations, add all of their complexities and divide with n to get the average case complexity.

$$T(n) = \sum_{i=1}^{n} \frac{1}{n}(runtime\ with\ i-split) + n + 1$$

$$= \frac{1}{n}\sum_{i=1}^{N}\big(T(i-1) + T(n-i)\big) + n + 1$$

//since we are dealing with best case we can assume $T(n-i)$ and $T(i-1)$ are equal

$$= \frac{2}{n}\sum_{i=1}^{n} T(i-1) + n + 1$$

$$= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n + 1$$

Multiply both sides by n.

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

Same formula for $n-1$.

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

Subtract the $n-1$ formula from n.

$$nT(n) - (n-1)T(n-1) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n - (2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1))$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n$$

$$nT(n) = (n+1)T(n-1) + 2n$$

Divide with $n(n+1)$.

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

.

.

$$= O(1) + 2\sum_{i=3}^{n}\frac{1}{i}$$

$$= O(1) + O(2\log n)$$

$$\frac{T(n)}{n+1} = O(\log n)$$

$$T(n) = O\big((n+1)\log n\big) = O(n\log n)$$

Time Complexity, $T(n) = O(n\log n)$.

Performance

Worst case Complexity: $O(n^2)$
Best case Complexity: $O(nlogn)$
Average case Complexity: $O(nlogn)$
Worst case space Complexity: O(1)

Randomized Quick sort

In average-case behavior of Quicksort, we assumed that all permutation of the input numbers are equally likely. However, we cannot always expect it to hold. We can add randomization to an algorithm in order to reduce the probability of getting worst case in Quick sort.

There are two ways of adding randomization in Quick sort: either by randomly placing the input data in the array or by randomly choosing an element in the input data for pivot. The second choice is easier to analyze and implement. The change will only be done at the Partition algorithm.

In normal Quicksort, *pivot* element was always the leftmost element in the list to be sorted. Instead of always using $A[low]$ as the *pivot* we will use a randomly chosen element from the subarray $A[low..high]$ in the randomized version of Quicksort. It is done by exchanging element $A[low]$ with an element chosen at random from $A[low..high]$. This ensures that the *pivot* element is equally likely to be any of the $high - low + 1$ elements in the subarray. Since the pivot element is randomly chosen, we can expect the split of the input array to be reasonably well balanced on average. This can help in preventing the worst-case behavior of quick sort which in unbalanced partitioning occurs.

Even though, the randomized version improves the worst case complexity, its worst case complexity is still $O(n^2)$. One way to improve the $Randomized - QuickSort$ is to choose the pivot for partitioning more carefully than by picking a random element from the array. One common approach is to choose the pivot as the median of a set of 3 elements randomly selected from the array.

10.12 Tree Sort

Tree sort uses a binary search tree. It involves scanning each element of the input and placing it into its proper position in a binary search tree. This has two phases:

- First phase is creating a binary search tree using the given array elements.
- Second phase is traverse the given binary search tree in inorder, thus resulting in a sorted array.

Performance

The average number of comparisons for this method is $O(nlogn)$. But in worst case, number of comparisons is reduced by $O(n^2)$, a case which arises when the sort tree is skew tree.

10.13 Comparison of Sorting Algorithms

Name	Average Case	Worst Case	Auxiliary Memory	Is Stable?	Other Notes
Bubble	$O(n^2)$	$O(n^2)$	1	yes	Small code
Selection	$O(n^2)$	$O(n^2)$	1	no	Stability depends on the implementation.
Insertion	$O(n^2)$	$O(n^2)$	1	yes	Average case is also $O(n + d)$, where d is the number of inversions
Shell	-	$O(nlog^2n)$	1	no	
Merge	$O(nlogn)$	$O(nlogn)$	depends	yes	
Heap	$O(nlogn)$	$O(nlogn)$	1	no	
Quick Sort	$O(nlogn)$	$O(n^2)$	$O(logn)$	depends	Can be implemented as a stable sort depending on how the pivot is handled.

Chapter 12 SEARCHING

What is Searching?

In computer science, searching is the process of finding an item with specified properties among a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs or may be elements of other search space.

Why Searching?

Searching is one of core computer science algorithms. We know that today's computers store lot of information. To retrieve this information efficiently we need very efficient searching algorithms.

There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in some proper order then it is easy to search the required element. Sorting is one of the techniques for making the elements ordered.

In this chapter we will see different searching algorithms.

Types of Searching

The following are the types of searches which we will be discussing in this book.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search
- Symbol Tables and Hashing
- String Searching Algorithms: Tries, Ternary Search and Suffix Trees

Unordered Linear Search

Let us assume that given an array whose elements order is not known. That means the elements of the array are not sorted. In this case if we want to search for an element then we have to scan the complete array and see if the element is there in the given list or not.

int UnsorteddLinearSearch (int A[], int n, int data)

```
{
        for (int i  = 0; i < n; i++)
        {
                if (A[i] == data)
                        return i;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array.
Space complexity: $O(1)$.

Sorted/Ordered Linear Search

If the elements of the array are already sorted then in many cases we don't have to scan the complete array to see if the element is there in the given array or not. In the below algorithm, it can be seen that, at any point if the value at $A[i]$ is greater than the $data$ to be searched then we just return -1 without searching the remaining array.

```
int SortedLinearSearch(int A[], int n, int data)
{
        for (int i  = 0; i < n; i++)
        {
                if (A[i] == data)
                        return i;
                else if(A[i] > data)
                        return -1;
        }
        return -1;
}
```

Time complexity of this algorithm is $O(n)$. This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is same.
Space complexity: $O(1)$.

Note: For the above algorithm we can make further improvement by incrementing the index at faster rate (say, 2). This will reduce the number of comparisons for searching in the sorted list.

Binary Search

If we consider searching of a word in a dictionary, in general we directly go some approximate page [generally middle page] start searching from that point. If the *name* that we are searching is same then we are done with the search. If the page is before the selected pages then apply the same process for the first half otherwise apply the same process to the second half. Binary search also works in the same way. The algorithm applying such a strategy is referred to as *binary search* algorithm.

```
//Iterative Binary Search Algorithm
int BinarySearchIterative[int A[], int n, int data)
{
        int low =  0;
        int high = n-1;
        while (low <= high)
        {
                mid  = low + (high-low)/2; //To avoid overflow

                if (A[mid] == data)
                        return mid;
                else if (A[mid] < data)
                        low =  mid + 1;
                else
                        high =  mid - 1;
        }
        return -1;
}

//Recursive Binary Search Algorithm
int BinarySearchRecursive[int A[], int low, int high, int data)
{
        int mid  = low + (high-low)/2; //To avoid overflow

        if (A[mid] == data)
                return mid;
        else if (A[mid] < data)
                return BinarySearchRecursive (A, mid + 1, high, data);
        else
                return BinarySearchRecursive (A, low, mid - 1 , data);

        return -1;
}
```

Recurrence for binary search is $T(n) = T(\frac{n}{2}) + \Theta(1)$. This is because we are always considering only half of the input list and throwing out the other half. Using *Divide and Conquer* master theorem, we get, $T(n) = O(logn)$.

Time Complexity: $O(logn)$.
Space Complexity: $O(1)$ [for iterative algorithm].

Comparing Basic Searching Algorithms

Implementation	Search-Worst Case	Search-Avg. Case
Unordered Array	n	$\frac{n}{2}$
Ordered Array	$logn$	$logn$
Unordered List	n	$\frac{n}{2}$
Ordered List	n	$\frac{n}{2}$
Binary Search (arrays)	$logn$	$logn$
Binary Search Trees (for skew trees)	n	$logn$

Note: For discussion on binary search trees refer *Trees* chapter.

Symbol Tables and Hashing

Refer *Symbol Tables* and *Hashing* chapters.

String Searching Algorithms

Refer *String Algorithms* chapter.

Problems on Searching

Problem-1 Given an array of n numbers. Give an algorithm for checking whether there are any duplicated elements in the array or not?

Solution: This is one of the simplest problems. One obvious answer to this is, exhaustively searching for duplicated in the array. That means, for each input element check whether there is any element with same value. This we can solve just by using two simple *for* loops. The code for this solution can be given as:

```
void CheckDuplicatesBruteForce(int A[], int n)
{
        int i =  0, j=0;
```

```
        for(i = 0; i < n; i++)
        {
                for(j = i+1; j < n; j++)
                {
                        if(A[i] == A[j])
                        {
                                printf("Duplicates exist: %d", A[i]);
                                return;
                        }
                }

        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-2 Can we improve the complexity of Problem-1's solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see if there are elements with same value and adjacent.

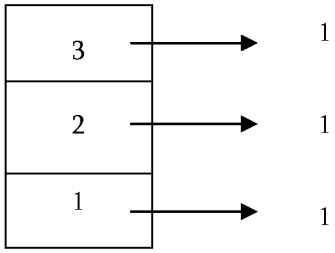
```
void CheckDuplicatesBruteForce(int A[], int n)
{
        //sort the array
        Sort(A, n);

        for(int i = 0; i < n-1; i++)
        {
                if(A[i] == A[i+1])
                {
                        printf("Duplicates exist: %d", A[i]);
                        return;
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(nlogn)$. This is because of sorting.
Space Complexity: $O(1)$.

Problem-3 Is there any other way of solving the Problem-1?

Solution: Yes, using hash table. Hash tables are a simple and effective method to implement dictionaries. *Average* time to search for an element is $O(1)$, while worst-case time is $O(n)$. Refer *Hashing* chapter for full details on hashing algorithms.

For example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. Scan the input array and insert the elements into the hash. For inserted element, keep the *counter* as 1. This indicates that the corresponding element has occurred already. For the given array, the hash table will look like (after inserting first three elements 3, 2 and 1):



If we try inserting 2, since the counter value of 2 is already 1, then we can say the element is appearing twice.

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

Problem-4 Can we further improve the complexity of Problem-1's solution?

Solution: Let us assume that the array elements are positive numbers and also all the elements are in the range 0 to $n - 1$. For each element $A[i]$, we go to the array element whose index is $A[i]$. That means we select $A[A[i]]$ and mark - $A[A[i]]$ (that means we negate the value at $A[A[i]]$). We continue this process until we encounter the element whose value is already negated. If one such element exists then we say duplicate elements exist in the given array. As an example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$.

Initially,

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

At step-1, negate A[abs(A[0])],

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

At step-2, negate A[abs(A[1])],

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

At step-3, negate A[abs(A[2])],

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

At step-4, negate A[abs(A[3])],

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

At step-4, we can observe that $A[abs(A[3])]$ is already negative. That means we have encountered the same value twice.

The code for this algorithm can be given as:

```
void CheckDuplicates(int A[], int n)
{
        int i =  0;
        for(i = 0; i < n; i++)
        {
                if(A[abs(A[i])] < 0)
                {
                        printf("Duplicates exist:%d", A[i]);
                        return;
                }
                else
                {
                        A[A[i]] = - A[A[i]];
                }
        }
        printf("No duplicates in given array.");
}
```

Time Complexity: $O(n)$. Since, only one scan is required.
Space Complexity: $O(1)$.

Note:

- This solution does not work if the given array is read only.
- This solution will work only if all the array elements are positive.
- If the elements range is not in 0 to $n - 1$ then it may give exceptions.

Problem-5 Given an array of n numbers. Give an algorithm for finding the first element in the array which is repeated?

For example, consider the array, $A = \{3, 2, 1, 2, 2, 3\}$. In this array the first repeated number is 3 (not 2). That means, we need to return the first element among the repeated elements.

Solution: We can use the brute force solution of Problem-1. Because it for each element it checks whether there is a duplicate for that element or not. So, whichever element duplicates first then that element is returned.

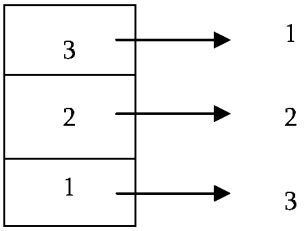
Problem-6 For Problem-5, can we use sorting technique?

Solution: No. For proving the failed case, let us consider the following array. For example, $A = \{3, 2, 1, 2, 2, 3\}$. Then after sorting we get $A = \{1, 2, 2, 2, 3, 3\}$. In this sorted array the first repeated element is 2 but the actual answer is 3.

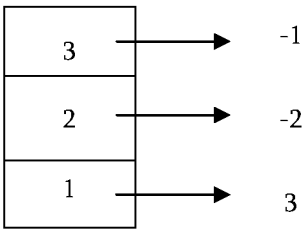
Problem-7 For Problem-5, can we use hashing technique?

Solution: Yes. But the simple technique which we used for Problem-3 will not work. For example, if we consider the input array as $A = \{3, 2, 1, 2, 3\}$, in this case the first repeated element is 3 but using our simple hashing technique we the answer as 2. This is because of the fact that 2 is coming twice before 3. Now let us change the hashing table behavior so that we get the first repeated element.

Let us say, instead of storing 1 value, initially we store the position of the element in the array. As a result the hash table will look like (after inserting 3, 2 and 1):



Now, if we see 2 again, we just negate the current value of 2 in the hash table. That means, we make its counter value as -2. The negative value in the hash table indicates that we have seen the same element two times. Similarly, for 3 (next element in input) also, we negate the current value of hash table and finally the hash table will look like:



After scanning the complete array, we scan the hash table and return the highest negative indexed value from it (i.e., -1 in our case). The highest negative value indicates that we have seen that element first (among repeated elements) and also repeating.

What if the element is repeated more than two times?

In this case, what we can do is, just skip the element if the corresponding value i already negative.

Problem-8 For Problem-5, can we use Problem-3's technique (negation technique)?

Solution: No. As a contradiction example, for the array $A = \{3, 2, 1, 2, 2, 3\}$ the first repeated element is 3. But with negation technique the result is 2.

Problem-9 Given an array of n elements. Find two elements in the array such that their sum is equal to given element K?

Solution: Brute Force Approach
One simple solution to this is, for each input element check whether there is any element whose sum is K. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int K)
{
        int i =  0, j = 0;
        for (i = 0; i < n; i++)
        {
                for(j = i; j < n; j++)
                {
                        if(A[i]+A[j] == K)
                        {
                                printf("Items Found:%d %d", i, j);
                                return;
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-10 Does the solution of Problem-9 works even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we get the pair of numbers if they exist.

Problem-11 For the Problem-9, can we improve the time complexity?

Solution: Yes. Let us assume that we have sorted the given array. This operation takes $O(n\ logn)$. On the sorted array, maintain indices $loIndex = 0$ and hiIndex = $n - 1$ and compute $A[loIndex] + A[hiIndex]$. If the sum equals K, then we are done with the solution. If the sum is less than K, decrement $hiIndex$, if the sum is greater than K, increment $loIndex$.

```
void Search[int A[], int n, int K)
{
        int i, j, temp;
        Sort(A, n);
        for(i = 0, j = n-1; i < j; )
        {
                temp = A[i] + A[j];
                if (temp == K)
                {
                        printf("Elements Found: %d %d", i, j);
                        return;
                }
                else if (temp < K)
                        i = i + 1;
                else
                        j = j - 1;
        }
        return;
}
```

Time Complexity: $O(nlogn)$. If the given array is already sorted then the complexity is $O(n)$.
Space Complexity: $O(1)$.

Problem-12 Is there any other way of solving the Problem-9?

Solution: Yes, using hash table.

Since our objective is to find two indexes of the array whose sum is K. Let us say those indexes are X and Y. That means, $A[X] + A[Y] = K$.

What we need is, for each element of the input array $A[X]$, check whether $K - A[X]$ also exists in input array. Now, let us simplify that searching with hash table.

Algorithm

- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Before proceeding to the next element we check whether $K - A[X]$ also exists in hash table or not.

- Existence of such number indicates that we are able to find the indexes.
- Otherwise proceed to the next input element.

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

Problem-13 Given an array A of n elements. Find three elements, i, j and k in the array such that $A[i]^2 + A[j]^2 = A[k]^2$?

Solution:

Algorithm

- For each array index i compute $A[i]^2$ and store in array.
- Now, the problem reduces to finding three indexes , i, j and k such that $A[i] + A[j] = A[k]$. This is same as that of Problem-9.

Problem-14 Two elements whose sum is closest to zero

Given an array with both positive and negative numbers. We need to find the two elements such that their sum is closest to zero. For the below array, algorithm should give -80 and 85.

Example: $1\ 60\ -10\ 70\ -80\ 85$

Solution: Brute Force Solution.

For each element, find the sum of it with every other element in the array and compare sums. Finally, return the minimum sum.

```
void TwoElementsWithMinSum(int A[], int n)
{
        int inv_count = 0;
        int i, j, min_sum, sum, min_i, min_j;

        if(n < 2)
        {
                printf("Invalid Input");
                return;
        }

        /* Initialization of values */
        min_i = 0;
        min_j = 1;
        min_sum = A[0] + A[1];
```

```
        for(i= 0; i < n - 1; i ++)
        {
                for(j = i + 1; j < n; j++)
                {
                        sum = A[i] + A[j];
                        if(abs(min_sum) > abs(sum))
                        {
                                min_sum = sum;
                                min_i = i;
                                min_j = j;
                        }
                }
        }
        printf(" The two elements are %d and %d", arr[min_i], arr[min_j]);
}
```

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

Problem-15 Can we improve the time complexity of Problem-14?

Solution: Use Sorting.

Algorithm

1) Sort all the elements of the given input array.
2) Find the two elements on either side of zero (if they are all positive or all negative then we are done with the solution)
3) If the one is positive and other is negative then add the two values at those positions. If the total is positive then increment the negative index, if it is negative then increment the positive index. If it is zero then stop.
4) loop step (3) until we hit a zero total or reached the end of array. Store the best total as you go.

Time Complexity: $O(nlogn)$, for sorting.

Problem-16 Given an array of n elements. Find three elements in the array such that their sum is equal to given element K?

Solution: Brute Force Approach.

The default solution to this is, for each pair of input elements check whether there is any element whose sum is K. This we can solve just by using three simple for loops. The code for this solution can be given as:

```
void BruteForceSearch[int A[], int n, int data)
{
        int i =  0, j = 0, k = 0;
        for (i = 0; i < n; i++)
        {
                for(j = i+1; j < n; j++)
                {
                        for(k = j+1; k < n; k++)
                        {
                                if(A[i] + A[j] + A[k]== data)
                                {
                                        printf("Items Found:%d %d %d", i, j, k);
                                        return;
                                }
                        }
                }
        }
        printf("Items not found: No such elements");
}
```

Time Complexity: $O(n^3)$. This is because of three nested for loops.
Space Complexity: $O(1)$.

Problem-17 Does the solution of Problem-16 works even if the array is not sorted?

Solution: Yes. Since we are checking all possibilities, the algorithm ensures that we can find three numbers whose sum is *K* if they exist.

Problem-18 Can we use sorting technique for solving Problem-16*?*

Solution: Yes.

```
void Search[int A[], int n, int data)
{
        int i, j;
        Sort(A, n);
        for(k = 0; k < n; k++)
        {
                for(i =  k + 1, j = n-1; i < j;  )
                {
                        if (A[k] + A[i] + A[j]  == data)
                        {
                                printf("Items Found:%d %d %d", i, j, k);
```

```
                        return;
                    }
                    else if (A[k] + A[i] + A[j] < data)
                        i = i + 1;
                    else
                        j = j - 1;
                }
            }
            return;
}
```

Time Complexity: Time for sorting + Time for searching in sorted list $= O(nlogn) + O(n^2) \approx O(n^2)$. This is because of two nested *for* loops.
Space Complexity: $O(1)$.

Problem-19 Can we use hashing technique for solving Problem-16*?*

Solution: Yes. Since our objective is to find three indexes of the array whose sum is K. Let us say those indexes are X, Y and Z. That means, $A[X] + A[Y] + A[Z] = K$.

Let us assume that we have kept all possible sums along with their pairs in hash table. That means the key to hash table is $K - A[X]$ and values for $K - A[X]$ are all possible pairs of input whose sum is $K - A[X]$.

Algorithm

- Before starting the searching, insert all possible sums with pairs of elements into the hash table.
- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Check whether there exists a hash entry in the table with key: $K - A[X]$.
- If such element exists then scan the element pairs of $K - A[X]$ and return all possible pairs by including $A[X]$ also.
- If no such element exists (with $K - A[X]$ as key) then go to next element.

Time Complexity: Time for storing all possible pairs in Hash table + searching = $O(n^2) + O(n^2) \approx O(n^2)$.
Space Complexity: $O(n)$.

Problem-20 Given an array of n integers, the $3 - sum\ problem$ is to determine find three integers whose sum is closest to $zero$.

Solution: This is same as that of Problem-16. In this case, K value is zero.

Problem-21 Given an array of n numbers. Give an algorithm for finding the element which appears maximum number of times in the array?

Solution: Brute Force.

One simple solution to this is, for each input element check whether there is any element with same value and for each such occurrence, increment the counter. Each time, check the current counter with the max counter and update it if this its value is greater than max counter. This we can solve just by using two simple for loops. The code for this solution can be given as:

```
int CheckDuplicatesBruteForce(int A[], int n)
{
        int i =  0, j=0;
        int counter =0, max=0;
        for(i = 0; i < n; i++)
        {
                counter=0;
                for(j = 0; j < n; j++)
                {
                        if(A[i] == A[j])
                                counter++;
                }
                if (counter > max)
                        max = counter;
        }
        return max;
}
```

Time Complexity: $O(n^2)$. This is because of two nested for loops.
Space Complexity: $O(1)$.

Problem-22 Can we improve the complexity of Problem-21 solution?

Solution: Yes. Sort the given array. After sorting all the elements with equal values come adjacent. Now, just do another scan on this sorted array and see which element is appearing maximum number of times.

Time Complexity: $O(nlogn)$. (for sorting).
Space Complexity: $O(1)$.

Problem-23 Is there any other way of solving Problem-21?

Solution: Yes, using hash table. For each element of the input keep track of how many times that element appeared in the input. That means the counter value represents the number of occurrences for that element.

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

Problem-24 For Problem-21, can we improve the time complexity? Assume that the elements range is o to $n - 1$. That means all the elements are within this range only.

Solution: Yes. We solve this problem in two scans. We *cannot* use the negation technique of Problem-3 for this problem because of number of repetitions.

In the first scan, instead of negating we add the value n. That means for each of occurrence of an element we add the array size to that element.

In the second scan we check the element value by dividing it with n and we return the element whichever gives the maximum value. The code based on this method is given below.

```
void MaxRepititions(int A[], int n)
{
        int i =  0;
        int max = 0;
        for(i = 0; i < n; i++)
        {
                A[A[i]%n] +=n;
        }
        for(i = 0; i < n; i++)
        {
                if(A[i]/n > max)
                {
                        max = A[i]/n;
                        max =i;
                }
        }
        return max;
}
```

Note:

- This solution does not work if the given array is read only.
- This solution will work only if the array elements are positive.
- If the elements range is not in 0 to $n - 1$ then it may give exceptions.

Time Complexity: $O(n)$. Since no nested for loops are required.

Space Complexity: $O(1)$.

Problem-25 Let A be an array of n distinct integers. Suppose A has the following property: there exists an index $1 \leq k \leq n$ such that $A[1], \ldots, A[k]$ is an increasing sequence and $A[k+1], \ldots, A[n]$ is a decreasing sequence. Design and analyze an efficient algorithm for finding k.

Similar question:

Lets us assume that the given array is sorted but starts with negative numbers and ends with positive numbers [such functions are called monotonically increasing function]. In this array find the starting index of the positive numbers. Let us assume that we know the length of the input array. Design a $O(logn)$ algorithm.

Solution: We use a variant of the binary search.

```
int Search (int A[], int first, int last)
{
        int first = 0;
        int last = n-1;
        int mid;

        while(first <= last)
        {
                // if the current array has size 1
                if(first == last)
                        return A[first];
                // if the current array has size 2
                else if(first == last-1)
                        return max(A[first], A[last]);
                // if the current array has size 3 or more
                else
                {
                        mid = first + (last-first)/2;

                        if(A[mid-1] < A[mid] && A[mid] > A[mid+1])
                                return A[mid];
                        else if(A[mid-1] < A[mid] && A[mid] < A[mid+1])
                                first = mid+1;
                        else if(A[mid-1] > A[mid] && A[mid] > A[mid+1])
                                last = mid-1;
                        else
                                return INT_MIN ;
                } // end of else
        } // end of while
```

}

The recursion equation is $T(n) = 2T(n/2) + c$. Using master theorem, we get $O(logn)$.

Problem-26 If we don't know n, how do we solve the Problem-25?

Solution: Repeatedly compute $A[1], A[2], A[4], A[8], A[16]$, and so on until we find a value of n such that $A[n] > 0$.

Time Complexity: $O(logn)$, since we are moving at the rate of 2.
Refer *Introduction to Analysis of Algorithms* chapter for details on this.

Problem-27 Given an input array of size unknown with all $1's$ in the beginning and $0's$ in the end. Find the index in the array from where $0's$ start. Consider there are millions of $1's$ and $0's$ in the array. E.g. array contents 1111111.....1100000.....0000000.

Solution: This problem is almost similar to Problem-26. Check the bits at the rate of $2^K$where $k = 0, 1, 2$ ....

Since we are moving at the rate of 2, the complexity is $O(logn)$.

Problem-28 Given a sorted array of n integers that has been rotated an unknown number of times, give a $O(log\ n)$ algorithm that finds an element in the array.
Example: Find 5 in array (15 16 19 20 25 1 3 4 5 7 10 14)
Output: 8 (the index of 5 in the array)

Solution: Let us assume that the given array is $A[]$. Using solution of Problem-25, with extension. The below function $FindPivot$ returns the k value (let us assume that this function return the index instead of value). Find the pivot point, divide the array in two sub-arrays and call binary search.

The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only element for which next element to it is smaller than it. Using above criteria and binary search methodology we can get pivot element in $O(logn)$ time

Algorithm

1) Find out pivot point and divide the array in two sub-arrays.
2) Now call binary search for one of the two sub-arrays.
    a. if element is greater than first element then search in left subarray
    b. else search in right subarray
3) If element is found in selected sub-array then return index *else* return -1.

```
int FindPivot(int A[], int start, int finish)
{
```

```
        if(finish - start == 0)
                return start;
        else if(start) == finish - 1)
        {
                if (A[start] >= A[finish])
                        return start;
                else
                        return finish;
        }
        else
        {
                mid = start + (finish-start)/2;
                if (A[mid] >= A[mid + 1])
                        return FindPivot(A, start, mid);
                else
                        return FindPivot(A, mid, finish);
        }
}

int Search(int A[], int n, int x)
{
        int pivot = FindPivot(A, 0, n-1);
        if(A[pivot] == x)
                return pivot;
        if(A[pivot] <= x)
                return BinarySearch(A, 0, pivot-1, x);
        else
                return BinarySearch(A, pivot+1, n-1, x);
}

int BinarySearch(int A[], int low, int high, int x)
{
        if(high >= low)
        {
                int mid = low + (high - low)/2;

                if(x == A[mid])
                        return mid;
                if(x > A[mid])
                        return BinarySearch(A, (mid + 1), high, x);
                else
                        return BinarySearch(A, low, (mid -1), x);
        }
        /*Return -1 if element is not found*/
```

```
        return -1;
}
```

Time complexity:$O(logn)$.

Problem-29 For Problem-28, can we solve in one scan?

Solution: Yes.

```
int BinarySearchRotated(int A[], int start, int finish, int data)
{
        int mid;
        if (start > finish)
                return -1;

        mid = start + (finish - start) / 2;

        if (data == A[mid])
                return mid;
        else if (A[start] <= A[mid])
        {       // start half is in sorted order.
                if (data >= A[start] && data < A[mid])
                        return BinarySearchRotated(A, start, mid - 1, data);
                else
                        return BinarySearchRotated(A, mid + 1, finish, data);
        }
        else
        {       // A[mid] <= A[finish], finish half is in sorted order.
                if (data > A[mid] && data <= A[finish])
                        return BinarySearchRotated(A, mid + 1, finish, data);
                else
                        return BinarySearchRotated(A, start, mid - 1, data);
        }
}
```

Time complexity:$O(logn)$.

Problem-30 **Bitonic search.**

An array is *bitonic* if it is comprised of an increasing sequence of integers followed immediately by a decreasing sequence of integers. Given a bitonic array A of n distinct integers, describe how to determine whether a given integer is in the array in $O(logn)$ steps.

Solution: This is same as Problem-25.

Problem-31 Yet, other way of asking Problem-25?
Let $A[]$ be an array that starts out increasing, reaches a maximum, and then decreases. Design an $O(logn)$ algorithm to find the index of the maximum value.

Problem-32 Give an $O(nlogn)$ algorithm for computing the median of a sequence of n integers.

Solution: Sort and return element at $n/2$.

Problem-33 Given two sorted lists of size m and n, find the median of all elements in $O(log\ (m + n))$ time.

Solution: Refer *Divide and Conquer* chapter.

Problem-34 Give a sorted array A of n elements, possibly with duplicates, find the index of the first occurrence of a number in $O(logn)$ time.

Solution: To find the first occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == low && A[mid] == data || A[mid] == data && A[mid-1] < data

```
int BinarySearchFirstOccurrence(int A[], int n, int low, int high, int data)
{
        int mid;
        if (high >= low)
        {
                mid = low + (high-low) / 2;

                if ((mid == low && A[mid] == data) || (A[mid] == data && A[mid - 1] < data))
                        return mid;

                // Give preference to left half of the array
                else if (A[mid] >= data)
                        return BinarySearchFirstOccurrence (A, n, low, mid - 1, data);
                else
                        return BinarySearchFirstOccurrence (A, n, mid + 1, high, data);
        }
        return -1;
}
```

Time Complexity: $O(logn)$.

Problem-35 Give a sorted array A of n elements, possibly with duplicates, find the index of the last occurrence of a number in $O(logn)$ time.

Solution: To find the last occurrence of a number we need to check for the following condition. Return the position if any one of the following is true:

mid == high && A[mid] == data || A[mid] == data && A[mid+1] > data

```
int BinarySearchLastOccurrence(int A[], int n, int low, int high, int data)
{
        int mid;
        if (high >= low)
        {
                mid = low + (high-low) / 2;

                if ((mid == high && A[mid] == data) || (A[mid] == data && A[mid + 1] > data))
                        return mid;

                // Give preference to right half of the array
                else if (A[mid] <= data)
                        return BinarySearchLastOccurrence (A, n, mid + 1, high, data);
                else
                        return BinarySearchLastOccurrence (A, n, low, mod - 1, data);
        }
        return -1;
}
```

Time Complexity: $O(logn)$.

Problem-36 Give a sorted array of n elements, possibly with duplicates, find the number of occurrences of a number.

Solution: Brute For Approach.

Do a linear search over the array and increment count as and when we find the element data in the array.

```
int LinearSearchCount(int A[], int n, int data)
{
        int count = 0;

        for (int i = 0; i < n; i++)
        {
                if (a[i] == k)
                        count++;
        }
        return count;
```

}

Time Complexity: $O(n)$.

Problem-37 Can we improve the time complexity of Problem-36?

Solution: Yes. We can solve this by using one binary search call followed by another small scan.

Algorithm

- Do a binary search for the $data$ in the array. Let us assume its position be K.
- Now traverse towards left from K and count the number of occurrences of $data$. Let this count be $leftCount$.
- Similarly, traverse towards right and count the number of occurrences of $data$. Let this count be $rightCount$.
- Total number of occurrences = $leftCount + 1 + rightCount$

Time Complexity – $O(logn + S)$ where S is the number of occurrences of $data$.

Problem-38 Is there any alternative way of solving the Problem-36*?*

Solution:

Algorithm

- Find the first occurrence of $data$ and call its index as $firstOccurrence$ (for algorithm refer Problem-34)
- Find the last occurrence of data and call its index as $lastOccurrence$ (for algorithm refer Problem-35)
- Return $lastOccurrence - firstOccurrence + 1$

Time Complexity = $O(logn + logn) = O(logn)$.

Problem-39 What is the next number in the sequence 1, 11, 21 and why?

Solution: Read the given number loudly. This is just a fun problem.

One one
Two Ones
One two, one one→ 1211

So answer is, the next number is the representation of previous number by reading it loudly.

Problem-40 Finding second smallest number efficiently.

Solution: We can construct a heap of the given elements using up just less than n comparisons (Refer *Priority Queues* chapter for algorithm). Then we find the second smallest using $logn$ comparisons for the GetMax() operation. Overall, we get $n + logn + constant$.

Problem-41 Is there any other solution for Problem-40?

Solution: Alternatively, split the n numbers into groups of 2, perform $n/2$ comparisons successively to find the largest using a tournament-like method. The first round will yield the maximum in $n - 1$ comparisons. The second round will be performed on the winners of the first round and the ones the maximum popped. This will yield $logn - 1$ comparisons for a total of $n + lg\, n - 2$. The above solution is called as *tournament problem.*

Problem-42 An element is a majority if it appears more than $n/2$ times. Give and algorithm takes an array of n element as argument and identifies a majority (if it exists).

Solution: The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than $n/2$ then break the loops and return the element having maximum count. If maximum count doesn't become more than $n/2$ then majority element doesn't exist.

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

Problem-43 Can we improve the Problem-42 time complexity to $O(nlogn)$?

Solution: Using binary search we can achieve this.

Node of the Binary Search Tree (used in this approach) will be as follows.

```
struct TreeNode
{
        int element;
        int count;
        struct TreeNode *left;
        struct TreeNode *right;
}BST;
```

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than $n/2$ then return. The method works well for the cases where $n/2 + 1$ occurrences of the majority element is present in the starting of the array, for example $\{1, 1, 1, 1, 1, 2, 3, \text{and } 4\}$.

Time Complexity: If a binary search tree is used then worst time complexity will be $O(n^2)$. If a balanced-binary-search tree is used then $O(nlogn)$.

Space Complexity: $O(n)$.

Problem-44 Is there any other of achieving $O(nlogn)$ complexity for the Problem-42?

Solution: Sort the input array and scan the sorted array to find the majority element.

Time Complexity: $O(nlogn)$.
Space Complexity: $O(1)$.

Problem-45 Can we improve the complexity for the Problem-42?

Solution: If an element occurs more than $n/2$ times in A then it must be the median of A. But, the reverse is not true, so once the median is found, we must check to see how many times it occurs in A. We can use linear selection which takes $O(n)$ time (for algorithm refer *Selection Algorithms* chapter).

int CheckMajority(int A[], in n)
{
1) Use linear selection to find the median m of A.
2) Do one more pass through A and count the number of occurrences of m.
   a. If m occurs more than $n/2$ times then return true;
   b. Otherwise return false.
}

Problem-46 Is there any other way of solving the Problem-42?

Solution: Since only one element is repeating, we can use simple scan of the input array by keeping track of count for the elements. If the count is 0 then we can assume that the element is coming first time otherwise that the resultant element.

```
int MajorityNum(int[] A, int n)
{
    int majNum, count;
    element = -1; count = 0;
    for(int i = 0; i < n; i++)
    {
            // If the counter is 0 then set the current candidate to majority num and
            // we set the counter to 1.
            if(count == 0)
            {
                    element = A[i];
```

```
                count = 1;
        }
        else if(element == A[i])
        {
                // Increment counter If the counter is not 0 and
                // element is same as current candidate.
                count++;
        }
        else
        {
                // Decrement counter If the counter is not 0 and
                // element is different from current candidate.
                count--;
        }
    }
    return element;
}
```

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

Problem-47 Given an array of $2n$ elements of which n elements are same and the remaining n elements are all different. Find the majority element.

Solution: The repeated elements will occupy half the array. No matter what arrangement it is, only one of the below will be true,

- All duplicate elements will be at a relative distance of 2 from each other. Ex: n, 1, n, 100, n, 54, n ...
- At least two duplicate elements will be next to each other
  Ex: $n, n, 1, 100, n, 54, n, \ldots.$
  $n, 1, n, n, n, 54, 100 \ldots$
  $1, 100, 54, n, n, n, n \ldots.$

So, in worst case, we need will two passes over the array,

First Pass: compare $A[i]$ and $A[i+1]$
Second Pass: compare $A[i]$ and $A[i+2]$

Something will match and that's your element.

This will cost $O(n)$ in time and $O(1)$ in space.

Problem-48 Given an array with $2n + 1$ integer elements, n elements appear twice in arbitrary places in the array and a single integer appears only once somewhere inside. Find the lonely integer with $O(n)$ operations and $O(1)$ extra memory.

Solution: Since except one element all other elements are repeated. We know that $A\ XOR\ A\ = 0$. Based on this if we XOR all the input elements then we get the remaining element.

```
int solution(int* A)
{
        int i, res;
        for (i = res = 0; i < 2n+1; i++)
                res = res ^ A[i];
         return res;
}
```

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

Problem-49 **Throwing eggs from an n-story building.**
Suppose that we have an n story building and a set of eggs. Also assume that an egg breaks if it is thrown off floor F or higher, and will not break otherwise. Devise a strategy to determine the floor F, while breaking $O(logn)$ eggs.

Solution: Refer *Divide and Conquer* chapter.

Problem-50 **Local minimum of an array.**
Given an array A of n distinct integers, design an $O(logn)$ algorithm to find a *local minimum*: an index i such that $A[i-1] < A[i] < A[i+1]$.

Solution: Check the middle value $A[n/2]$, and two neighbors $A[n/2 - 1]$ and $A[n/2 + 1]$. If $A[n/2]$ is local minimum, stop; otherwise search in half with smaller neighbor.

Problem-51 Give an $n \times n$ array of elements such that each row is in ascending order and each column is in ascending order, devise an $O(n)$ algorithm to determine if a given element x in the array. You may assume all elements in the $n \times n$ array are distinct.

Solution: Let us assume that the given matrix is $A[n][n]$. Start with the last row, first column [or first row - last column]. If the element we are searching for is greater than the element at $A[1][n]$, then the column 1 can be eliminated. If the search element is less than the element at $A[1][n]$, then the last row can be completely eliminated. Now, once the first column or the last row is eliminated, now, start over the process again with left-bottom end of the remaining array. In this algorithm, there would be maximum n elements that the search element would be compared with.

Time Complexity: $O(n)$. This is because we will traverse at most 2 n points.

Space Complexity: $O(1)$.

Problem-52 Given an $n \times n$ array a of n^2 numbers, Give an $O(n)$ algorithm to find a pair of indices i and j such that $A[i][j] < A[i+1][j], A[i][j] < A[i][j+1], A[i][j] < A[i-1][j]$, and $A[i][j] < A[i][j-1]$.

Solution: This problem is same as Problem-51.

Problem-53 Given $n \times n$ matrix, and in each row all 1's are followed $0's$. Find row with maximum number of $0's$.

Solution: Start with first row, last column. If the element is 0 then move to the previous column in the same row and at the same time increase the counter to indicate the maximum number of $0's$. If the element is 1 then move to the next row in the same column. Repeat this process until we reach last row, first column.

Time Complexity: $O(2n) \approx O(n)$ (very much similar to Problem-51).

Problem-54 Given an input array of size unknown with all numbers in the beginning and special symbols in the end. Find the index in the array from where special symbols start.

Solution: Refer *Divide and Conquer* chapter.

Problem-55 Finding the Missing Number

We are given a list of $n-1$ integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Given an algorithm to find the missing integer.

Example:

I/P [1, 2, 4, 6, 3, 7, 8]

O/P 5

Solution: Use sum formula

1) Get the sum of numbers, $sum = n * (n+1)/2$
2) Subtract all the numbers from sum and you will get the missing number.

Time Complexity: $O(n)$, this is because we need to scan the complete array.

Problem-56 In Problem-55, if the sum of the numbers goes beyond maximum allowed integer, then there can be integer overflow and we may not get correct answer. Can we solve this problem?

Solution:

1) XOR all the array elements, let the result of XOR be X.
2) XOR all numbers from 1 to n, let XOR be Y.
3) XOR of X and Y gives the missing number.

```
int FindMissingNumber(int A[], int n)
{
        int i, X, Y;
        for (i = 0; i < 9; i ++)
                X ^= A[i];
        for (i = 1; i <= 10; i ++)
                Y ^= i;
        //In fact, one variable is enough.
        return X ^ Y;
}
```

Time Complexity: $O(n)$, this is because we need to scan the complete array.

Problem-57 Find the Number Occurring Odd Number of Times

Given an array of positive integers, all numbers occurs even number of times except one number which occurs odd number of times. Find the number in $O(n)$ time & constant space.

Example:

I/P = $[1, 2, 3, 2, 3, 1, 3]$

O/P = 3

Solution: Do a bitwise XOR of all the elements. Finally we get the number which has odd occurrences. This is because of the fact that, $A\ XOR\ A = 0$.

Time Complexity: $O(n)$.

Problem-58 Find the two repeating elements in a given array

Given an array with $n + 2$ elements, all elements of the array are in range 1 to n and also all elements occur only once except two numbers which occur twice. Find those two repeating numbers.

Example: $6, 2, 6, 5, 2, 3, 1$ and $n = 5$

The above input has $n + 2 = 7$ elements with all elements occurring once except 2 and 6 which occur twice. So the output should be 6 2.

Solution: One simple way to scan the complete array for each element of the input elements. That means use two loops. In the outer loop, select elements one by one and count the number of occurrences of the selected element in the inner loop.

```
void PrintRepeatedElements(int A[], int n)
{
        int i, j;
        for(i = 0; i < n; i++)
                for(j = i+1; j <n; j++)
                        if(A[i] == A[j])
                                printf("%d", A[i]);
}
```

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

Problem-59 For the Problem-58, can we improve the time complexity?

Solution: Sort the array using any comparison sorting algorithm and see if there are any elements which contiguous with same value.

Time Complexity: $O(nlogn)$.
Space Complexity: $O(1)$.

Problem-60 For the Problem-58, can we improve the time complexity?

Solution: Use Count Array. This solution is like using a hash table. But for simplicity we can use array for storing the counts. Traverse the array once. While traversing, keep track of count of all elements in the array using a temp array $count[]$ of size n, when we see an element whose count is already set, print it as duplicate.

```
void PrintRepeatedElements(int A[], int n)
{
        int *count = (int *)calloc(sizeof(int), (n - 2));
        for(int i = 0; i < size; i++)
        {
                if(count[A[i]] == 1)
                        printf("%d", A[i]);
                else
                        count[A[i]]++;
        }
}
```

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

Problem-61 Consider the Problem-58. Let us assume that the numbers are in the range 1 to n. Is there any other way of solving the problem?

Solution: Using XOR Operation. Let the repeating numbers be X and Y, if we xor all the elements in the array and all integers from 1 to n, then the result is $X\ XOR\ Y$.

The 1's in binary representation of $X\ XOR\ Y$ is corresponding to the different bits between X and Y. Suppose that the k^{th} bit of $X\ XOR\ Y$ is 1, we can XOR all the elements in the array and all integers from 1 to n, whose k^{th} bits are 1. The result will be one of X and Y.

```
void PrintRepeatedElements (int A[], int size)
{
        int XOR = A[0];
        int right_most_set_bit_no;
        int n = size - 2;
        int X= 0, Y = 0;

        /* Compute XOR of all elements in A[]*/
        for(int i = 0; i < n; i++)
                XOR ^= A[i];

        /* Compute XOR of all elements {1, 2 ..n} */
        for(i = 1; i <= n; i++)
                XOR ^= i;

         /* Get the rightmost set bit in right_most_set_bit_no */
         right_most_set_bit_no = XOR & ~( XOR -1);

        /* Now divide elements in two sets by comparing rightmost set */
        for(i = 0; i < n; i++)
        {
                if(A[i] & right_most_set_bit_no)
                        X = X^ A[i]; /*XOR of first set in A[] */
                else
                        Y = Y ^ A[i]; /*XOR of second set inA[] */
        }
        for(i = 1; i <= n; i++)
        {
                if(i & right_most_set_bit_no)
                        X = X ^ i; /*XOR of first set in A[] and {1, 2, ...n }*/
                else
                        Y = Y ^ i; /*XOR of second set in A[] and {1, 2, ...n } */
        }

        printf("%d and %d",X, Y);
}
```

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

Problem-62 Consider the Problem-58. Let us assume that the numbers are in the range 1 to n. Is there yet other way of solving the problem?

Solution: We can solve this by creating two simple mathematical equations. Let us assume that two numbers which we are going to find are X and Y. We know the sum of n numbers is $n(n+1)/2$ and product is $n!$. Make two equations using these sum and product formulae, and get values of two unknowns using the two equations.

Let summation of all numbers in array be S and product be P and the numbers which are being repeated are X and Y.

$X + Y = S - n(n+1)/2$
$XY = P/n!$

Using above two equations, we can find out X and Y.
There can be addition and multiplication overflow problem with this approach.

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

Problem-63 Similar to Problem-58. Let us assume that the numbers are in the range 1 to n. Also, $n-2$ elements are repeating thrice and remaining two elements are repeating twice. Find the element which is repeating twice.

Solution: If we xor all the elements in the array and all integers from 1 to n, then the all the elements which are trice will become zero This is because, since the element is repeating trice and XOR with another time from range makes that element appearing four times. As a result, output of a $XOR\ a\ XOR\ a\ XOR\ a = 0$. Same is case with all elements which repeated thrice.

With the same logic, for the element which repeated twice, if we XOR the input elements and also the range, then the total number of appearances for that element is 3. As a result, output of $a\ XOR\ a\ XOR\ a = a$. Finally, we get the element which repeated twice.

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

Problem-64 Separate Even and Odd numbers

Given an array $A[\ ]$, write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers.

Example:
Input = {12, 34, 45, 9, 8, 90, 3}
Output = {12, 34, 90, 8, 9, 45, 3}

In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.

Solution: The problem is very similar to *Separate 0's and 1's (*Problem-65*)* in an array, and both of these problems are variation of famous *Dutch national flag problem.*

Algorithm: Logic is little similar to Quick sort.

1) Initialize two index variables left and right: $left = 0$, $right = n - 1$
2) Keep incrementing left index until we see an odd number.
3) Keep decrementing right index until we see an even number.
4) If $left < right$ then swap $A[left]$ and $A[right]$

Implementation:

```
void DutchNationalFlag(int A[], int n)
{
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
        while(left < right)
        {
                /* Increment left index while we see 0 at left */
                while(A[left]%2 == 0 && left < right)
                        left++;

                /* Decrement right index while we see 1 at right */
                while(A[right]%2 == 1 && left < right)
                        right--;

                if(left < right)
                {
                        /* Swap A[left] and A[right]*/
                        swap(&A[left], &A[right]);
                        left++;
                        right--;
                }
        }
}
```

Time Complexity: $O(n)$.

Problem-65 Other way of asking Problem-64 but with little difference.

Separate 0's and 1's in an array

We are given an array of 0's and 1's in random order. Separate 0's on left side and 1's on right side of the array. Traverse array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

Solution: Counting 0's or 1's

1. Count the number of $0's$. Let count be C.
2. Once we have count, we can put C 0's at the beginning and $1's$ at the remaining $n - C$ positions in array.

Time Complexity: $O(n)$. This solution scans the array two times.

Problem-66 Can we solve the Problem-65 in once scan?

Solution: Yes. Use two indexes to traverse:

Maintain two indexes. Initialize first index left as 0 and second index right as $n - 1$.

Do following while $left < right$:

1) Keep incrementing index left while there are 0s at it
2) Keep decrementing index right while there are 1s at it
3) If left < right then exchange $A[left]$ and $A[right]$

```
/*Function to put all 0s on left and all 1s on right*/
void Separate0and1(int A[], int n)
{
        /* Initialize left and right indexes */
        int left = 0, right = n-1;
       while(left < right)
       {
              /* Increment left index while we see 0 at left */
              while(A[left] == 0 && left < right)
                     left++;

              /* Decrement right index while we see 1 at right */
              while(A[right] == 1 && left < right)
                     right--;

              /* If left is smaller than right then there is a 1 at left
              and a 0 at right.  Swap A[left] and A[right]*/
              if(left < right)
              {
```

```
                    A[left] = 0;
                    A[right] = 1;
                    left++;
                    right–;
                }
        }
}
```

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

Problem-67 Maximum difference between two elements

Given an array $A[]$ of integers, find out the difference between any two elements such that larger element appears after the smaller number in $A[]$.

Examples: If array is $[2, 3, 10, 6, 4, 8, 1]$ then returned value should be 8 (Diff between 10 and 2). If array is $[\ 7, 9, 5, 6, 3, 2\ ]$ then returned value should be 2 (Difference between 7 and 9)

Solution: Refer *Divide and Conquer* chapter.

Problem-68 Given an array of 101 elements. Out of them 25 elements are repeated twice, 12 elements are repeated 4 times and one element is repeated 3 times. Find the element which repeated 3 times in $O(1)$.

Solution: Before solving this problem let us consider the following XOR operation property.

$$a\ XOR\ a = 0$$

That means, if we apply the XOR on same elements then the result is 0. Let us apply this logic for this problem.

Algorithm:

- XOR all the elements of the given array and assume the result is A.
- After this operation, 2 occurrences of number which appeared 3 times becomes 0 and one occurrence will remain.
- The 12 elements which are appearing 4 times become 0.
- The 25 elements which are appearing 2 times become 0.

So just $XOR'ing$ all the elements give the result.

Time Complexity: $O(n)$, because we are doing only once scan.
Space Complexity: $O(1)$.

Problem-69 Given an array A of n numbers. Find all pairs of X and Y in the array such that $K = X * Y$. Give an efficient algorithm without sorting.

Solution: Create a hash table from the numbers that divide K. Divide K by a number and check for quotient in the table.

Problem-70 Given a number n, give an algorithm for finding the number of trailing zeros in $n!$.

Solution:

```
int NumberOfTrailingZerosInNumber(int n)
{
        int i, count = 0;
        if (n < 0)
                return -1;
        for (i = 5; n / i > 0; i *= 5)
                count += n / i;
        return count;
}
```

Time Complexity: $O(logn)$,

SALIENT FEATURES OF BOOK

- ☞ All code written in *C*
- ☞ Enumeration of possible solutions for each problem
- ☞ Covers all topics for competitive exams
- ☞ Covers interview questions on data structures and algorithms
- ☞ Reference Manual for working people
- ☞ Campus Preparation
- ☞ Degree Masters Course Preparation
- ☞ Big Job Hunters: Microsoft, Google, Amazon, Yahoo, Oracle, Facebook and many more

ABOUT THE AUTHOR

Narasimha Karumanchi is the Senior Software Developer at **Amazon Corporation, India.** Most recently he worked for **IBM Labs, Hyderabad** and prior to that he served for **Mentor Graphics** and **Microsoft, Hyderabad.** He received his B-TECH. in Computer Science from **JNT University** and his M-Tech. in Computer Science from **IIT Bombay.**

He has experience in teaching data structures and algorithms at various training centers and colleges. He was born and bought up in Kambhampadu, Macherla (Palnadu), Guntur, Andhra Pradesh.

ISBN 978-0-615-45981-3

CareerMonk Publications

9 780615 459813 >