DATA STRUCTURES AND ALGORITHMS MADE EASY

Data Structures and Algorithmic Puzzles

5TH EDITION

CareerMonk
Together we are better..

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

Data Structures
And
Algorithms
Made Easy

-To All My Readers

By
Narasimha Karumanchi

☼ Concepts ☼ Problems ☼ Interview Questions

Table of Contents

CHAPTER

1

INTRODUCTION

The objective of this chapter is to explain the importance of the analysis of algorithms, their notations, relationships and solving as many problems as possible. Let us first focus on understanding the basic elements of algorithms, the importance of algorithm analysis, and then slowly move toward the other topics as mentioned above. After completing 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 this equation. The important thing that we need to understand is that the equation has names (*x* and *y*), which hold values (data). That means the *names* (*x* and *y*) are placeholders for representing data. Similarly, in computer science programming we need something for holding data, and *variables* is the way to do that.

1.2 Data Types

In the above-mentioned equation, the variables *x* and y can take any values such as integral numbers (10, 20), real numbers (0.23, 5.5), or just 0 and 1. To solve the equation, we need to relate them to the kind of values they can take, and *data type* is the name used in computer science programming for this purpose. A *data type* in a programming language is a set of data with predefined values. 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 we want to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers provide us with 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 an *integer*. Similarly, combining 4 bytes (32 bits) and calling it a *float*. A data type reduces the coding effort. 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 that are defined by system are called *primitive* data types. The primitive data types 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, the compiler and the 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 change.

For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits), then the total possible values are minus 32,768 to plus 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 and +2,147,483,647 (-2^{31} *to* $2^{31}-1$). The same is the case with other data types.

User defined data types

If the system-defined data types are not enough, then most programming languages allow the users

to define their own data types, called *user – defined data types*. Good examples of user defined data types are: structures in *C/C + +* and classes in *Java*. For example, in the snippet below, we are combining many system-defined data types and calling the user defined data type by the 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 Structures

Based on the discussion above, 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. A *data structure* is a special 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. *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, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also 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 problems, we combine the data structures with their operations and we call this *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 others. 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 worry about the implementation details. They come into the 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 *omelette*. To prepare an omelette, we follow the steps given below:

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 omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as:

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

In the traditional study of algorithms, there are two main criteria for judging the merits of algorithms: correctness (does the algorithm give solution to the problem in a finite number of steps?) and efficiency (how much resources (in terms of memory and time) does it take to execute the).

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

1.6 Why the 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 bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

1.7 Goal of the Analysis of Algorithms

The goal of the *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, developer 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 the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs.

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

1.9 How to Compare Algorithms

To compare algorithms, let us define a *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 express the running time of a 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 go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

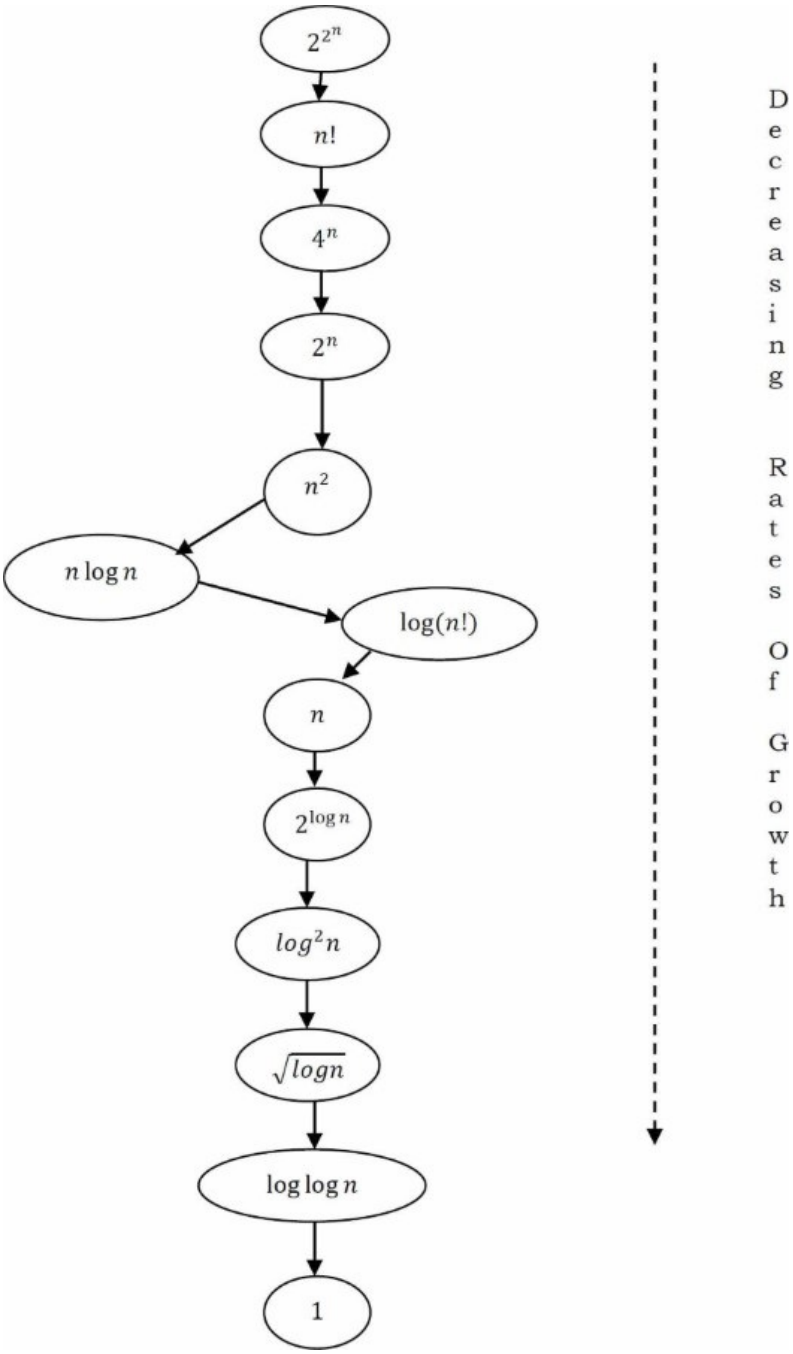
$$Total\ Cost = cost\_of\_car + cost\_of\_bicycle$$
$$Total\ Cost \approx cost\_of\_car\ (approximation)$$

For the above-mentioned example, we can represent the cost of the car and the cost of the bicycle 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 case below, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate 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 Rates of Growth

The diagram below shows the relationship between different rates of growth.



Below is the list of growth rates you will come across in the following 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

1.12 Types of Analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing wel1) and with which inputs the algorithm takes a long 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 the case where it takes less time and another for the case where it takes more time.

In general, the first case is called the *best case* and the second case is called the *worst case* for 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 a long time (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.
- **Best case**
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

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

$$f(n) = n^2 + 500, \text{ for worst case}$$
$$f(n) = n + 100n + 500, \text{ for best case}$$

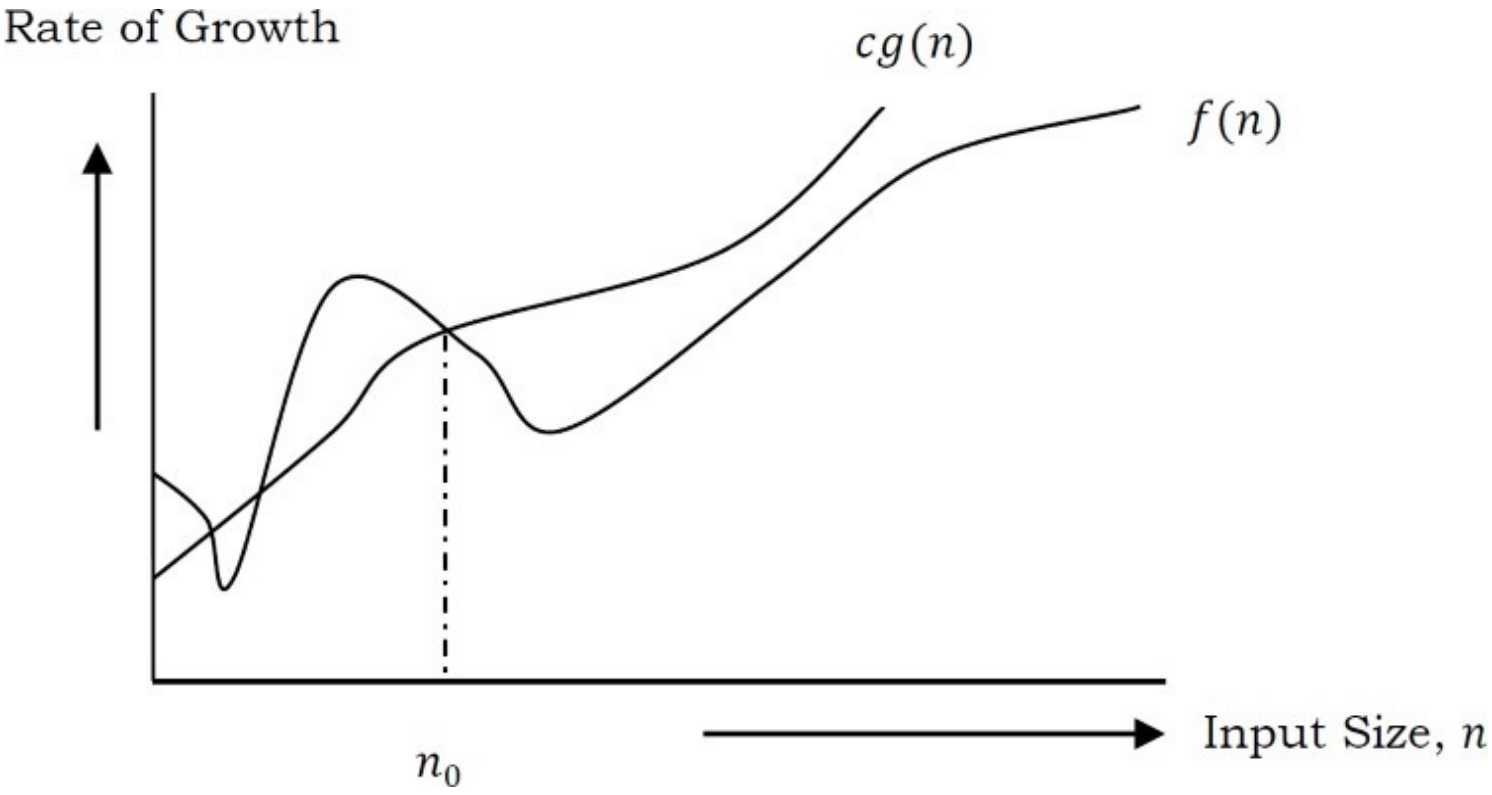
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

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

1.14 Big-O Notation [Upper Bounding Function]

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.



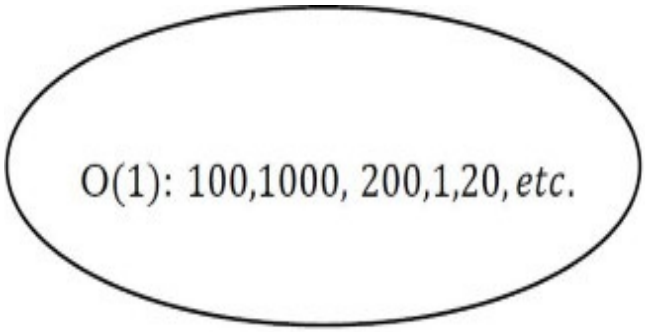
Let us see the O–notation with a little more detail. O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n > n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth $g(n)$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

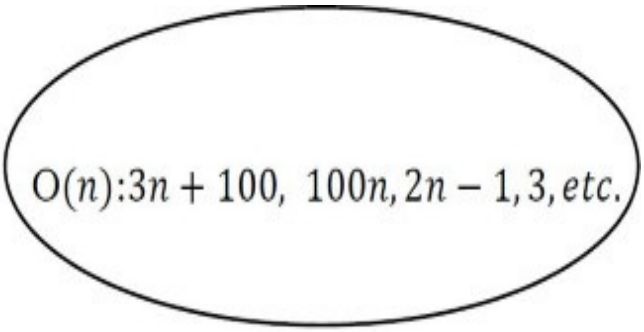
Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0, the rate of growth could be different. n_0 is called threshold for the given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

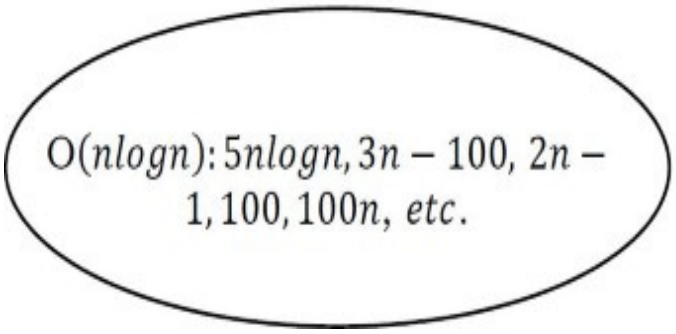
Big-O Visualization

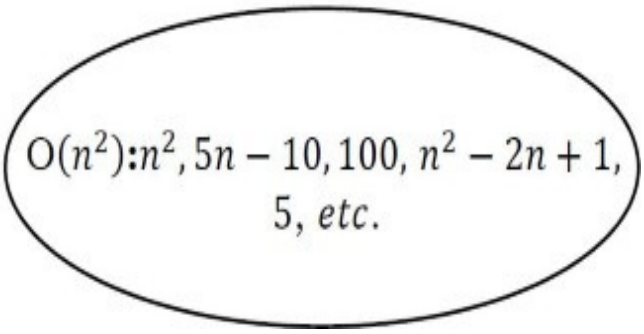
$O(g(n))$ is the set of functions with smaller or the same order of growth as $g(n)$. For example; $O(n^2)$ includes $O(1)$, $O(n)$, $O(nlogn)$, etc.

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









Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \le 4n$, for all $n \ge 8$

$\therefore 3n + 8 = O(n)$ 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 $n > 1$
$\therefore 2n^3 - 2n^2 = O(n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$

Solution: $n \leq n$, for all $n \geq 1$
$\therefore n = O(n)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n > 1$
$\therefore 410 = O(1)$ with $c = 1$ and $n_0 = 1$

No Uniqueness?

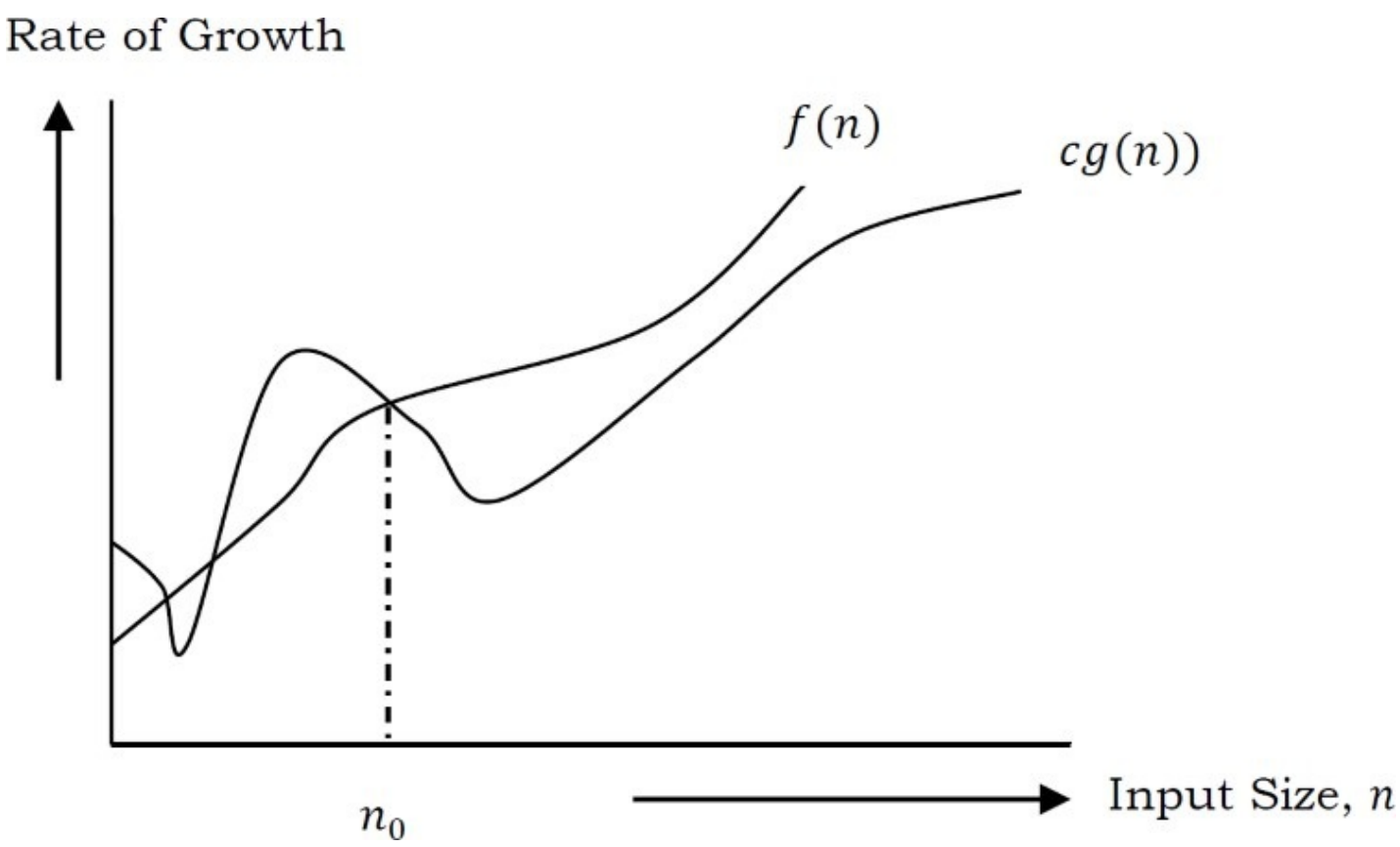
There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n$, for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.

Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n$, for all $n > 1$, $n_0 = 1$ and $c = 105$ is also a solution.

1.15 Omega-Q Notation [Lower Bounding Function]

Similar to the 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 $0 \le cg(n) \le f(n)$ for all $n \ge n_0$}. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give the largest rate of growth $g(n)$ which is less than or equal to the given algorithm's rate of growth $f(n)$.

Ω Examples

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

Solution: ∃ c, n_0 Such that: $0 \le cn^2 \le 5n^2 \Rightarrow cn^2 \le 5n^2 \Rightarrow c = 5$ and $n_0 = 1$

$\therefore 5n^2 = \Omega(n^2)$ with $c = 5$ and $n_0 = 1$

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

Solution: ∃ c, n_0 Such that: $0 \le cn^2 \le 100n + 5$

$100n + 5 \le 100n + 5n(\forall n \ge 1) = 105n$

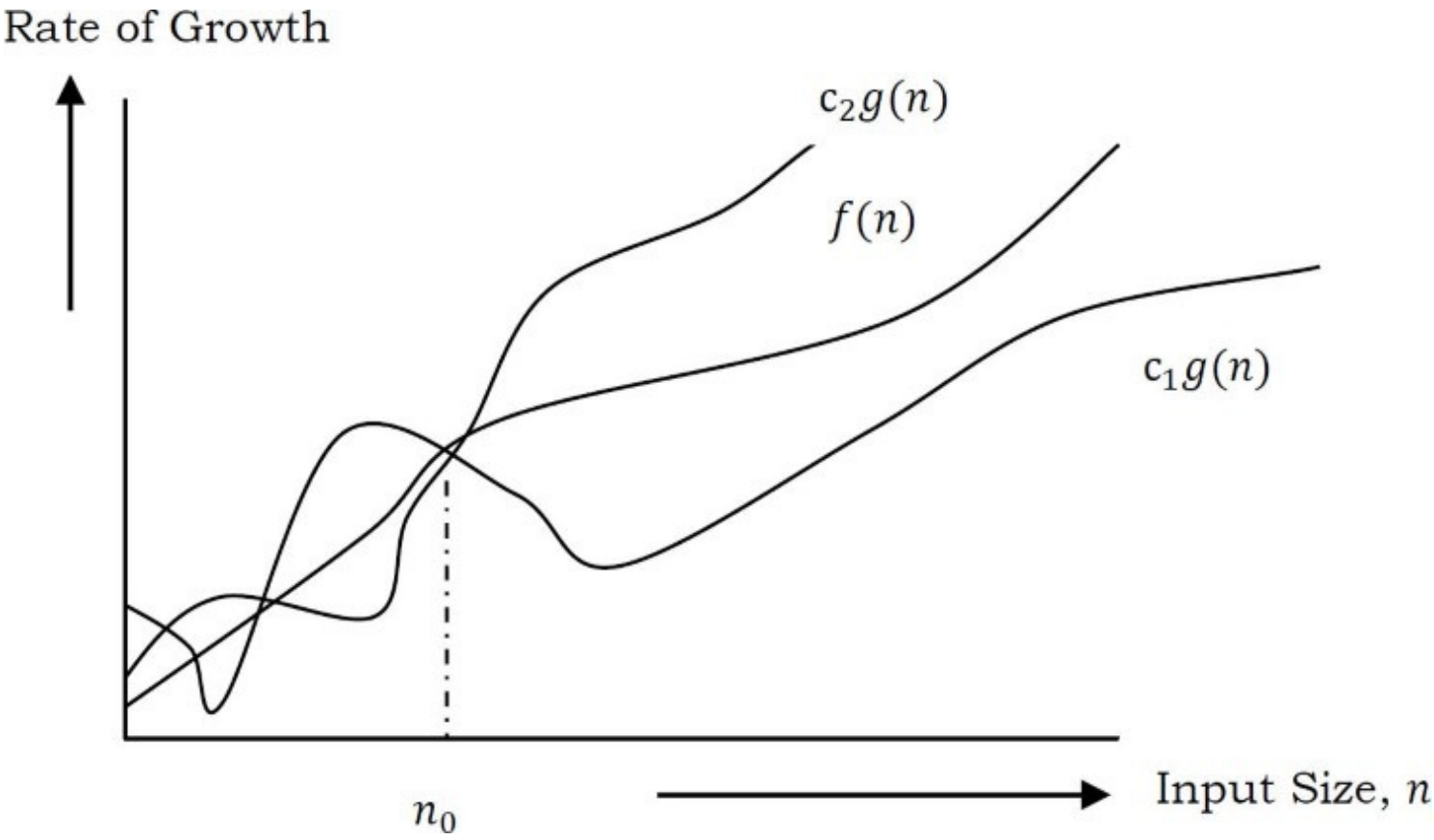
$cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

⇒ Contradiction: n cannot be smaller than a constant

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

1.16 Theta-Θ Notation [Order Function]



This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound (O) and lower bound (Ω) give the same result, then the 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 the best case is $g(n) = O(n)$.

In this case, the rates of growth in the best case and worst case are the same. As a result, the average case will also be the same. For a given function (algorithm), if the rates of growth (bounds) for O and Ω are not the same, then the rate of growth for the Θ case may not be the same. In this case, we need to consider all possible time complexities and take the average of those (for example, for a quick sort average case, refer to the *Sorting* chapter).

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 $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ 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, $n \geq 2$

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

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

1.17 Important Notes

For analysis (best case, worst case and average), we try to give the 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 the upper bound (O) and lower bound (Ω) and average running time (Θ) may not always be possible. For example, if we are discussing the best case of an algorithm, we try to give the upper bound (O) and lower bound (Ω) and average running time (Θ).

In the remaining chapters, we generally focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

1.18 Why is it called Asymptotic Analysis?

From the discussion above (for all 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 another 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 a curve an *asymptotic curve*. In other terms, $g(n)$ is the asymptotic

curve for $f(n)$. For this reason, we call algorithm analysis *asymptotic analysis*.

1.19 Guidelines for Asymptotic Analysis

There are some general rules to help us determine 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 $c \times n = c\,n = O(n)$.

2) **Nested loops:** Analyze from the 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 executes 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
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executes 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 $2^k = n$, and at $(k + 1)^{th}$ step we come out of the *loop*. Taking logarithm on both sides, gives

$$log(2^k) = logn$$
$$klog2 = logn$$
$$k = logn \qquad //\text{if we assume base-2}$$

Total time = O(*logn*).

Note: Similarly, for the case below, the worst case rate of growth is O(*logn*). The same discussion holds good for the 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 the word towards the left or right of center?
- Repeat the process with the left or right part of the dictionary until the word is found.

1.20 Simplyfying properties of asymptotic 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.
- 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 $g(n) = \Omega(f(n))$.
- If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n)), (g_1(n)))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)\ f_2(n)$ is in $O(g_1(n)\ g_1(n))$.

1.21 Commonly used Logarithms and Summations

Logarithms

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

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

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

1.22 Master Theorem for Divide and Conquer Recurrences

All divide and conquer algorithms (also discussed in detail in the *Divide and Conquer* chapter) divide the problem into sub-problems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, a merge sort algorithm [for details, refer to *Sorting* chapter] operates on two sub-problems, 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 $T(n) = O(n^k)$

1.23 Divide and Conquer Master Theorem: Problems & Solutions

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

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

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

Problem-4 $T(n) = 2^n T(n/2) + n^n$
Solution: $T(n) = 2^n T(n/2) + n^n =>$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$
Solution: $T(n) = 16T(n/4) + n => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

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

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

Problem-7 $T(n) = 2T(n/2) + n/logn$
Solution: $T(n) = 2T(n/2) + n/logn => T(n) = \Theta(nloglogn)$ (Master Theorem Case 2. b)

Problem-8 $T(n) = 2T(n/4) + n^{051}$
Solution: $T(n) = 2T(n/4) + n^{051} => T(n) = \Theta(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$
Solution: $T(n) = 0.5T(n/2) + 1/n$ => Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$
Solution: $T(n) = 6T(n/3) + n^2logn => T(n) = \Theta(n^2logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2logn$
Solution: $T(n) = 64T(n/8) - n^2logn$ => Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$
Solution: $T(n) = 7T(n/3) + n^2 => T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$
Solution: $T(n) = 4T(n/2) + logn => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$
Solution: $T(n) = 16T(n/4) + n! => T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$
Solution: $T(n) = \sqrt{2}T(n/2) + logn => T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$
Solution: $T(n) = 3T(n/2) + n => T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$
Solution: $T(n) = 3T(n/3) + \sqrt{n} => T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$
Solution: $T(n) = 4T(n/2) + cn => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$
Solution: $T(n) = 3T(n/4) + nlogn => T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$
Solution: $T(n) = 3T(n/3) + n/2 => T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

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

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

1.25 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 $O(nlogn)$.

1.26 Method of Guessing and Confirming

Now, let us discuss a method which can be used to solve any recurrence. The basic idea behind this method is:

guess the answer; and then *prove* it correct by induction.

In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

As an example, consider the recurrence $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems. Carefully observing the recurrence gives us the impression that it is similar to the divide and conquer method (dividing the problem into $\sqrt{n}$ subproblems each with size $\sqrt{n}$). As we can see, the size of the subproblems at the first level of recursion is n. So, let us guess that $T(n) = O(nlogn)$, and then try to prove that our guess is correct.

Let's start by trying to prove an *upper* bound $T(n) < cnlogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.\, c\sqrt{n}\, log\sqrt{n} + n \\ &= n.\, c\, log\sqrt{n} + n \\ &= n.c.\frac{1}{2}.logn + n \\ &\leq cnlogn \end{aligned}$$

The last inequality assumes only that $1 \leq c.\frac{1}{2}.logn$. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for the upper bound. Now, let us prove the *lower* bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.\, k\, \sqrt{n}\, log\sqrt{n} + n \\ &= n.\, k\, log\sqrt{n} + n \\ &= n.k.\frac{1}{2}.logn + n \\ &\geq knlogn \end{aligned}$$

The last inequality assumes only that $1 \geq k.\frac{1}{2}.logn$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for the lower bound.

From the above discussion, we understood that $\Theta(nlogn)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \quad \geq n$$

Now, let us prove the upper bound for this $\Theta(n)$.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n} + n \\ &= n.\, c + n \\ &= n\,(c + 1) \\ &\nleq cn \end{aligned}$$

From the above induction, we understood that $\Theta(n)$ is too small and $\Theta(nlogn)$ is too big. So, we need something bigger than n and smaller than $nlogn$. How about $n\sqrt{logn}$?

Proving the upper bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ c.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\leq cnlog\sqrt{n} \end{aligned}$$

Proving the lower bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ k.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\not\geq knlog\sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n\sqrt{logn})$ doesn't work. What else is between *n* and *nlogn?* How about *nloglogn?* Proving upper bound for *nloglogn:*

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ c.loglogn\text{-}c.n + n \\ &\leq cnloglogn, \text{ if } c \geq 1 \end{aligned}$$

Proving lower bound for *nloglogn:*

$$\begin{aligned} T(n) &= \sqrt{n}\,T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ k.loglogn\text{-}k.n + n \\ &\geq knloglogn, \text{ if } k \leq 1 \end{aligned}$$

From the above proofs, we can see that $T(n) \leq cnloglogn$, if $c \geq 1$ and $T(n) \geq knloglogn$, if $k \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(nloglogn)$.

1.27 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 *change 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 the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. 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 the next few operations become easier.

Example: Let us consider an array of elements from which we want to find the 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. The cost of performing the 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.

1.28 Algorithms Analysis: Problems & Solutions

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

Problem-21 Find the complexity of the below recurrence:

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

Solution: Let us try solving this function with substitution.

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

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

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

.

.

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

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.

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

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

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

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

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

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

$T(n) = 1$

∴ Time 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 the 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 '*s*' terms according to the relation $s_i = s_{i-1} + i$. The value oft' increases by 1 for each iteration. The value contained in '*s*' at the i^{th} iteration is the sum of the first '('positive integers. If *k* is the total number of iterations taken by the program, then the *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-mentioned function the loop will end, if $i^2 > n \Rightarrow T(n) = O(\sqrt{n})$. This is similar to Problem-23.

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

```
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+1)
                        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+1)
                        //inner 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 program given below:

```
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)
            //inner 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 program below.

```
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 function below.

```
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, 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 the function below:

```
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 3^i 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:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$
$$\dots$$
$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$
$$T(n) = T(1) + \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i$$
$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$
$$T(n) = \Theta(n^3)$$

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 the running time of this program is: T(n) = $T(n-1) + T(n-2) + c$. Note $T(n)$ 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) computations 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 the function below:

```
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 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 the 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 the function below:

```
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 the function below:

```
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 format. Let us try to convert this to the master theorem format by assuming $n = 2^m$. Applying the logarithm on both sides gives, $logn = mlogl \Rightarrow m = logn$. 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 $S(m) = T(2^m) \Longrightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Longrightarrow S(m) = 2S\left(\frac{m}{2}\right) + m.$

Applying the master theorem format would result in $S(m) = O(mlogm)$.
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 in $S(m) = O(logm)$. 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 $S(m) = O(m^{log_2^2})$. 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 the function below:

```
int Function (int n) {
   if(n <= 2) return 1;          //constant time
   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 pseudo-code 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 pseudo-code 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 pseudocode:

```
temp = 1
repeat
        for i = 1 to n
                temp = temp + 1;
        n = n/2;
until n <= 1
```

Solution: Consider the comments in the pseudocode below:

```
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 the 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 the 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 the 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 the 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 the below function:

```
void function(int n) {
    if(n <= 1) return; //constant time
    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 $\sum_{i \le k \le n} O(n)$, 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). $\sum_{i \le k \le n} O(n) = O(n) \sum_{i \le k \le n} 1 = O(n^2)$.

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) $(n + k)^m = n^h + c1*n^{k-1} + \ldots k^m = \Theta(n^h)$ 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 behavior 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) $h(n) = O(f(n)); g(n) = \Omega(f(n))$

Solution: (D). According to the rate of growth: $h(n) < f(n) < g(n)$ ($g(n)$ is asymptotically greater than $f(n)$, and $f(n)$ is asymptotically greater than $h(n)$). We can easily see the 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) $\text{ceil}(log_2^n) + 1$
(B) n
(C) $\text{ceil}(log_2^n)$
(D) $\text{floor}(log_2^n) + 1$

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 the loop, the answer is $\text{ceil}(log_2^n) + 1$.

Problem-54 Consider the following C code segment. Let $T(n)$ denote 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 $T(n) = \Omega(\sqrt{n})$
(B) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(1)$
(C) $T(n) = O(n)$ and $T(n) = \Omega(\sqrt{n})$
(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) $\Theta(log_2^n)$
(B) Ω(n)
(C) $\Theta(log_2 log_2^n)$
(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, the running time is O(1) which contradicts every option.

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

(A) $T(n) = O(n^2)$
(B) $T(n) = \Theta(nlogn)$
(C) $T(n) = Q(n^2)$
(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 the same. That means, O($nlogn$) and Ω($nlogn$) are correct for given recurrence. So option (C) is wrong.

Problem-57 Find the complexity of the below function:

```
function(int n) {
    for (int i = 0; i<n; i++)
        for(int j=i; j<i*i; j++)
            if (j %i == 0){
                for (int k = 0; k < j; k++)
                    printf(" * ");
            }

}
```

Solution:

```
function(int n) {
    for (int i = 0; i<n; i++)                  // Executes n times
        for(int j=i; j<i*i; j++)               // Executes n*n times
            if (j %i == 0){
                for (int k = 0; k < j; k++)    // Executes j times = (n*n) times
                    printf(" * ");
            }
}
```

Time Complexity: $O(n^5)$.

Problem-58 To calculate 9^n, give an algorithm and discuss its complexity.

Solution: Start with 1 and multiply by 9 until reaching 9^n.

Time Complexity: There are $n - 1$ multiplications and each takes constant time giving a $\Theta(n)$ algorithm.

Problem-59 For Problem-58, can we improve the time complexity?

Solution: Refer to the *Divide and Conquer* chapter.

Problem-60 Find the time complexity of recurrence $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$.

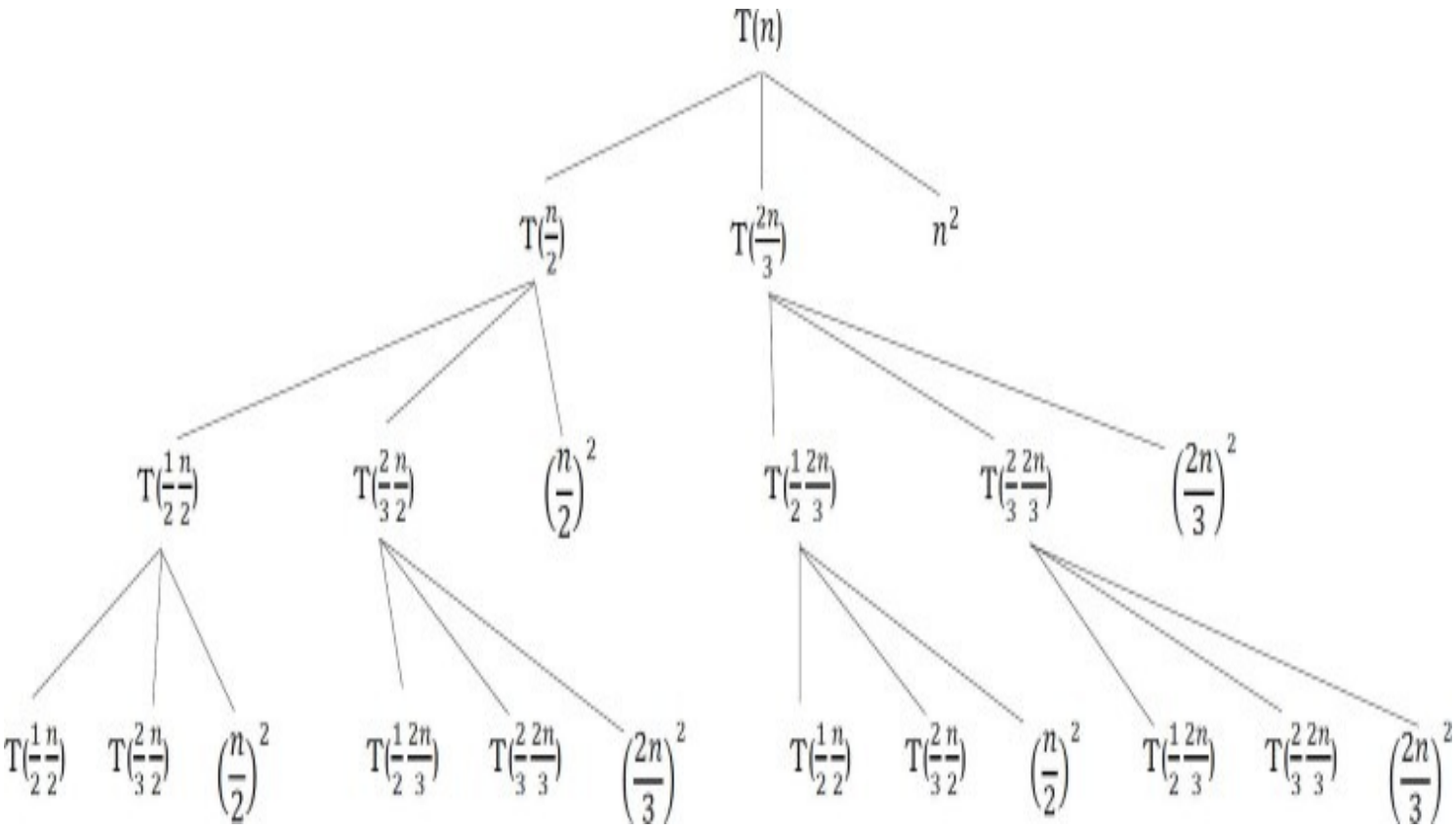
Solution: Let us solve this problem by method of guessing. The total size on each level of the recurrance tree is less than n, so we guess that $f(n) = n$ will dominate. Assume for all $i < n$ that $c_1 n \leq T(i) < c_2 n$. Then,

$$
\begin{aligned}
c_1\frac{n}{2} + c_1\frac{n}{4} + c_1\frac{n}{8} + kn \quad &\le T(n) \le \quad c_2\frac{n}{2} + c_2\frac{n}{4} + c_2\frac{n}{8} + kn \\
c_1 n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_1}\right) \quad &\le T(n) \le \quad c_2 n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_2}\right) \\
c_1 n\left(\frac{7}{8} + \frac{k}{c_1}\right) \quad &\le T(n) \le \quad c_2 n\left(\frac{7}{8} + \frac{k}{c_2}\right)
\end{aligned}
$$

If $c_1 \geq 8k$ and $c_2 \leq 8k$, then $c_1 n = T(n) = c_2 n$. So, $T(n) = \Theta(n)$. In general, if you have multiple recursive calls, the sum of the arguments to those calls is less than n (in this case $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} < n$), and $f(n)$ is reasonably large, a good guess is $T(n) = \Theta(f(n))$.

Problem-61 Solve the following recurrence relation using the recursion tree method: $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{2n}{3}\right) + n^2$.

Solution: How much work do we do in each level of the recursion tree?



In level 0, we take n^2 time. At level 1, the two subproblems take time:

$$\left(\frac{1}{2}n\right)^2 + \left(\frac{2}{3}n\right)^2 = \left(\frac{1}{4} + \frac{4}{9}\right)n^2 = \left(\frac{25}{36}\right)n^2$$

At level 2 the four subproblems are of size $\frac{1}{2}\frac{n}{2}$, $\frac{2}{3}\frac{n}{2}$, $\frac{1}{2}\frac{2n}{3}$ and $\frac{2}{3}\frac{2n}{3}$ respectively. These two subproblems take time:

$$\left(\frac{1}{4}n\right)^2+\left(\frac{1}{3}n\right)^2+\left(\frac{1}{3}\right)n^2+\left(\frac{4}{9}\right)n^2=\frac{625}{1296}n^2=\left(\frac{25}{36}\right)^2 n^2$$

Similarly the amount of work at level *k* is at most $\left(\frac{25}{36}\right)^k n^2$.

Let $\alpha = \frac{25}{36}$, the total runtime is then:

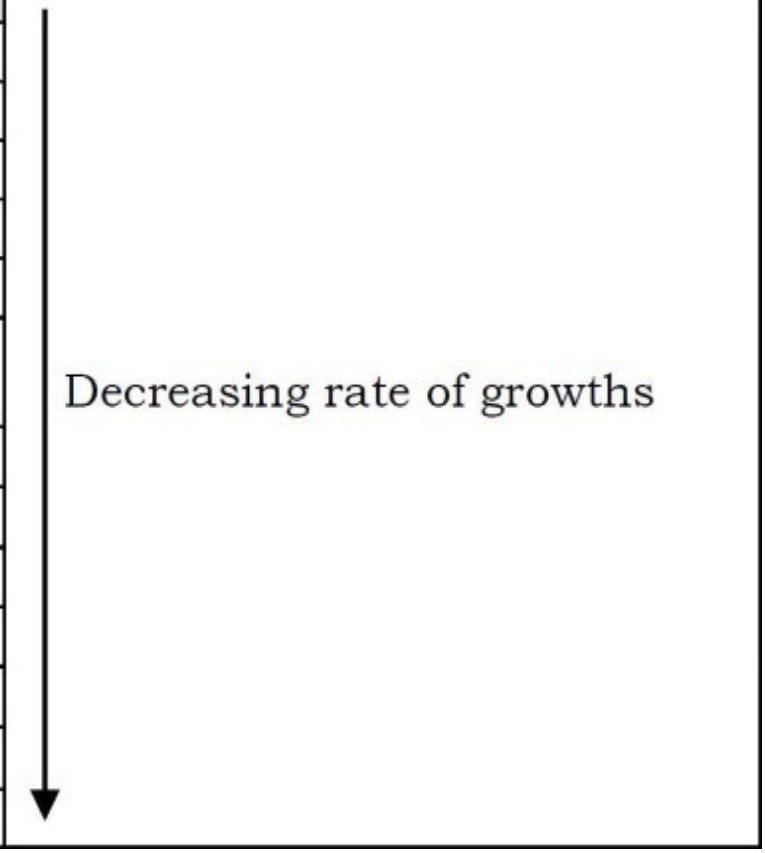
$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\infty} \alpha^k n^2 \\ &= \frac{1}{1-\alpha} n^2 \\ &= \frac{1}{1-\frac{25}{36}} n^2 \\ &= \frac{1}{\frac{11}{36}} n^2 \\ &= \frac{36}{11} n^2 \\ &= O(n^2) \end{aligned}$$

That is, the first level provides a constant fraction of the total runtime.

Problem-62 Rank the following functions by order of growth: $(n + 1)!$, $n!$, 4^n, $n \times 3^n$, $3^n + n^2 + 20n$, $(\frac{3}{2})^n$, $n^2 + 200$, $20n + 500$, 2^{lgn}, $n^{2/3}$, 1.

Solution:

Function	Rate of Growth
$(n+1)!$	$O(n!)$
$n!$	$O(n!)$
4^n	$O(4^n)$
$n \times 3^n$	$O(n3^n)$
$3^n + n^2 + 20n$	$O(3^n)$
$(\frac{3}{2})^n$	$O((\frac{3}{2})^n)$
$4n^2$	$O(n^2)$
4^{lgn}	$O(n^2)$
$n^2 + 200$	$O(n^2)$
$20n + 500$	$O(n)$
2^{lgn}	$O(n)$
$n^{2/3}$	$O(n^{2/3})$
1	$O(1)$



Problem-63 Find the complexity of the below function:

```
function(int n) {
    int sum = 0;
    for (int i = 0; i<n; i++)
        if (i>j)
            sum = sum +1;
        else {
            for (int k = 0; k < n; k++)
                sum = sum -1;
        }
    }
}
```

Solution: Consider the worst-case.

```
function(int n) {
   int sum = 0;
      for (int i = 0; i<n; i++)                  // Executes n times
      if (i>j)
         sum = sum +1;                           // Executes n times
      else {
           for (int k = 0; k < n; k++)           // Executes n times
              sum = sum -1;
      }
   }
}
```

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

Problem-64 Can we say $3^{n^{0.75}} = O(3^n)$??

Solution: Yes: because $3^{n^{0.75}} < 3^{n^1}$

Problem-65 Can we say $2^{3n} = O(2^n)$?

Solution: No: because $2^{3n} = (2^3)^n = 8^n$ not less than 2^n.

CHAPTER

2

RECURSION AND BACKTRACKING

2.1 Introduction

In this chapter, we will look at one of the important topics, *"recursion"*, which will be used in almost every chapter, and also its relative *"backtracking"*.

2.2 What is Recursion?

Any function which calls itself is called *recursive*. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the recursion step. The recursion step can result in many more such recursive calls.

It is important to ensure that the recursion terminates. Each time the function calls itself with a slightly simpler version of the original problem. The sequence of smaller problems must eventually converge on the base case.

2.3 Why Recursion?

Recursion is a useful technique borrowed from mathematics. Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.

Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

2.4 Format of a Recursive Function

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called the *base case*. The former, where the function calls itself to perform a subtask, is referred to as the *ecursive case*. We can write all recursive functions using the format:

```
if(test for the base case)
   return some base case value
else if(test for another base case)
   return some other base case value
// the recursive case
else
   return (some work and then a recursive call)
```

As an example consider the factorial function: $n!$ is the product of all integers between n and 1. The definition of recursive factorial looks like:

$$n! = 1, \quad \text{if } n = 0$$
$$n! = n * (n-1)! \text{ if } n > 0$$

This definition can easily be converted to recursive implementation. Here the problem is determining the value of $n!$, and the subproblem is determining the value of $(n - 1)!$. In the recursive case, when n is greater than 1, the function calls itself to determine the value of $(n - 1)!$ and multiplies that with n.

In the base case, when n is 0 or 1, the function simply returns 1. This looks like the following:

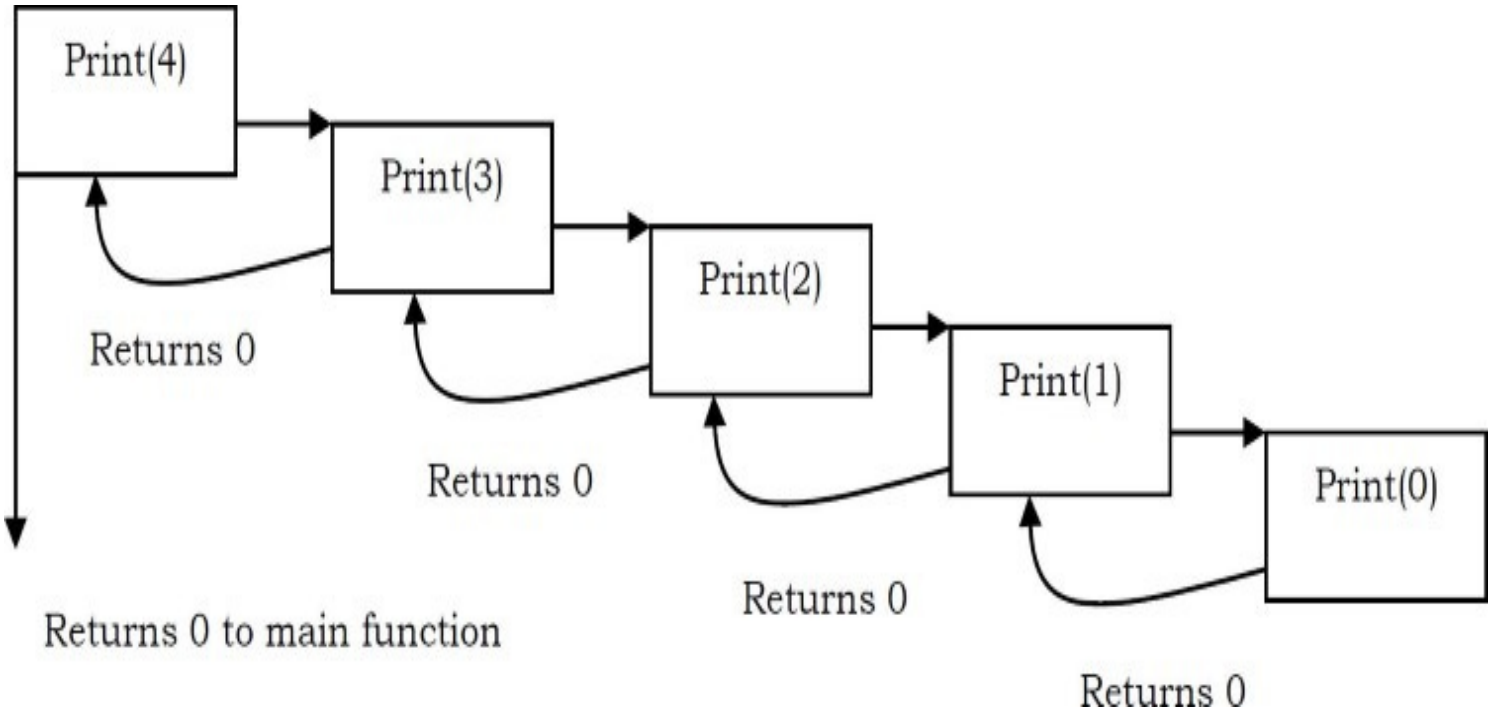
```
// calculates factorial of a positive integer
int Fact(int n) {
   if(n == 1)   // base cases: fact of 0 or 1 is 1
      return 1;
   else if(n == 0)
      return 1;
   else    // recursive case: multiply n by (n – 1) factorial
      return n*Fact(n-1);
}
```

2.5 Recursion and Memory (Visualization)

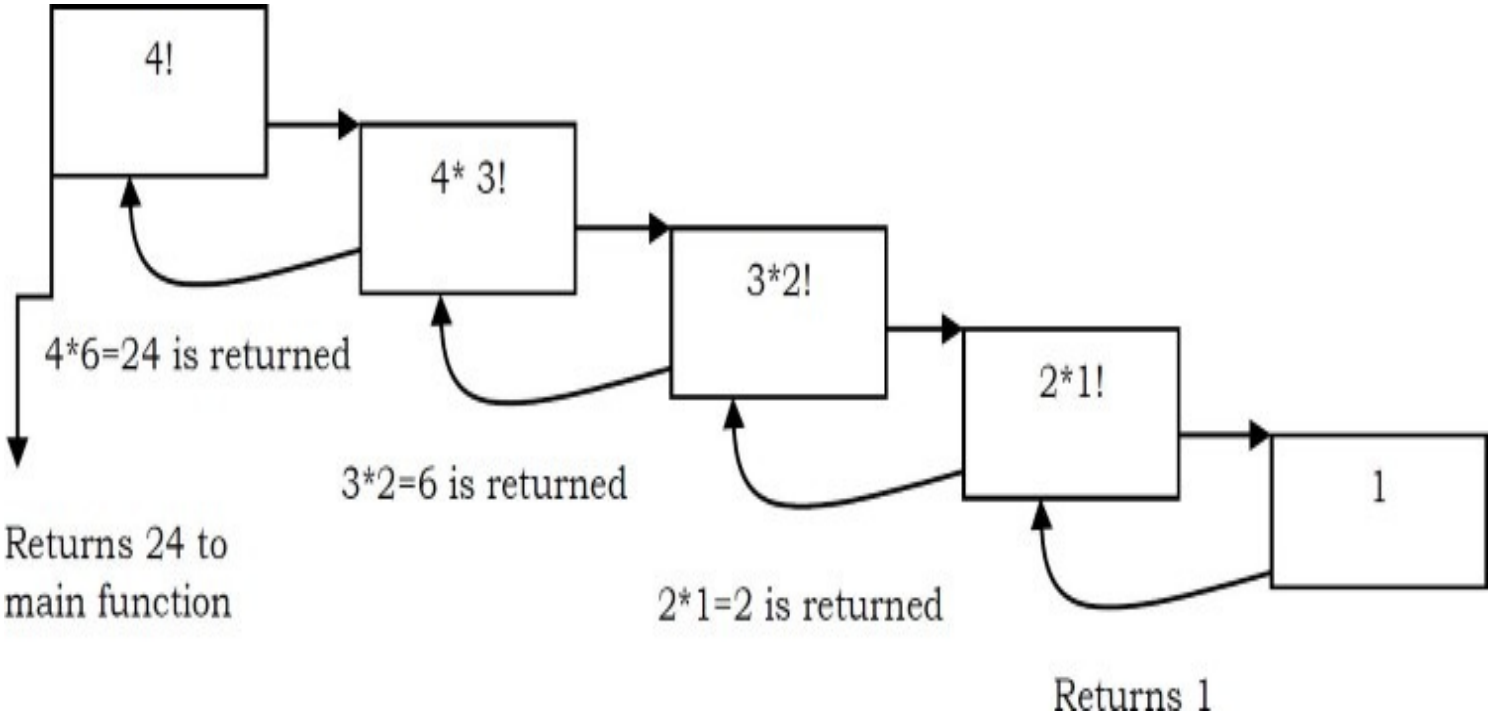
Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes time. For better understanding, let us consider the following example.

```
//print numbers 1 to n backward
int Print(int n) {
 if( n = = 0) // this is the terminating base case
    return 0;
 else {
    printf ("%d",n);
    return Print(n-1);       // recursive call to itself again
 }
}
```

For this example, if we call the print function with n=4, visually our memory assignments may look like:



Now, let us consider our factorial function. The visualization of factorial function with n=4 will look like:



2.6 Recursion versus Iteration

While discussing recursion, the basic question that comes to mind is: which way is better? – iteration or recursion? The answer to this question depends on what we are trying to do. A recursive approach mirrors the problem that we are trying to solve. A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers. But, recursion adds

overhead for each recursive call (*n*eeds space on the stack frame).

Recursion

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stack overflow.
- Solutions to some problems are easier to formulate recursively.

Iteration

- Terminates when a condition is proven to be false.
- Each iteration does not require extra space.
- An infinite loop could loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.

2.7 Notes on Recursion

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

2.8 Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamic Programming Examples
- Divide and Conquer Algorithms
- Towers of Hanoi

- Backtracking Algorithms [we will discuss in next section]

2.9 Recursion: Problems & Solutions

In this chapter we cover a few problems with recursion and we will discuss the rest in other chapters. By the time you complete reading the entire book, you will encounter many recursion problems.

Problem-1 Discuss Towers of Hanoi puzzle.

Solution: The Towers of Hanoi is a mathematical puzzle. It consists of three rods (or pegs or towers), and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks on one rod in ascending order of size, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, satisfying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Algorithm:

- Move the top $n - 1$ disks from *Source* to *Auxiliary* tower,
- Move the n^{th} disk from *Source* to *Destination* tower,
- Move the $n - 1$ disks from *Auxiliary* tower to *Destination* tower.
- Transferring the top $n - 1$ disks from *Source* to *Auxiliary* tower can again be thought of as a fresh problem and can be solved in the same manner. Once we solve *Towers of Hanoi* with three disks, we can solve it with any number of disks with the above algorithm.

```
void TowersOfHanoi(int n, char frompeg, char topeg, char auxpeg) {
    /* If only 1 disk, make the move and return */
    if(n==1) {
        printf("Move disk 1 from peg %c to peg %c",frompeg, topeg);
        return;
    }
    /* Move top n-1 disks from A to B, using C as auxiliary */
    TowersOfHanoi(n-1, frompeg, auxpeg, topeg);

    /* Move remaining disks from A to C */
    printf("\nMove disk %d from peg %c to peg %c", n, frompeg, topeg);

    /* Move n-1 disks from B to C using A as auxiliary */
    TowersOfHanoi(n-1, auxpeg, topeg, frompeg);
}
```

Problem-2 Given an array, check whether the array is in sorted order with recursion.

Solution:

```
int isArrayInSortedOrder(int A[],int n){
    if(n == 1)
      return 1;
    return (A[n-1] < A[n-2])?0:isArrayInSortedOrder(A,n-1);
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*) for recursive stack space.

2.10 What is Backtracking?

Backtracking is an improvement of the brute force approach. It systematically searches for a solution to a problem among all available options. In backtracking, we start with one possible option out of many available options and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other option and try to solve it. If none if the options work out we will claim that there is no solution for the problem.

Backtracking is a form of recursion. The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of

options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't.

Backtracking can be thought of as a selective tree/graph traversal method. The tree is a way of representing some initial starting position (the root node) and a final goal state (one of the leaves). Backtracking allows us to deal with situations in which a raw brute-force approach would explode into an impossible number of options to consider. Backtracking is a sort of refined brute force. At each node, we eliminate choices that are obviously not possible and proceed to recursively check only those that have potential.

What's interesting about backtracking is that we back up only as far as needed to reach a previous decision point with an as-yet-unexplored alternative. In general, that will be at the most recent decision point. Eventually, more and more of these decision points will have been fully explored, and we will have to backtrack further and further. If we backtrack all the way to our initial state and have explored all alternatives from there, we can conclude the particular problem is unsolvable. In such a case, we will have done all the work of the exhaustive recursion and known that there is no viable solution possible.

- Sometimes the best algorithm for a problem is to try all possibilities.
- This is always slow, but there are standard tools that can be used to help.
- Tools: algorithms for generating basic objects, such as binary strings [2^n possibilities for n-bit string], permutations [$n!$], combinations [$n!/r!(n - r)!$], general strings [k –ary strings of length n has k^n possibilities], etc...
- Backtracking speeds the exhaustive search by pruning.

2.11 Example Algorithms of Backtracking

- Binary Strings: generating all binary strings
- Generating k – ary Strings
- N-Queens Problem
- The Knapsack Problem
- Generalized Strings
- Hamiltonian Cycles [refer to *Graphs* chapter]
- Graph Coloring Problem

2.12 Backtracking: Problems & Solutions

Problem-3 Generate all the strings of n bits. Assume $A[0..n - 1]$ is an array of size n.

Solution:

```
void Binary(int n) {
    if(n < 1)
        printf("%s", A);              //Assume array A is a global variable
    else {
        A[n-1] = 0;
        Binary(n - 1);
        A[n-1] = 1;
        Binary(n - 1);
    }
}
```

Let $T(n)$ be the running time *of binary*(n). Assume function *printf* takes time O(1).

$$T(n) = \begin{cases} c, & \text{if } n < 0 \\ 2T(n-1) + d, & \text{otherwise} \end{cases}$$

Using Subtraction and Conquer Master theorem we get: $T(n) = O(2^n)$. This means the algorithm for generating bit-strings is optimal.

Problem-4 Generate all the strings of length n drawn from 0... $k - 1$.

Solution: Let us assume we keep current k-ary string in an array $A[0..\ n - 1]$. Call function *k-string*(n, k):

```
void k-string(int n, int k) {
    //process all k-ary strings of length m
    if(n < 1)
        printf("%s",A);               //Assume array A is a global variable
    else {
        for (int j = 0 ; j < k ; j++) {
            A[n-1] = j;
            k-string(n- 1, k);
        }
    }
}
```

Let $T(n)$ be the running time of $k - string(n)$. Then,

$$T(n) = \begin{cases} c, & if\ n < 0 \\ kT(n-1) + d, & otherwise \end{cases}$$

Using Subtraction and Conquer Master theorem we get: $T(n) = O(k^n)$.

Note: For more problems, refer to *String Algorithms* chapter.

Problem-5 Finding the length of connected cells of 1s (regions) in an matrix of Os and 1s: Given a matrix, each of which may be 1 or 0. The filled cells that are connected form a region. Two cells are said to be connected if they are adjacent to each other horizontally, vertically or diagonally. There may be several regions in the matrix. How do you find the largest region (in terms of number of cells) in the matrix?

Sample Input: 11000 Sample Output: 5
01100
00101
10001
01011

Solution: The simplest idea is: for each location traverse in all 8 directions and in each of those directions keep track of maximum region found.

```
int getval(int (*A)[5],int i,int j,int L, int H){
    if (i< 0 || i >= L || j< 0 || j >= H)
        return 0;
    else
        return A[i][j];
}
void findMaxBlock(int (*A)[5], int r, int c,int L,int H,int size, bool **cntarr,int &maxsize){
    if ( r >= L || c >= H)
        return;
    cntarr[r][c]=true;
    size++;
    if  (size > maxsize)
        maxsize = size;
    //search in eight directions
    int direction[][2]={{-1,0},{-1,-1},{0,-1},{1,-1},{1,0},{1,1},{0,1},{-1,1}};
    for(int i=0; i<8; i++)        {
        int newi =r+direction[i][0];
        int newj=c+direction[i][1];
        int val=getval (A,newi,newj,L,H);
        if (val>0  && (cntarr[newi][newj]==false)){
            findMaxBlock(A,newi,newj,L,H,size,cntarr,maxsize);
        }
    }
    cntarr[r][c]=false;
}
int getMaxOnes(int (*A)[5], int rmax, int colmax){
    int maxsize=0;
    int size=0;
    bool **cntarr=create2darr(rmax,colmax);
    for(int i=0; i< rmax; i++){
        for(int j=0; j< colmax; j++){
            if (A[i][j] == 1){
                findMaxBlock(A,i,j,rmax,colmax, 0,cntarr,maxsize);
            }
        }
    }
    return maxsize;
}
```

Sample Call:

```
int zarr[][5]={{1,1,0,0,0},{0,1,1,0,1},{0,0,0,1,1},{1,0,0,1,1},{0,1,0,1,1}};
cout << "Number of maximum 1s are " << getMaxOnes(zarr,5,5) << endl;
```

Problem-6 Solve the recurrence $T(n) = 2T(n - 1) + 2^n$.

Solution: At each level of the recurrence tree, the number of problems is double from the previous level, while the amount of work being done in each problem is half from the previous level. Formally, the i^{th} level has 2^i problems, each requiring 2^{n-i} work. Thus the i^{th} level requires exactly 2^n work. The depth of this tree is n, because at the i^{th} level, the originating call will be $T(n - i)$. Thus the total complexity for $T(n)$ is $T(n2^n)$.

CHAPTER

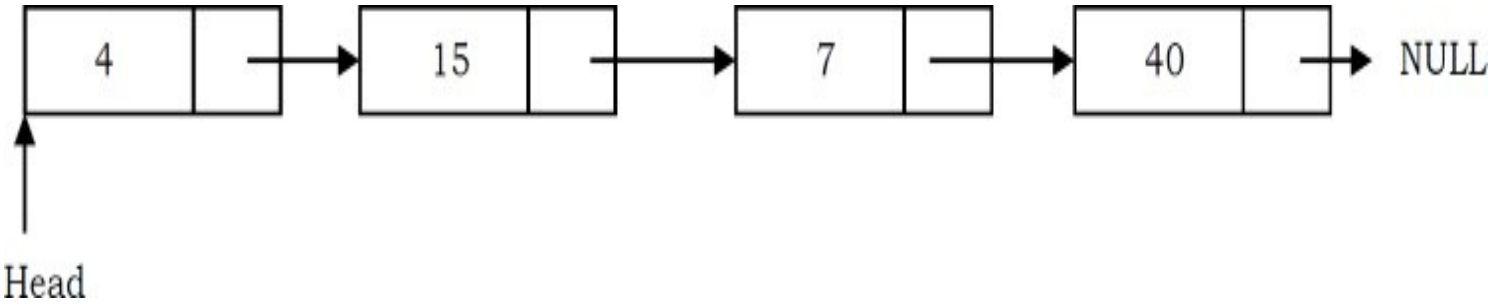
3

LINKED LISTS

3.1 What is a Linked List?

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

- Successive elements are connected by pointers
- The 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)
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as list grows.



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

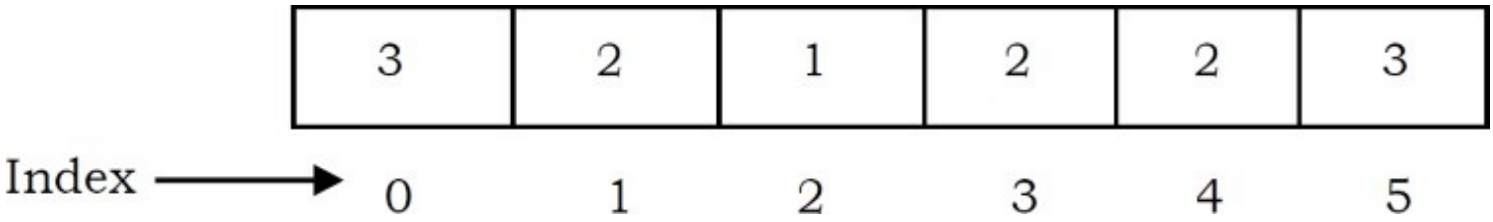
- Delete List: removes all elements of the list (disposes the list)
- Count: returns the number of elements in the list
- Find n^{th} node from the end of the list

3.3 Why Linked Lists?

There are many other data structures that do the same thing as 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, and since both are used for the same purpose, we need to differentiate their usage. 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 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, the 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

- Preallocates all needed memory up front and wastes memory space for indices in the array that are empty.
- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation:** To allocate the array itself at the beginning, 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, 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 to initially start with some fixed size array. As soon as that array becomes full, create the new array double the size of 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 the *Stacks, Queues* and *Hashing* chapters.

Advantages of Linked Lists

Linked lists have both 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 when full, we must create a new array and copy the old array into the new array. This can take a lot of time.

We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory. With a linked list, we can start with space for just one allocated element 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 with 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 take O(*n*) for access to an element in the list in the worst case. Another advantage of arrays in access time is *spacial 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 then 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.

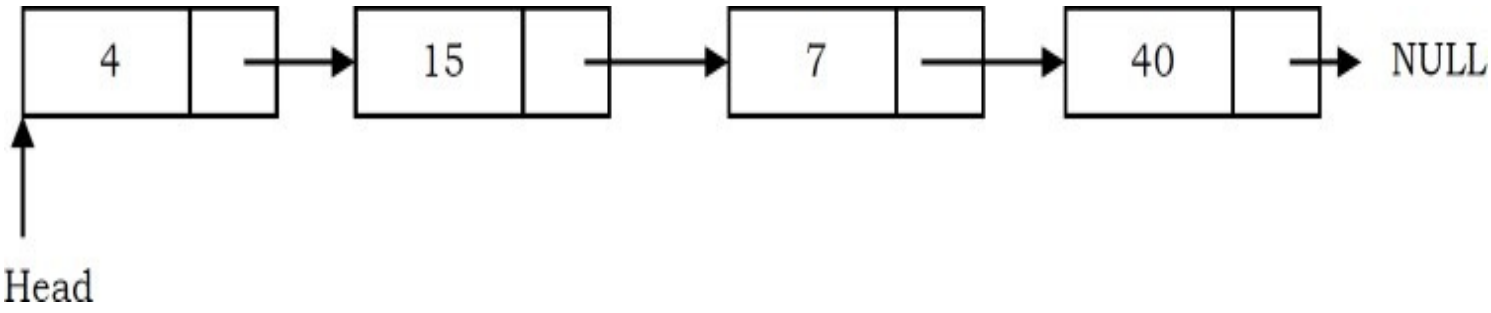
Finally, linked lists waste memory in terms of extra reference points.

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*), if array is not full (for shifting the elements)	O(*n*)
Insertion at ending	O(*n*)	O(1), if array is not full	O(1), if array is not full O(*n*), if array is full
Deletion at ending	O(*n*)	O(1)	O(*n*)
Insertion in middle	O(*n*)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Deletion in middle	O(*n*)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Wasted space	O(*n*) (for pointers)	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 the end of the list.



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

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

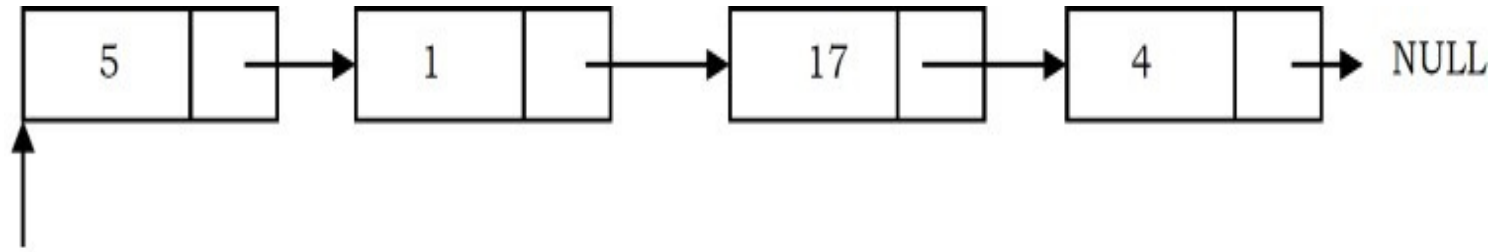
Basic Operations on a List

- 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.



Head

The ListLength() function takes a linked list as input and counts the number of nodes in the list. The function given below 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 list of size *n*.
Space Complexity: O(1), for creating a temporary variable.

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

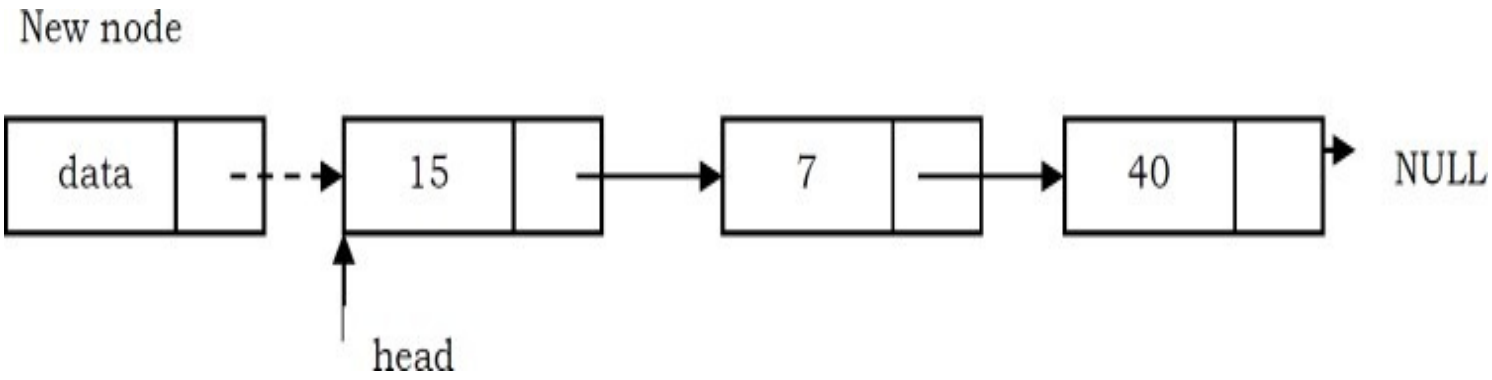
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

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*.

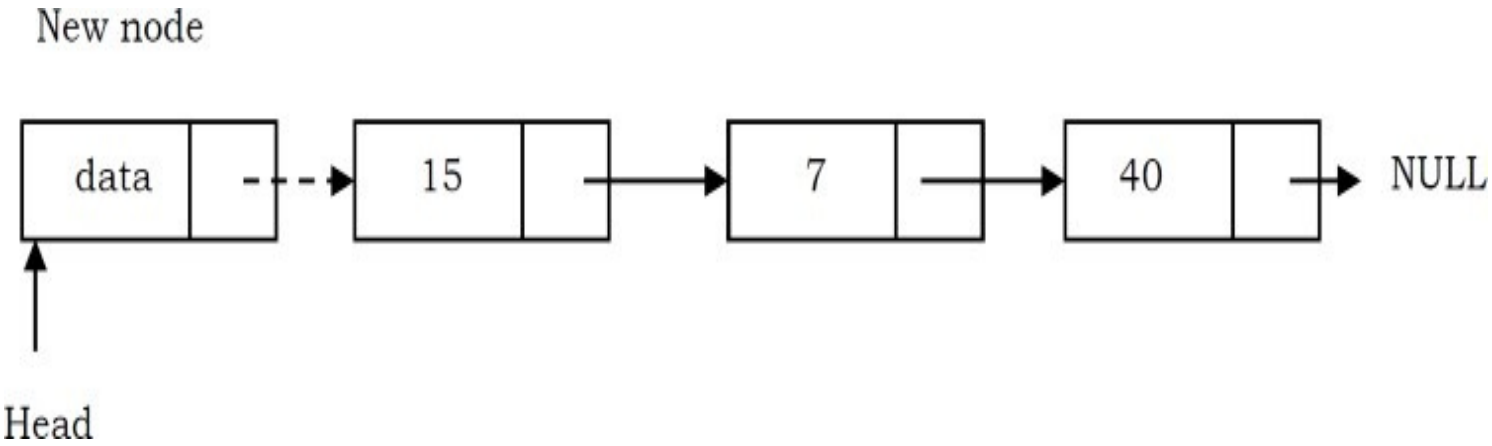
Inserting a Node in Singly Linked List at the Beginning

In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (*n*ew 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.



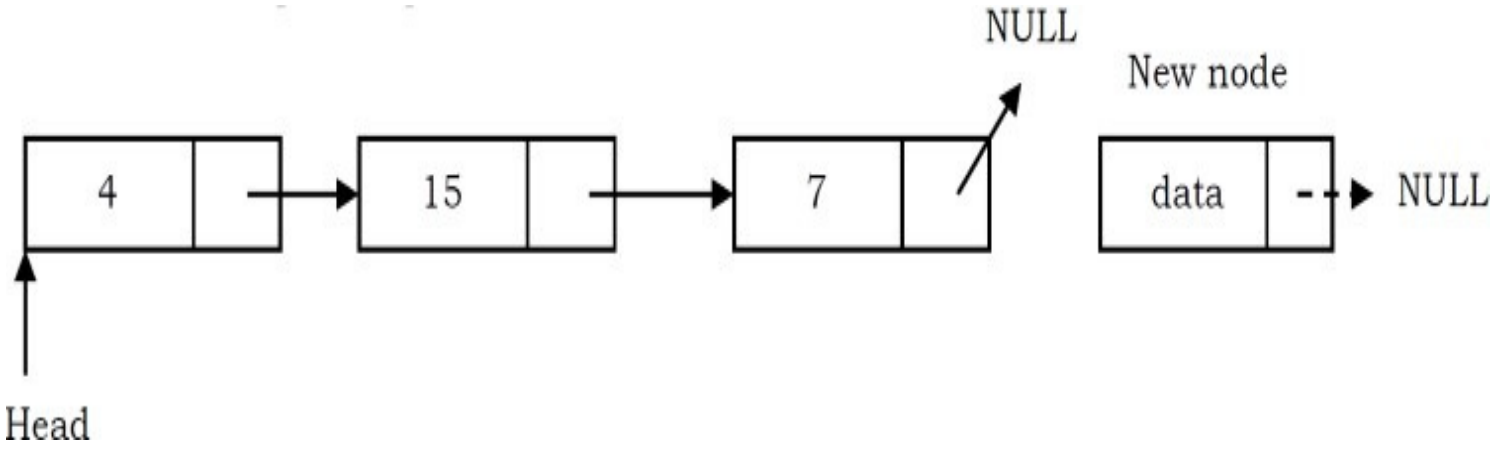
- Update head pointer to point to the new node.



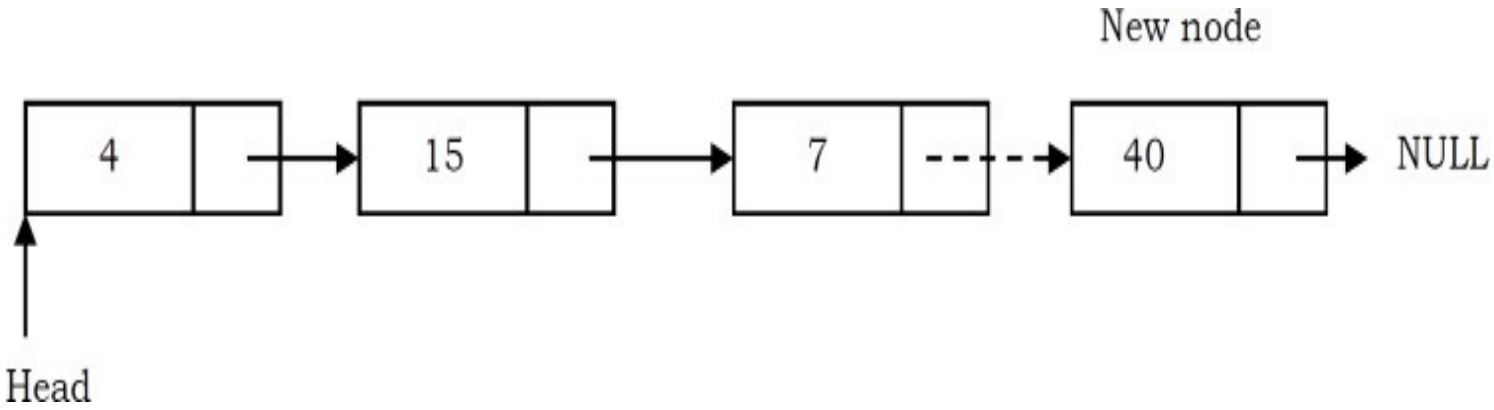
Inserting a Node in Singly Linked List at the Ending

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.



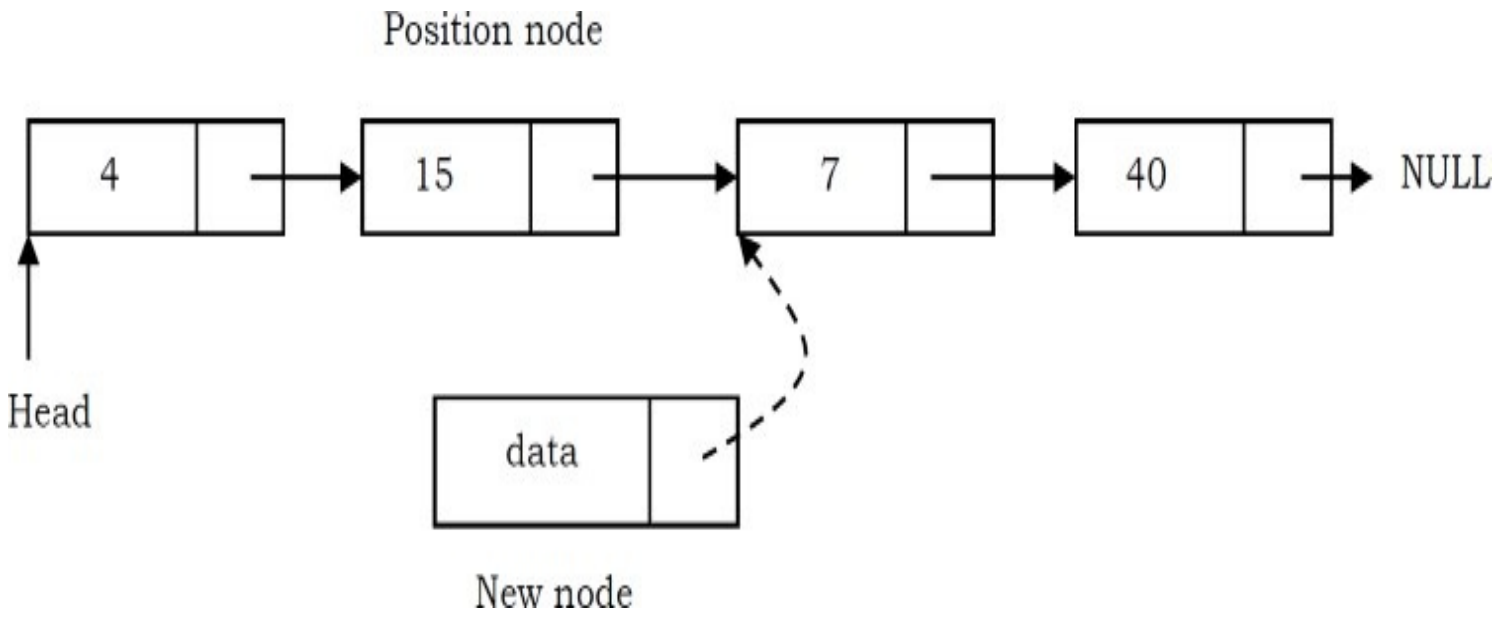
- Last nodes next pointer points to the new node.



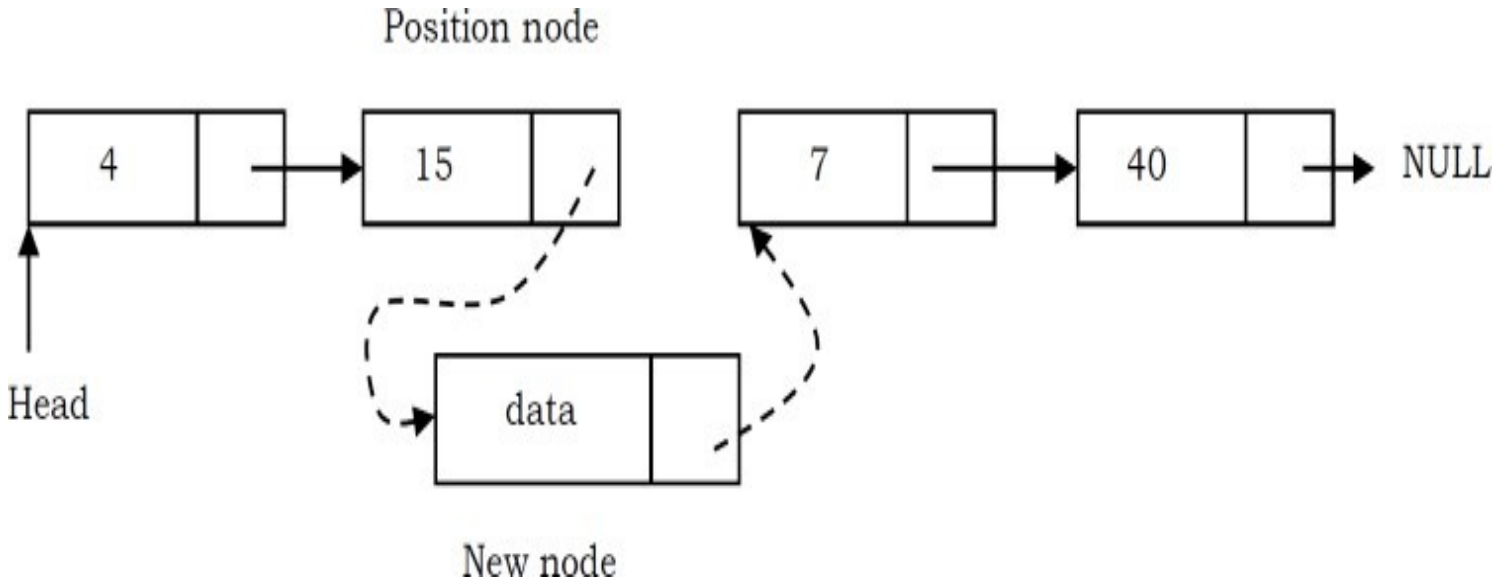
Inserting a Node in Singly Linked List at the Middle

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 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called *position* node. The new node points to the next node of the position where we want to add this node.



- Position node's next pointer now points to the new node.



Let us write the code for all 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 a 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){
        printf("Memory Error");
        return;
    }
    newNode→data=data;
    p=*head;
    //Inserting at the beginning
    if(position == 1){
        newNode→next=p;
        *head=newNode;
    }
    else{
        //Traverse the list until the position where we want to insert
        while((p!=NULL) && (k<position)){
            k++;
            q=p;
            p=p→next;
        }
        q→next=newNode;   //more optimum way to do this
        newNode→next=p;
    }
}
```

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

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

Singly Linked List Deletion

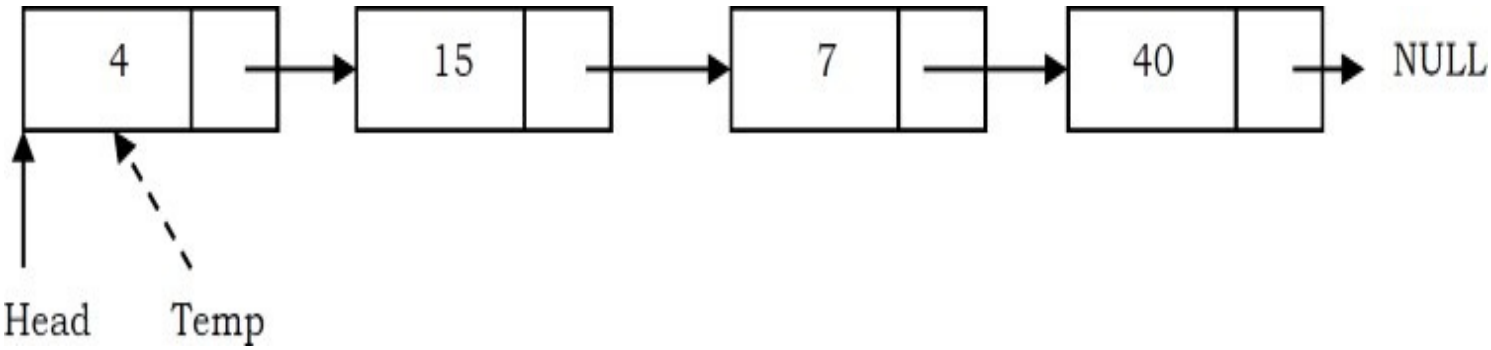
Similar to insertion, here we also have three cases.

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

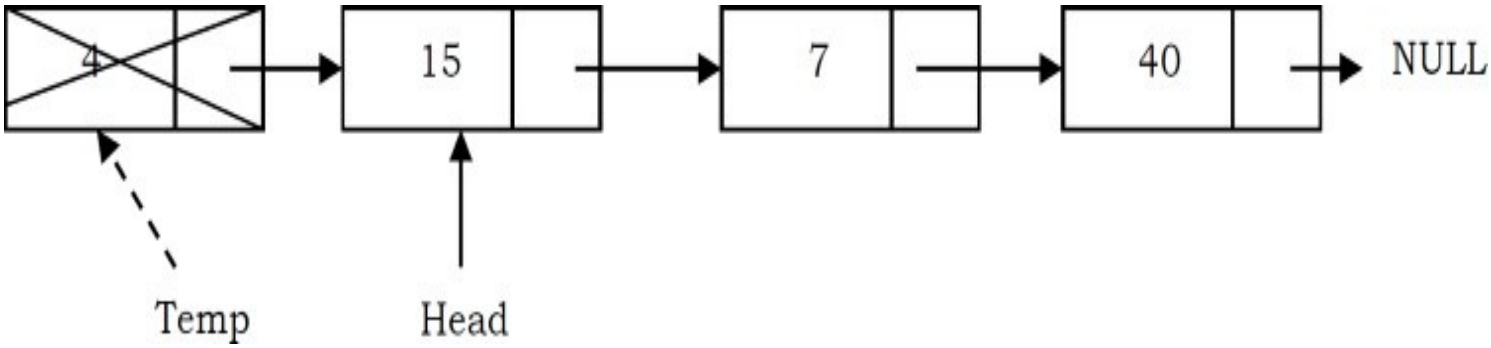
Deleting the First Node in Singly Linked List

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 the same node as that of head.



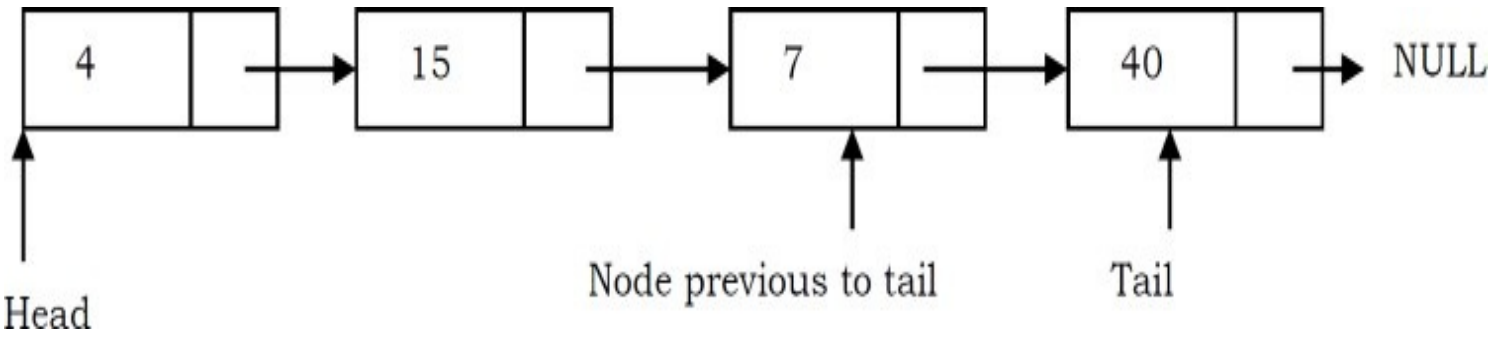
- Now, move the head nodes pointer to the next node and dispose of the temporary node.



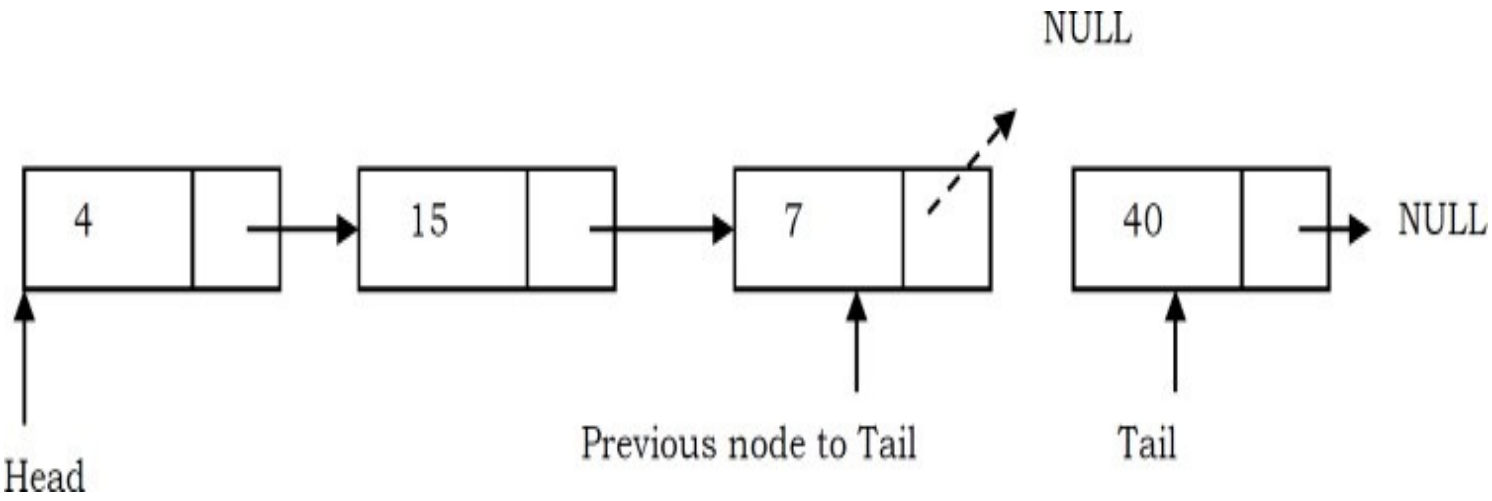
Deleting the Last Node in Singly Linked List

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

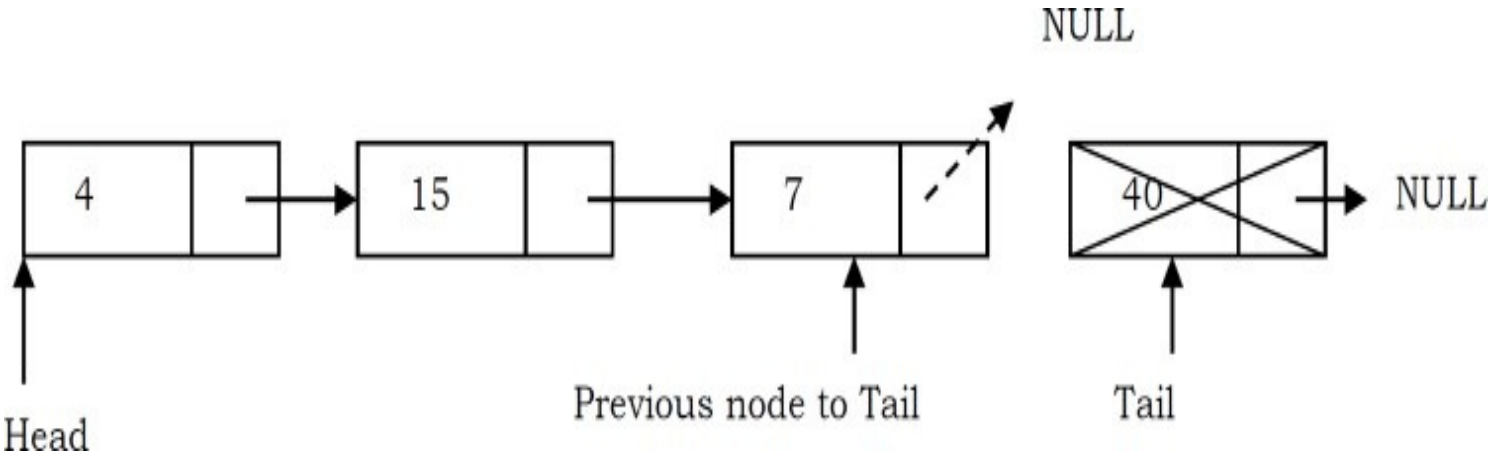
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.



- Update previous node's next pointer with NULL.



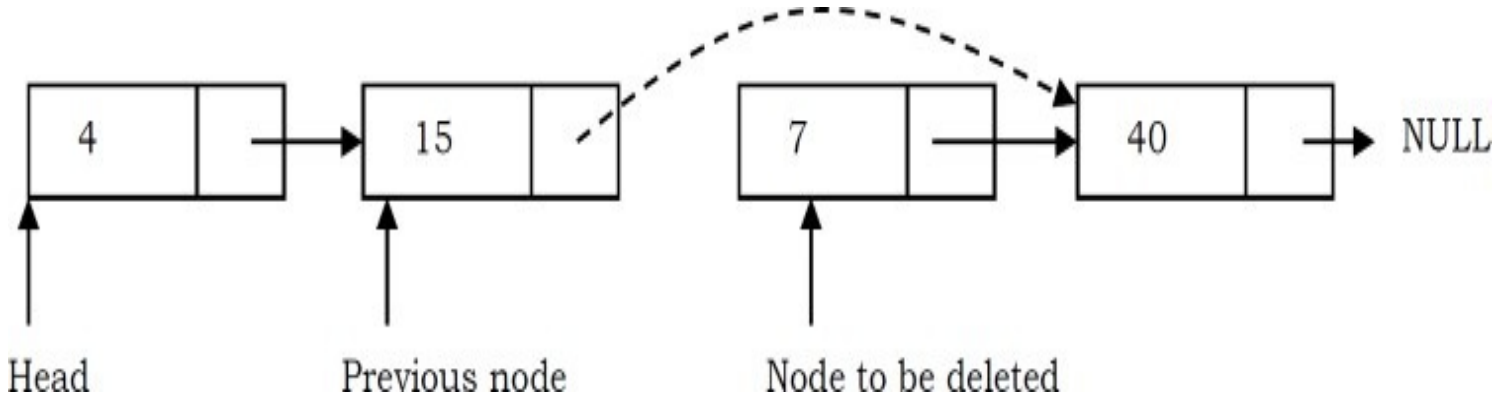
- Dispose of the tail node.



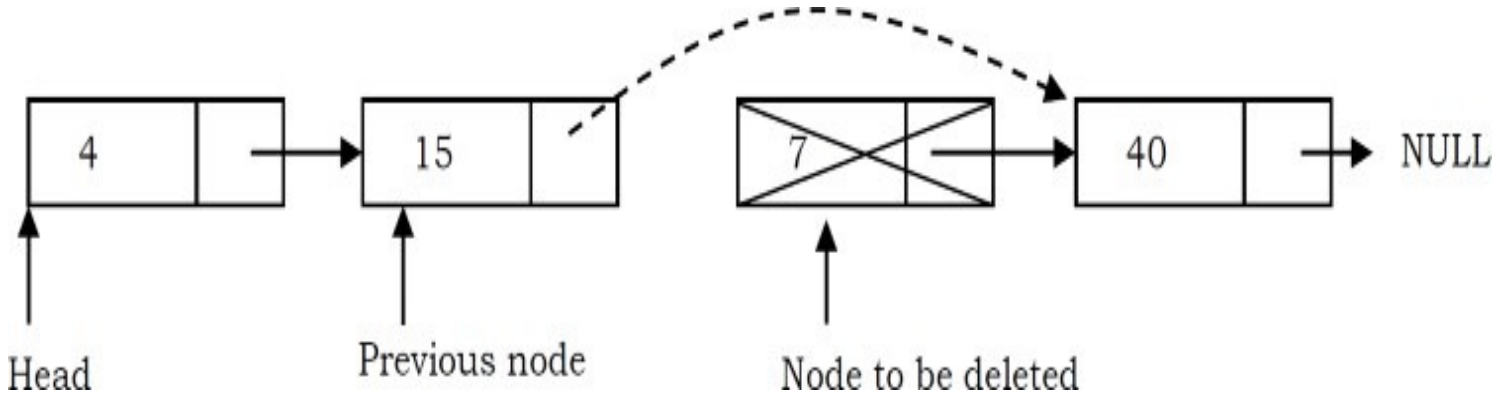
Deleting an Intermediate Node in Singly Linked List

In this case, the 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:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



- Dispose of 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;
    /* from the beginning */
    if(position == 1) {
        *head = (*head)→next;
        free (p);
        return;
    }
    else {
        //Traverse the list until arriving at the position from which we want to delete
        while ((p != NULL) && (k < position )) {
            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 case, we may need to delete the node at the end of the list.
Space Complexity: O(1), for 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 the next node with a 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 that 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 a doubly linked list, we can delete a node even if we don't have the previous node's address (since each node has a left pointer pointing to the 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).

Similar to a singly linked list, let us implement the operations of a doubly linked list. If you understand the singly linked list operations, then doubly linked list operations are 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

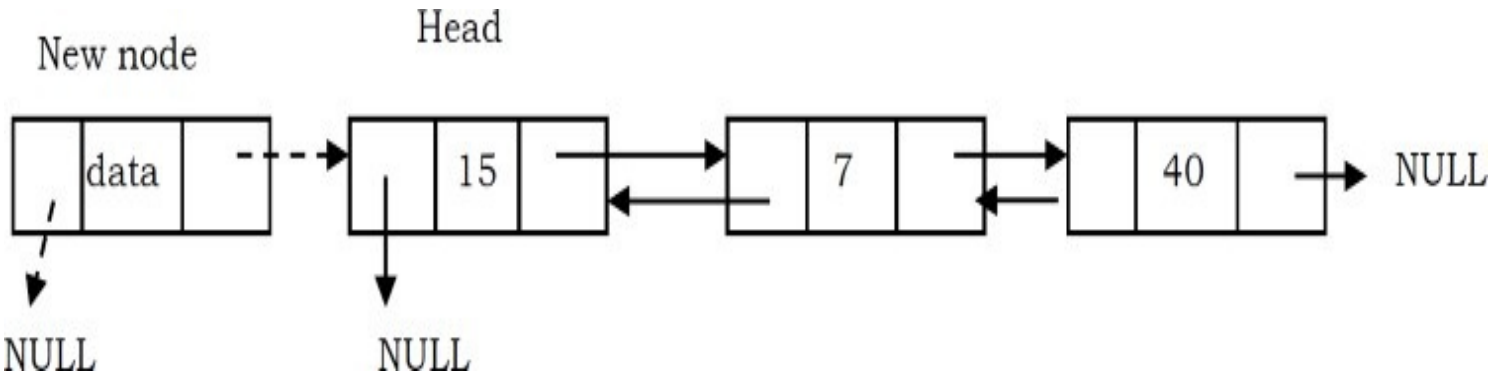
Insertion into a doubly-linked list has three cases (same as singly linked list):

- 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.

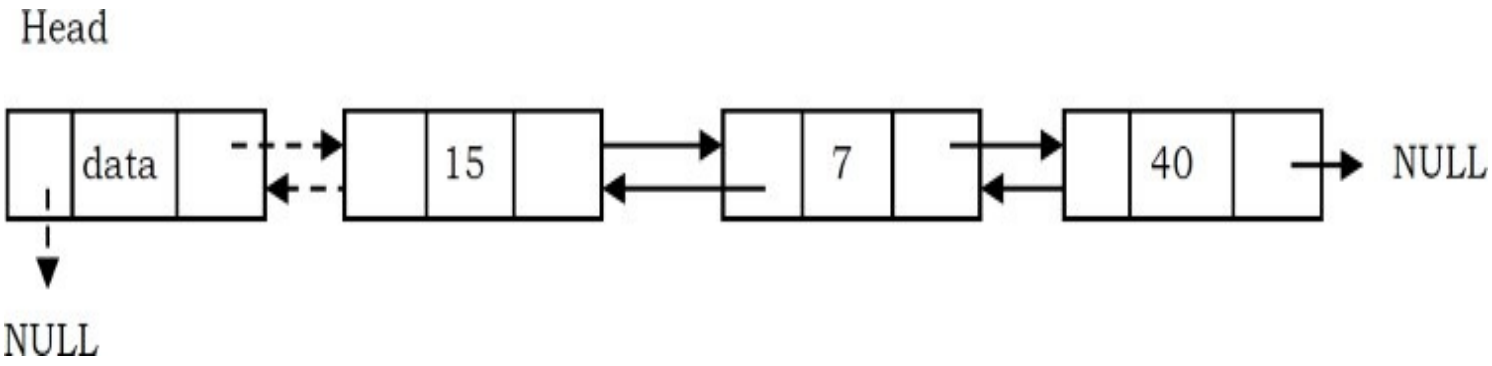
Inserting a Node in Doubly Linked List at the Beginning

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



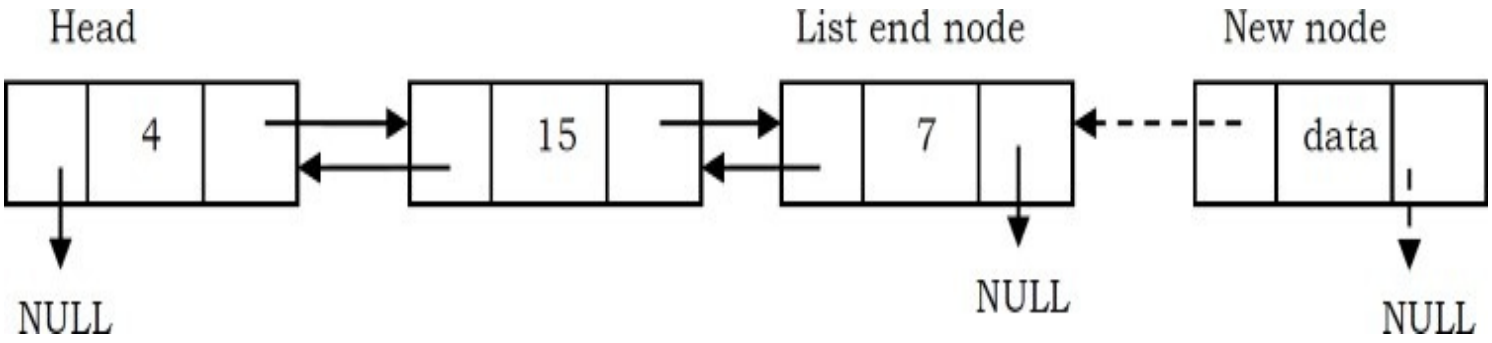
- Update head node's left pointer to point to the new node and make new node as head. Head



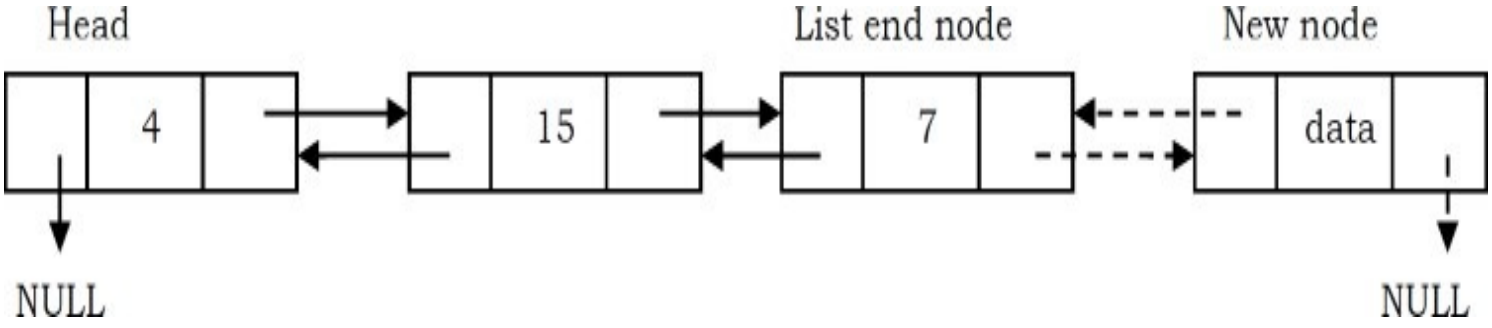
Inserting a Node in Doubly Linked List at the Ending

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.



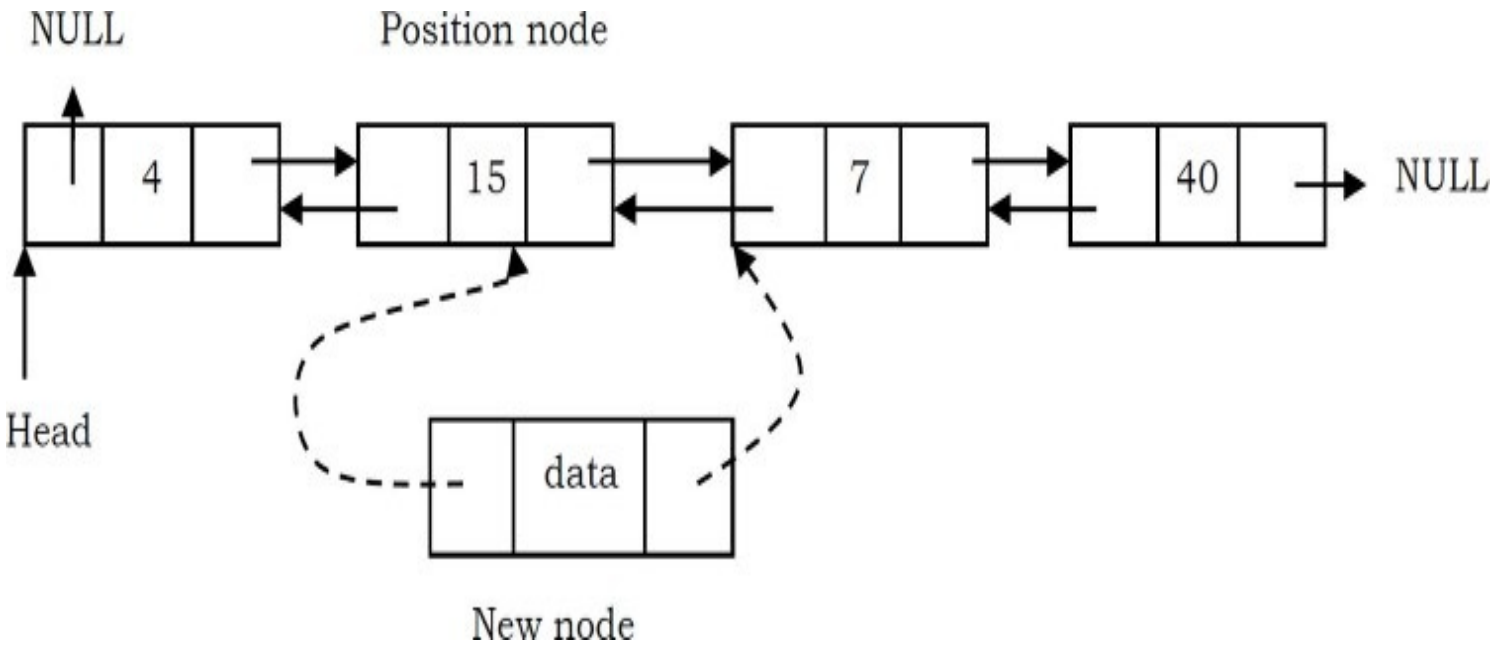
- Update right pointer of last node to point to new node.



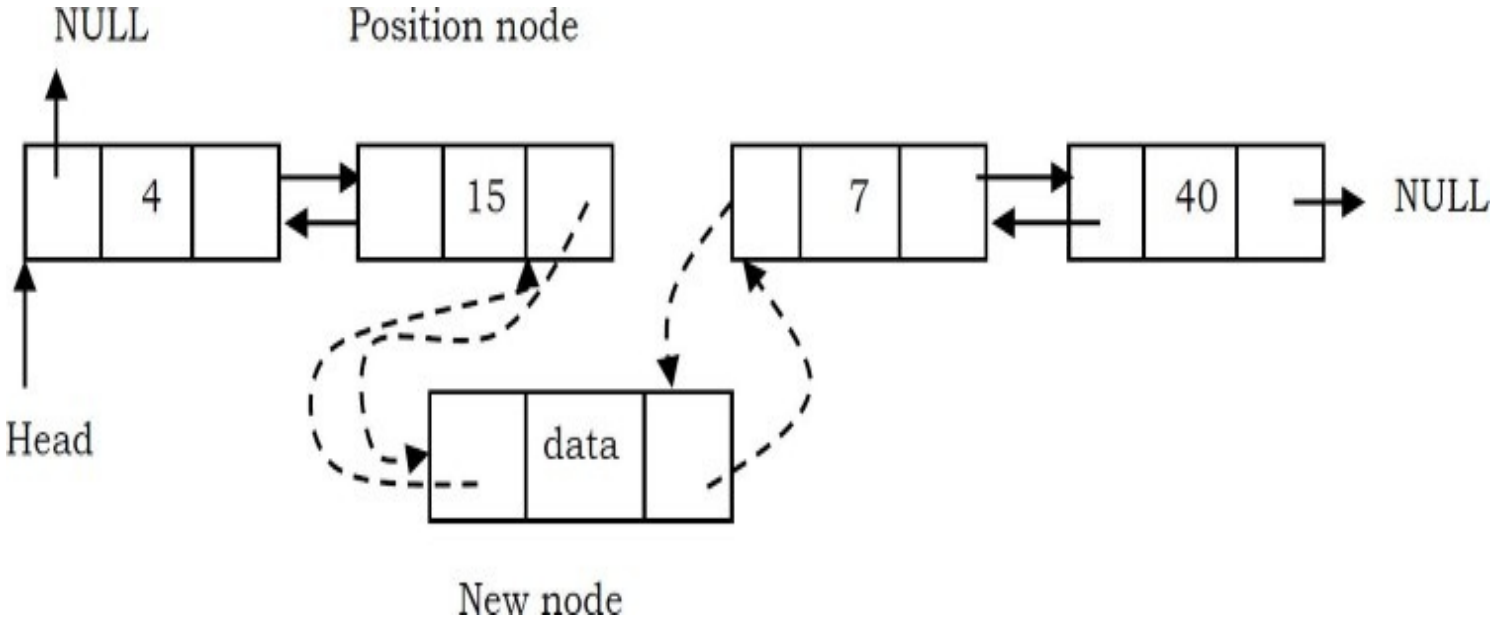
Inserting a Node in Doubly Linked List at the Middle

As discussed in singly linked lists, traverse the list to 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 node left pointer points to new node.



Now, let us write the code for all of 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 a 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;

        if(*head)
          (*head)→prev = newNode;

        *head = newNode;
        return;
    }
    temp = *head;
    while ( (k < position - 1) && temp→next!=NULL) {
        temp = temp→next;
        k++;
    }
    if(k!=position){
       printf("Desired position does not exist\n");
    }
    newNode→next=temp→next;
    newNode→prev=temp;
    if(temp→next)
       temp→next→prev=newNode;

    temp→next=newNode;
    return;
}
```

Time Complexity: O(*n*). In the worst case, 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

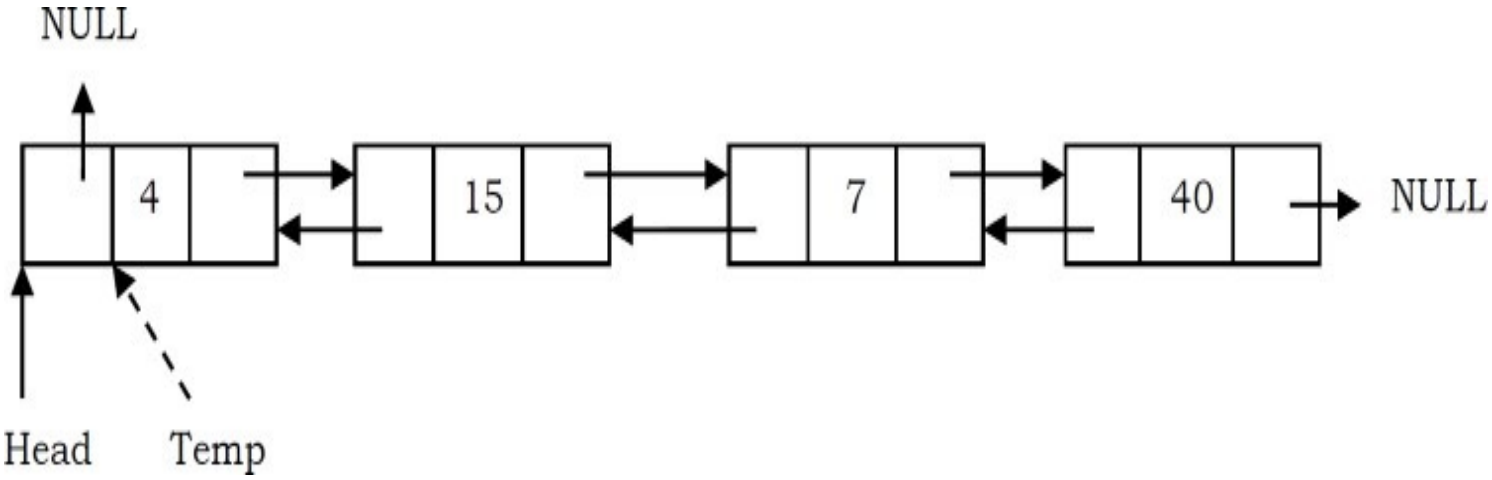
Similar to singly linked list deletion, here we have three cases:

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

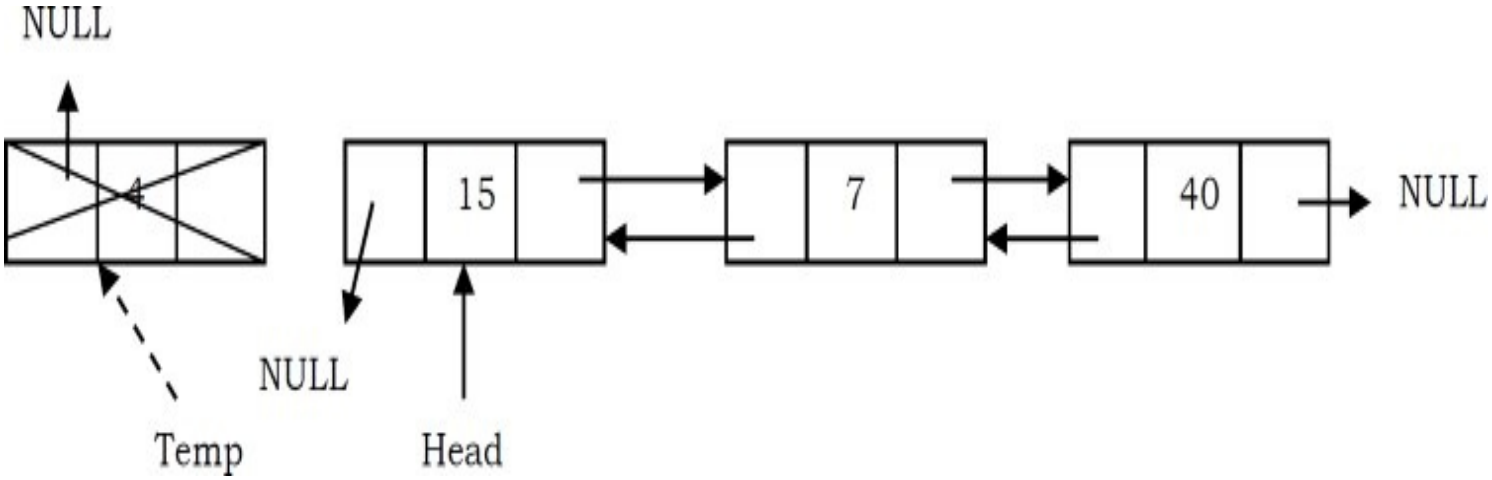
Deleting the First Node in Doubly Linked List

In this case, the 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 the same node as that of head.



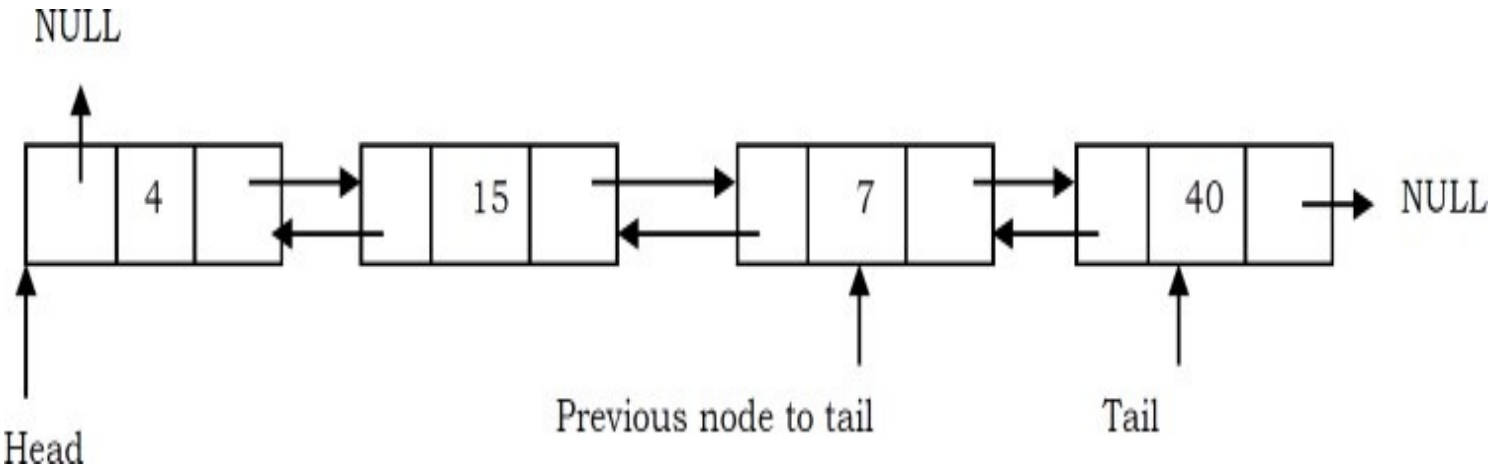
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose of the temporary node.



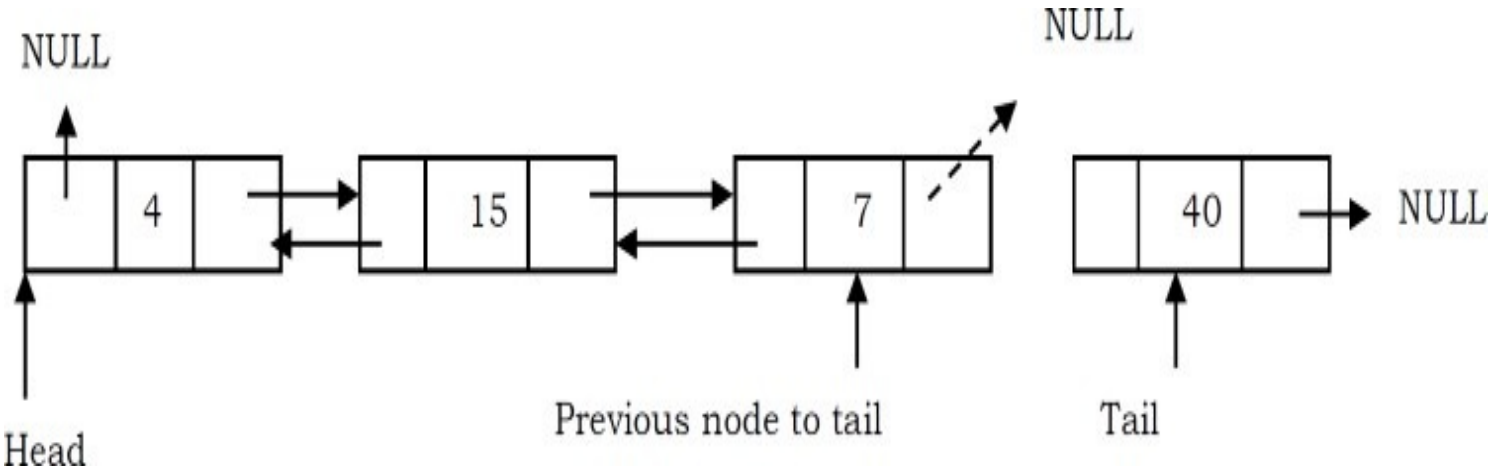
Deleting the Last Node in Doubly Linked List

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

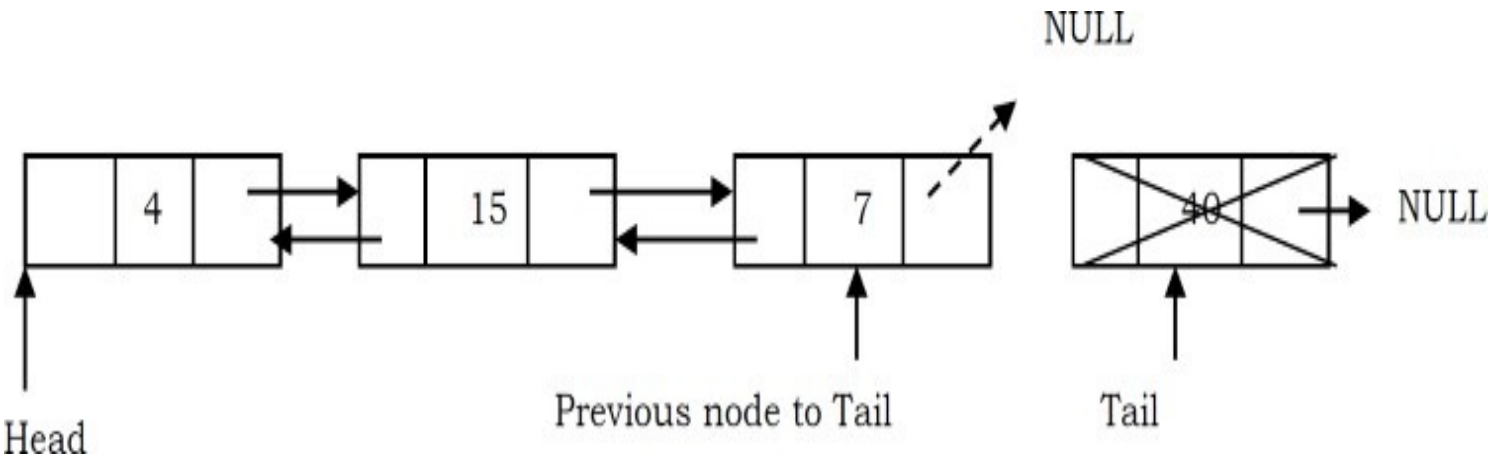
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.



- Update the next pointer of previous node to the tail node with NULL.



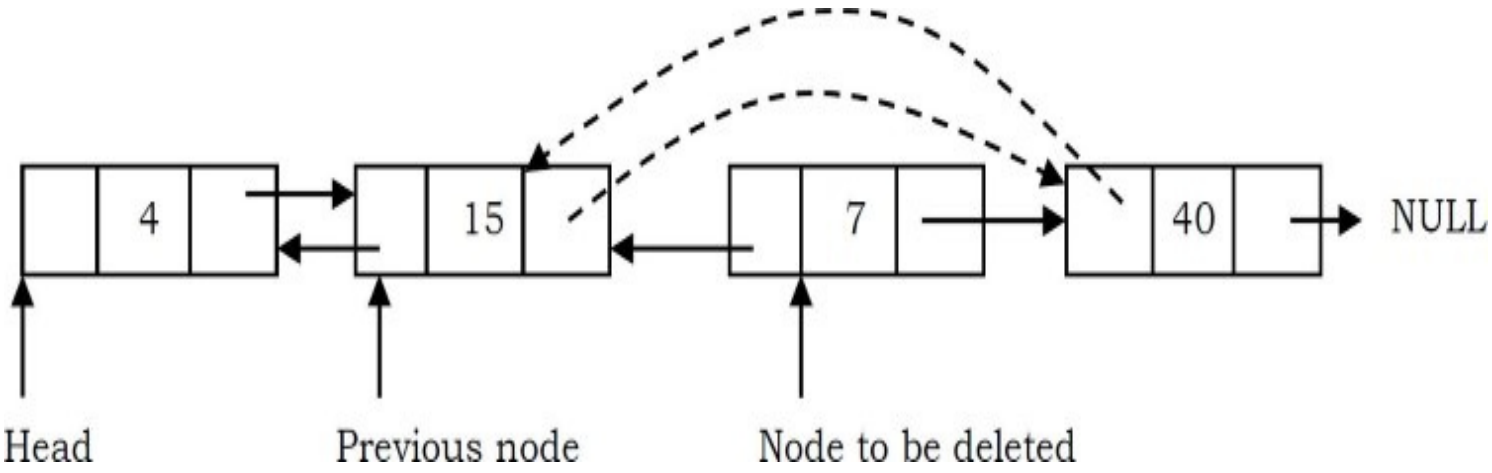
- Dispose the tail node.



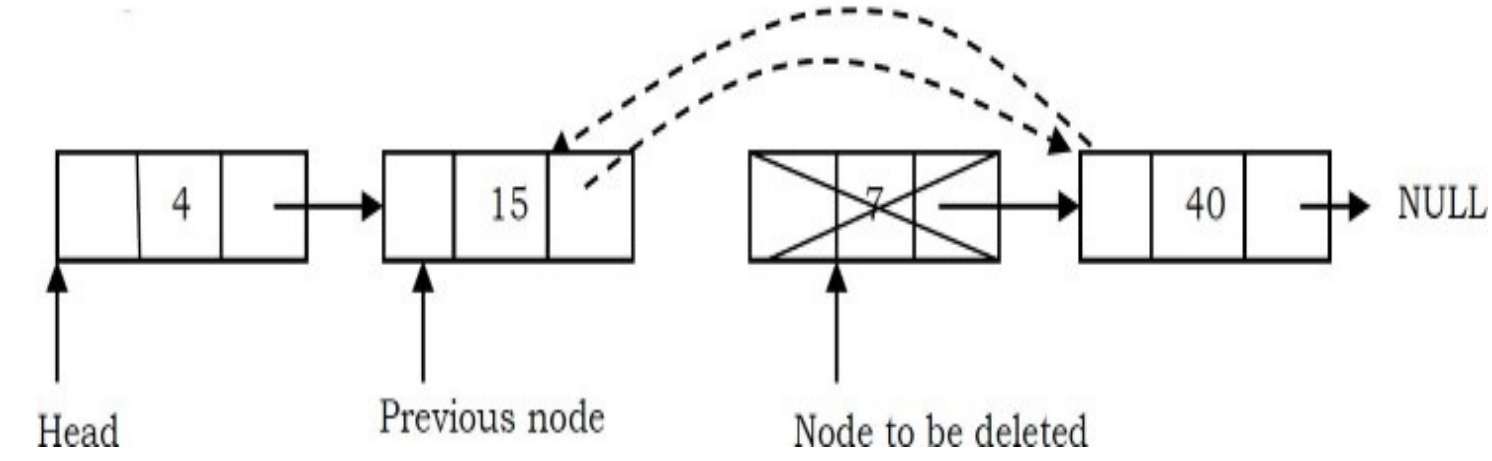
Deleting an Intermediate Node in Doubly Linked List

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

- Similar to the previous case, maintain the previous node while also traversing the list. Upon locating the node to be deleted, change the previous node's next pointer to the next node of the node to be deleted.



- Dispose of 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) && temp→next!=NULL) {
       temp = temp→next;
       k++;
    }
    if(k!=position-1){
       printf("Desired position does not exist\n");
    }
    temp2=temp→prev;
    temp2→next=temp→next;
    if(temp→next) // Deletion from Intermediate Node
       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

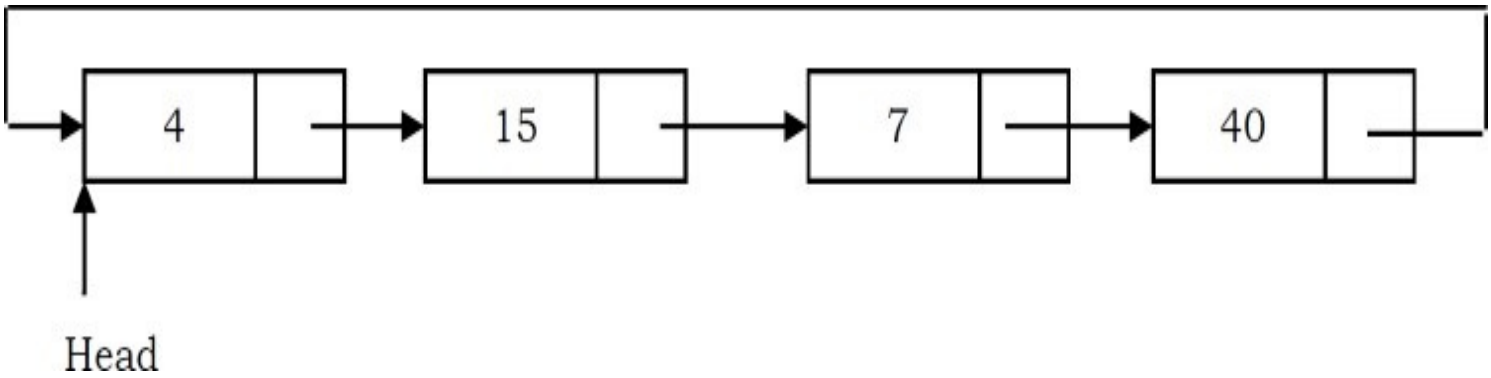
In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value. But circular linked lists do not have ends. While traversing the circular linked lists we should be careful; otherwise we will be traversing the list infinitely. In circular linked lists, each node has a successor. 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, we have to assure that no process accesses the resource before all other processes do (round robin algorithm). The following is a type declaration for a circular linked list of integers:

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

In a 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 Linked List



The circular list is accessible through the node marked *head*. To count the nodes, the list has to be traversed from the node marked *head*, with the help of a dummy node *current*, and stop the counting when *current* reaches the starting node *head*.

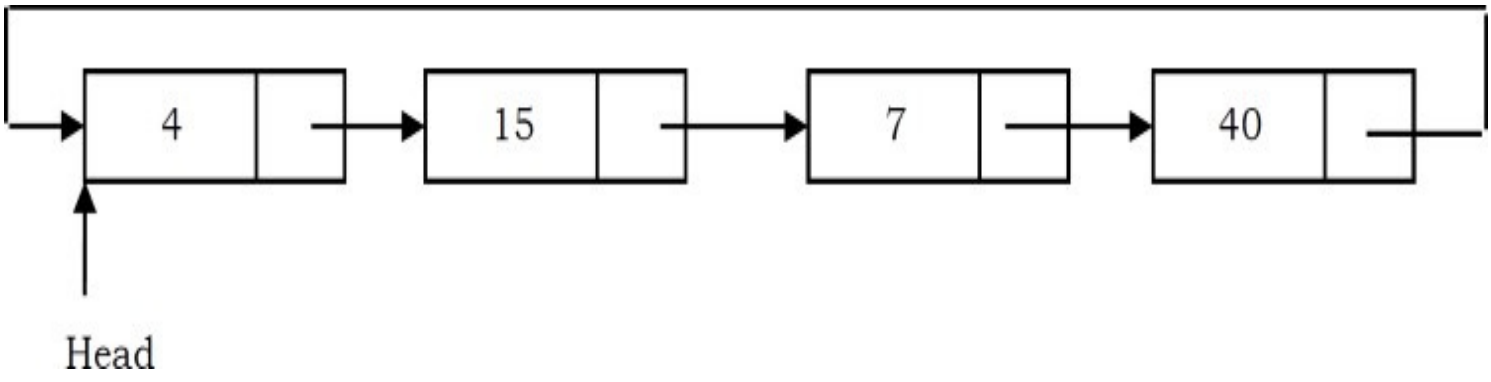
If the list is empty, *head* will be NULL, and in that case set *count* = 0. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

```
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 Linked 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 previous 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.



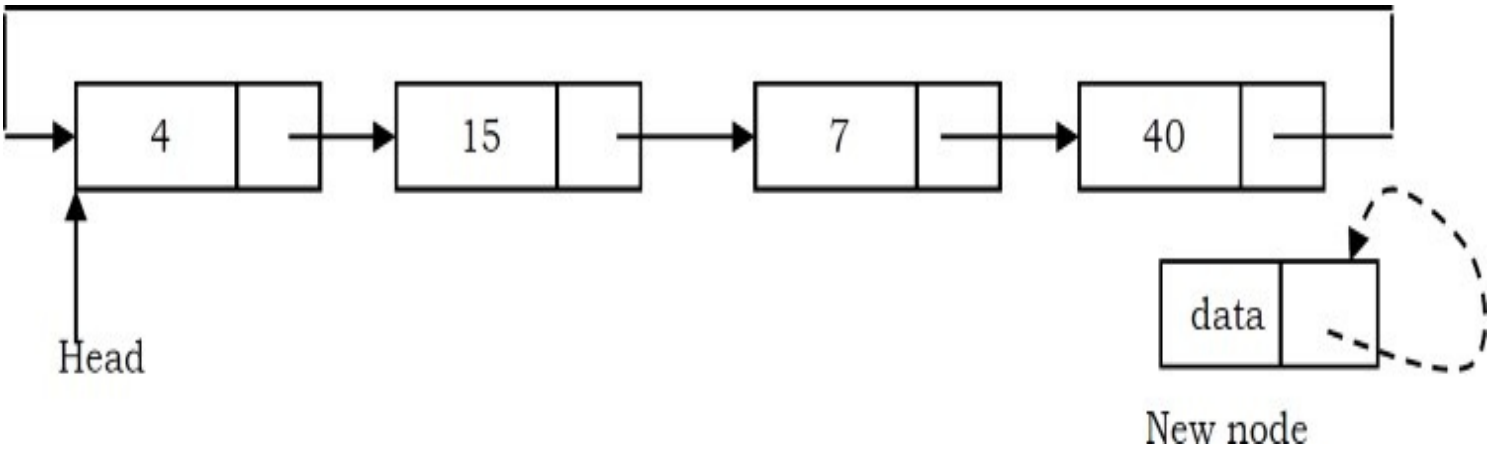
```
void PrintCircularListData(struct CLLNode *head) {
      struct CLLNode *current = head;
      if(head == NULL)
         return;

      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 temporary variable.

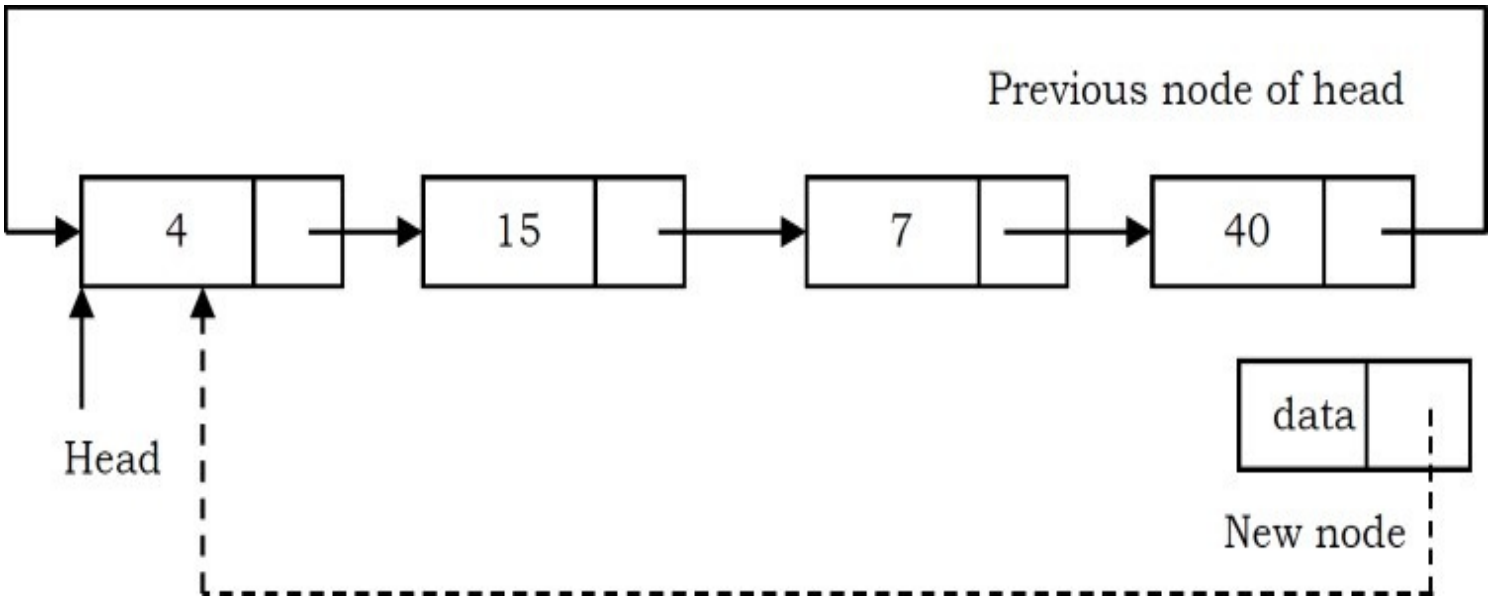
Inserting a Node at the End of a Circular Linked List

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.

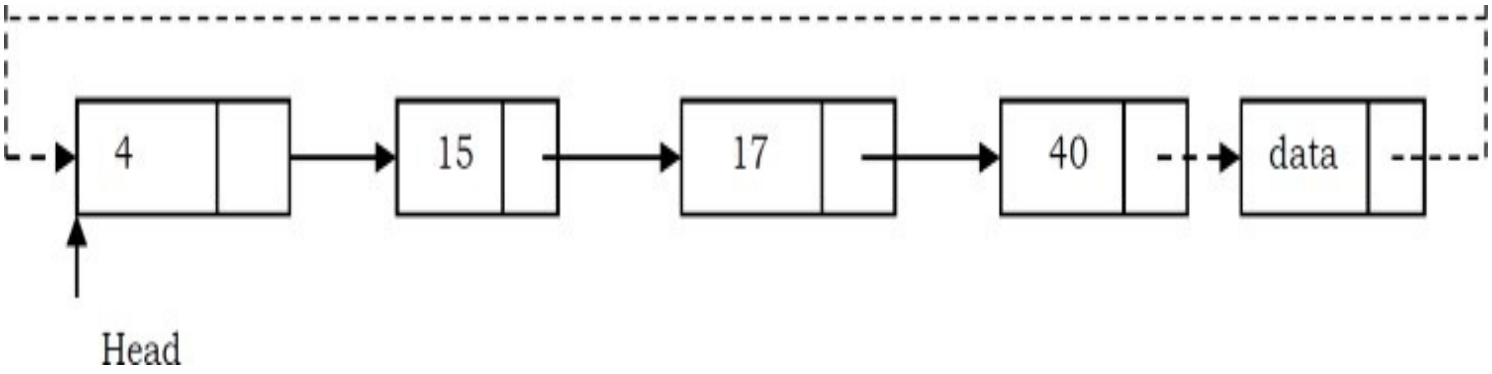
- Create a new node and initially keep its next pointer pointing to itself.



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



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



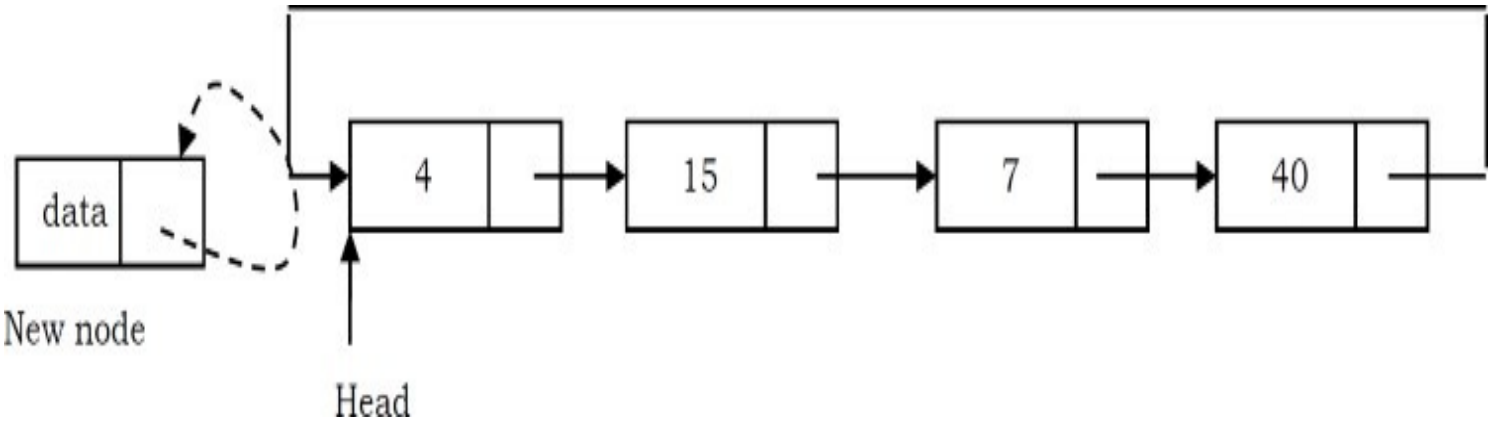
```
void InsertAtEndInCLL (struct CLLNode **head, int data) {
    struct CLLNode *current = *head;
    struct CLLNode *newNode = (struct CLLNode *) (malloc(sizeof(struct CLLNode)));
    if(!newNode) {
        printf("Memory Error");
        return;
    }
    newNode→data = data;
    while (current→next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

    newNode→next = newNode;

    if(*head ==NULL)
        *head = newNode;
    else {
        newNode→next = *head;
        current→next = newNode;
    }
}
```

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

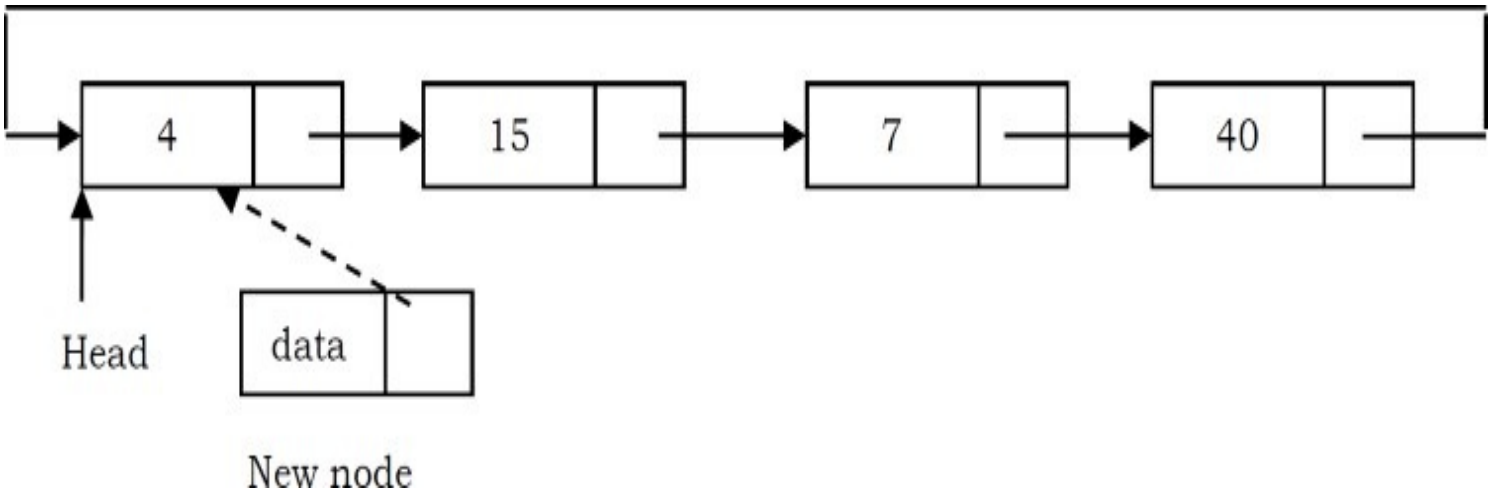
Inserting a Node at the Front of a Circular Linked List

The only difference between inserting a node at the beginning and at the end is that, after inserting the new node, we just need to update the pointer. The steps for doing this are given below:

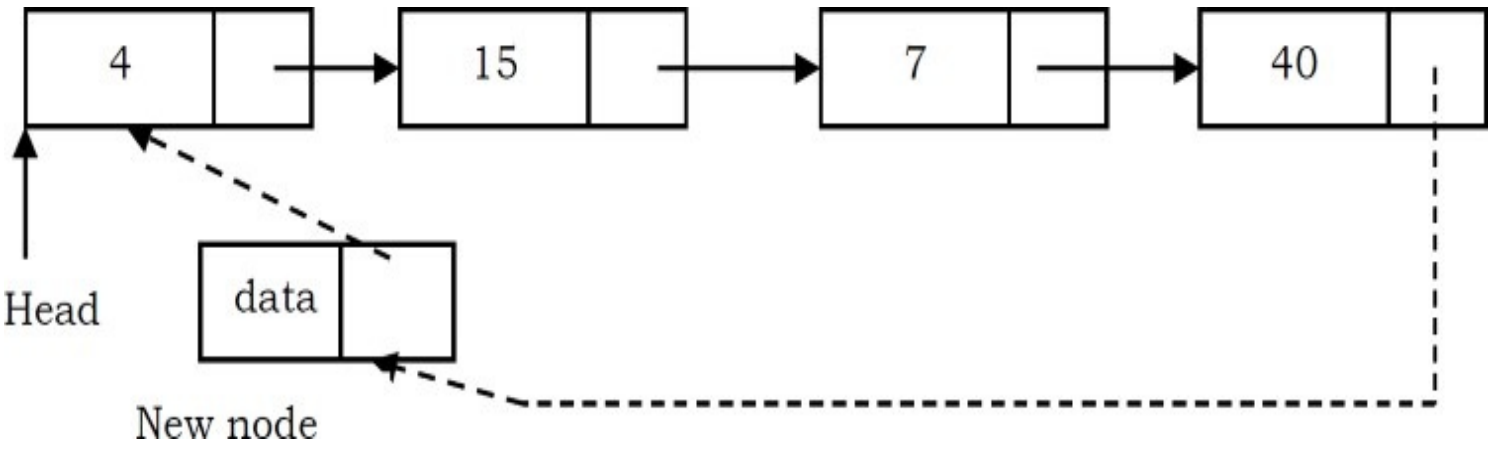
- Create a new node and initially keep its next pointer pointing to itself.



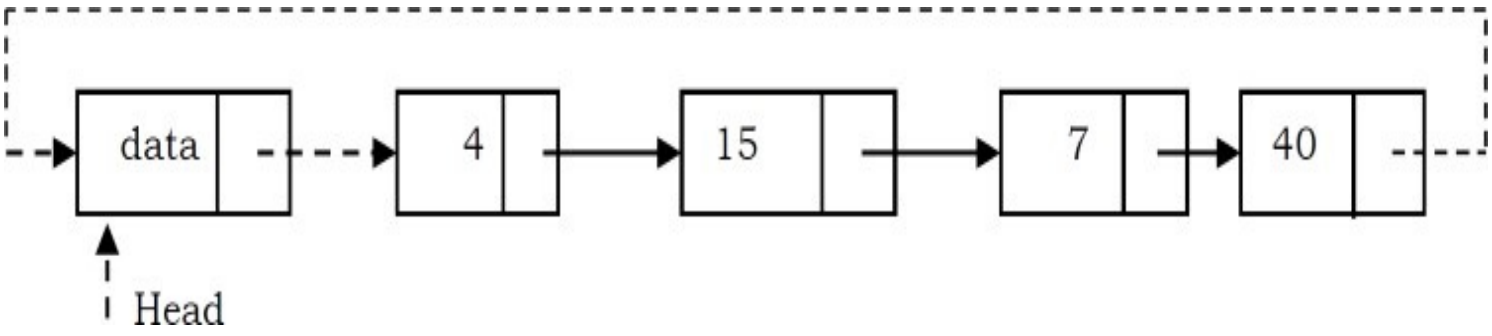
- Update the next pointer of the new node with the head node and also traverse the list until the tail. That means in a circular list we should stop at the node which is its previous node in the list.



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



- Make the new node as the head.




```
void InsertAtBeginInCLL (struct CLLNode **head, int data) {
    struct CLLNode *current = *head;
    struct CLLNode * newNode = (struct CLLNode *) (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 temporary variable.

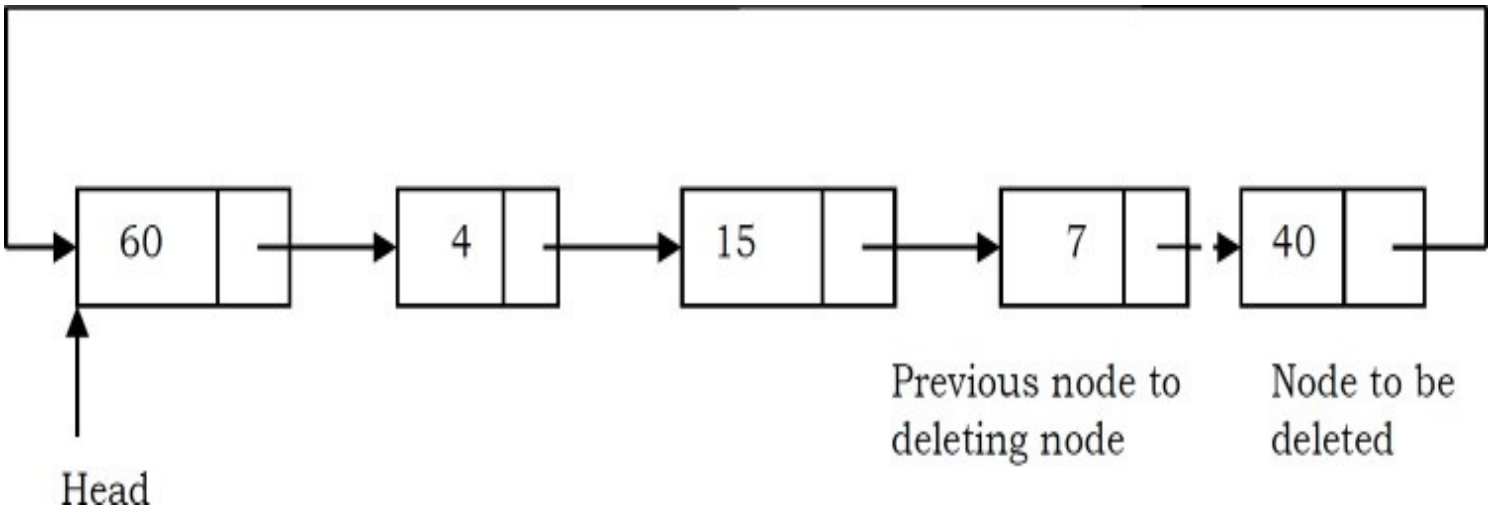
Deleting the Last Node in a Circular Linked List

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.

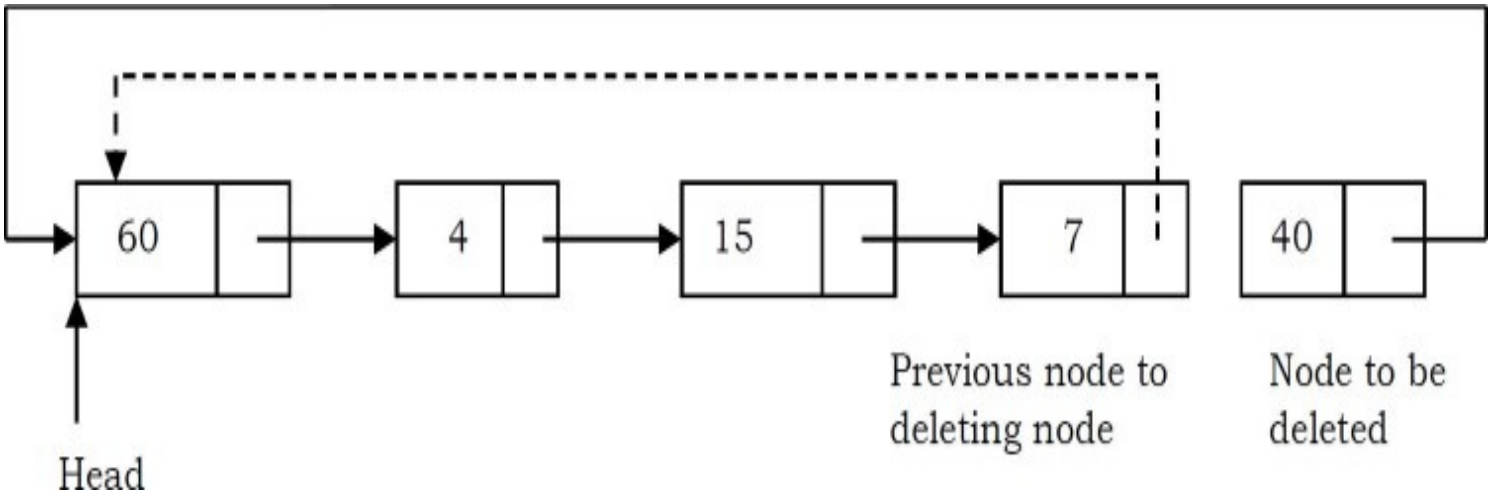
To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to

be changed to point to 60, and this node must be renamed *pTail*.

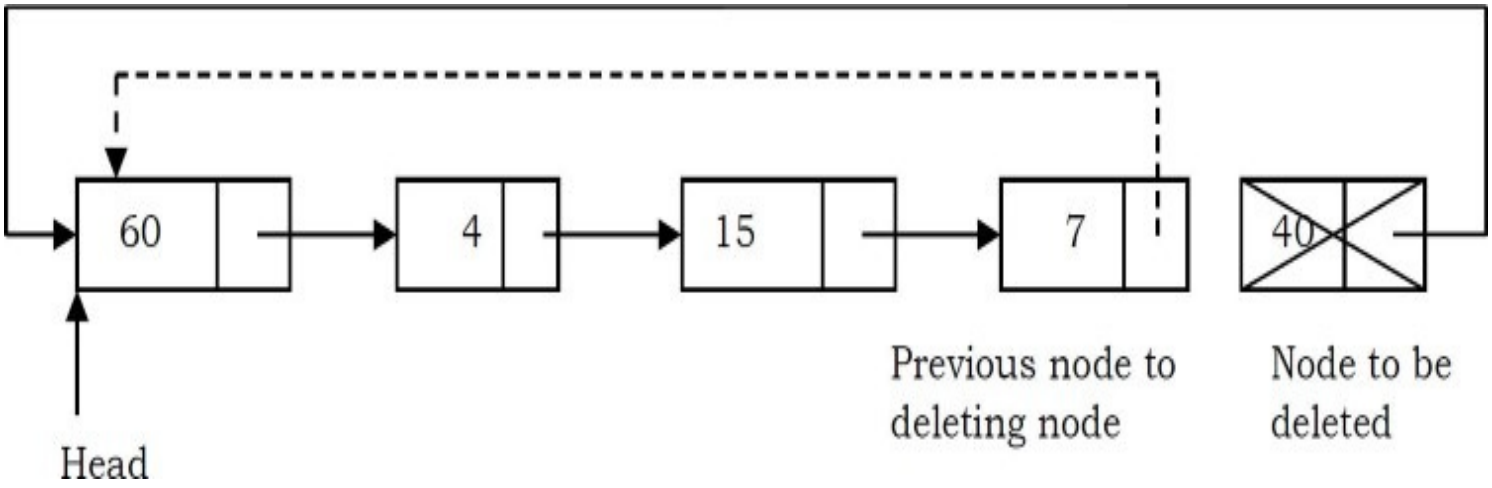
- Traverse the list and find the tail node and its previous node.



- Update the next pointer of tail node's previous node to point to head.



- Dispose of the tail node.



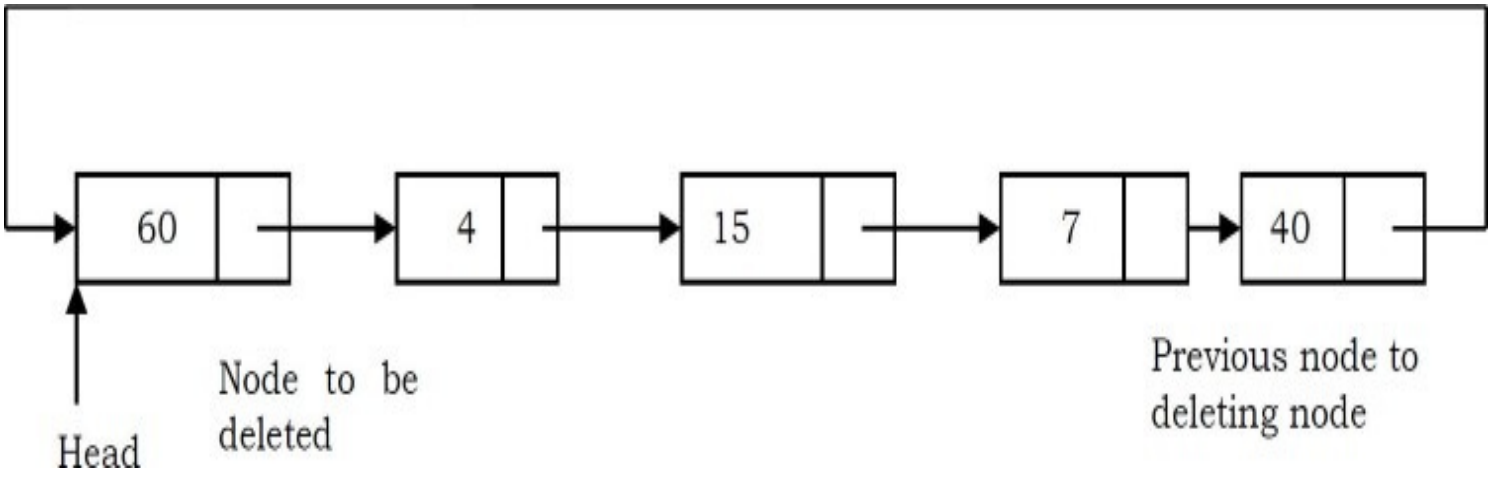
```
void DeleteLastNodeFromCLL (struct CLLNode **head) {
    struct CLLNode *temp = *head, *current = *head;
    if(*head == NULL) {
        printf( "List Empty");  return;
    }
    while (current→next != *head) {
        temp = current;
        current = current→next;
    }
    temp→next = current→next;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

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

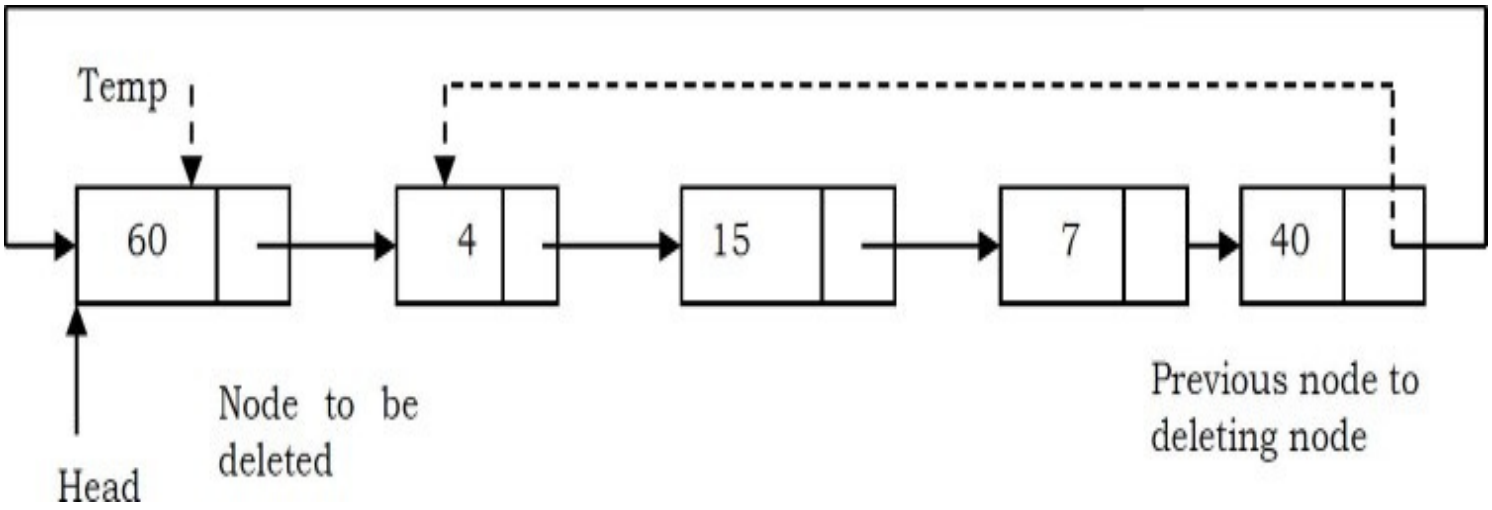
Deleting the First Node in a Circular List

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

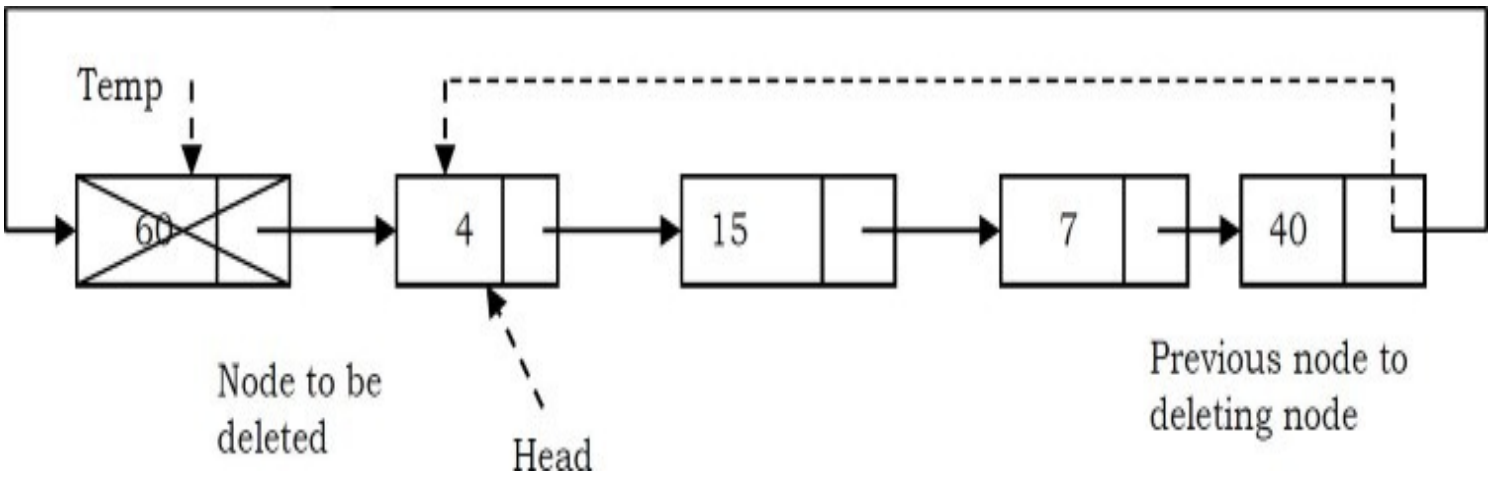
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to the 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 node 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 a 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 consist 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.

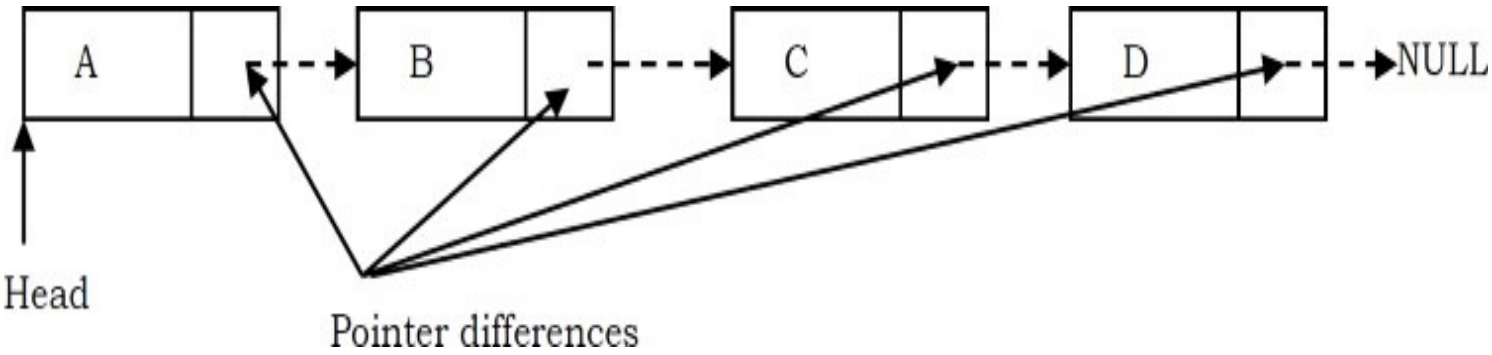
New Node Definition

```
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. The pointer difference is calculated by using exclusive-or (⊕) operation.

ptrdiff = pointer to previous node ⊕ pointer to next node.

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



In the example above,

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

(B ⊕ D) ⊕ D = B(since, D ⊕ D= 0)

Similarly, if we want to move to D, then we have to apply ⊕ to C's *ptrdiff* with B to give D.

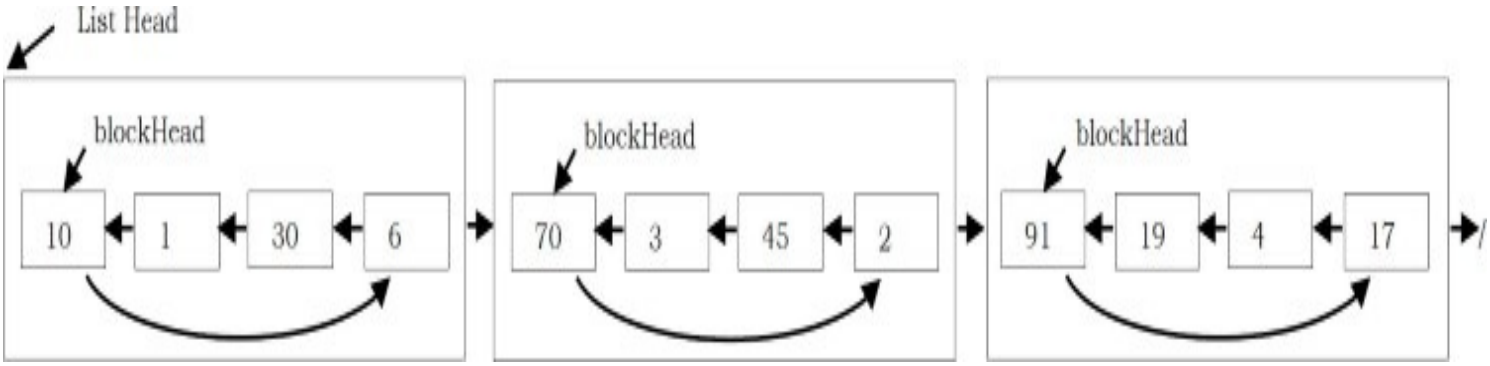
(B ⊕ D) ⊕ B = D (since, B © B=0)

From the above discussion we can see that just by using a single pointer, we can move back and forth. A memory-efficient implementation of a doubly linked list is possible with minimal compromising of timing efficiency.

3.10 Unrolled Linked Lists

One of the biggest advantages of linked lists over arrays is that inserting an element at any location takes only O(1) time. However, it takes O(*n*) to search for an element in a linked list. There is a simple variation of the singly linked list called *unrolled linked lists.*

An unrolled linked list stores multiple elements in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.



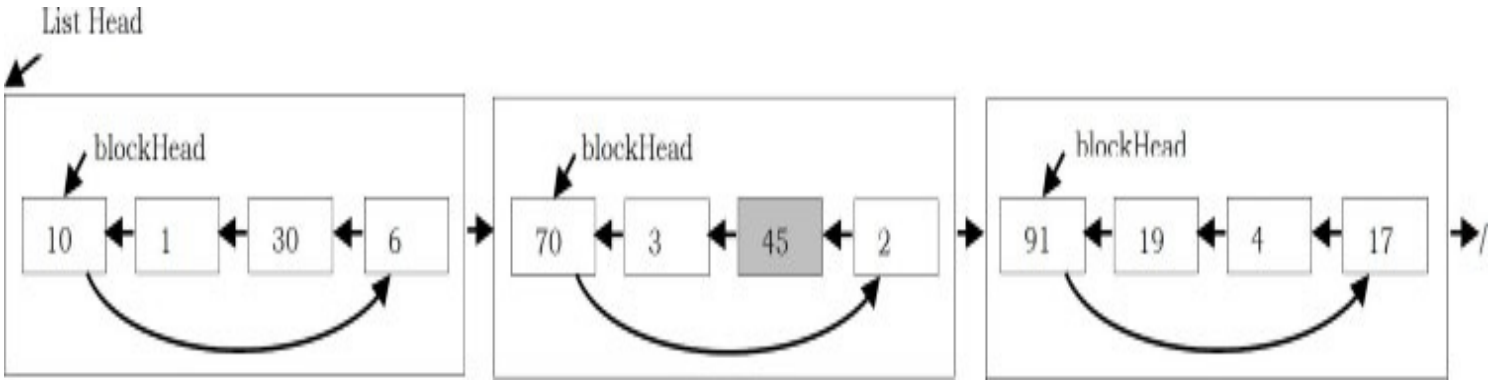
Assume that there will be no more than n elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly $\lceil \sqrt{n} \rceil$ elements. Thus,

there will be no more than $\lceil\sqrt{n}\rceil$ blocks at any time.

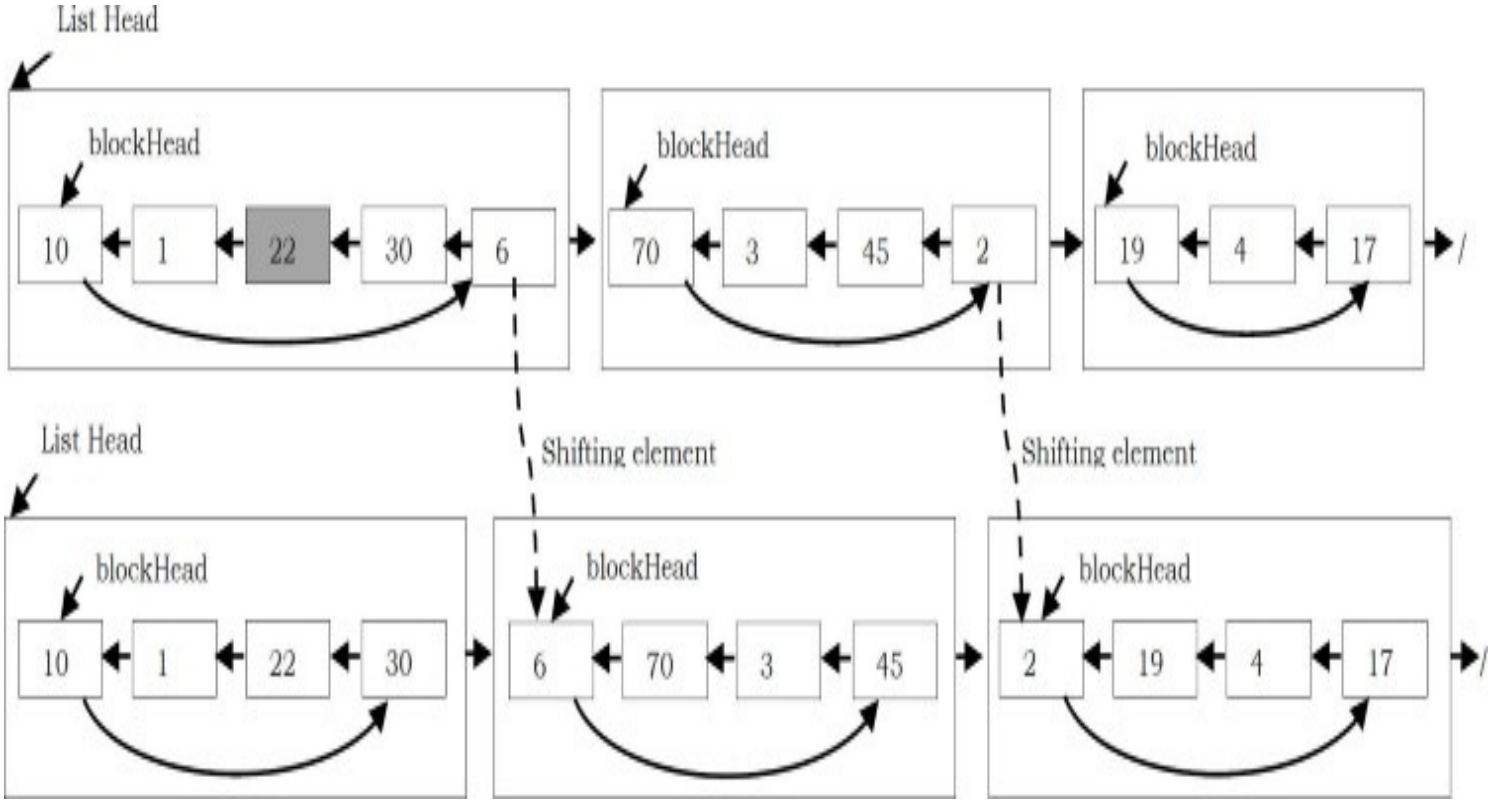
Searching for an element in Unrolled Linked Lists

In unrolled linked lists, we can find the k^{th} element in O($\sqrt{n}$):

1. Traverse the *list of blocks* to the one that contains the k^{th} node, i.e., the $\left\lceil\frac{k}{\lceil\sqrt{n}\rceil}\right\rceil^{\text{th}}$ block. It takes O($\sqrt{n}$) since we may find it by going through no more than $\sqrt{n}$ blocks.
2. Find the (k mod $\lceil\sqrt{n}\rceil$)$^{\text{th}}$ node in the circular linked list of this block. It also takes O($\sqrt{n}$) since there are no more than $\lceil\sqrt{n}\rceil$ nodes in a single block.



Inserting an element in Unrolled Linked Lists

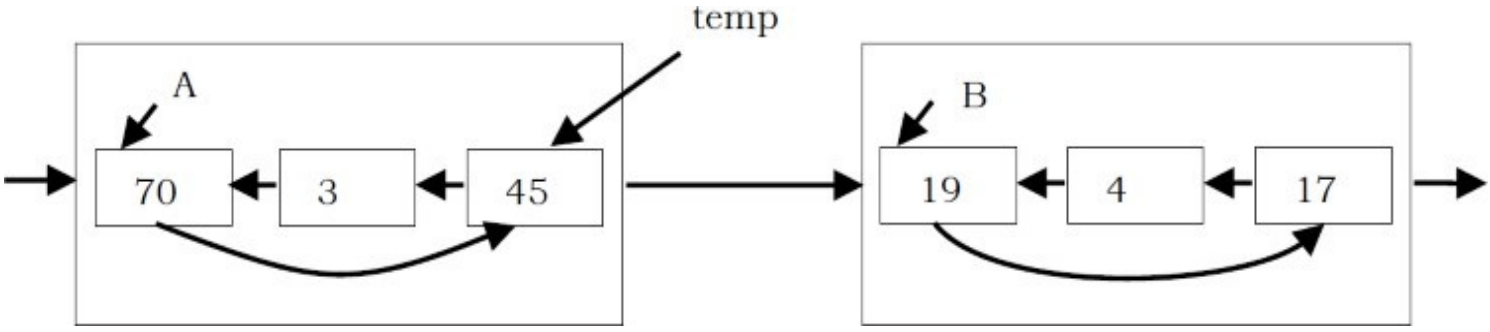


When inserting a node, we have to re-arrange the nodes in the unrolled linked list to maintain the properties previously mentioned, that each block contains $\lceil\sqrt{n}\rceil$ nodes. Suppose that we insert a node *x* after the i^{th} node, and *x* should be placed in the j^{th} block. Nodes in the j^{th} block and in the blocks after the j^{th} block have to be shifted toward the tail of the list so that each of them still have $\lceil\sqrt{n}\rceil$ nodes. In addition, a new block needs to be added to the tail if the last block of the list is out of space, i.e., it has more than $\lceil\sqrt{n}\rceil$ nodes.

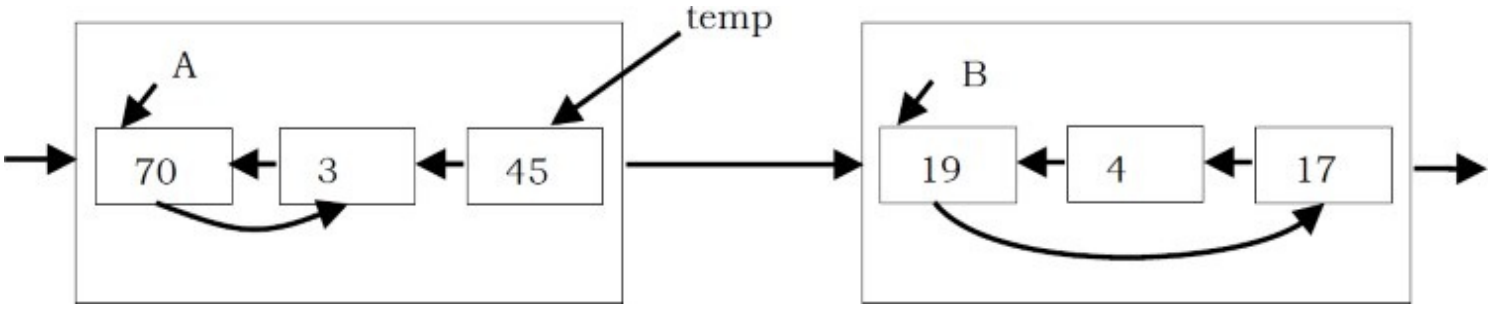
Performing Shift Operation

Note that each *shift* operation, which includes removing a node from the tail of the circular linked list in a block and inserting a node to the head of the circular linked list in the block after, takes only O(1). The total time complexity of an insertion operation for unrolled linked lists is therefore $O(\sqrt{n})$; there are at most $O(\sqrt{n})$ blocks and therefore at most $O(\sqrt{n})$ shift operations.

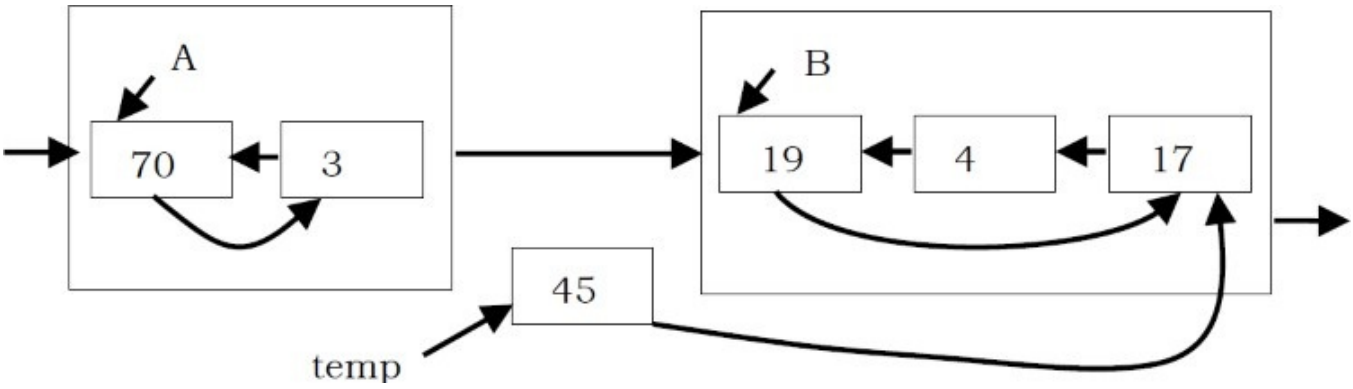
1. A temporary pointer is needed to store the tail of *A*.



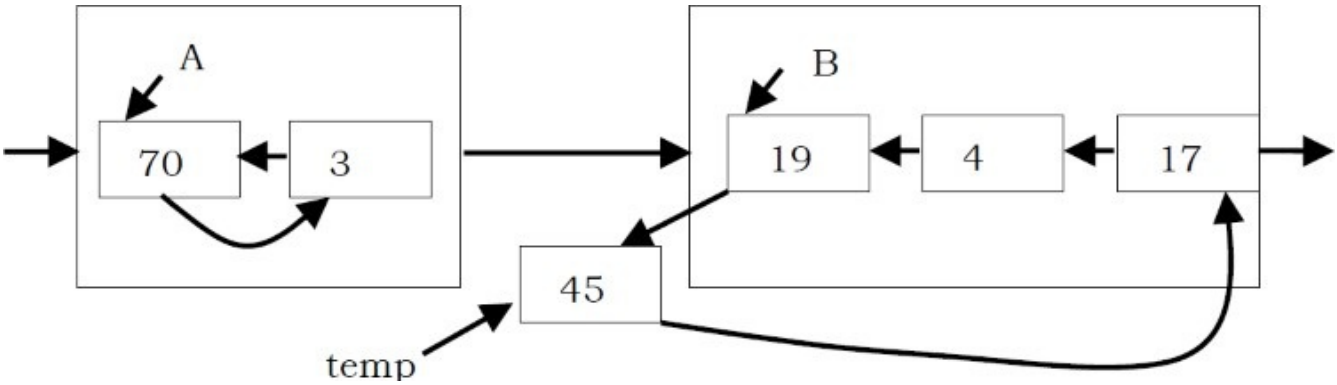
2. In block *A*, move the next pointer of the head node to point to the second-to-last node, so that the tail node of *A* can be removed.



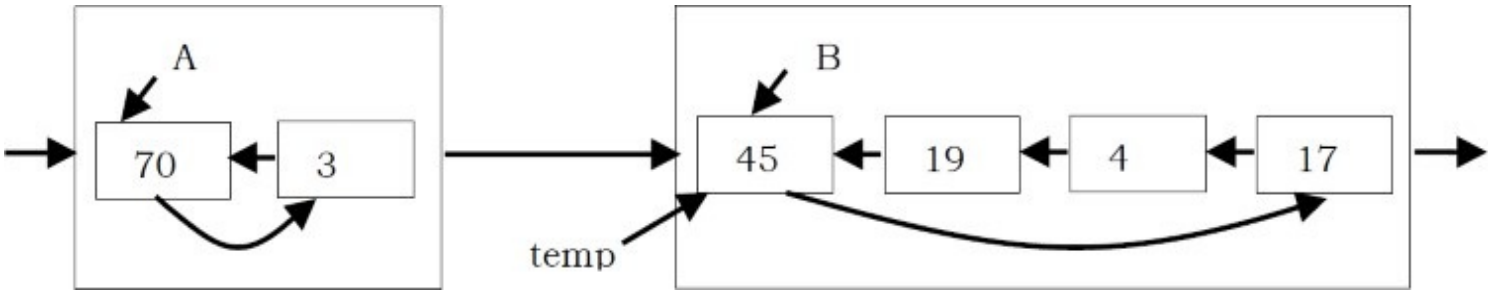
3. Let the next pointer of the node, which will be shifted (the tail node of *A*), point to the tail node of *B*.



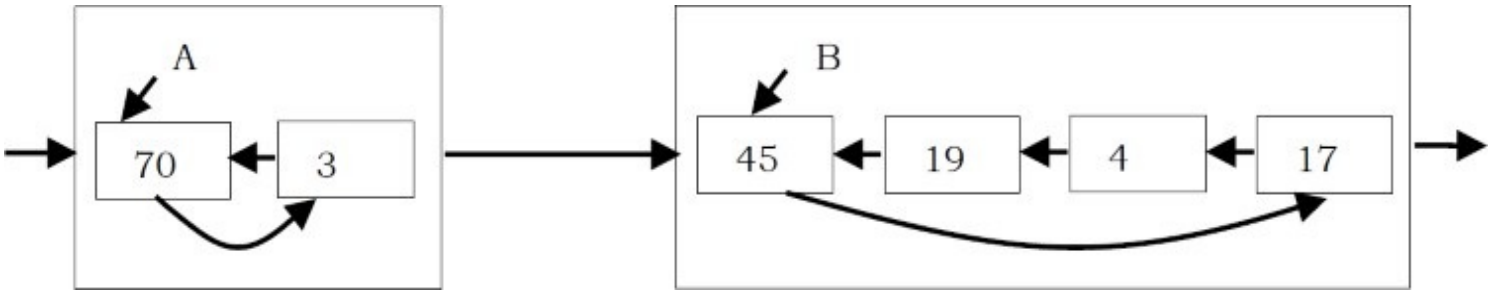
4. Let the next pointer of the head node of *B* point to the node *temp* points to.



5. Finally, set the head pointer of *B* to point to the node *temp* points to. Now the node *temp* points to becomes the new head node of *B*.



6. *temp* pointer can be thrown away. We have completed the shift operation to move the original tail node of *A* to become the new head node of *B*.



Performance

With unrolled linked lists, there are a couple of advantages, one in speed and one in space. First, if the number of elements in each block is appropriately sized (e.g., at most the size of one cache line), we get noticeably better cache performance from the improved memory locality. Second, since we have O(*n*/*m*) links, where *n* is the number of elements in the unrolled linked list and *m* is the number of elements we can store in any block, we can also save an appreciable amount of space, which is particularly noticeable if each element is small.

Comparing Linked Lists and Unrolled Linked Lists

To compare the overhead for an unrolled list, elements in doubly linked list implementations consist of data, a pointer to the next node, and a pointer to the previous node in the list, as shown below.

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

Assuming we have 4 byte pointers, each node is going to take 8 bytes. But the allocation overhead for the node could be anywhere between 8 and 16 bytes. Let's go with the best case and assume it will be 8 bytes. So, if we want to store IK items in this list, we are going to have 16KB of overhead.

Now, let's think about an unrolled linked list node (let us call it *LinkedBlock*). It will look something like this:

```
struct LinkedBlock{
    struct LinkedBlock *next;
    struct ListNode *head;
    int nodeCount;
};
```

Therefore, allocating a single node (12 bytes + 8 bytes of overhead) with an array of 100 elements (400 bytes + 8 bytes of overhead) will now cost 428 bytes, or 4.28 bytes per element. Thinking about our IK items from above, it would take about 4.2KB of overhead, which is close to 4x better than our original list. Even if the list becomes severely fragmented and the item arrays are only 1/2 full on average, this is still an improvement. Also, note that we can tune the array size to whatever gets us the best overhead for our application.

Implementation

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int blockSize; //max number of nodes in a block

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

```
struct LinkedBlock{
    struct LinkedBlock *next;
    struct ListNode *head;
    int nodeCount;
};
struct LinkedBlock* blockHead;

//create an empty block
struct LinkedBlock* newLinkedBlock(){
    struct LinkedBlock* block=(struct LinkedBlock*)malloc(sizeof(struct LinkedBlock));
    block→next=NULL;
    block→head=NULL;
    block→nodeCount=0;
    return block;
}
//create a node
struct ListNode* newListNode(int value){
    struct ListNode* temp=(struct ListNode*)malloc(sizeof(struct ListNode));
    temp→next=NULL;
    temp→value=value;
    return temp;
}
void searchElement(int k,struct LinkedBlock **fLinkedBlock,struct ListNode **fListNode){
    //find the block
    int j=(k+blockSize-1)/blockSize; //k-th node is in the j-th block
    struct LinkedBlock* p=blockHead;
    while(--j){
        p=p→next;
    }
    *fLinkedBlock=p;
    //find the node
    struct ListNode* q=p→head;
    k=k%blockSize;
    if(k==0) k=blockSize;
    k=p→nodeCount+1-k;
    while(k--){
        q=q→next;
    }
    *fListNode=q;
}
//start shift operation from block *p
void shift(struct LinkedBlock *A){
    struct LinkedBlock *B;
    struct ListNode* temp;
    while(A→nodeCount > blockSize){ //if this block still have to shift
        if(A→next==NULL){ //reach the end. A little different
            A→next=newLinkedBlock();
            B=A→next;
            temp=A→head→next;
            A→head→next=A→head→next→next;
            B→head=temp;
            temp→next=temp;
            A→nodeCount--;
            B→nodeCount++;
        }else{
            B=A→next;
            temp=A→head→next;
            A→head→next=A→head→next→next;
            temp→next=B→head→next;
            B→head→next=temp;
            B→head=temp
            A→nodeCount--;
```

```
            B→nodeCount++;
        }
        A=B;
    }
}
void addElement(int k,int x){
    struct ListNode *p,*q;
    struct LinkedBlock *r;

    if(!blockHead){ //initial, first node and block
        blockHead=newLinkedBlock();
        blockHead→head=newListNode(x);
        blockHead→head→next=blockHead→head;
        blockHead→nodeCount++;
    }else{
        if(k==0){ //special case for k=0.
        p=blockHead→head;
        q=p→next;
        p→next=newListNode(x);
        p→next→next=q;
        blockHead→head=p→next;
        blockHead→nodeCount++;
        shift(blockHead);
        }else{
            searchElement(k,&r,&p);
            q=p;
            while(q→next!=p) q=q→next;
            q→next=newListNode(x);
            q→next→next=p;
            r→nodeCount++;
            shift(r);
        }
    }
}
int searchElement(int k){
    struct ListNode *p;
    struct LinkedBlock *q;
    searchElement(k,&q,&p);
    return p→value;
}
int testUnRolledLinkedList(){
    int tt=clock();
    int m,i,k,x;
    char cmd[10];
    scanf("%d",&m);
    blockSize=(int)(sqrt(m-0.001))+1;

    for( i=0; i<m; i++ ){
        scanf("%s",cmd);
        if(strcmp(cmd,"add")==0){
            scanf("%d %d",&k,&x);
            addElement(k,x);
        }else if(strcmp(cmd,"search")==0){
            scanf("%d",&k);
            printf("%d\n",searchElement(k));
        }else{
             fprintf(stderr,"Wrong Input\n");
        }
    }
    return 0;
}
```

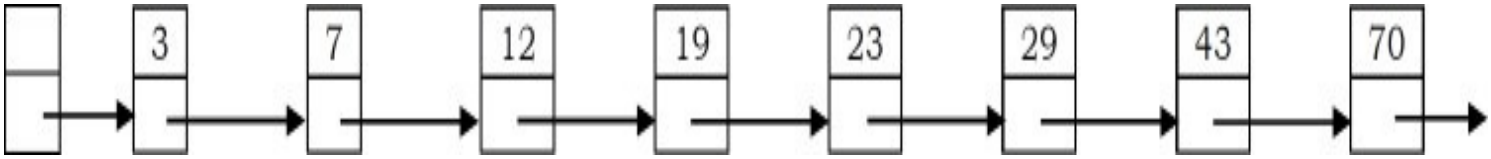
3.11 Skip Lists

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. Balanced tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

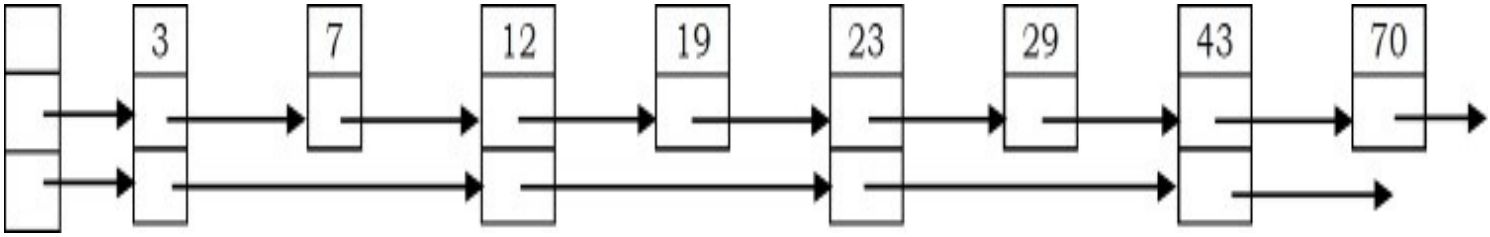
Skip lists are a probabilistic alternative to balanced trees. Skip list is a data structure that can be used as an alternative to balanced binary trees (refer to *Trees* chapter). As compared to a binary tree, skip lists allow quick search, insertion and deletion of elements. This is achieved by using probabilistic balancing rather than strictly enforce balancing. It is basically a linked list with additional pointers such that intermediate nodes can be skipped. It uses a random number generator to make some decisions.

In an ordinary sorted linked list, search, insert, and delete are in O(*n*) because the list must be scanned node-by-node from the head to find the relevant node. If somehow we could scan down the list in bigger steps (skip down, as it were), we would reduce the cost of scanning. This is the fundamental idea behind Skip Lists.

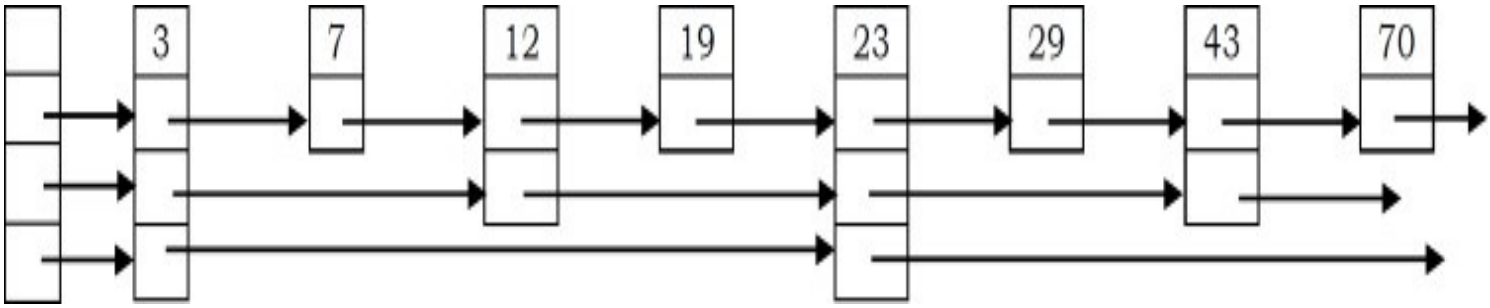
Skip Lists with One Level



Skip Lists with Two Levels



Skip Lists with Three Levels



Performance

In a simple linked list that consists of n elements, to perform a search n comparisons are required in the worst case. If a second pointer pointing two nodes ahead is added to every node, the number of comparisons goes down to $n/2 + 1$ in the worst case.

Adding one more pointer to every fourth node and making them point to the fourth node ahead reduces the number of comparisons to $\lceil n/2 \rceil + 2$. If this strategy is continued so that every node with i pointers points to $2 * i - 1$ nodes ahead, O(*logn*) performance is obtained and the number of pointers has only doubled ($n + n/2 + n/4 + n/8 + n/16 + ... = 2n$).

The find, insert, and remove operations on ordinary binary search trees are efficient, O(*logn*), when the input data is random; but less efficient, O(*n*), when the input data is ordered. Skip List performance for these same operations and for any data set is about as good as that of randomly-built binary search trees - namely O(*logn*).

Comparing Skip Lists and Unrolled Linked Lists

In simple terms, Skip Lists are sorted linked lists with two differences:

- The nodes in an ordinary list have one next reference. The nodes in a Skip List have many *next* references (also called *forward* references).
- The number of *forward* references for a given node is determined probabilistically.

We speak of a Skip List node having levels, one level per forward reference. The number of levels in a node is called the *size* of the node. In an ordinary sorted list, insert, remove, and find operations require sequential traversal of the list. This results in O(*n*) performance per operation. Skip Lists allow intermediate nodes in the list to be skipped during a traversal - resulting in an expected performance of O(*logn*) per operation.

Implementation

```
#include <stdio.h>
#include <stdlib.h>
#define MAXSKIPLEVEL 5
struct ListNode {
   int data;
   struct ListNode *next[1];
};
struct SkipList {
   struct ListNode *header;
   int listLevel;              //current level of list */
};
struct SkipList list;
struct ListNode *insertElement(int data) {
   int i, newLevel;
   struct ListNode *update[MAXSKIPLEVEL+1];
   struct ListNode *temp;
   temp = list.header;
   for (i = list.listLevel; i >= 0; i--) {
      while (temp→next[i] !=list.header && temp→next[i]→data < data)
         temp = temp→next[i];
      update[i] = temp;
   }
   temp = temp→next[0];
   if (temp != list.header && temp→data == data)
      return(temp);
   //determine level
   for (newLevel = 0; rand() < RAND_MAX/2 && newLevel < MAXSKIPLEVEL; newLevel++);
   if (newLevel > list.listLevel) {
      for (i = list.listLevel + 1; i <= newLevel; i++)
         update[i] = list.header;
      list.listLevel = newLevel;
   }
   // make new node
   if ((temp = malloc(sizeof(Node) +
     newLevel*sizeof(Node *))) == 0) {
      printf ("insufficient memory (insertElement)\n");
      exit(1);
   }
   temp→data = data;
   for (i = 0; i <= newLevel; i++) {    // update next links
      temp→next[i] = update[i]→next[i];
      update[i]→next[i] = temp;
   }
   return(temp);
}
// delete node containing data
void deleteElement(int data) {
   int i;
   struct ListNode *update[MAXSKIPLEVEL+1], *temp;
   temp = list.header;
   for (i = list.listLevel; i >= 0; i--) {
      while (temp→next[i] != list.header && temp→next[i]→data < data)
         temp = temp→next[i];
      update[i] = temp;
   }
   temp = temp→next[0];
   if (temp == list.header || !(temp→data == data) return;
   //adjust next pointers
   for (i = 0; i <= list.listLevel; i++) {
      if (update[i]→next[i] != temp) break;
      update[i]→next[i] = temp→next[i];
   }
```

```
    free (temp);
    //adjust header level
    while ((list.listLevel > 0) && (list.header→next[list.listLevel] == list.header))
        list.listLevel--;
}
// find node containing data
struct ListNode *findElement(int data) {
    int i;
    struct ListNode *temp = list.header;
    for (i = list.listLevel; i >= 0; i--) {
        while (temp→next[i] != list.header
          && temp→next[i]→data < data)
            temp = temp→next[i];
    }
    temp = temp→next[0];
    if (temp != list.header && temp→data == data) return (temp);
    return(0);
}
// initialize skip list
void initList() {
    int i;
    if ((list.header = malloc(sizeof(struct ListNode) + MAXSKIPLEVEL*sizeof(struct ListNode *))) == 0) {
        printf ("Memory Error\n");
        exit(1);
    }
    for (i = 0; i <= MAXSKIPLEVEL; i++)
        list.header→next[i] = list.header;
    list.listLevel = 0;
}
/* command-line: skipList maxnum skipList 2000: process 2000 sequential records */
int main(int argc, char **argv) {
    int i, *a, maxnum = atoi(argv[1]);
    initList();
    if ((a = malloc(maxnum * sizeof(*a))) == 0) {
        fprintf (stderr, "insufficient memory (a)\n");
        exit(1);
    }
    for (i = 0; i < maxnum; i++) a[i] = rand();
        printf ("Random, %d items\n", maxnum);
    for (i = 0; i < maxnum; i++) {
        insertElement(a[i]);
    }
    for (i = maxnum-1; i >= 0; i--) {
        findElement(a[i]);
    }
    for (i = maxnum-1; i >= 0; i--) {
        deleteElement(a[i]);
    }
    return 0;
}
```

3.12 Linked Lists: Problems & Solutions

Problem-1 Implement Stack using Linked List.

Solution: Refer to *Stacks* chapter.

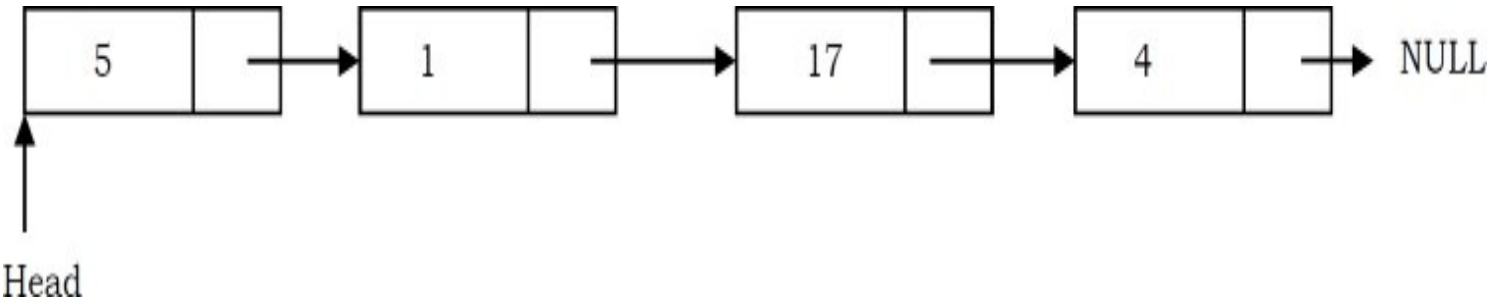
Problem-2 Find n^{th} node from the end of a Linked List.

Solution: Brute-Force Method: Start with the first node and count the number of nodes present after that node. If the number of nodes is $< n - 1$ then return saying "fewer number of nodes in the list". If the number of nodes is $> n - 1$ 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 the hash table), we can find the list length. Let us say the list length is M. To find n^{th} from the 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 is just a matter of

returning M- n + 1^{th} key value from the hash table.

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

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

Solution: Yes. If we observe the Problem-3 solution, what we are actually doing is finding the size of the linked list. That means we are using the 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 the list and the 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: $O(1)$. Hence, no need to create 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 point to head node of the list. *pNthNode* starts moving only after *pTemp* has made n moves.

From there both move forward until *pTemp* reaches the end of the list. As a result *pNthNode* points to n^{th} node from the end of the linked list.

Note: At any point of time both move one node at a time.

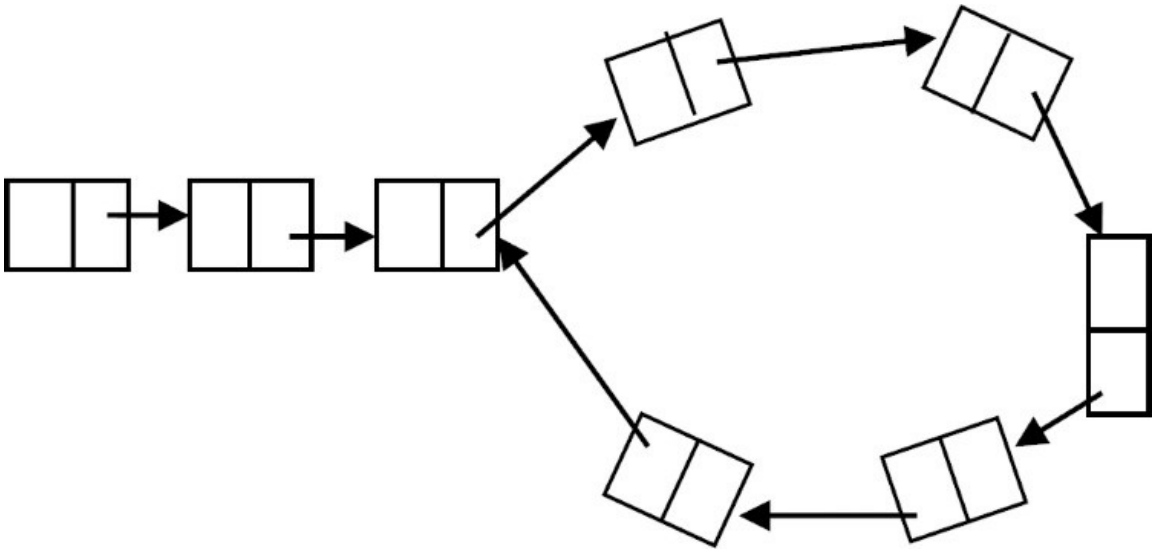
```
struct ListNode *NthNodeFromEnd(struct ListNode *head, int NthNode){
    struct ListNode *pNthNode = NULL, *pTemp = head;
    for(int count =1; count< NthNode;count++) {
        if(pTemp)
            pTemp = pTemp→next;
    }
    while(pTemp) {
        if(pNthNode == NULL)
            pNthNode = head;
        else
            pNthNode = pNthNode→next;
        pTemp = pTemp→next;
    }
    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 the regular list is that, in this list, there are two nodes whose next pointers are the same. In regular singly linked lists (without a loop) each node's next pointer is unique.

That means the repetition of next pointers indicates the existence of a 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 the current node's address. If there is a node with the same address then that indicates that some other node is pointing to the current node and we can say a loop exists. Continue this process for all the nodes of the linked list.

Does this method work? 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 an infinite loop)?

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

Problem-7 Can we use the 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 address of the node is available in the hash table or not.
- If it is already available in the hash table, that indicates that we are visiting the node that 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 available in the hash table, insert that node's address into the hash table.
- Continue this process until we reach the end of the linked list *or* we find the loop.

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

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

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

algorithm fails.

Algorithm:

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

Time Complexity; O(*nlogn*) for sorting the next pointers array.
Space Complexity; O(*n*) for the next pointers array.

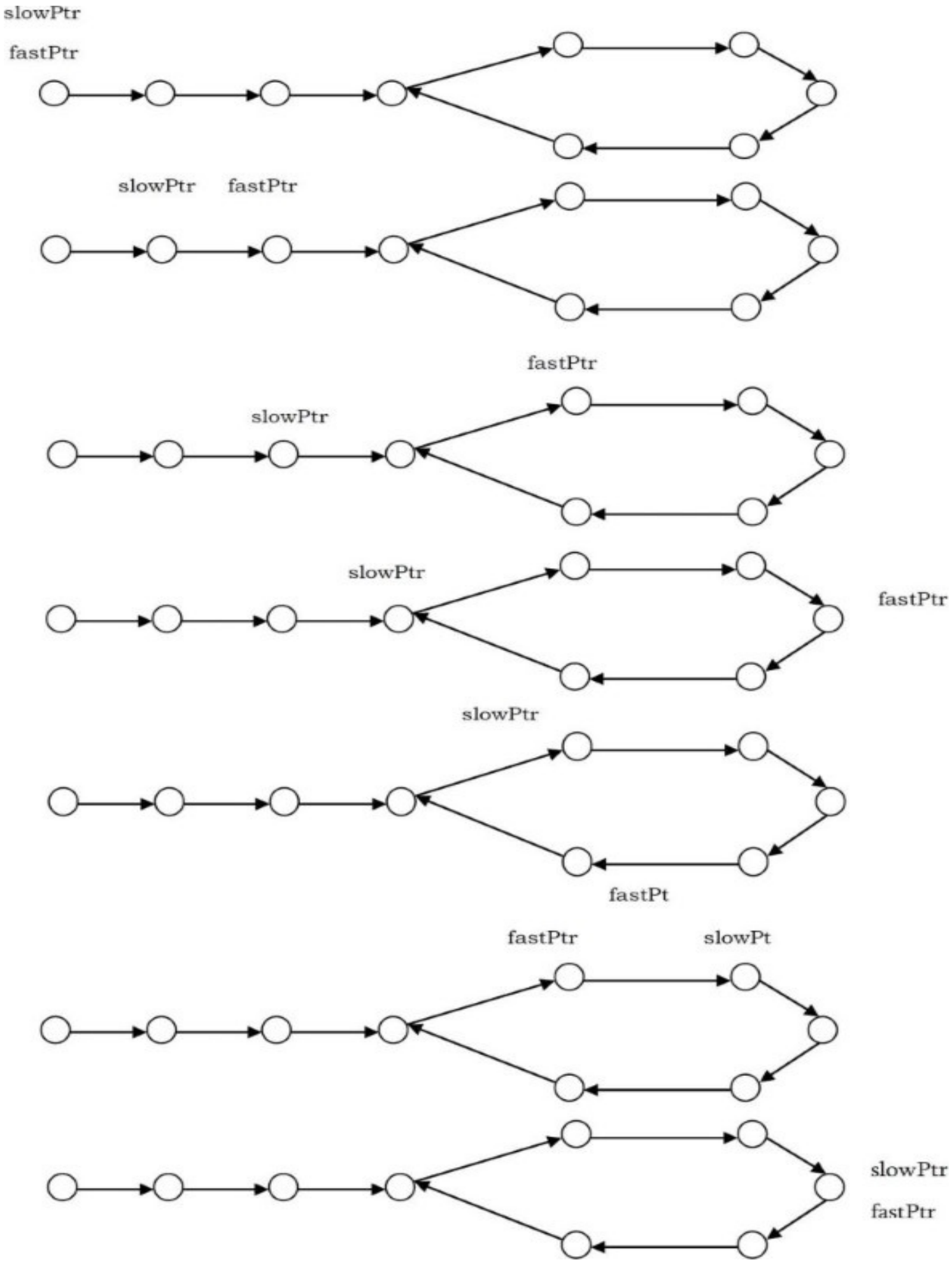
Problem with the above algorithm: The above algorithm works only if we can find the length of the list. But if the list has a loop then we may end up in an infinite loop. Due to this reason the algorithm fails.

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

Solution: Yes. Efficient Approach (Memoryless Approach): This problem was solved by *Floyd*. The solution is named the Floyd cycle finding algorithm. It uses *two* 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 diagrams below we can see that after the final step they are meeting at some point in the loop which may not be the starting point of the loop.

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


slowPtr
fastPtr
slowPtr
fastPtr
fastPtr
slowPtr
slowPtr
fastPtr
slowPtr
fastPt
fastPtr
slowPt
slowPtr
fastPtr

```
int DoesLinkedListHasLoop(struct ListNode * head) {
  struct ListNode *slowPtr = head, *fastPtr = head;
  while (slowPtr && fastPtr && fastPtr→next) {
    slowPtr = slowPtr→next;
    fastPtr = fastPtr→next→next;
    if (slowPtr == fastPtr)
      return 1;
  }
  return 0;
}
```

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

Problem-10 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 the same as Problem-6.

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

Solution: The solution is an extension to the solution in Problem-9. After finding the loop in the linked list, we initialize the *slowPtr* to the head of the linked list. From that point onwards both *slowPtr* and *fastPtr* move 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→next) {
           slowPtr = slowPtr→next;
           fastPtr = fastPtr→next→next;
           if (slowPtr == fastPtr){
              loopExists = 1;
              break;
           }
        }
        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 the meeting of tortoise and hare concludes the existence of the loop, but how does moving the tortoise to the beginning of the linked list while keeping the hare at the meeting place, followed by moving both one step at a time, make them meet at the starting point of the cycle?

Solution: This problem is at the heart of number theory. In the 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 position of the tortoise 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 the Floyd cycle finding algorithm, does it work if we use steps 2 and 3 instead of 1 and 2?

Solution: Yes, but the complexity might be high. Trace out an example.

Problem-14 Check whether the given linked list is NULL-terminated. If there is a cycle, find the length of the loop.

Solution: This solution is also an extension of the basic cycle detection problem. After finding the loop in the linked list, keep the *slowPtr* as it is. The *fastPtr* keeps on moving until it again comes back to *slowPtr*. While moving *fastPtr*, 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→next) {
            slowPtr = slowPtr→next;
            fastPtr = fastPtr→next→next;
            if (slowPtr == fastPtr){
                loopExists = 1;
                break;
            }
        }
        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).

Recursive version: We will find it easier to start from the bottom up, by asking and answering tiny questions (this is the approach in The Little Lisper):

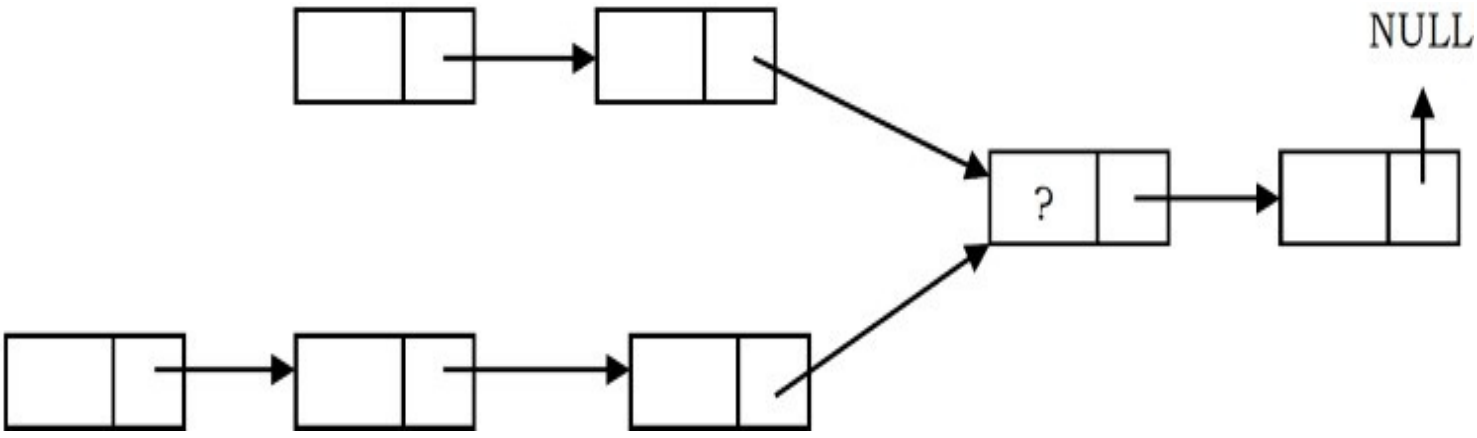
- What is the reverse of NULL (the empty list)? NULL.
- What is the reverse of a one element list? The element itself.

- What is the reverse of an *n* element list? The reverse of the second element followed by the first element.

```
struct ListNode * RecursiveReverse(struct ListNode *head) {
  if (head == NULL)
        return NULL;
  if (head→next == NULL)
        return list;
  struct ListNode *secondElem = head→next;
  // Need to unlink list from the rest or you will get a cycle
  head→next = NULL;
  // reverse everything from the second element on
  struct ListNode *reverseRest = RecursiveReverse(secondElem);
  secondElem→next = head;        // then we join the two lists
  return reverseRest;
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*),for recursive stack.

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 lists before they intersect is unknown and may be different in each list. *List*1 may have n nodes before it reaches the intersection point, and *List*2 might have *m* nodes before it reaches the intersection point where *m* and n may be $m = n, m < n$ or $m > n$. Give an algorithm for finding 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 be O(*mn*) which will be high.

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

Problem-18 Can we solve Problem-17 using the 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 them in some array and sort them.
- Take second list node pointers and keep them in some array and sort them.
- After sorting, use two indexes: one for the first sorted array and the other for the second sorted array.
- Start comparing values at the indexes and increment the index according to whichever has the lower value (increment only if the values are not equal).
- At any point, if we are able to find two indexes whose values are the 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 that gives the maximum value.

Space Complexity: O(1).

Any 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 the same for both the lists. The algorithm works fine only in one case and it is when both lists have the ending node at their merge point.

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

Solution: Yes.

Algorithm:

- Select a list which has 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.
- If there is a merge point for the given 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*), depending on which list we select for creating the hash table. But in both

cases the time complexity is the 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 addresses onto the first stack.
- Traverse the second list and push all the node addresses onto 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 the same, take the top elements from both the stacks and keep them in some temporary variable (since both node addresses are node, it is enough if we use one temporary variable).
- Continue this process until the top node addresses of the stacks are not the same.
- This point is the one where the lists merge into a 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 Problem-17?

Solution: Yes. Using "finding the first repeating number" approach in an array (for algorithm refer to *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 [Refer to *Searching* chapter for algorithm].
- The first repeating number indicates the merging point of both the lists.

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

Problem-22 Can we still think of finding an alternative solution for 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 elements, 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 that 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 Problem-17?

Solution: Yes.

Efficient Approach:

- Find lengths (L1 and L2) of both lists - 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;
    }
    if(L1 < L2) {
      head1 = list2;  head2 = list1;  diff = L2 - L1;
    }else{
      head1 = list1;  head2 = list2;  diff = L1 - L2;
    }
    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 node of the list.

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 the hash table for solving Problem-24?

Solution: Yes. The reasoning is the 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 the end of the list, the second pointer will be pointing to the middle node.

Note: If the list has an 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 the 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 a 2x pointer. Take a pointer that moves at 2x [two nodes at a time]. At the end, if the length is even, then the pointer will be NULL; otherwise it will point to the last node.

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

Time Complexity: O(⌊n/2⌋) ≈ 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, how to merge them into the third list in sorted order?

Solution: Assume the sizes of lists are m and n.

Recursive:

```
struct ListNode *MergeSortedList(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 = MergeSortedList(a→next, b);
   }
   else {
      result =b;
      result→next = MergeSortedList(b→next,a);
   }
   return result;
}
```

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

Iterative:

```
struct ListNode *MergeSortedListIterative(struct ListNode *head1, struct ListNode *head2){
    struct ListNode * newNode = (struct ListNode*) (malloc(sizeof(struct ListNode)));
    struct ListNode *temp;
    newNode = new Node;
    newNode→next = NULL;
    temp = newNode;
    while (head1!=NULL and head2!=NULL){
        if (head1→data<=head2→data){
            temp→next = head1;
            temp = temp→next;
            head1 = head1→next;
        }else{
            temp→next = head2;
            temp = temp→next;
            head2 = head2→next;
        }
    }
    if (head1!=NULL)
        temp→next = head1;
    else
        temp→next = head2;

    temp = newNode→next;
    free(newNode);
    return temp;
}
```

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 → 2 → 3 → 4 → *X*, then after the function has been called the linked list would hold 2 → 1 → 4 → 3 → *X*.

Solution:

Recursive:

```
struct ListNode *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;
                head = temp;
                //Call the method recursively for the rest of the list
                head→next→next = ReversePairRecursive(head→next→next);
                return head;
        }
}
```

Iterative:

```
struct ListNode *ReversePairIterative(struct ListNode *head) {
        struct ListNode *temp1=NULL, *temp2=NULL, *current = head;
        while(current != NULL && current→next != NULL)  {
              if (temp1 != null) {
                temp1→next→next = current→next;
              }
              temp1 = current→next;
              current→next = current→next→next;
              temp1.next = current;
              if (temp2 == null)
                 temp2 = temp1;
             current = current→next;
        }
       return temp2;
}
```

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.

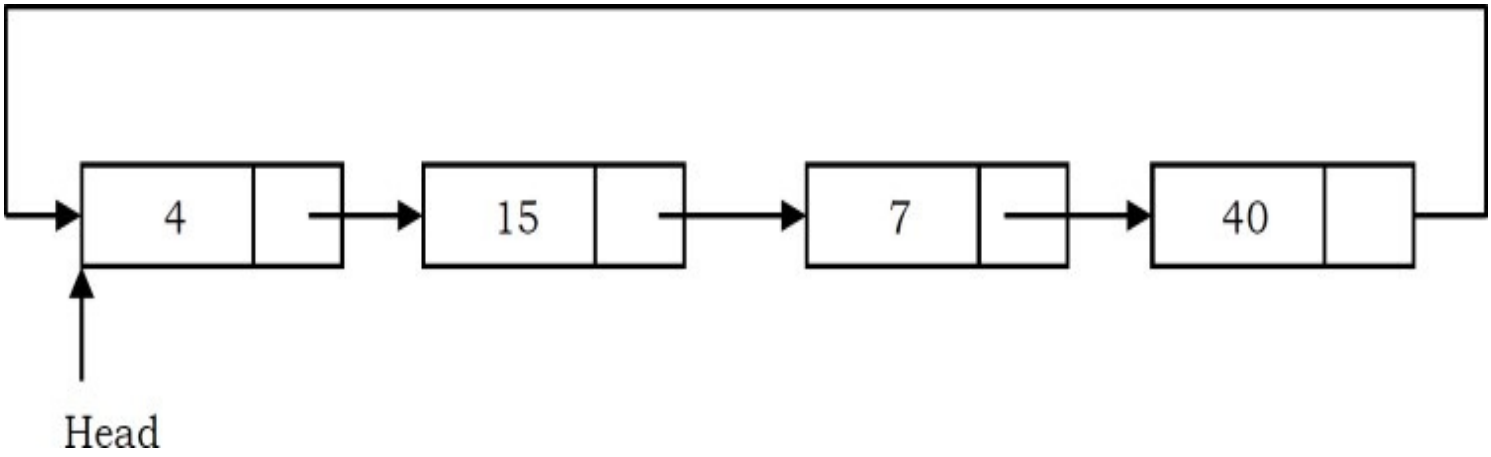
Problem-35 Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

Solution:

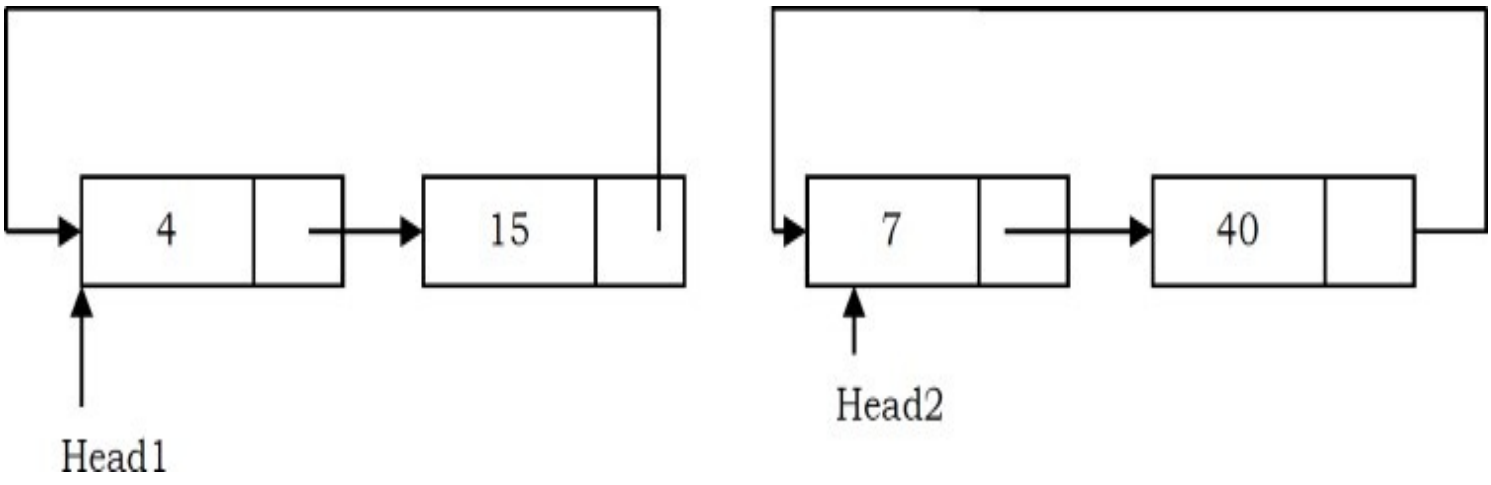
Algorithm:

- 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-36 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 the case of circular doubly linked lists we don't have to traverse the lists.

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 and at the same time keep it in a hash table. For n elements we have to keep all the elements in a hash table which gives a 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 Table.

Problem-40 **Josephus Circle:** 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;
        }
       // Close the circular linked list by having the last node point to the first.
        p→next = q;
        // 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 eliminated 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, a 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 a 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 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 the original list again and set the pointers building 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.

```
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 temp→next→random is the new copy.
    temp = head;
    while (temp != NULL) {
      temp2 = temp→random;
      temp2→random = temp→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) ≈ O(*n*). Space Complexity: O(1).

Problem-43 We are given a pointer to a node (*n*ot the tail node) in a singly linked list. Delete that node from the linked list.

Solution: To delete a node, we have to adjust the next pointer of the previous node to point to the

next node instead of the current one. Since we don't have a pointer to the previous node, we can't redirect its next pointer. So what do we do? We can easily get away by moving the data from the next node into the current node and then deleting the next node.

```
void deleteaNodeinLinkedList( struct ListNode * node ){
    struct ListNode * temp = node->next;
    node->data = node->next->data;
    node->next = temp->next;
    free(temp);
}
```

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

Problem-44 Given a linked list with even and odd numbers, create an algorithm for making changes to the list in such a way that all even numbers appear at the beginning.

Solution: To solve this problem, we can use the splitting logic. While traversing the list, split the linked list into two: one contains all even nodes and the other contains all odd nodes. Now, to get the final list, we can simply append the odd node linked list after the even node linked list.

To split the linked list, traverse the original linked list and move all odd nodes to a separate linked list of all odd nodes. At the end of the loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes the same, we must insert all the odd nodes at the end of the odd node list.

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

Problem-45 In a linked list with n nodes, the time taken to insert an element after an element pointed by some pointer is

(A) O(1)
(B) O($logn$)
(C) O(n)
(D) O($nlogn$)

Solution: A.

Problem-46 **Find modular node:** Given a singly linked list, write a function to find the last element from the beginning whose $n\%k == 0$, where n is the number of elements in the list and k is an integer constant. For example, if $n = 19$ and $k = 3$ then we should return 18^{th} node.

Solution: For this problem the value of n is not known in advance.

```
struct ListNode *modularNodeFromBegin(struct ListNode *head, int k){
    struct ListNode * modularNode;
    int i=0;
    if(k<=0)
        return NULL;
    for (;head != NULL; head = head→next){
        if(i%k == 0){
            modularNode = head;
        }
        i++;
    }
    return modularNode;
}
```

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

Problem-47 **Find modular node from the end:** Given a singly linked list, write a function to find the first from the end whose $n\%k == 0$, where n is the number of elements in the list and k is an integer constant. If $n = 19$ and $k = 3$ then we should return 16^{th} node.

Solution: For this problem the value of n is not known in advance and it is the same as finding the k^{th} element from the end of the the linked list.

```
struct ListNode *modularNodeFromEnd(struct ListNode *head, int k){
    struct ListNode *modularNode=NULL;
    int i=0;
    if(k<=0)
        return NULL;
    for (i=0; i < k; i++){
        if(head)
            head = head→next;
        else
            return NULL;
    }
    while(head != NULL)
        modularNode = modularNode→next;
        head = head→next;
    }
    return modularNode;
}
```

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

Problem-48 **Find fractional node:** Given a singly linked list, write a function to find the $\frac{n}{k}th$ element, where n is the number of elements in the list.

Solution: For this problem the value of n is not known in advance.

```
struct ListNode *fractionalNodes(struct ListNode *head, int k){
    struct ListNode *fractionalNode = NULL;
    int i=0;
    if(k<=0)
        return NULL;
    for (;head != NULL; head = head→next){
        if(i%k == 0){
            if(fractionalNode == NULL)
                fractionalNode = head;
            else fractionalNode = fractionalNode→next;
        }
        i++;
    }
    return fractionalNode;
}
    int i=0;
    if(k<=0)
        return NULL;
    for (;head != NULL; head = head→next){
        if(i%k == 0){
            if(fractionalNode == NULL)
                fractionalNode = head;
            else fractionalNode = fractionalNode→next;
        }
        i++;
    }
    return fractionalNode;
}
```

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

Problem-49 **Find $\sqrt{n}^{th}$ node:** Given a singly linked list, write a function to find the $\sqrt{n}^{th}$ element, where *n* is the number of elements in the list. Assume the value of *n* is not known in advance.

Solution: For this problem the value of *n* is not known in advance.

```
struct ListNode *sqrtNode(struct ListNode *head){
    struct ListNode *sqrtN = NULL;
    int i=1, j=1;
    for (;head != NULL; head = head→next){
        if(i == j*j){
            if(sqrtN == NULL)
                sqrtN = head;
            else
                sqrtN = sqrtN→next;

            j++;
        }
        i++;
    }
    return sqrtN;
}
```

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

Problem-50 Given two lists List 1 = $\{A_1, A_2, \ldots, A_n\}$ and List2 = $\{B_1, B_2, \ldots, B_m\}$ with data (both lists) in ascending order. Merge them into the third list in ascending order so that the merged list will be:

$$\{A_1, B_1, A_2, B_2 \ldots\ldots A_m, B_m, A_{m+1} \ldots\ldots A_n\} \text{ if } n >= m$$
$$\{A_1, B_1, A_2, B_2 \ldots\ldots A_n, B_n, B_{n+1} \ldots\ldots B_m\} \text{ if } m >= n$$

Solution:

```
struct ListNode*AlternateMerge(struct ListNode *List1, struct ListNode *List2){
    struct ListNode *newNode = (struct ListNode*) (malloc(sizeof(struct ListNode)));
    struct ListNode *temp;
    newNode→next = NULL;
    temp = newNode;
    while (List1!=NULL and List2!=NULL){
        temp→next = List1;
        temp = temp→next;
        List1 = List1→next;
        temp→next = List2;
        List2 = List2→next;
        temp = temp→next;
    }
    if (List1!=NULL)
        temp→next = List1;
    else
        temp→next = List2;
    temp = newNode→next;
    free(newNode);
    return temp;
}
```

Time Complexity: The *while* loop takes O(*min*(*n*,*m*)) time as it will run for *min*(*n*,*m*) times. The other steps run in O(1). Therefore the total time complexity is O(*min*(*n*,*m*)). Space Complexity: O(1).

Problem-51 Median in an infinite series of integers

Solution: Median is the middle number in a sorted list of numbers (if we have an odd number of elements). If we have an even number of elements, the median is the average of two middle numbers in a sorted list of numbers. We can solve this problem with linked lists (with both sorted and unsorted linked lists).

First, let us try with an *unsorted* linked list. In an unsorted linked list, we can insert the element either at the head or at the tail. The disadvantage with this approach is that finding the median takes O(*n*). Also, the insertion operation takes O(1).

Now, let us try with a *sorted* linked list. We can find the median in O(1) time if we keep track of

the middle elements. Insertion to a particular location is also O(1) in any linked list. But, finding the right location to insert is not O(*logn*) as in a sorted array, it is instead O(*n*) because we can't perform binary search in a linked list even if it is sorted. So, using a sorted linked list isn't worth the effort as insertion is O(*n*) and finding median is O(1), the same as the sorted array. In the sorted array the insertion is linear due to shifting, but here it's linear because we can't do a binary search in a linked list.

Note: For an efficient algorithm refer to the *Priority Queues and Heaps* chapter.

Problem-52 Given a linked list, how do you modify it such that all the even numbers appear before all the odd numbers in the modified linked list?

Solution:

```
struct ListNode *exchangeEvenOddList(struct ListNode *head){
    // initializing the odd and even list headers
    struct ListNode *oddList = NULL, *evenList =NULL;
    // creating tail variables for both the list
    struct ListNode *oddListEnd = NULL, *evenListEnd = NULL;
    struct ListNode *itr=head;
    if( head == NULL ){
        return;
    }
    else{
        while( itr != NULL ){
            if( itr→data % 2 == 0 ){
              if( evenList == NULL ){
                  // first even node
                  evenList = evenListEnd = itr;
              }
              else{
                  // inserting the node at the end of linked list
                  evenListEnd→next = itr;
                  evenListEnd = itr;
              }
            }
            else{
              if( oddList == NULL ){
                  // first odd node
                  oddList = oddListEnd = itr;
              }
              else{
                  // inserting the node at the end of linked list
                  oddListEnd→next = itr;
                  oddListEnd = itr;
              }
            }
            itr = itr→next;
        }
        evenListEnd→next = oddList;
        return head;
    }
}
```

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

Problem-53 Given two linked lists, each list node with one integer digit, add these two linked lists. The result should be stored in the third linked list. Also note that the head node contains the most significant digit of the number.

Solution: Since the integer addition starts from the least significant digit, we first need to visit the last node of both lists and add them up, create a new node to store the result, take care of the carry if any, and link the resulting node to the node which will be added to the second least significant node and continue.

First of all, we need to take into account the difference in the number of digits in the two numbers. So before starting recursion, we need to do some calculation and move the longer list pointer to the appropriate place so that we need the last node of both lists at the same time. The other thing we need to take care of is *carry*. If two digits add up to more than 10, we need to forward the *carry* to the next node and add it. If the most significant digit addition results in a *carry*, we need to create an extra node to store the *carry*.

The function below is actually a wrapper function which does all the housekeeping like calculating lengths of lists, calling recursive implementation, creating an extra node for the *carry* in the most significant digit, and adding any remaining nodes left in the longer list.

```
void addListNumbersWrapper(struct ListNode *list1, struct ListNode *list2, int *carry, struct ListNode **result){
    int list1Length = 0, list2Length = 0, diff =0;
    struct ListNode  *current  = list1;
    while(current){
        current = current→next;
        list1Length++;
    }
    current = list2;
     while(current){
        current = current→next;
        list2Length++;
    }
     if(list1Length < list2Length){
        current = list1;
        list1 = list2;
        list2 = current;
    }
    diff  = abs(list1Length-list2Length);
    current = list1;
    while(diff--)
        current = current→next;
    addListNumbers(current, list2, carry, result);
    diff  = abs(list1Length-list2Length);
    addRemainingNumbers(list1, carry, result, diff);
    if(*carry){
        struct ListNode  * temp = (struct ListNode  *)malloc(sizeof(struct ListNode ));
        temp→next =  (*result);
        *result = temp;
    }
    return;
}
void addListNumbers(struct ListNode  *list1, struct ListNode  *list2, int *carry, struct ListNode  **result){
    int sum;
    if(!list1)
        return;
    addListNumbers(list1→next, list2→next, carry, result);
    //End of both lists, add them
    struct ListNode  * temp = (struct ListNode  *)malloc(sizeof(struct ListNode ));
    sum = list1→data + list2→data + (*carry);
    // Store carry
    *carry = sum/10;
    sum = sum%10;
    temp→data = sum;
    temp→next =  (*result);
     *result = temp;
    return;
}
void addRemainingNumbers(struct ListNode  * list1, int *carry, struct ListNode  **result, int diff){
    int sum =0;
     if(!list1  || diff == 0)
        return;
    addRemainingNumbers(list1→next, carry, result, diff-1);
    struct ListNode  * temp = (struct ListNode  *)malloc(sizeof(struct ListNode ));
    sum = list1->data + (*carry);
    *carry = sum/10;
    sum = sum%10;
    temp→data = sum;
    temp→next =  (*result);
    *result = temp;
    return;
}
```

Time Complexity: O(*max*(*List*1 *length*,*List*2 *length*)).
Space Complexity: O(*min*(*List*1 *length*, *List*1 *length*)) for recursive stack.

Note: It can also be solved using stacks.

Problem-54 Which sorting algorithm is easily adaptable to singly linked lists?

Solution: Simple Insertion sort is easily adabtable to singly linked lists. To insert an element, the linked list is traversed until the proper position is found, or until the end of the list is reached. It is inserted into the list by merely adjusting the pointers without shifting any elements, unlike in the array. This reduces the time required for insertion but not the time required for searching for the proper position.

Problem-55 Given a list, List1 = $\{A_1, A_2, \ldots A_{n-1;} A_n)$ with data, reorder it to $\{A_1, A_n, A_2, A_{n-1}\}$ without using any extra space.

Solution: Find the middle of the linked list. We can do it by *slow* and *fast* pointer approach. After finding the middle node, we reverse the right halfl then we do a in place merge of the two halves of the linked list.

Problem-56 Given two sorted linked lists, given an algorithm for the printing common elements of them.

Solution: The solution is based on merge sort logic. Assume the given two linked lists are: list1 and list2. Since the elements are in sorted order, we run a loop till we reach the end of either of the list. We compare the values of list1 and list2. If the values are equal, we add it to the common list. We move list1/list2/both nodes ahead to the next pointer if the values pointed by list1 was less / more / equal to the value pointed by list2.

Time complexity O($m + n$), where m is the lengh of list1 and n is the length of list2. Space Complexity: O(1).

```
struct ListNode  *commonElement(struct ListNode  *list1, struct ListNode  *list2) {
        struct ListNode  *temp =  (struct ListNode  *)malloc(sizeof(struct ListNode));
        struct ListNode  *head = temp;
        while (list1 != null && list2 != null) {
                if (list1→data == list2→data) {
                        head→next = new ListNode(list1→data); // Copy common element.
                        list1 = list1→next;
                        list2 = list2→next;
                        head = head→next;
                } else if (list1→data > list2→data) {
                        list2 = list2→next;
                } else { // list1→data < list2→data
                        list1 = list1→next;
                }
        }
        return temp.next;
}
```

CHAPTER

4

STACKS

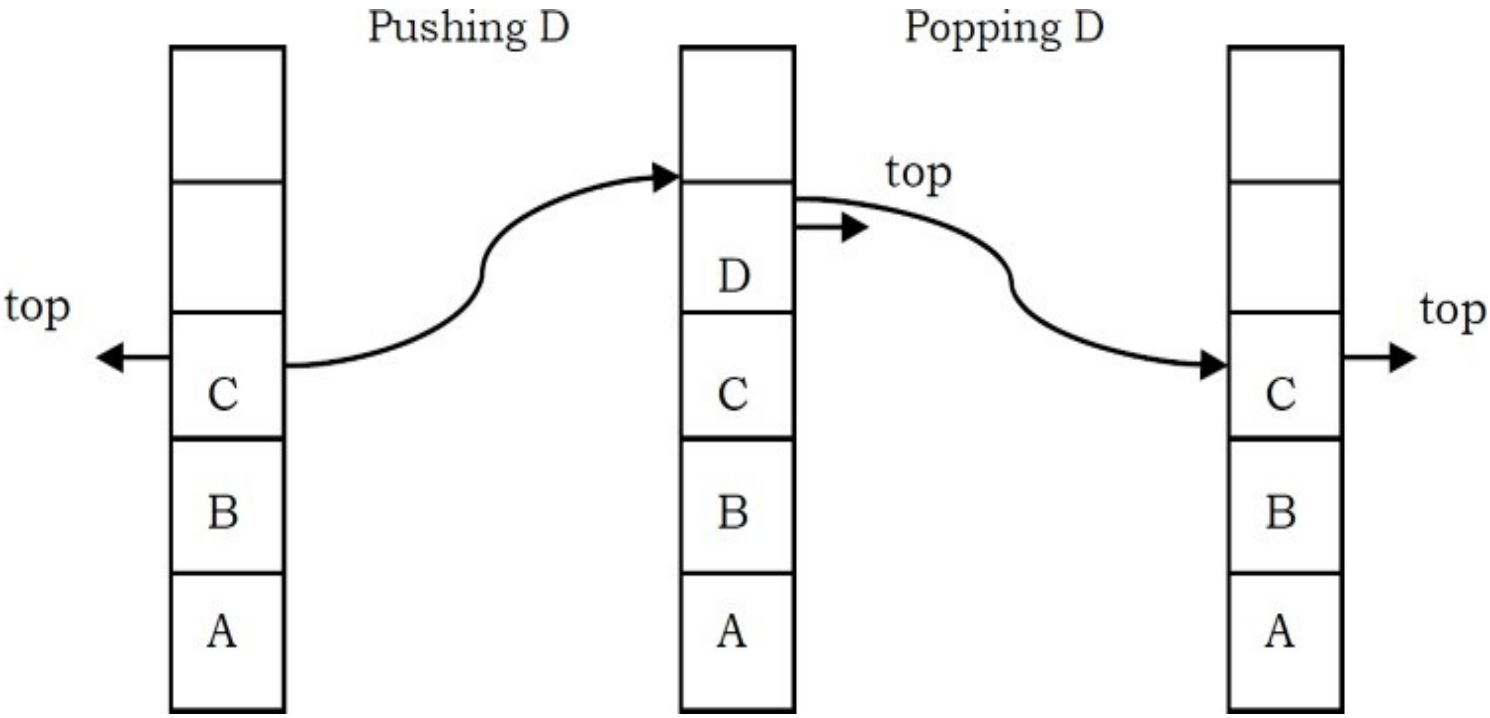
4.1 What is a Stack?

A stack is a simple data structure used for storing data (similar to Linked Lists). In a stack, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and 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, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the 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 *push*, and when an element is removed from the stack, the concept is called *pop*. Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *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 puts the long-term project aside and begins work on the new task. The phone rings, and 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 that was abandoned to answer the phone is retrieved from the pending tray and work progresses. To take another call, 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 an 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 the stack.
- int IsEmptyStack(): Indicates whether any elements are stored in the 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 some of the applications in which stacks play 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 to *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)

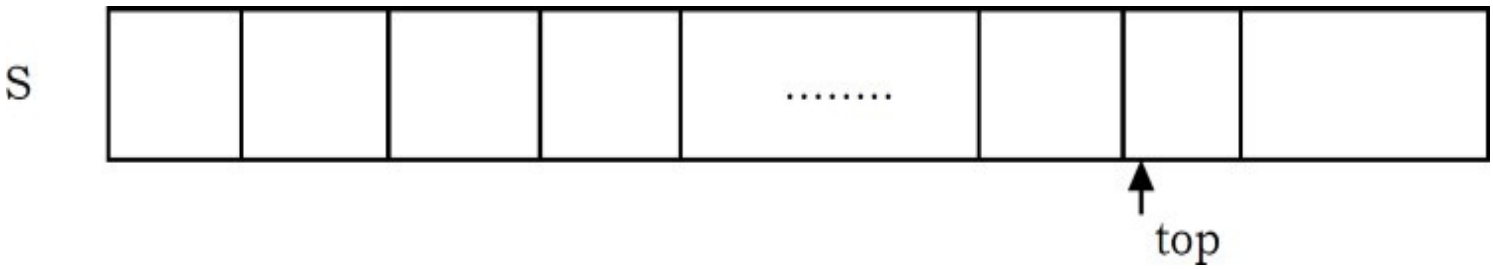
4.5 Implementation

There are many ways of implementing stack ADT; 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 an empty stack it will throw *stack empty exception.*

```
#define MAXSIZE 10
struct ArrayStack {
        int top;
        int capacity;
        int *array;
};
struct ArrayStack *CreateStack() {
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        if(!S)
           return NULL;
        S→capacity = MAXSIZE;
        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){
      /* S→top == - 1 indicates empty stack*/
        if(IsEmptyStack(S)){
           printf("Stack is Empty");
           return INT_MIN;;
        }
        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 IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Limitations

The maximum size of the stack must first be defined and it 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 an empty queue with *top* value equal to –1. The issue that still needs to be resolved is what we do when all the slots in the 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 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 the 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 the 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 the new array and at the end add the new element. After n push operations the total time $T(n)$ (*n*umber of copy operations) is proportional to $1 + 2 + ... + n \approx O(n^2)$.

Alternative Approach: Repeated Doubling

Let us improve the complexity by using the array *doubling* technique. If the array is full, create a new array of twice the size, and copy the 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 up to $n = 32$. That means, we do the doubling at 1,2,4,8,16. The other way of analyzing the same approach is: at $n = 1$, if we want to add (push) an element, double the current size of the array and copy all the elements of the old array to the new array.

At $n = 1$, we do 1 copy operation, at $n = 2$, we do 2 copy operations, and at $n = 4$, 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 2*n* 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 the expression below. The total time $T(n)$ of a series of n push operations is proportional to

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

$T(n)$ 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 * sizeof(int));
}
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);
    }
}
```

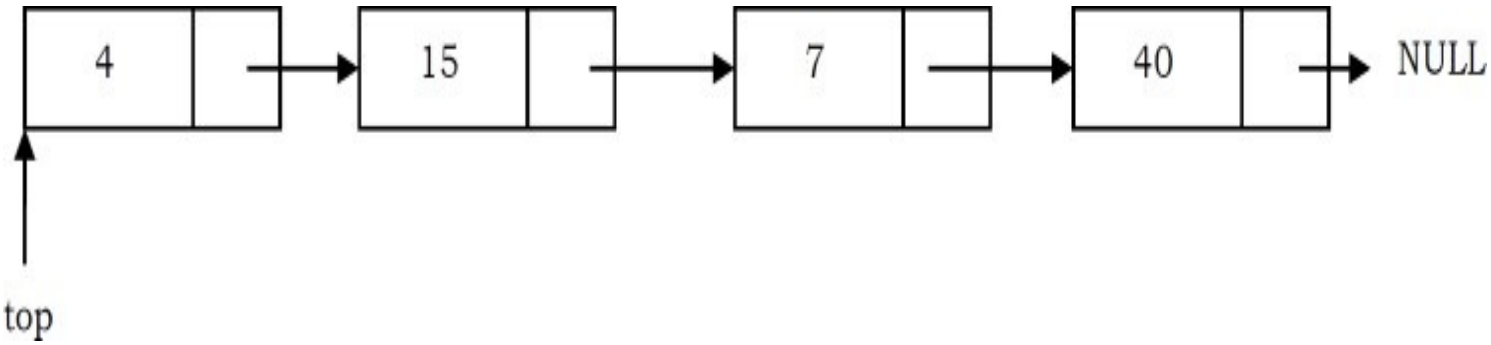
Performance

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 PushQ	O(1) (Average)
Time Complexity of PopQ	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmpryStackf)	O(1))
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	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. 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, that is, $T(n)/n$.

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 analysis, refer to the *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 Stacks: Problems & Solutions

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. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [- then it is written to the stack. When a closing delimiter is encountered like), }, or]-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 has a 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*), where 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 focus 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 associativity (order of evaluation) among operators.

Token	Operator	Precedence	Associativity
( ) [ ] → .	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 234*+. 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 an 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?

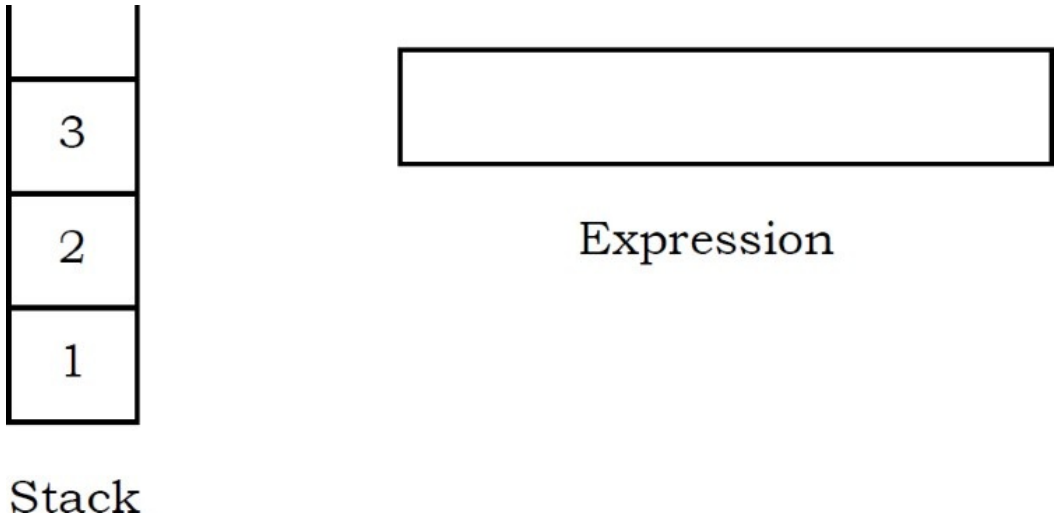
Solution:

Algorithm:

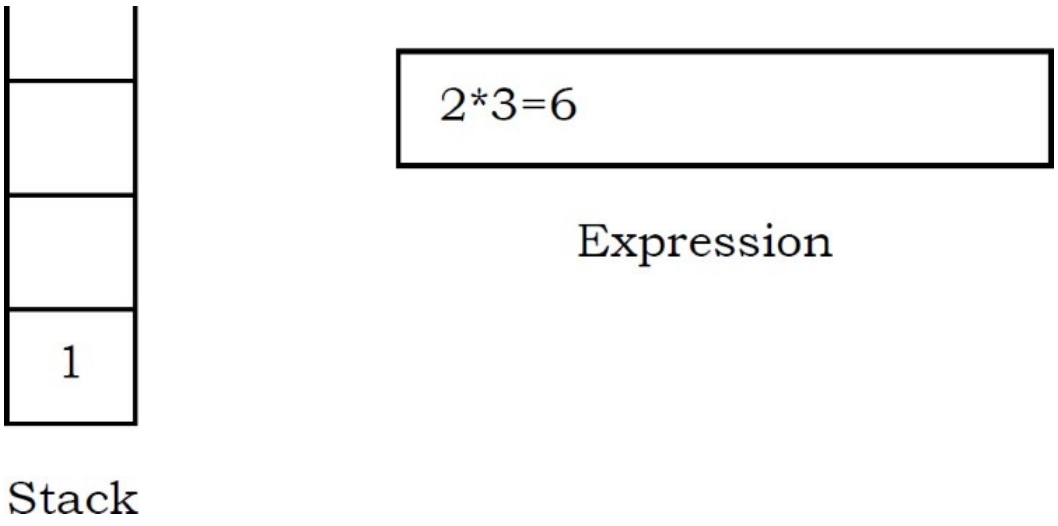
1 Scan the Postfix string from left to right.
2 Initialize an empty stack.
3 Repeat steps 4 and 5 till all the characters are scanned.
4 If the scanned character is an operand, push it onto the stack.
5 If the scanned character is an operator, and if the operator is a unary operator, then pop an element from the stack. If the operator is a 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-mentioned algorithm works using an example. Assume that the postfix string is 123*+5-.

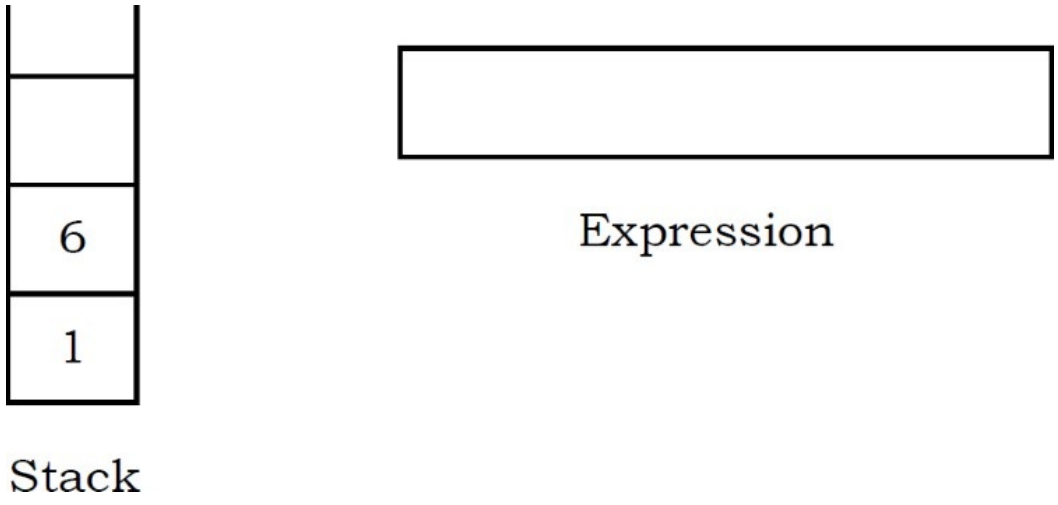
Initially the stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. They will be pushed into the stack in that order.



The 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.

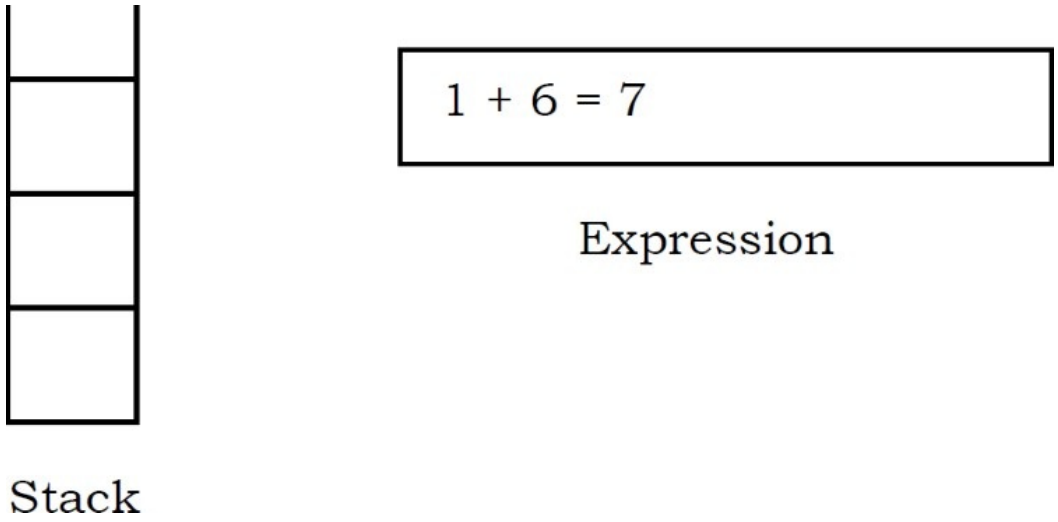


The value of the expression (2*3) that has been evaluated (6) is pushed into the stack.

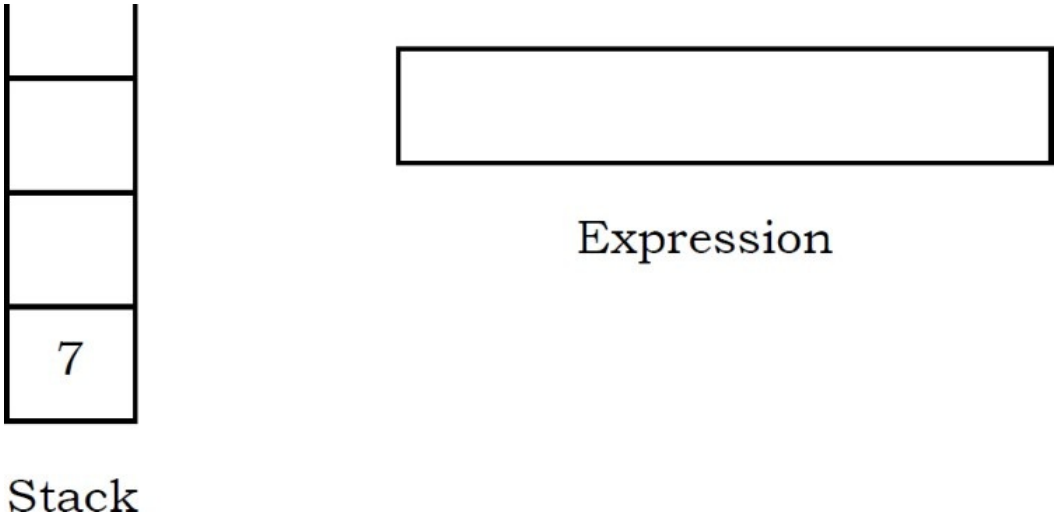


The next character scanned is "+", which is an operator. Thus, we pop the top two elements from

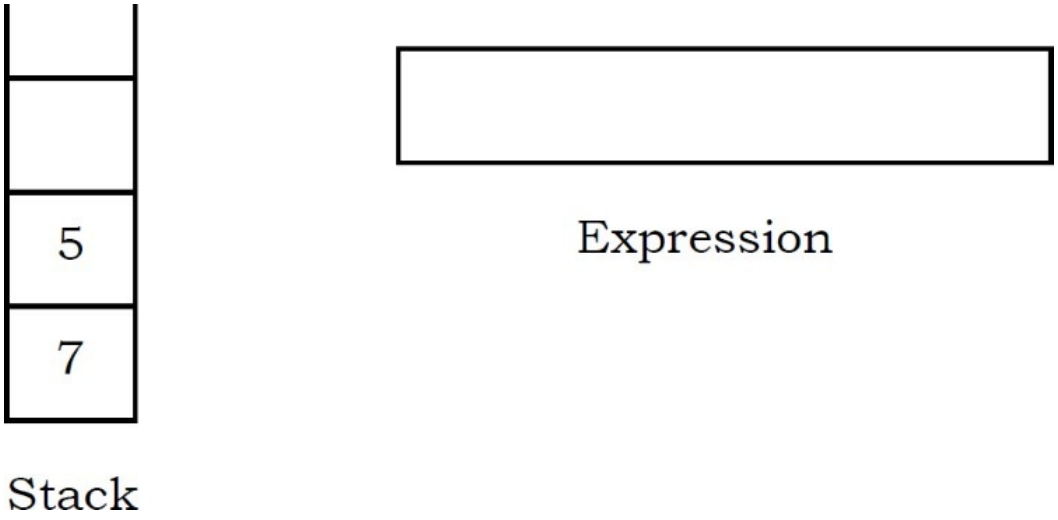
the stack and perform the “+” operation with the two operands. The second operand will be the first element that is popped.



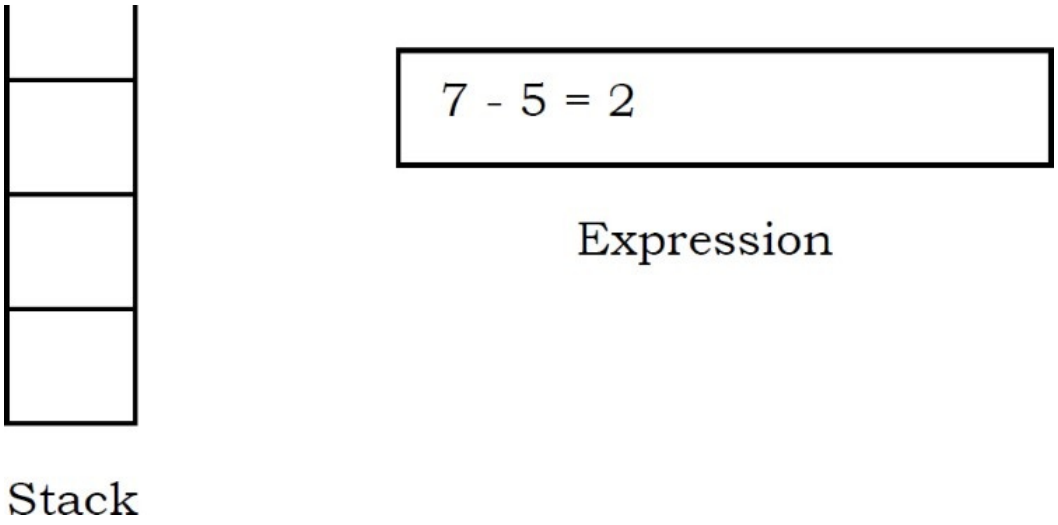
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



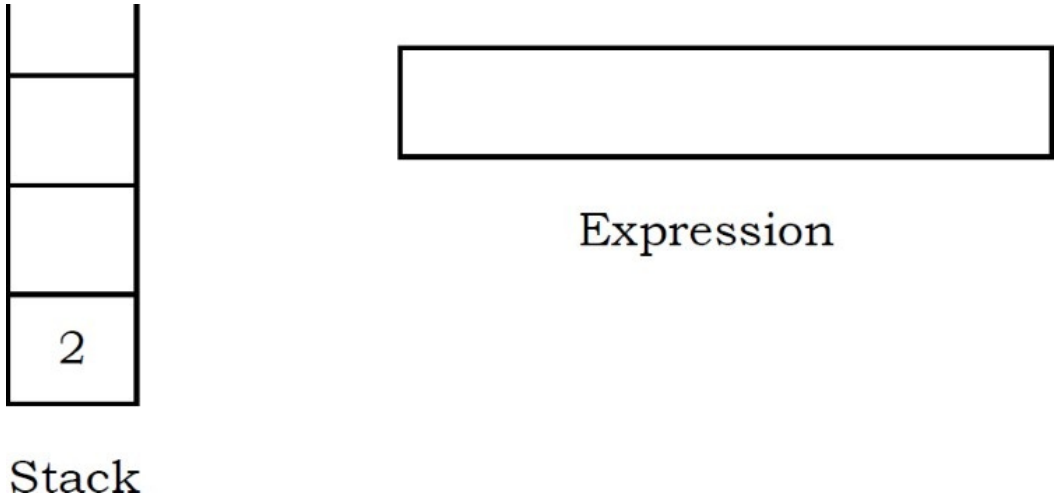
The next character scanned is “5”, which is added to the stack.



The 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 (*n*ext 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 that 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 *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 an example and trace it out. Initially let us assume that we have pushed 2, 6, 4, 1 and 5. Based on the above-mentioned 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 discussion above, 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 Problem-5 is it possible to improve the space complexity?

Solution: Yes. The main problem of the 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 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 equal* 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 the *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.

Solution: This is one of the simplest algorithms. What we do is, start two indexes, one at the beginning of the string and the other at the end of the string. Each time compare whether the values at both the indexes are the same or not. If the values are not the same then we say that the given string is not a palindrome.

If the values are the 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 Problem-8, if the input is in singly linked list then how do we check whether the list elements form a palindrome (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 element's content with top of the stack. If they are the same then pop the stack and go to the next element in the input list.
- If they are not the 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) ≈ O(n).

Problem-11 Given a stack, how to reverse the elements of the stack 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 the 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.

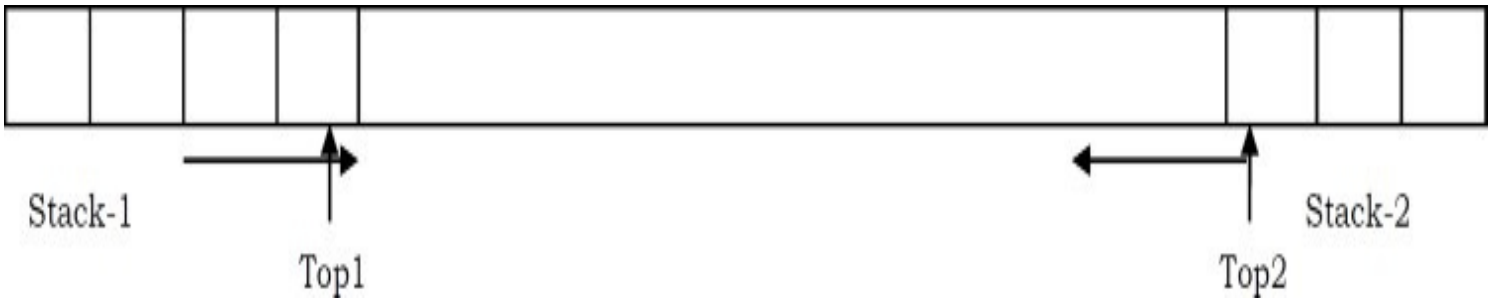
Solution: Refer Queues chapter.

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 *two* stacks using only one array? Our stack routines should not indicate an exception unless every slot in the array is used?

Solution:



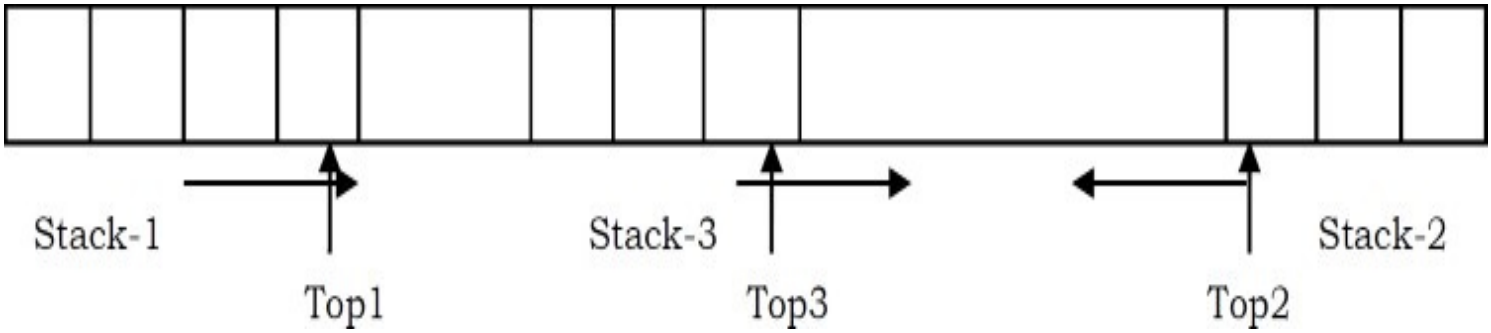
Algorithm:

- Start two indexes one at the left end and the 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 the left index.
- Similarly, if we want to push an element into the second stack then put the element at the right index.
- The first stack grows towards the right, and the second stack grows towards the 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 ways of solving it. Given below is one 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 (Topi): 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 into 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).

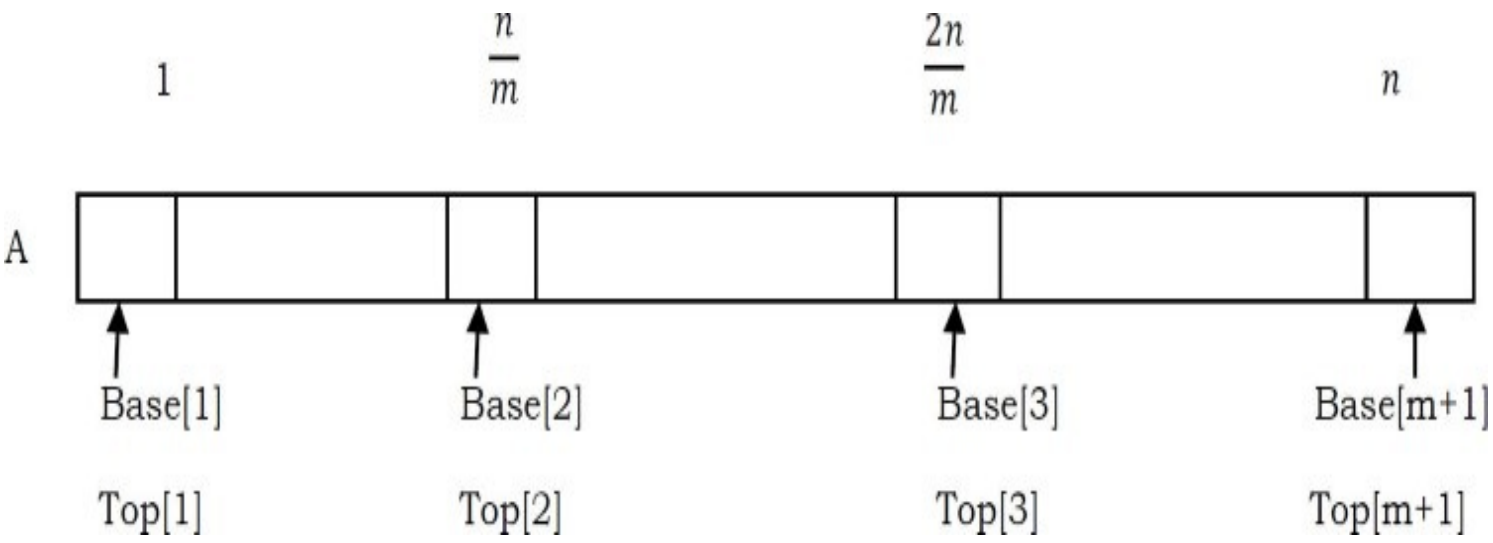
Problem-16 For Problem-15, is there any other way implementing the middle stack?

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 happens if a push on the middle stack causes it to bump into the right stack.

To solve the above-mentioned problem (*n*umber of shifts) what we can do is: alternating pushes can 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 we 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 we'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: 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*. Similar to the discussion in 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[l]), 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 $1 \le i \le m$ 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 $1 \le i \le m$.
- 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 the 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 toward the 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 toward the 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 in the order they appear from left to right. Let 5 indicate a push and *X* indicate a pop operation. Can they be permuted in to the order 325641(output) and order 154623?

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 in this chapter, we discussed that for dynamic array implementation of stacks, the 'repeated doubling' approach is used. For the same problem, what is the complexity if we create a new array whose size is n + if 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 is:

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

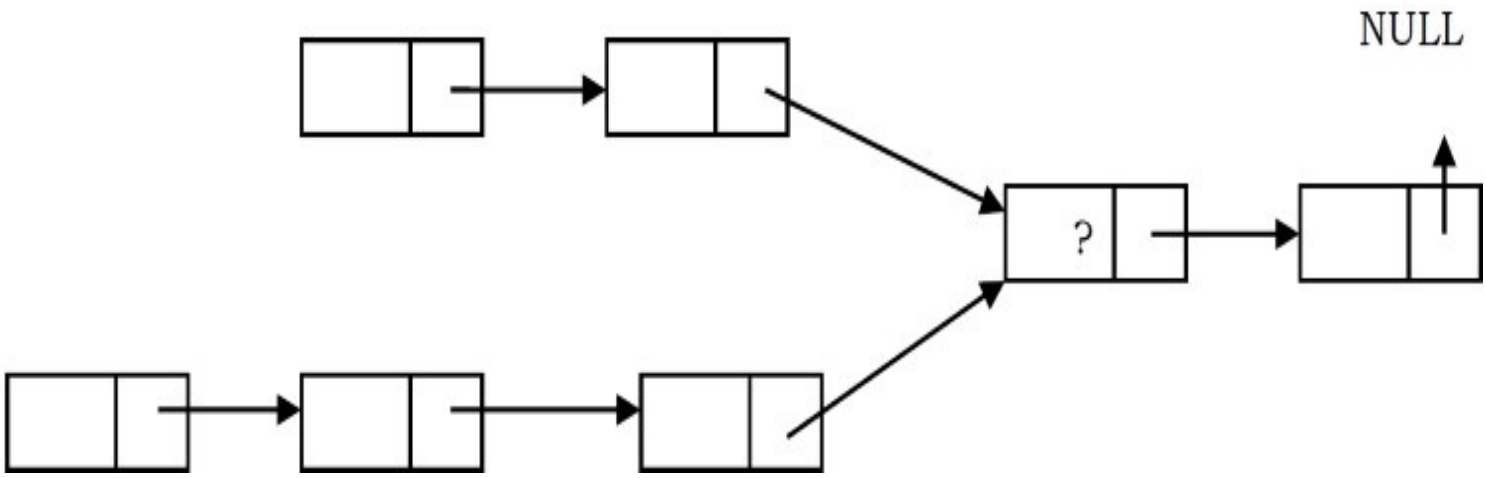
If we are performing *n* push operations, the cost per operation is O(*logn*).

Problem-20 Given a string containing *n S's* and *n X's* where 5 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 5 of operations is admissible or not?

Solution: Given a string of length 2*n*, 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 number of *S's* is always greater than or equal to that of *X*'s. Hence the condition is at any stage of processing of the string, the number of push operations (*S*) should be greater than the number of pop operations (*X*).

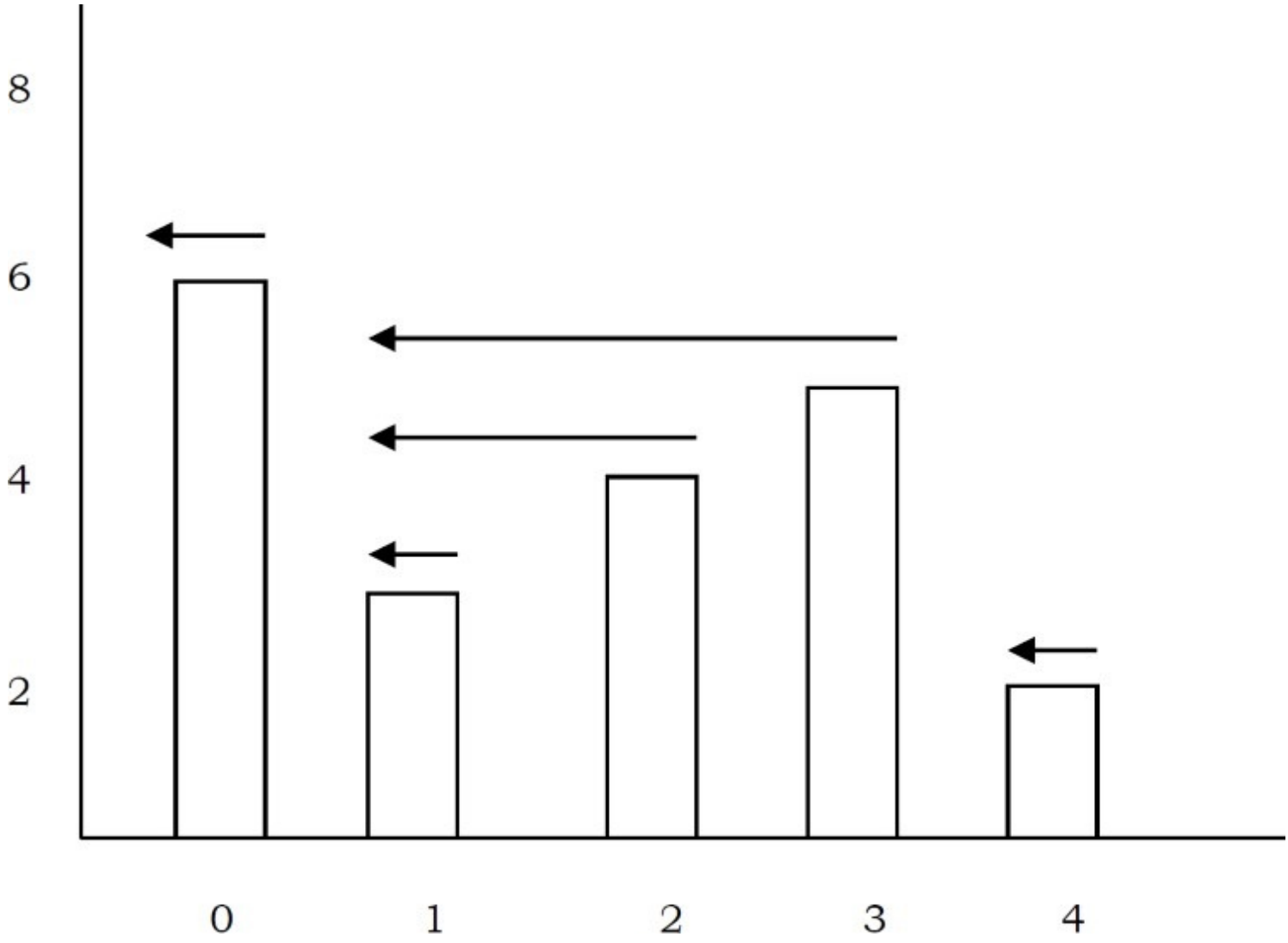
Problem-21 Suppose there are two singly linked lists which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect are unknown and both lists may have a different number. *List*1 may have *n* nodes before it reaches the intersection point and *List*2 may have *m* nodes before it reaches the 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 to *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*] < *A*[*i*]?
Other way of asking: Given an array *A* of integers, find the maximum of $j - i$ subjected to the constraint of *A*[*i*] < *A*[*j*].

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 are used in financial analysis (E.g., stock at 52-week high). The span of a stock 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 table and the corresponding spans diagram. In the figure the arrows indicate 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 have a stock price that is

less than the current price.

```
Algorithm: FindingSpans(int A[], int n) {                    Executes n times
    //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++) {                                      n
        j = 1;                                                     1 + 2 + ...+ (n - 1)
        while (j <= i && A[i] > A[i-j])                            1 + 2 + ...+ (n - 1)
            j = j + 1;                                             n
        S[i] = j;
    }
    return S;                                                      1
}
```

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

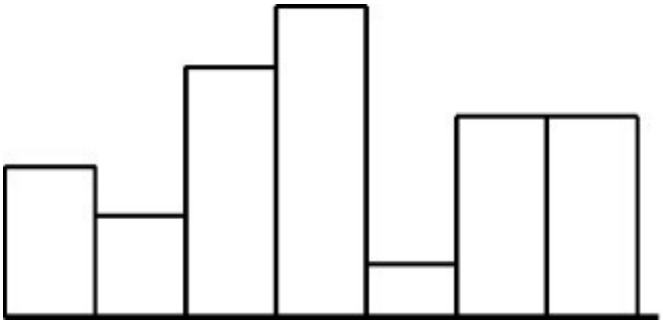
Problem-23 Can we improve the complexity of Problem-22?

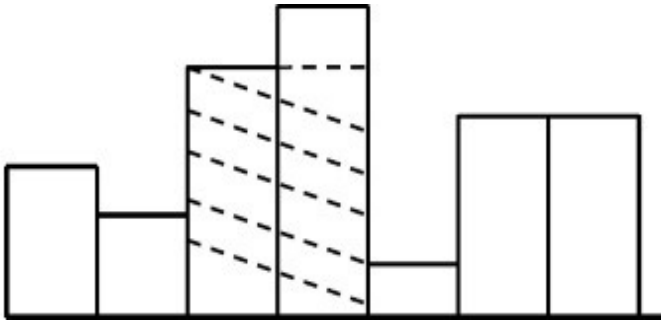
Solution: From the example above, we can see that span $S[i]$ on day i can be easily calculated if we know the closest day preceding i, such that the price is greater on that day than the price on day i. Let us call such a day as P. If such a day exists then the span is now defined as $S[i] = i - P$.

```
Algorithm: FindingSpans(int A[], int n) {
    struct Stack *D = CreateStack();
    int P;
    for (int i = 0 i< n; i++) {
        while (!IsEmptyStack(D) && 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 once 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 the algorithm (trace out an example and see how many times the inner loop becomes successful). 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 have equal widths but may have different heights. For example, the figure on the left shows a 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 rectangle possible. For the given example, the largest rectangle is the shared part.





Solution: A straightforward answer is to go to each bar in the histogram and find the maximum possible area in the 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 sub problems: 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 sub problem 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 sub problem 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 sub problems are finished when 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 sub problems by updating the maximum area with respect 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]→height > A[i])) {
                        if(top > 0)
                                left = S[top-1]→index;
                        else    left = -1;
                        currentArea = (i – left-1) * S[top]→height;
                        --top;
                        if(currentArea > maxArea)
                                maxArea = currentArea;
                }
                if(i<n) {
                        ++top;
                        S[top]→height = A[i];
                        S[top]→index = i;
                }
        }
        return maxArea;
}
```

At the 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].

Problem-26 On a given machine, how do you check whether the stack grows up or down?

Solution: Try noting down the address of a local variable. Call another function with a local variable declared in it and check the address of that local variable and compare.

```
int testStackGrowth() {
    int temporary;
    stackGrowth(&temporary);
    exit(0);
}
void stackGrowth(int *temp){
    int temp2;
    printf("\nAddress of first local valuable: %u", temp);
    printf("\nAddress of second local: %u", &temp2);
    if(temp < &temp2)
        printf("\n Stack is growing downwards");
    else
        printf("\n Stack is growing upwards");
}
```

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

Problem-27 Given a stack of integers, how do you check whether each successive pair of numbers in the stack is consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. For example, if the stack of elements are [4, 5, -2, -3, 11, 10, 5, 6, 20], then the output should be true because each of the pairs (4, 5), (-2, -3), (11, 10), and (5, 6) consists of consecutive numbers.

Solution: Refer to *Queues* chapter.

Problem-28 Recursively remove all adjacent duplicates: Given a string of characters, recursively remove adjacent duplicate characters from string. The output string should not have any adjacent duplicates.

Input: careermonk *Output:* camonk	*Input:* mississippi *Output:* m

Solution: This solution runs with the concept of in-place stack. When element on stack doesn't match the current character, we add it to stack. When it matches to stack top, we skip characters until the element matches the top of stack and remove the element from stack.

```
void removeAdjacentDuplicates(char *str){
    int stkptr=-1;
    int i=0;
    int len=strlen(str);
    while (i<len){
        if (stkptr == -1 || str[stkptr]!=str[i]){
            stkptr++;
            str[stkptr]=str[i];
            i++;
        }else {
            while(i < len&& str[stkptr]==str[i])
                i++;
            stkptr--;
        }
    }
    str[stkptr+1]='\0';
}
```

Time Complexity: O(*n*). Space Complexity: O(1) as the stack simulation is done inplace.

Problem-29 Given an array of elements, replace every element with nearest greater element on the right of that element.

Solution: One simple approach would involve scanning the array elements and for each of the elements, scan the remaining elements and find the nearest greater element.

```
void replaceWithNearestGreaterElement(int A[], int n){
    int nextNearestGreater = INT_MIN;
    int i = 0, j = 0;
    for (i=0; i<n; i++){
        nextNearestGreater = -INT_MIN;
        for (j = i+1; j<n; j++){
            if (A[i] < A[j]){
                nextNearestGreater = A[j];
                break;
            }
        }
        printf("For the element %d, %d is the nearest greater element\n", A[i], nextNearestGreater);
    }
}
```

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

Problem-30 For Problem-29, can we improve the complexity?

Solution: The approach is pretty much similar to Problem-22. Create a stack and push the first element. For the rest of the elements, mark the current element as *nextNearestGreater*. If stack is not empty, then pop an element from stack and compare it with *nextNearestGreater*. If *nextNearestGreater* is greater than the popped element, then *nextNearestGreater* is the next greater element for the popped element. Keep popping from the stack while the popped element is smaller than *nextNearestGreater*. *nextNearestGreater* becomes the next greater element for all such popped elements. If *nextNearestGreater* is smaller than the popped element, then push the popped element back.

```
void replaceWithNearestGreaterElement(int A[], int n){
    int i = 0;
    struct Stack *S = CreateStack();
    int element, nextNearestGreater;
    Push(S, A[0]);
    for (i=1; i<n; i++){
        nextNearestGreater = A[i];
        if (!IsEmptyStack(S)){
        element = Pop(S);
          while (element < nextNearestGreater){
            printf("For the element %d, %d is the nearest greater element\n", A[i], nextNearestGreater);
             if(IsEmptyStack(S))
               break;
             element = Pop(S);
          }
          if (element > nextNearestGreater)
             Push(S, element);
       }
       Push(S, nextNearestGreater);
    }
    while (!IsEmptyStack(S)){
       element = Pop(S);
       nextNearestGreater = -INT_MIN;
       printf("For the element %d, %d is the nearest greater element\n", A[i], nextNearestGreater);
    }
}
```

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

Problem-31 How to implement a stack which will support following operations in O(1) time complexity?

- Push which adds an element to the top of stack.
- Pop which removes an element from top of stack.
- Find Middle which will return middle element of the stack.
- Delete Middle which will delete the middle element.

Solution: We can use a LinkedList data structure with an extra pointer to the middle element.

Also, we need another variable to store whether the LinkedList has an even or odd number of elements.

- *Push:* Add the element to the head of the LinkedList. Update the pointer to the middle element according to variable.
- *Pop:* Remove the head of the LinkedList. Update the pointer to the middle element according to variable.
- *Find Middle:* Find Middle which will return middle element of the stack.
- *Delete Middle:* Delete Middle which will delete the middle element use the logic of Problem-43 from *Linked Lists* chapter.

CHAPTER

5

QUEUES

5.1 What is a Queue?

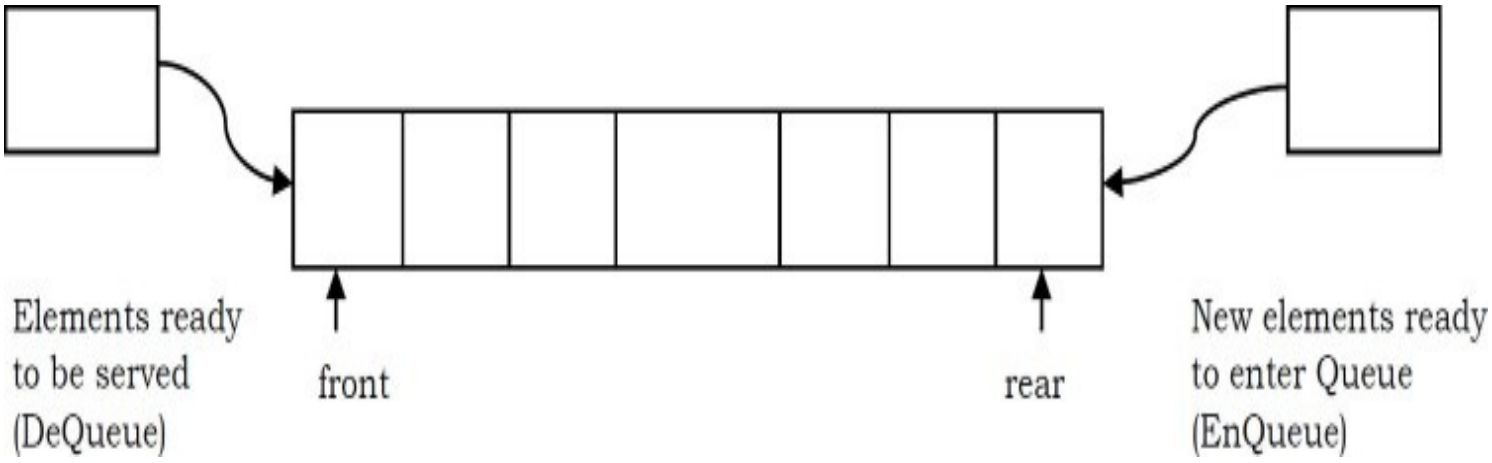
A queue is a data structure used for storing data (similar to Linked Lists and Stacks). In queue, the order in which data arrives is important. In general, a queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

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 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 *EnQueue*, and when an element is removed from the queue, the concept is called *DeQueue*.

DeQueueing an empty queue is called *underflow* and *EnQueuing* an element in a full queue is called *overflow*. Generally, we treat them as exceptions. As an example, consider the snapshot of

the queue.



5.2 How are Queues Used?

The concept of a queue can be explained by observing a line at a reservation counter. When we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

As this happens, the next person will come at the head of the line, will exit the queue and will be served. As each person at the head of the line keeps exiting the queue, we move towards the head of the line. Finally we will reach the head of the line and we will exit the queue and be served. This behavior is very useful in cases where there is a need to maintain the order of arrival.

5.3 Queue ADT

The following operations make a queue an ADT. Insertions and deletions in the 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 in the queue
- int IsEmptyQueueQ: Indicates whether no elements are stored in the queue or not

5.4 Exceptions

Similar to other ADTs, executing *DeQueue* on an empty queue throws an "*Empty Queue Exception*" and executing *EnQueue* on a full queue throws "*Full Queue Exception*".

5.5 Applications

Following are some of the applications that use queues.

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 some of the commonly used methods are listed below.

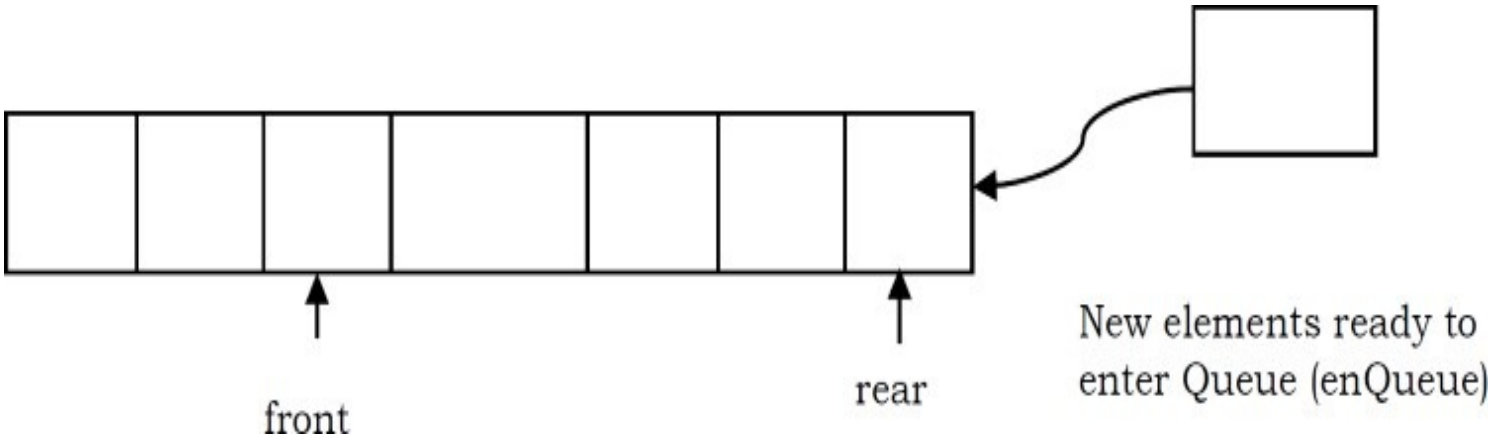
- Simple circular array based implementation
- Dynamic circular array based implementation
- Linked list implementation

Why Circular Arrays?

First, let us see whether we can use simple arrays for implementing queues as we have done for stacks. We know that, in queues, the insertions are performed at one end and deletions are performed at the other end. After performing some insertions and deletions the process becomes easy to understand.

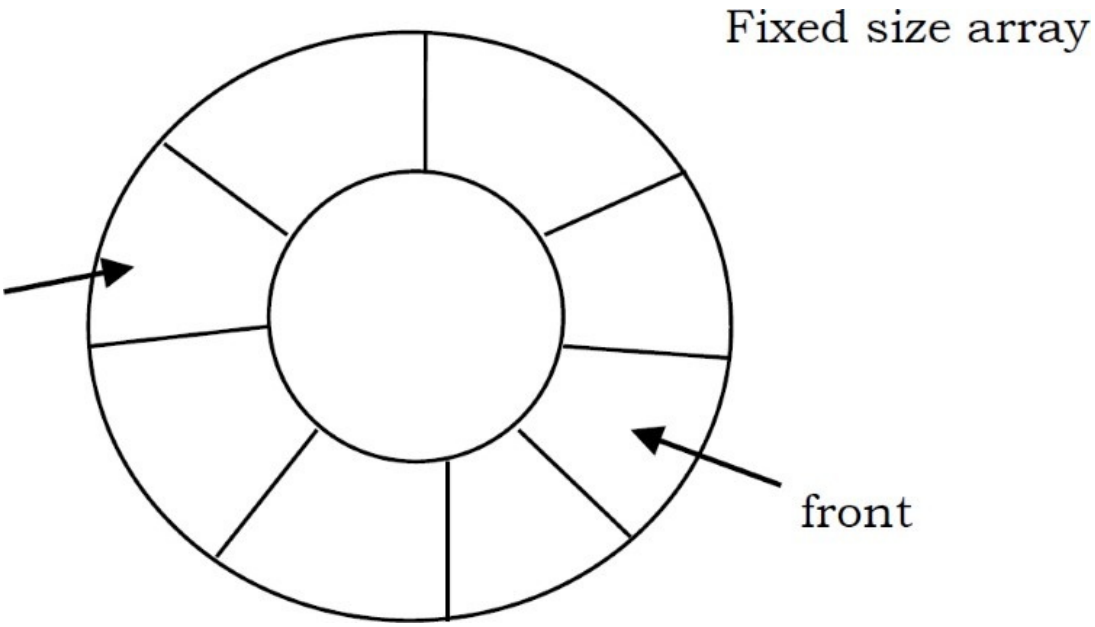
In the example shown below, it can be seen clearly that the initial slots of the array are getting wasted. So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat the last element and the first array

elements as 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 similar to stack array implementations. Refer to *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 the 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 an empty queue 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 and 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 as 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);
    }
}
```

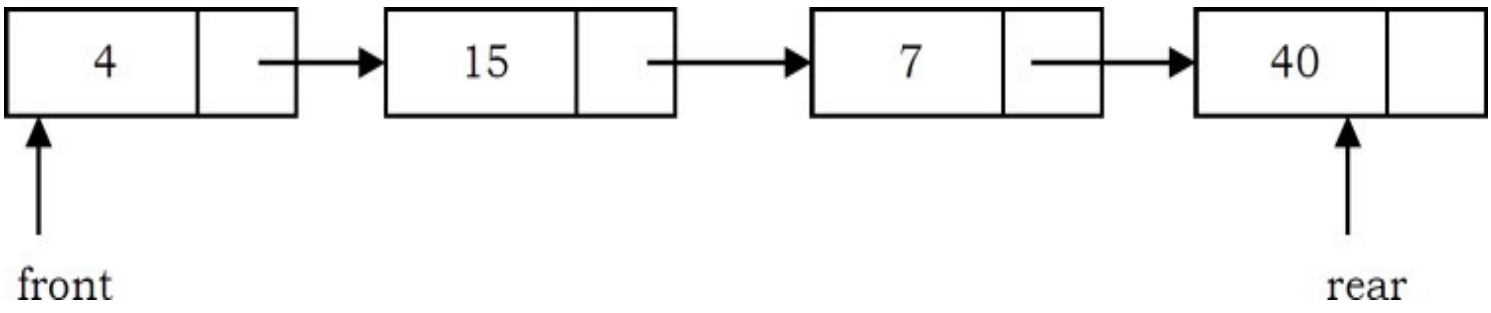
Performance

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

Another way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting an element at the end 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;
    if(Q→rear) 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 similar to stack implementations and *Stacks* chapter.

5.7 Queues: Problems & Solutions

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 can you implement a queue using two stacks?

Solution: Let SI 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 SI to S2 and pop the top element from S2 and return that popped element [we can optimize the code a little by transferring only $n - 1$ elements from SI to S2 and pop the n^{th} element from SI 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 SI 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).The amortized complexity of pop operation is O(1).

Problem-3 Show how you can efficiently implement one stack using two queues. Analyze the

running time of the stack operations.

Solution: Yes, it is possible to implement the Stack ADT using 2 implementations of the Queue ADT. One of the queues will be used to store the elements and the other to hold them temporarily during the *pop* and *top* methods. The *push* method would *enqueue* the given element onto the storage queue. The *top* method would transfer all but the last element from the storage queue onto the temporary queue, save the front element of the storage queue to be returned, transfer the last element to the temporary queue, then transfer all elements back to the storage queue. The *pop* method would do the same as top, except instead of transferring the last element onto the temporary queue after saving it for return, that last element would be discarded. 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 the algorithms below, we make sure that one queue is always empty.

Push Operation Algorithm: Insert the element in whichever queue is not empty.

- 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 the 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 the 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 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 supports insertion and deletion at both ends). Refer *Priority Queues* chapter for algorithms.

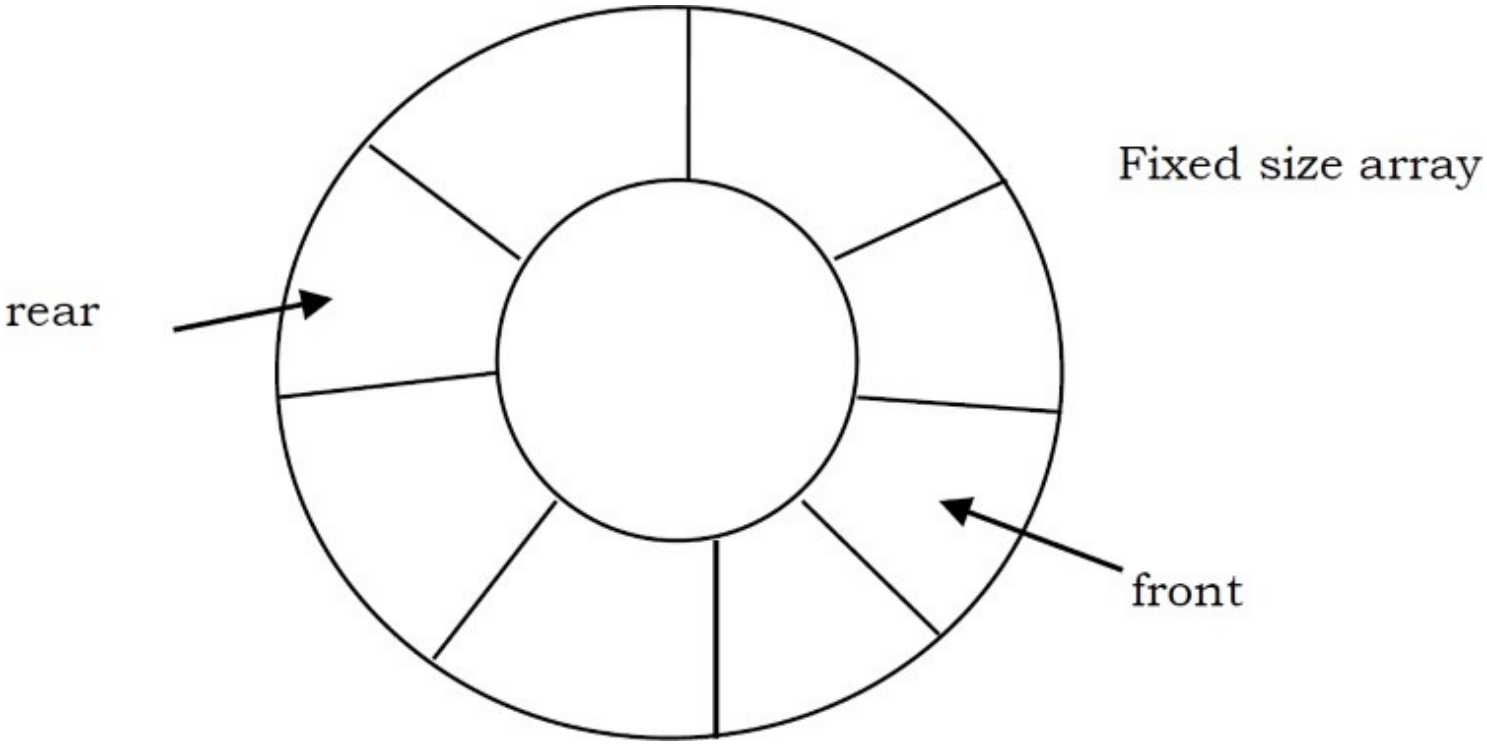
Problem-5 Given a queue Q containing *n* elements, transfer these items on to a stack S (initially empty) so that front element of Q appears at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue, and push and pop operations for the stack, 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 $a_1.a_2$ ...a_n. Dequeuing all elements and pushing them onto the stack will result in a stack with a_n at the top and a_1 at the bottom. This is done in O(*n*) time as dequeue and each push require constant time per operation. The queue is now empty. By popping all elements and pushing them on 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[O..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 figure to get a clear idea of 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:

$$Number\ Of\ Elements = \begin{cases} rear - front + 1 & \text{if } rear == front \\ rear - front + n & \text{otherwise} \end{cases}$$

Problem-7 What is the most appropriate data structure to print elements of queue in reverse order?

Solution: Stack.

Problem-8 Implement doubly ended queues. A double-ended queue is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a head-tail linked list.

Solution:

```
void pushBackDEQ(struct ListNode **head, int data){
    struct ListNode *newNode = (struct ListNode*) malloc(sizeof(struct ListNode));
    newNode→data = data;
    if(*head == NULL){
        *head = newNode;
        (*head)→next = *head;
        (*head)→prev = *head;
    }
    else{
        newNode→prev = (*head)→prev;
        newNode→next = *head;
        (*head)→prev→next = newNode;
        (*head)→prev = newNode;
    }
}
void pushFrontDEQ(struct ListNode **head, int data){
    pushBackDEQ(head,data);
    *head = (*head)→prev;
}
int popBackDEQ(struct ListNode **head){
    int data;
    if( (*head)→prev == *head ){
        data = (*head)→data;
        free(*head);
        *head = NULL;
    }
    else{
        struct ListNode *newTail = (*head)→prev→prev;
        data = (*head)→prev→data;
        newTail→next = *head;
        free((*head)→prev);
        (*head) →prev = newTail;
    }
    return data;
}
int popFront(struct ListNode **head){
    int data;
    *head = (*head)→next;
    data = popBackDEQ(head);
    return data;
}
```

Problem-9 Given a stack of integers, how do you check whether each successive pair of numbers in the stack is consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. For example, if the stack of elements are [4, 5, -2, -3, 11, 10, 5, 6, 20], then the output should be true because each of the pairs (4, 5), (-2, -3), (11, 10), and (5, 6) consists of consecutive numbers.

Solution:

```
int checkStackPairwiseOrder(struct Stack *s) {
    struct Queue *q = CreateQueue();
    int pairwiseOrdered = 1;
    while (!isEmptyStack(s))
        EnQueue(q, Pop(s));
    while (!IsEmptyQueue(q))
        Push(s, DeQueue(q));
    while (!isEmptyStack(s)) {
        int n = Pop(s);
        EnQueue(q, n);
        if (!isEmptyStack(s)) {
            int m = Pop(s);
            EnQueue(q, m);
            if (abs(n - m) != 1) {
                pairwiseOrdered = 0;
            }
        }
    }
    while (!IsEmptyQueue(q))
        Push(s, DeQueue(q));
    return pairwiseOrdered;
}
```

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

Problem-10 Given a queue of integers, rearrange the elements by interleaving the first half of the list with the second half of the list. For example, suppose a queue stores the following sequence of values: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Consider the two halves of this list: first half: [11, 12, 13, 14, 15] second half: [16, 17, 18, 19, 20]. These are

combined in an alternating fashion to form a sequence of interleave pairs: the first values from each half (11 and 16), then the second values from each half (12 and 17), then the third values from each half (13 and 18), and so on. In each pair, the value from the first half appears before the value from the second half. Thus, after the call, the queue stores the following values: [11, 16, 12, 17, 13, 18, 14, 19, 15, 20].

Solution:

```
void interLeavingQueue(struct Queue *q) {
    if (Size(q) % 2 != 0)
        return;
    struct Stack *s = CreateStack();
    int halfSize = Size(q) / 2;
    for (int i = 0; i < halfSize; i++)
        Push(s, DeQueue(q));
    while (!isEmptyStack(s))
        EnQueue(q, Pop(s));
    for (int i = 0; i < halfSize; i++)
        EnQueue(q, DeQueue(q));
    for (int i = 0; i < halfSize; i++)
        Push(s, DeQueue(q));
    while (!isEmptyStack(s)) {
        EnQueue(q, Pop(s));
        EnQueue(q, DeQueue(q));
    }
}
```

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

Problem-11 Given an integer *k* and a queue of integers, how do you reverse the order of the first *k* elements of the queue, leaving the other elements in the same relative order? For example, if *k*=4 and queue has the elements [10, 20, 30, 40, 50, 60, 70, 80, 90]; the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90].

Solution:

```
void reverseQueueFirstKElements(int k, struct Queue *q) {
    if (q == Null || k > Size(q)) {
        return;
    }
    else if (k > 0) {
        struct Stack *s = CreateStack();
        for (int i = 0; i < k; i++) {
            Push(s, DeQueue(q));
        }
        while (!isEmptyStack(s)) {
            EnQueue(q, Pop(s));
        }
        for (int i = 0; i < Size(q) - k; i++) { // wrap around rest of elements
            EnQueue(q, DeQueue(q));
        }
    }
}
```

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

CHAPTER

6

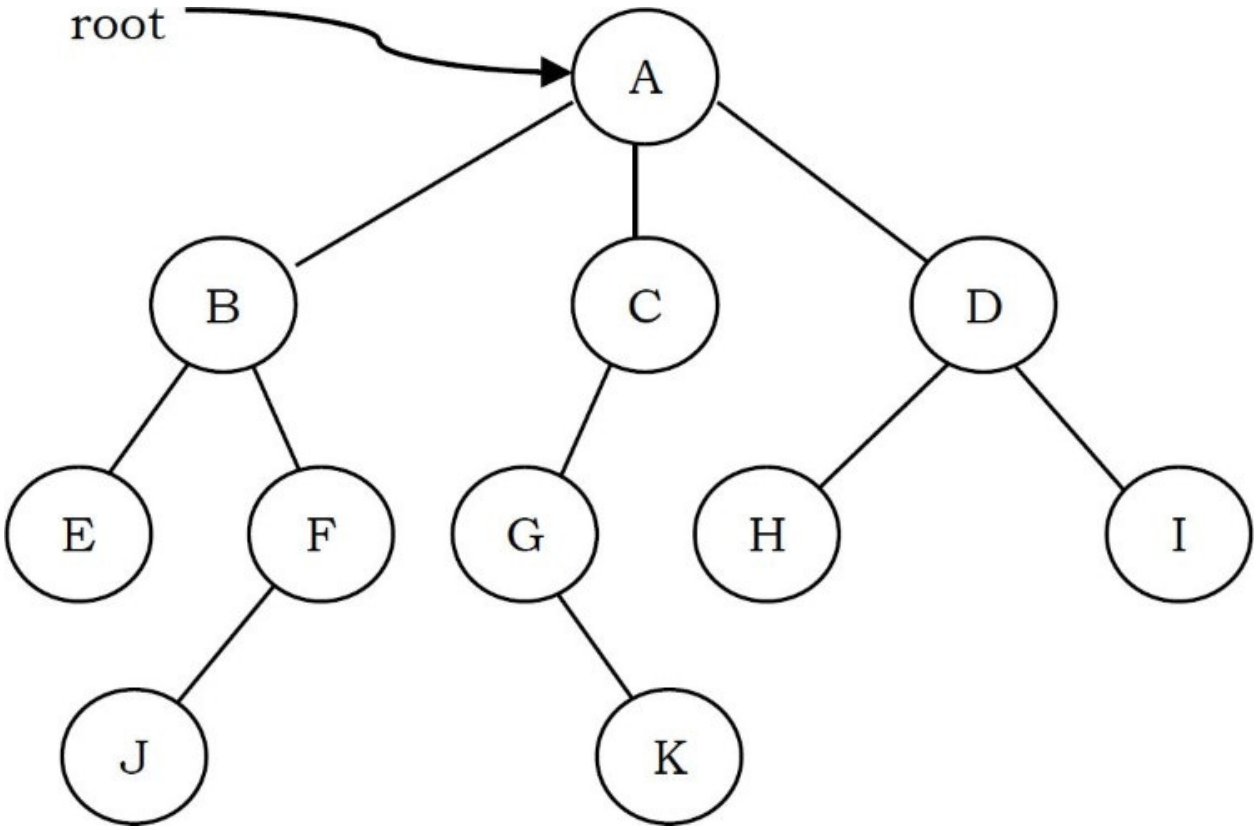
TREES

6.1 What is a Tree?

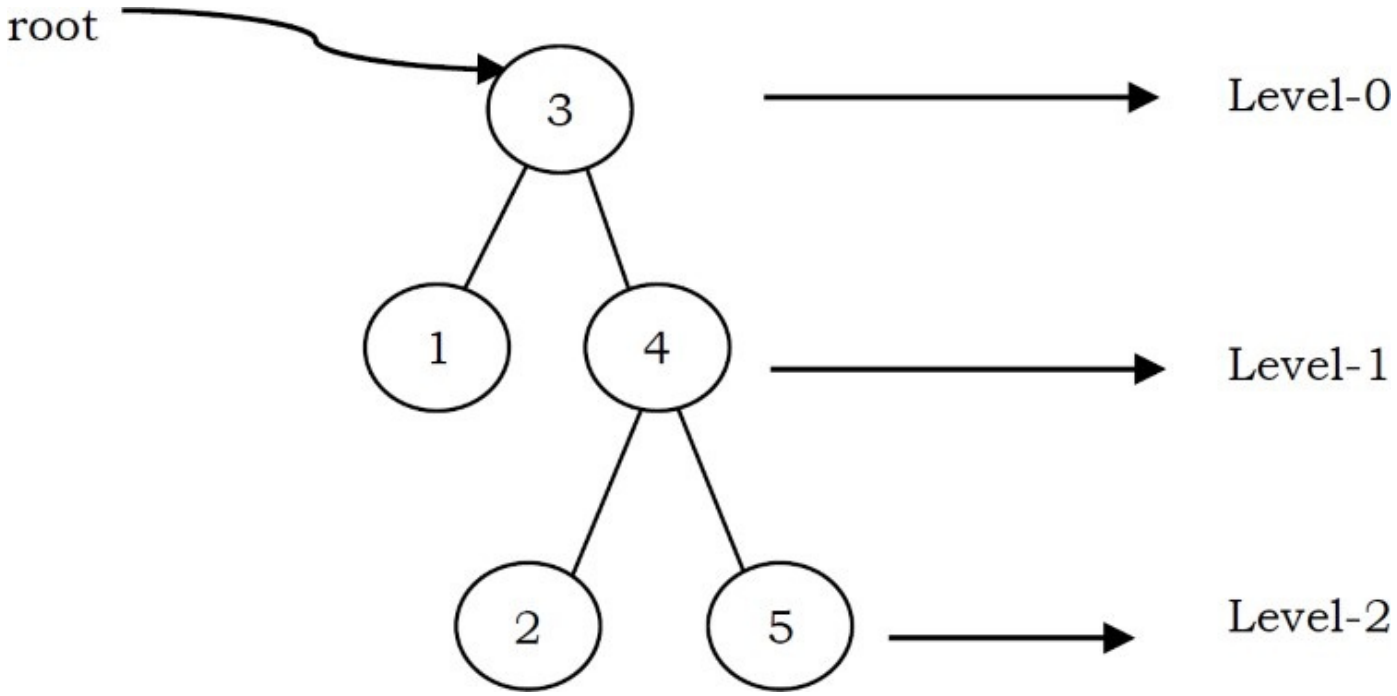
A *tree* is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes. Tree is an example of a non-linear data structure. A *tree* structure is a way of representing the hierarchical nature of a structure in a graphical form.

In trees ADT (Abstract Data Type), the order of the elements is not important. If we need ordering information, linear data structures like linked lists, stacks, queues, etc. can be used.

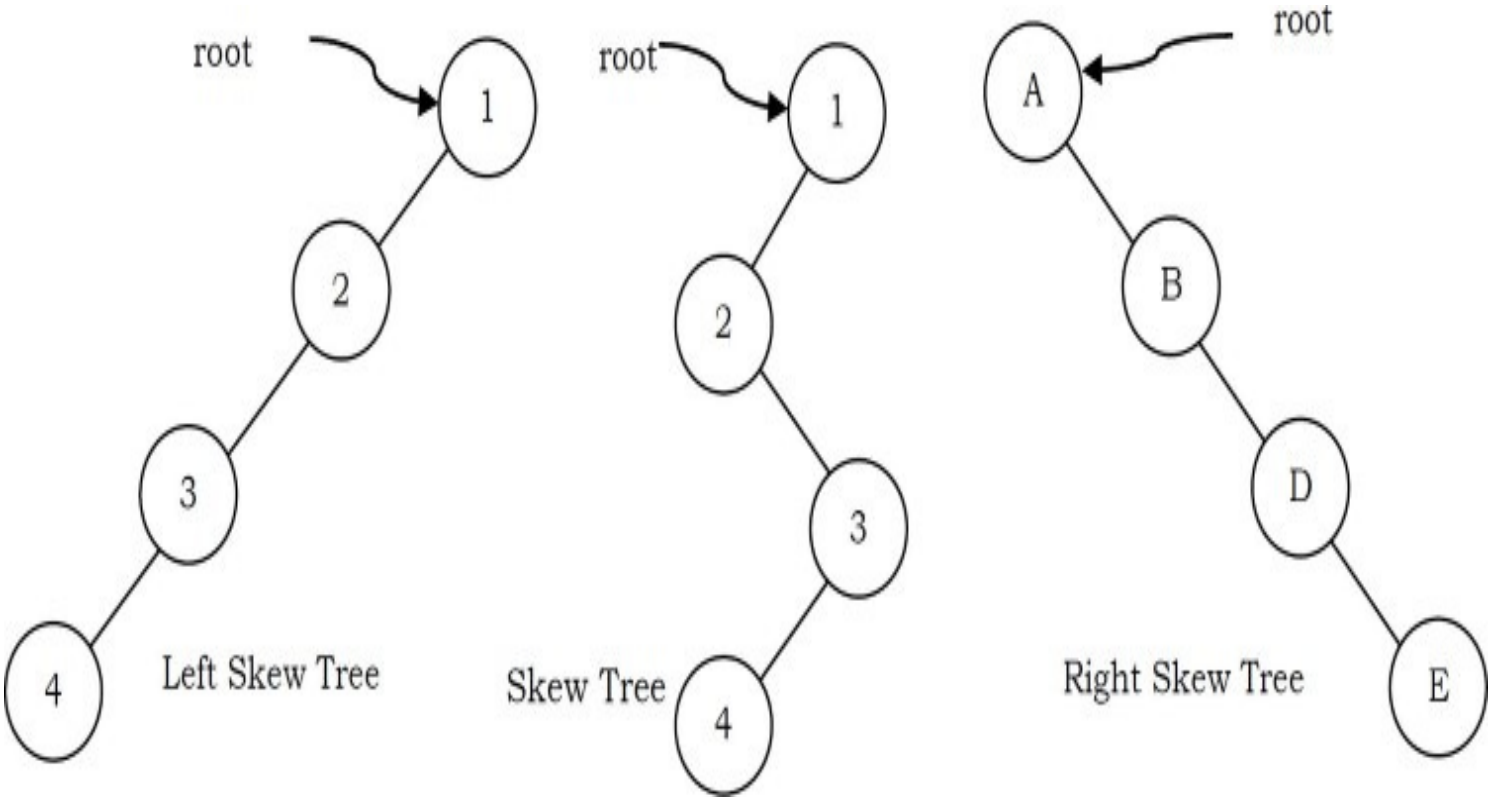
6.2 Glossary



- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (*n*ode *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 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 of if.
- The set of all nodes at a given depth is called the *level* of the tree (*B, C* and *D* are the same level). The root node is at level zero.



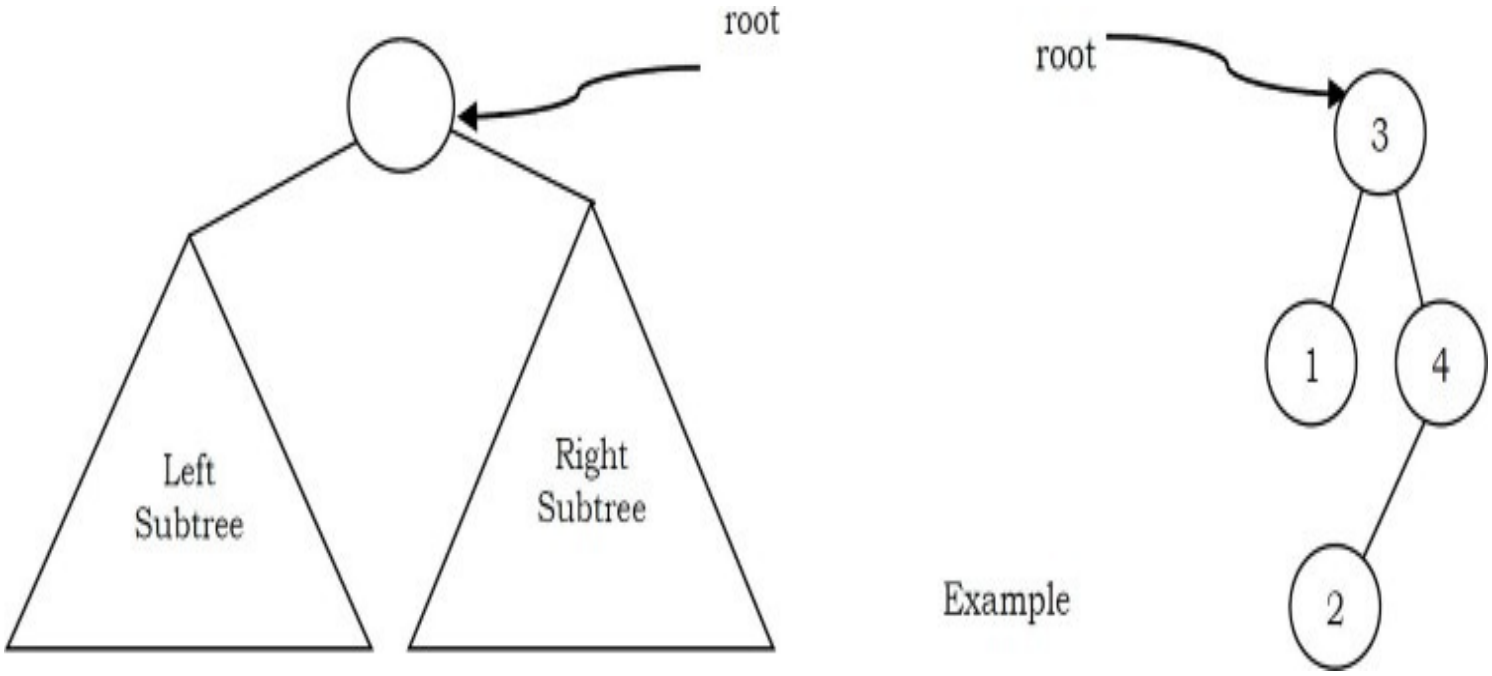
- The *depth* of a node is the length of the path from the root to the node (depth of *G* is 2, *A* – *C* – *G*).
- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the 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.
- The size of a node is the number of descendants it has including itself (the 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 *skew trees*. If every node has only left child then we call them *left skew trees*. Similarly, if every node has only right child then we call them *right skew trees*.



6.3 Binary Trees

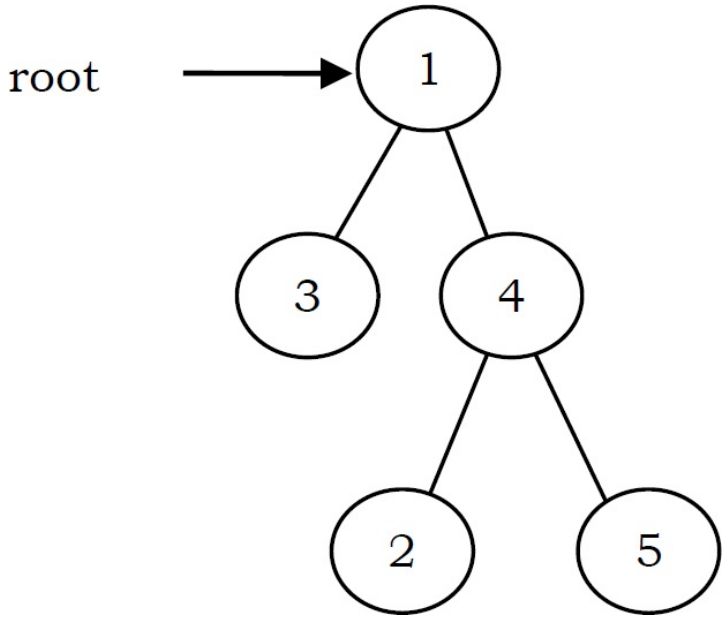
A tree is called *binary tree* if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

Generic Binary Tree

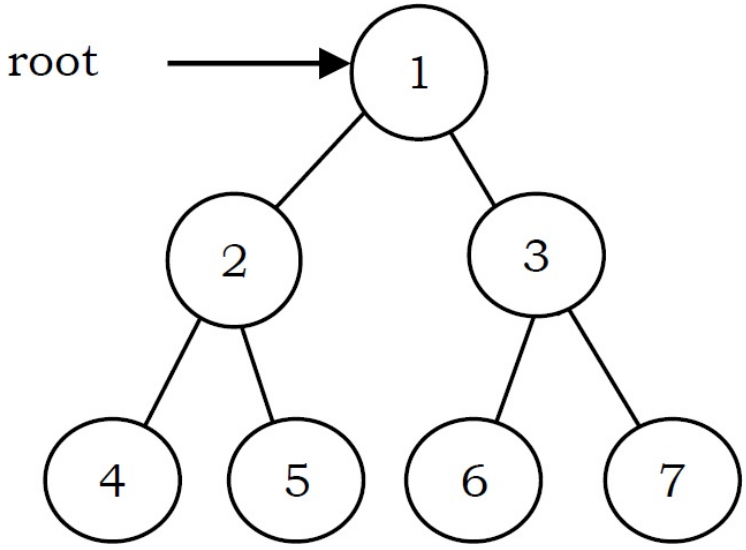


6.4 Types of Binary Trees

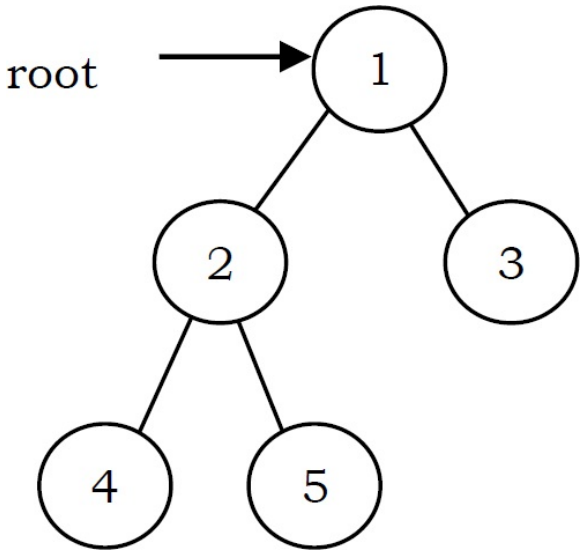
Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



Full Binary Tree: A binary tree is called *full binary tree* if each node has exactly two children and all leaf nodes are at the same level.



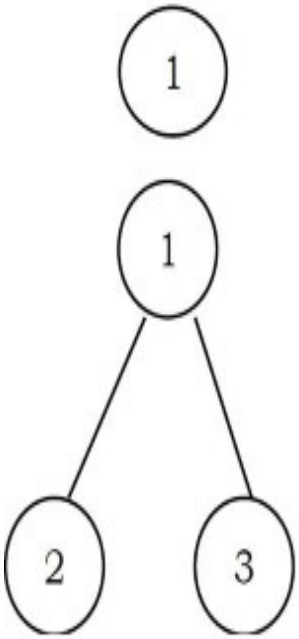
Complete Binary Tree: Before defining the *complete binary tree*, let us assume that the height of the binary tree is *h*. In complete binary trees, if we give numbering for the nodes by starting at the root (let us say the root node has 1) then we get a complete sequence from 1 to the number of nodes 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.

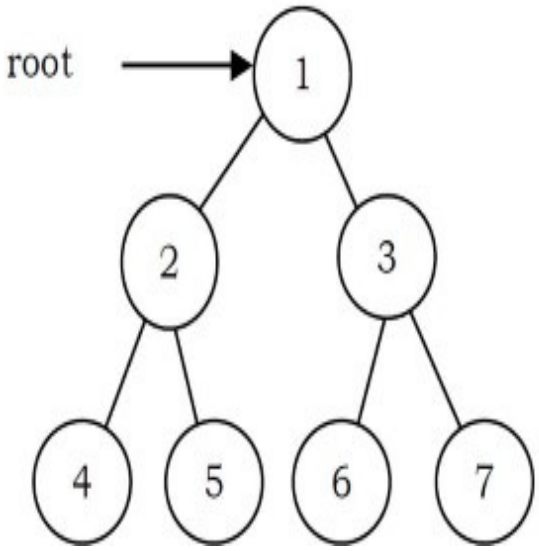


6.5 Properties of Binary Trees

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
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$
$h = 2$	$2^2 = 4$



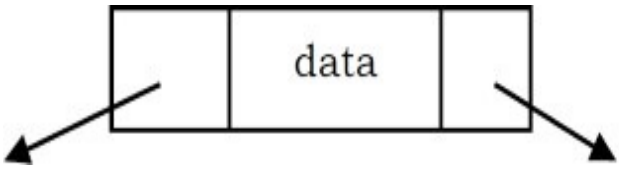


From the diagram we can infer the following properties:

- 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 to *Priority Queues* chapter.
- The number of leaf nodes in a full binary tree is 2^h.
- The number of NULL links (wasted pointers) in a complete binary tree of n nodes is $n + 1$.

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 data) is to have two links which point 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 it is not mandatory to show directed branches. For our discussion, we assume both the representations shown below are the same.

Operations on Binary Trees

Basic Operations

- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree
- Finding the level which has maximum sum
- Finding the 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 an important role:

- Expression trees are used in compilers.
- Huffman coding trees that 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 Queue (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 node is processed only once but it may be visited more than once. As we have already seen 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 a 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 denoted with "*D*"), traversing to the left child node (denoted with "*L*"), and traversing to the right child node (denoted 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 (*n*odes) are processed defines a particular traversal method. The classification is 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* is on left side of *D* or *R* is on left side of *D*.

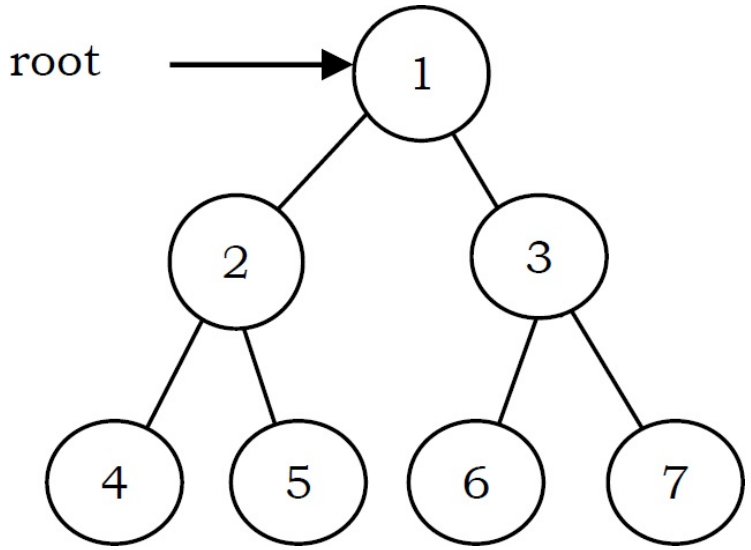
Similarly, it does not matter whether *L* is on right side of *D* or *R* is on right side of *D*. Due to this, the total 6 possibilities are reduced to 3 and these are:

- Preorder (*DLR*) Traversal
- Inorder (*LDR*) Traversal
- Postorder (*LRD*) Traversal

There is another traversal method which does not depend on the above orders and it is:

- Level Order Traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

Let us use the diagram below for the remaining discussion.



PreOrder Traversal

In preorder traversal, each node is processed before (pre) either of its subtrees. 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, 1 is processed first, then the left subtree, and this is followed by the right subtree.

Therefore, processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing the 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 the 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 the right subtree. To simulate the same, first we process the current node and before going to the 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

The Non-recursive version of Inorder traversal is similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which is indicated 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 the 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 popping 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 the left subtree we will visit the current node and after processing the right subtree we will visit 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 the left subtree or the right subtree.

We use a *previous* variable to keep track of the earlier traversed node. Let's assume *current* is the current node that is on top of the stack. When *previous* is *current's* parent, we are traversing down the tree. In this case, we try to traverse to *current's* left child if available (i.e., push left child to the stack). If it is not available, we look at *current's* right child. If both left and right child do not exist (ie, *current* is a leaf node), we print *current's* value and pop it off the stack.

If prev is *current's* left child, we are traversing up the tree from the left. We look at *current's* right child. If it is available, then traverse down the right child (i.e., push right child to the stack); otherwise print *current's* value and pop it off the stack. If *previous* is *current's* right child, we are traversing up the tree from the right. In this case, we print *current's* value and pop it off the stack.

```
void PostOrderNonRecursive(struct BinaryTreeNode *root) {
    struct SimpleArrayStack *S = CreateStack();
    struct BinaryTreeNode *previous = NULL;
    do{
        while (root!=NULL){
            Push(S, root);
            root = root->left;
        }
        while(root == NULL && !IsEmptyStack(S)){
            root = Top(S);
            if(root->right == NULL || root->right == previous){
                printf("%d ", root->data);
                Pop(S);
                previous = root;
                root = NULL;
            }
            else
                root = root->right;
        }
    }while(!IsEmptyStack(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 (, keep all the elements at level (+ 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 the tree are 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.

Binary Trees: Problems & Solutions

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 the 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 the maximum element in binary tree without recursion.

Solution: Using level order traversal: just observe the element's 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 data is found in the tree. Recurse down the tree, choose the left or right branch by comparing data with each node's 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 find 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*).

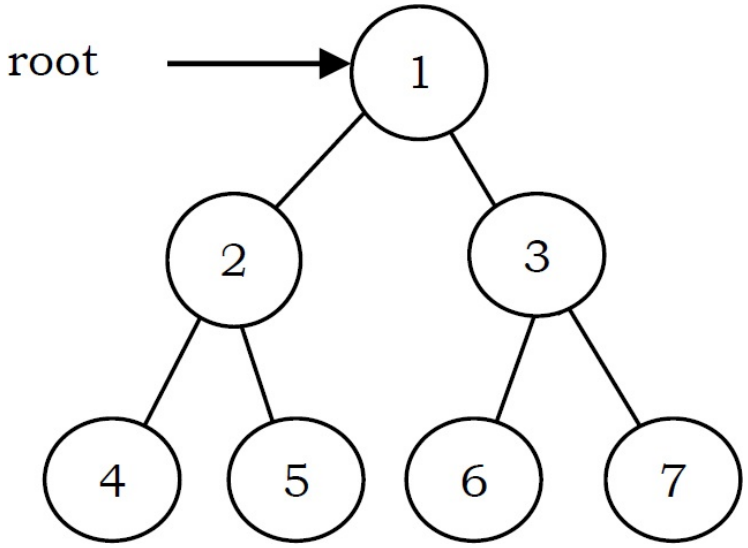
Problem-7 Can we solve Problem-6 without recursion?

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 reverse order. For example, the output for the below tree should be: 4 5 6 7 2 3 1



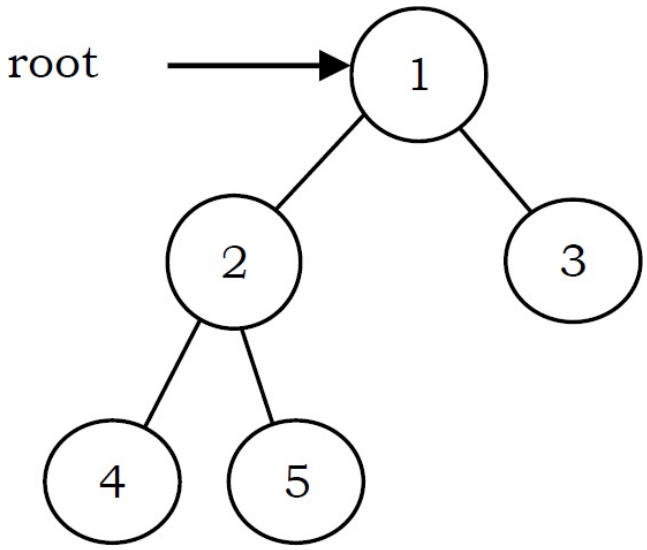
Solution:

```
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 should we 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 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 = 0;
    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 (assuming data is given) from binary tree.

Solution: The deletion of a node in binary tree can be implemented as

- Starting at root, find the node which we want to delete.
- Find the deepest node in the tree.
- Replace the deepest node's 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 nodes whose both left and right children are NULL are called leaf nodes.

```
int 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.

```
int 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 (*n*odes with only one child) in the binary tree without using recursion.

Solution: The set of all nodes with either left or right child (but not both) are called half nodes.

```
int 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 subtrees recursively. Among these two values, we need to send maximum value 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;
}

//Alternative Coding
static int diameter(struct BinaryTreeNode *root) {
    if (root == NULL)
        return 0;
    int lHeight = height(root->eft);
    int rHeight = height(root-right);
    int lDiameter = diameter(root-left);
    int rDiameter = diameter(root-right);
    return max(lHeight + rHeight + 1, max(lDiameter, rDiameter));

}
/* The function Compute the "height" of a tree. Height is the number of nodes along
the longest path from the root node down to the farthest leaf node.*/
static int  height(Node root) {
    if (root == null)
        return 0;
    return 1 + max(height(root.left), height(root.right));
}
```

There is another solution and the complexity is O(*n*). The main idea of this approach is that the node stores its left child's and right child's maximum diameter if the node's child is the "root", therefore, there is no need to recursively call the height method. The drawback is we need to add two extra variables in the node structure.

```
int findMaxLen(Node root) {
    int nMaxLen = 0;
    if (root == null)
        return 0;
    if (root.left == null)
        root.nMaxLeft = 0;
    if (root.right == null)
        root.nMaxRight = 0;
    if (root.left != null)
        findMaxLen(root.left);
    if (root.right != null)
        findMaxLen(root.right);
    if (root.left != null) {
        int nTempMaxLen = 0;
        nTempMaxLen = (root.left.nMaxLeft > root.left.nMaxRight) ?
            root.left.nMaxLeft : root.left.nMaxRight;
        root.nMaxLeft = nTempMaxLen + 1;
    }
    if (root.right != null) {
        int nTempMaxLen = 0;
        nTempMaxLen = (root.right.nMaxLeft > root.right.nMaxRight) ?
            root.right.nMaxLeft : root.right.nMaxRight;
        root.nMaxRight = nTempMaxLen + 1;
    }
    if (root.nMaxLeft + root.nMaxRight > nMaxLen)
        nMaxLen = root.nMaxLeft + root.nMaxRight;
    return nMaxLen;
}
```

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

Problem-19 Give an algorithm for finding the level that has the maximum sum in the binary tree.

Solution: The logic is very much similar to finding the number of levels. The only change is, we

need to keep track of the 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 its root-to-leaf paths.

Solution: Refer to 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.

```
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]);
}
        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*).

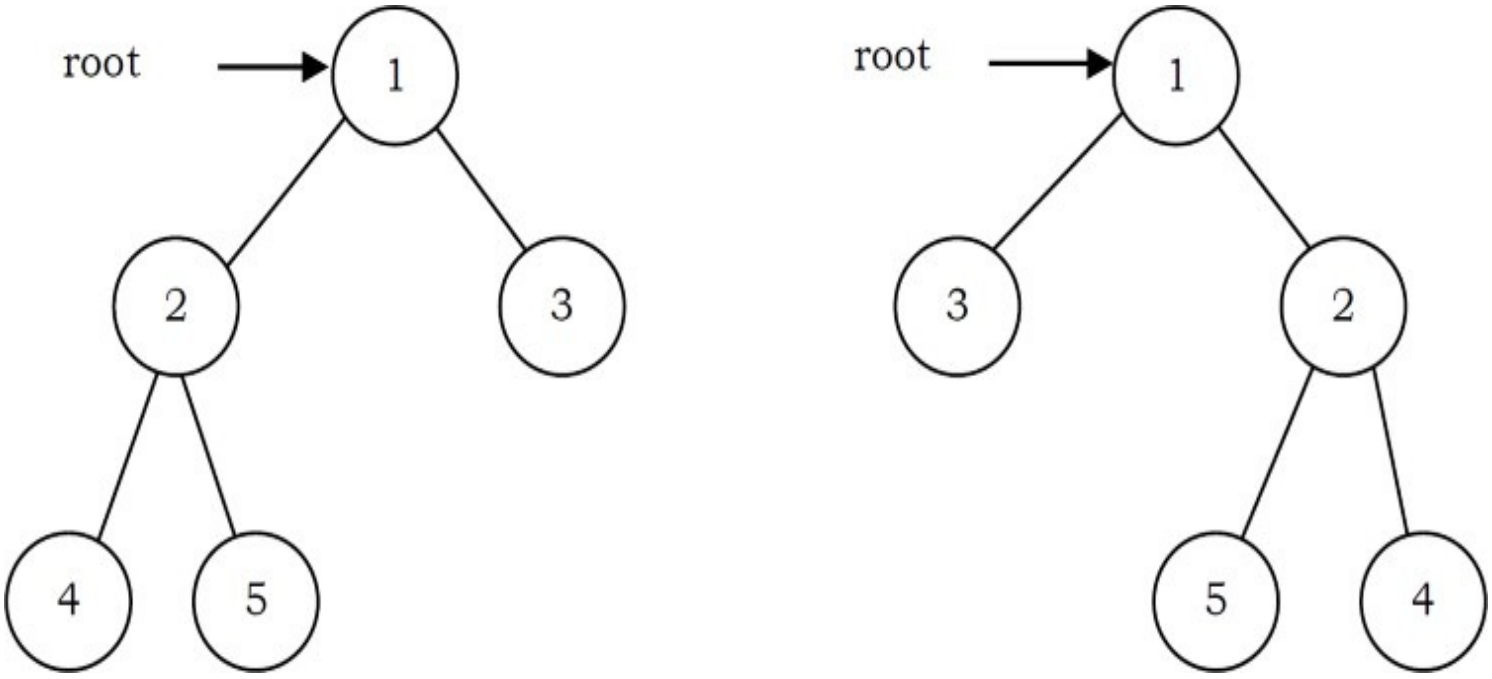
Problem-23 Can we solve Problem-22 without recursion?

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. The trees below are mirrors to each other.



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:

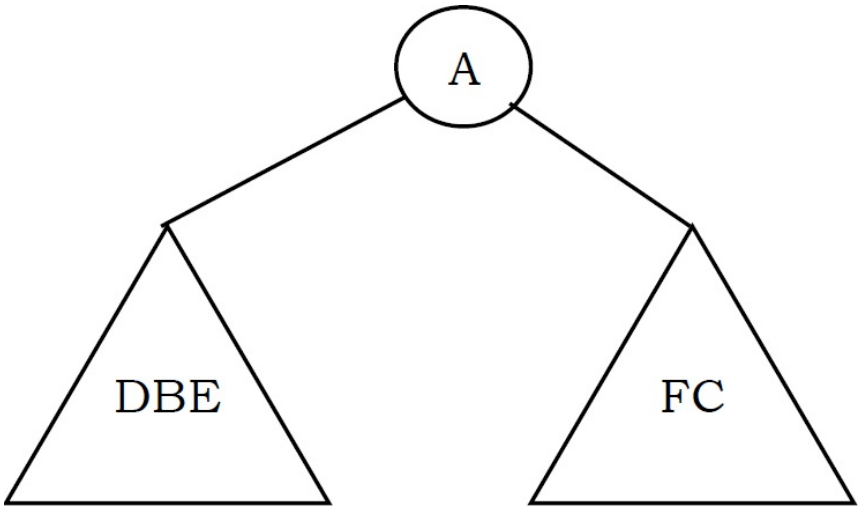
```
struct BinaryTreeNode *LCA(struct BinaryTreeNode *root, struct BinaryTreeNode *α,
                           struct BinaryTreeNode *β){
    struct BinaryTreeNode *left, *right;
    if(root == NULL)
        return root;
    if(root == α || root == β)
        return root;
    left = LCA (root→left, 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.

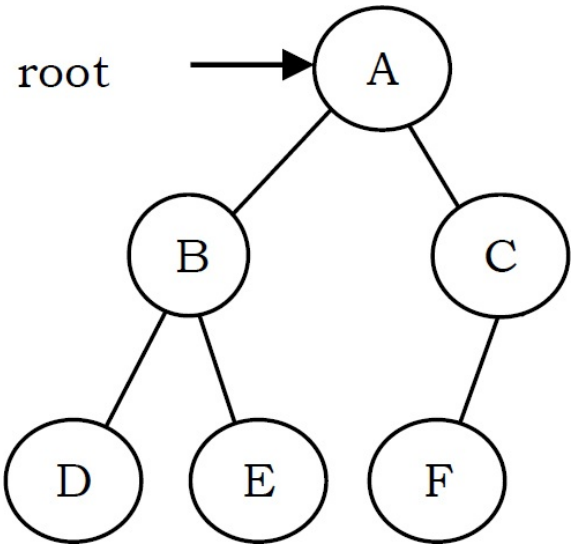
Solution: Let us consider the traversals below:

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 the root for given sequences. By searching '*A*' in Inorder sequence we can find out all elements on the left side of '*A*', which come under the left subtree, and elements on the right side of '*A*', which come under the right subtree. So we get the structure as seen below.

We recursively follow the above steps and get the following tree.



Algorithm: BuildTree()

1 Select an element from *Preorder*. Increment a *Preorder* index variable (*preOrderIndex* in code below) to pick next element in next recursive call.
2 Create a new tree node (*newNode*) from heap with the data as selected element.
3 Find the selected element's index in Inorder. Let the index be *inOrderIndex*.
4 Call BuildBinaryTree for elements before *inOrderIndex* and make the built tree as left subtree of *newNode*.
5 Call BuildBinaryTree for elements after *inOrderIndex* and make the built tree as right subtree of *newNode*.
6 return *newNode*.

```
struct BinaryTreeNode *BuildBinaryTree(int inOrder[], int preOrder[], int inOrderStart, int inOrderEnd){
    static int preOrderIndex = 0;
    struct BinaryTreeNode *newNode;
    if(inOrderStart > inOrderEnd)
        return NULL;
    newNode = (struct BinaryTreeNode *) malloc (sizeof(struct BinaryTreeNode));
    if(!newNode) {
        printf("Memory Error");
        return NULL;
    }
    // Select current node from Preorder traversal using preOrderIndex
    newNode→data = preOrder[preOrderIndex];
    preOrderIndex++;
    if(inOrderStart == inOrderEnd)
        return newNode;
    // find the index of this node in Inorder traversal
    int inOrderIndex = Search(inOrder, inOrderStart, inOrderEnd, newNode→data);
    //Fill the left and right subtrees using index in Inorder traversal
    newNode→left = BuildBinaryTree(inOrder, preOrder, inOrderStart, inOrderIndex -1);
    newNode→right = BuildBinaryTree(inOrder, preOrder, inOrderIndex +1, inOrderEnd);
    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, the following combinations can uniquely identify a tree:

- Inorder and Preorder
- Inorder and Postorder
- Inorder and Level-order

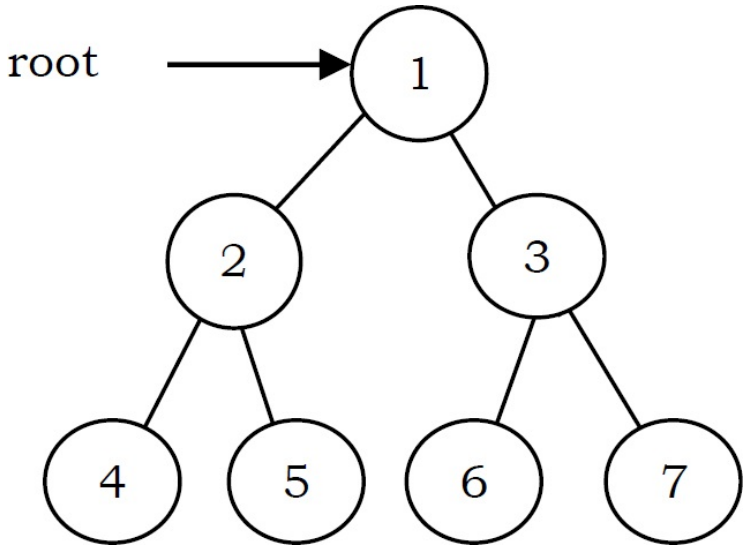
The following combinations do not uniquely identify a tree.

- Postorder and Preorder
- Preorder and Level-order
- Postorder and Level-order

For example, Preorder, Level-order and Postorder traversals are the same for the above trees:

So, even if three of them (PreOrder, Level-Order and PostOrder) are given, the tree cannot be constructed uniquely.

Problem-29 Give an algorithm for printing all the ancestors of a node in a Binary tree. For the tree below, for 7 the ancestors are 1 3 7.

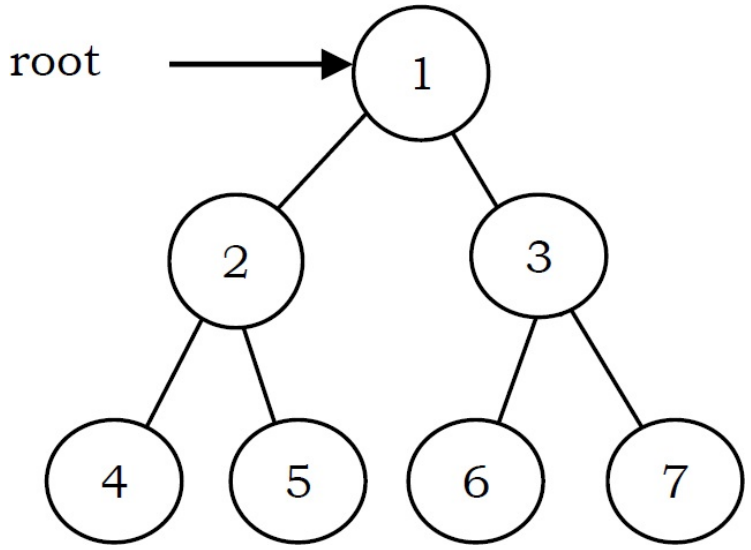


Solution: Apart from the Depth First Search of this tree, we can use the following recursive way to print the ancestors.

```
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 tree below 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 to left).

We pop from *currentLevel* stack and print the node's value. Whenever the current level order is from left to right, push the node's left child, then its right child, to stack *nextLevel*. Since a stack is a Last In First Out (*LIFO*) structure, the next time that nodes are popped off nextLevel, it will be in the reverse order.

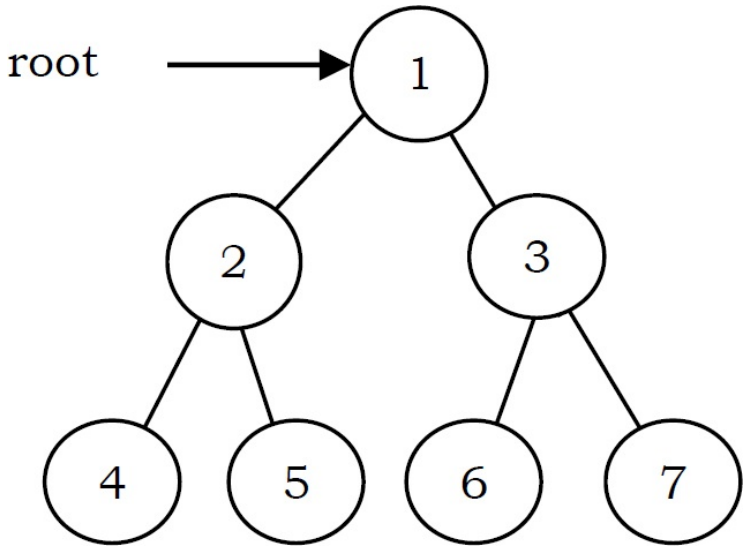
On the other hand, when the current level order is from right to left, we would push the node's right child first, then its left child. Finally, don't forget to swap those two stacks at the end of each level (*i. e.*, when *currentLevel* is empty).

```
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

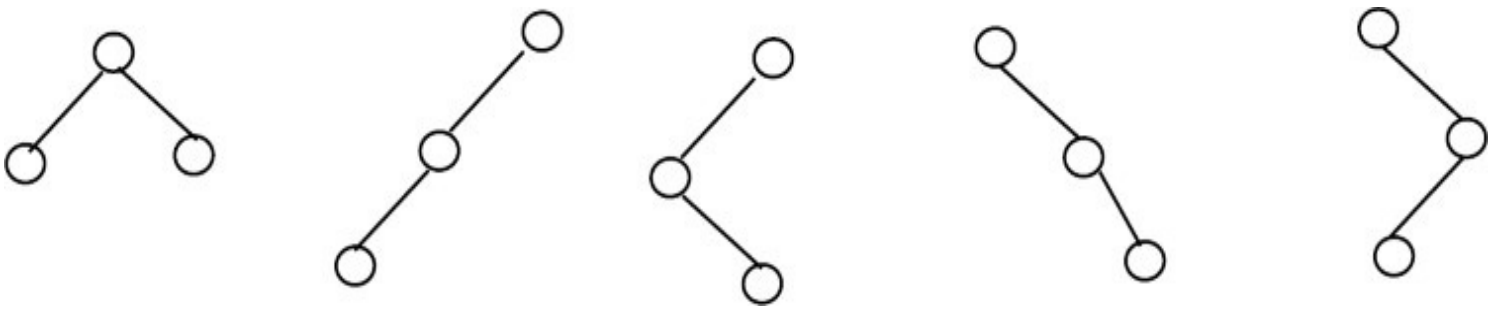


Solution: We can do an inorder traversal and hash the column. We call VerticalSumInBinaryTreefroot, 0) which means the root is at column 0. While doing the traversal, hash the column and increase its value by *root → data*.

```
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?

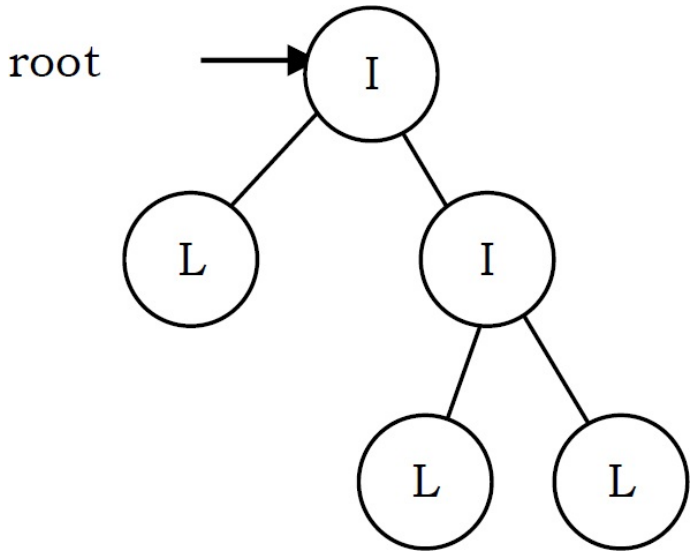
Solution: For example, consider a tree with 3 nodes (n = 3). It will have the maximum combination of 5 different (i.e., 2^3 -3 = 5) trees.



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.

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 a 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 when 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 the next subtree to compute.

Keeping the above invariant in mind, we can easily determine when a subtree ends and the next one starts. 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 the 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 Is there any other way of solving Problem-34?

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 pass the *left* and *right* to the recursion function itself, we connect the right child's *nextSibling* to the current node's nextSibling left child. In order for this to work, the current node *nextSibling* pointer must be populated, which is true in this case.

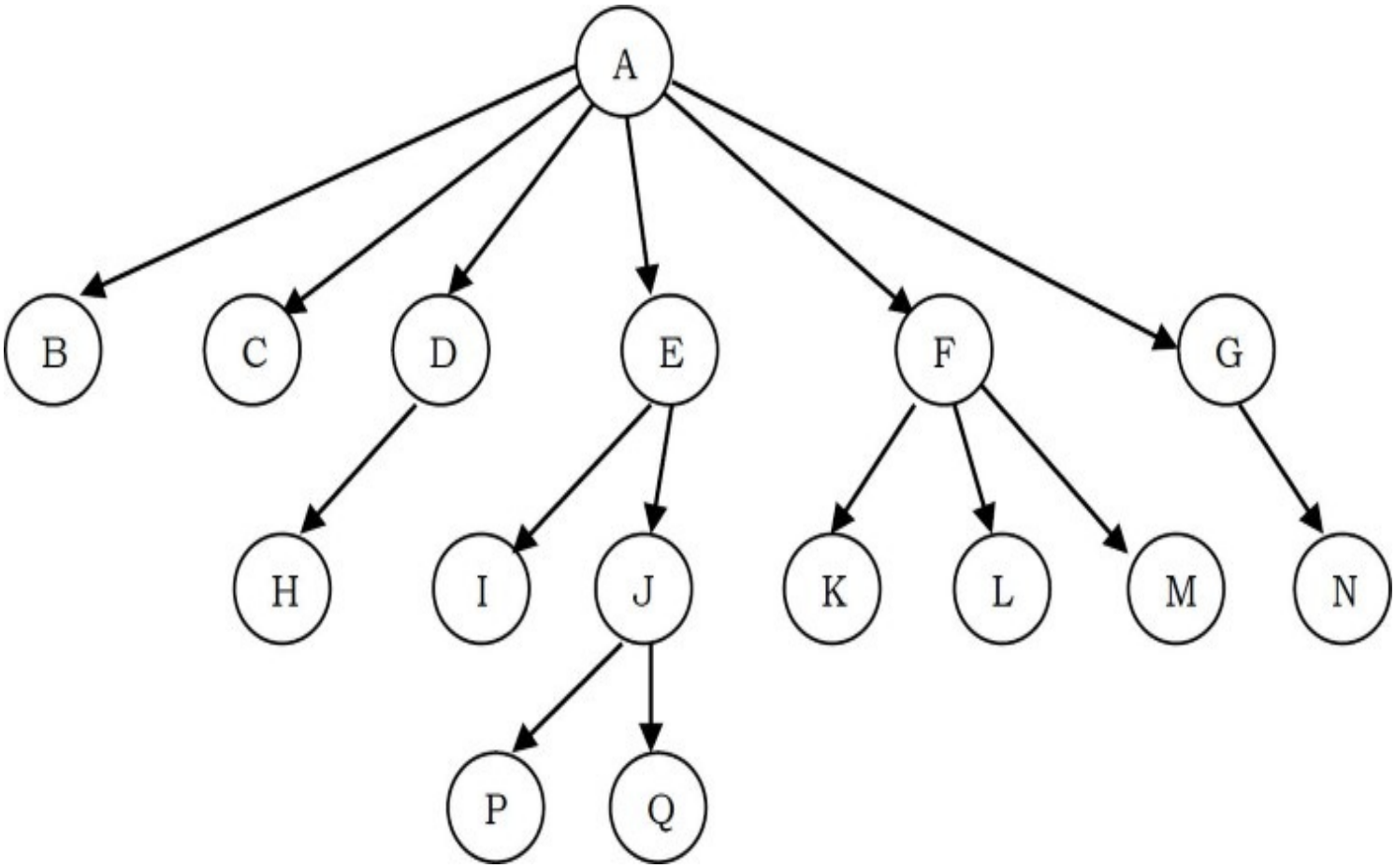
```
void FillNextSiblings(struct BinaryTreeNode* root) {
    if (!root)
        return;
    if (root→left)
        root→left→nextSibling = root→right;
    if (root→right)
        root→right→nextSibling = (root→nextSibling) ? root→nextSibling→left : NULL;
    FillNextSiblings(root→left);
    FillNextSiblings(root→right);
}
```

Time Complexity: O(*n*).

6.7 Generic Trees (N-ary Trees)

In the previous section we discussed binary trees where each node can have a maximum of two children and these are represented 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 below.



How do we represent the tree?

In the above tree, there are nodes with 6 children, with 3 children, with 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 that 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;
};
```

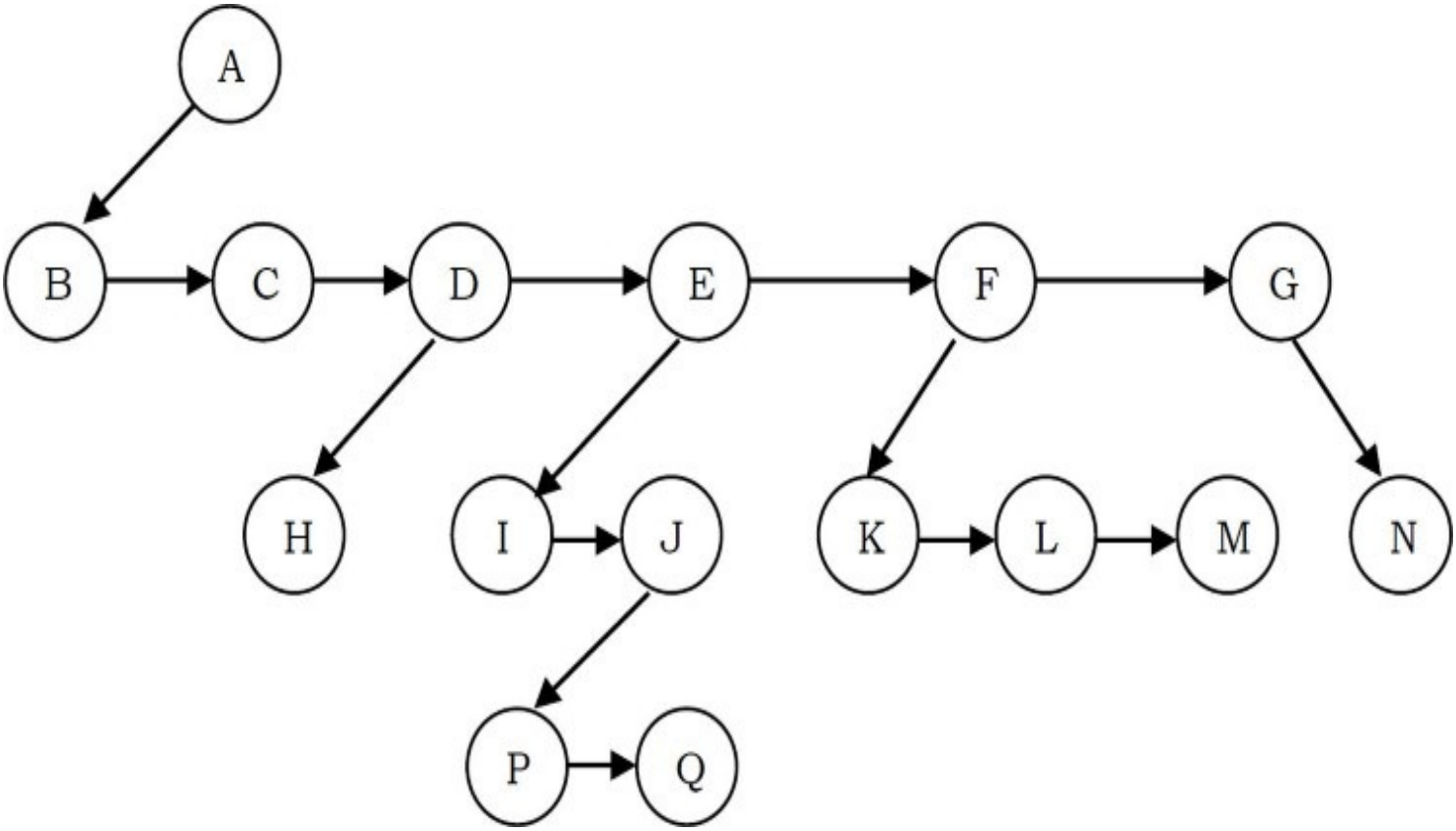
Since we are not using all the pointers in all the cases, there is a lot of memory wastage. Another problem is that we do not know the number of children for each node in advance. In order to

solve this problem we need a representation that minimizes the wastage and also accepts nodes with any number of children.

Representation of Generic Trees

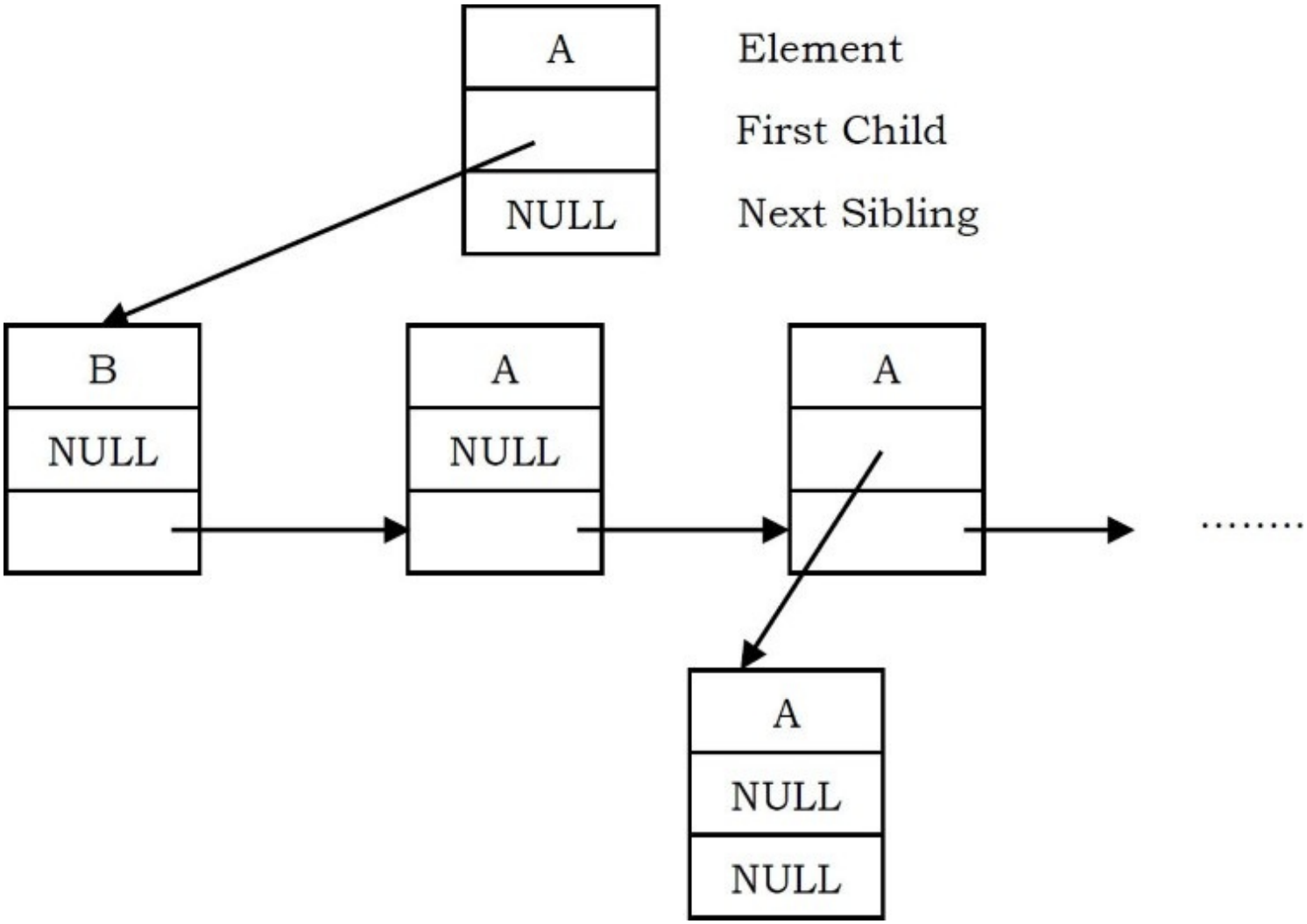
Since our objective is to reach all nodes of the tree, a possible solution to this is as follows:

- 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 children 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 a 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 convert any generic tree to binary representation; in practice we use binary trees. We can treat all generic trees with a first child/next sibling representation as binary trees.

Generic Trees: Problems & Solutions

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→nextSibling);
}
```

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 has 4 children, then we keep one node with 4 children and the remaining 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 the 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:

$$n = \frac{4^{h+1}-1}{3} \Rightarrow 4^{h+1} = 3n+1 \Rightarrow (h+1)log4 = \log(3n+1) \Rightarrow h+1 = \log_4(3n+1) \Rightarrow h = \log_4(3n+1) - 1$$

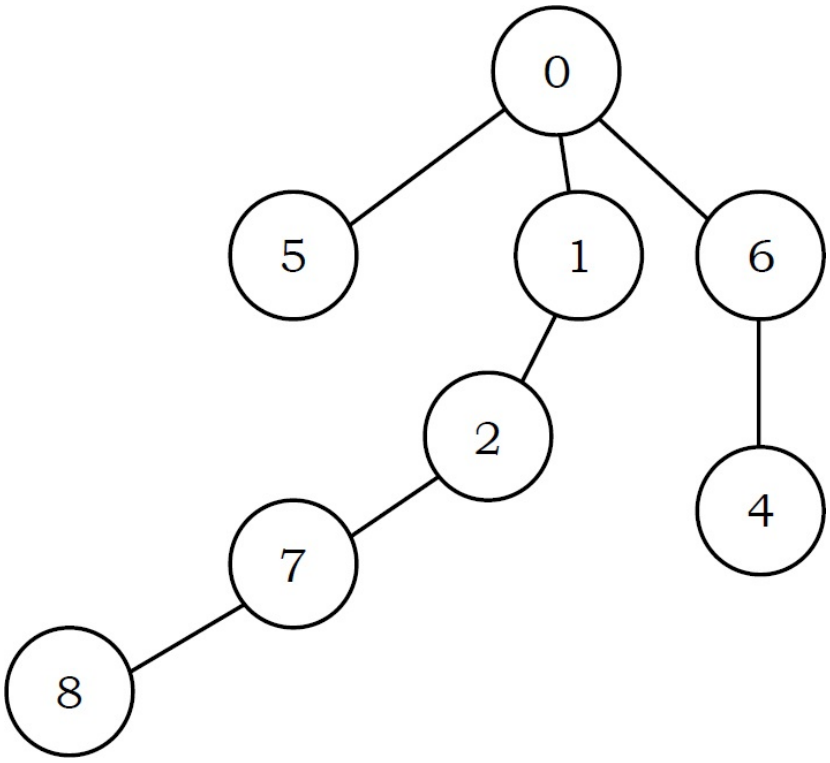
Problem-39 Given a parent array P, where $P[i]$ indicates the parent of i^{th} node in the tree (assume parent of root node is indicated with –1). Give an algorithm for finding the height or depth of the tree.

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 represents 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 the 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 next siblings.

```
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;
};
```

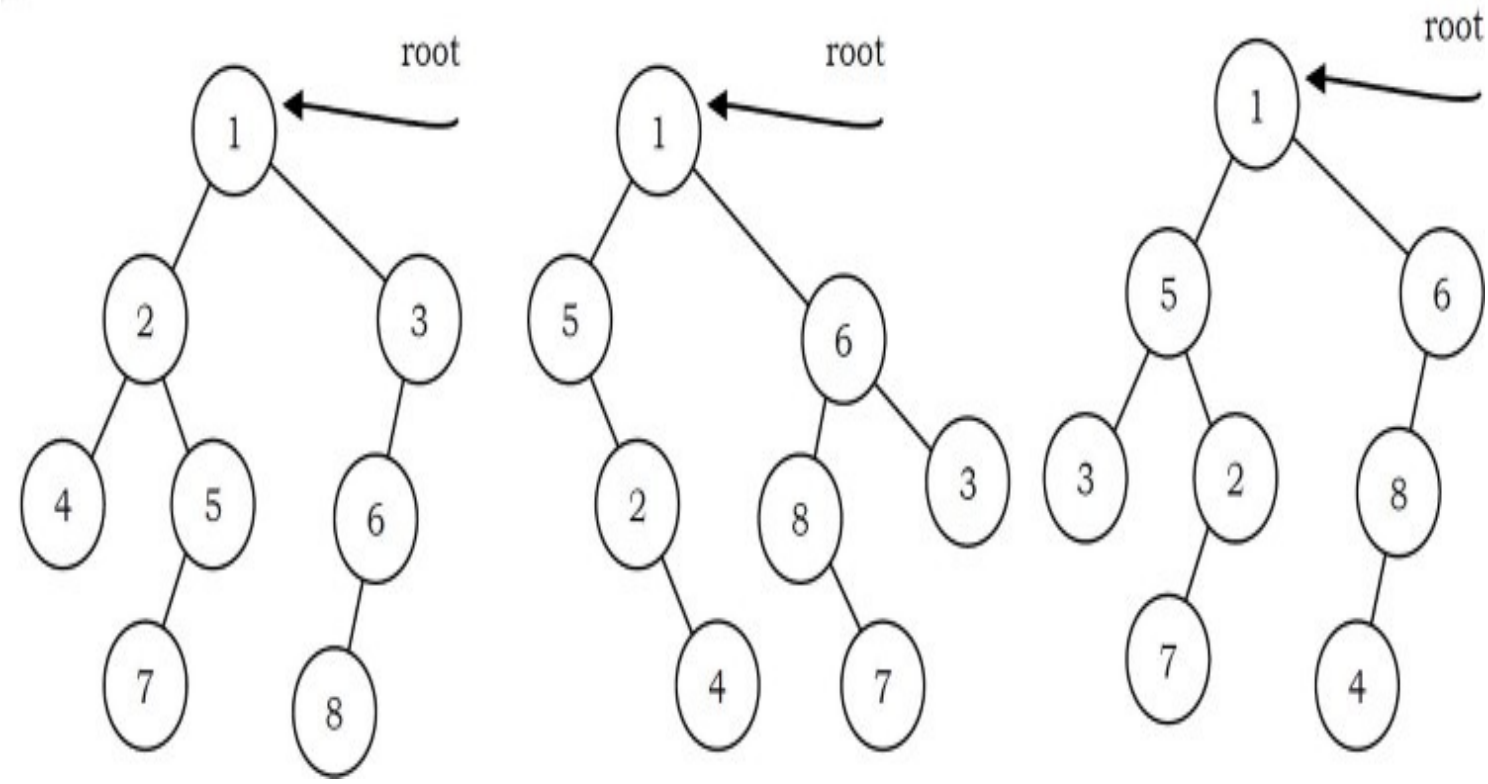
For a given node in the tree, we just need to point to its first child and keep traversing all its next siblings.

```
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:



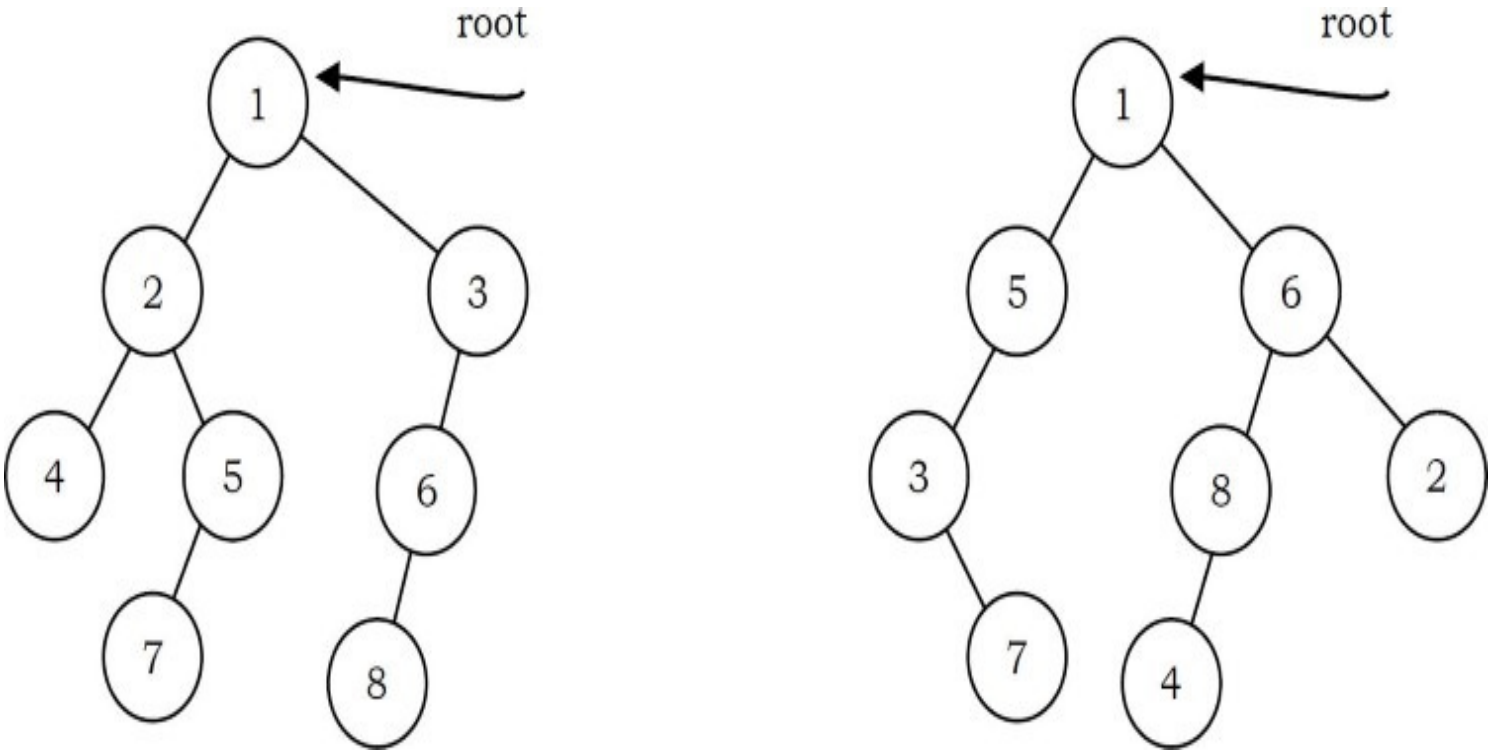
Two binary trees *root*1 and *root*2 are isomorphic if they have the same structure. The values of the nodes does not affect whether two trees are isomorphic or not. In the diagram below, the tree in the middle is not isomorphic to the other trees, but the tree on the right is isomorphic to the tree on the left.

```
int IsIsomorphic(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:



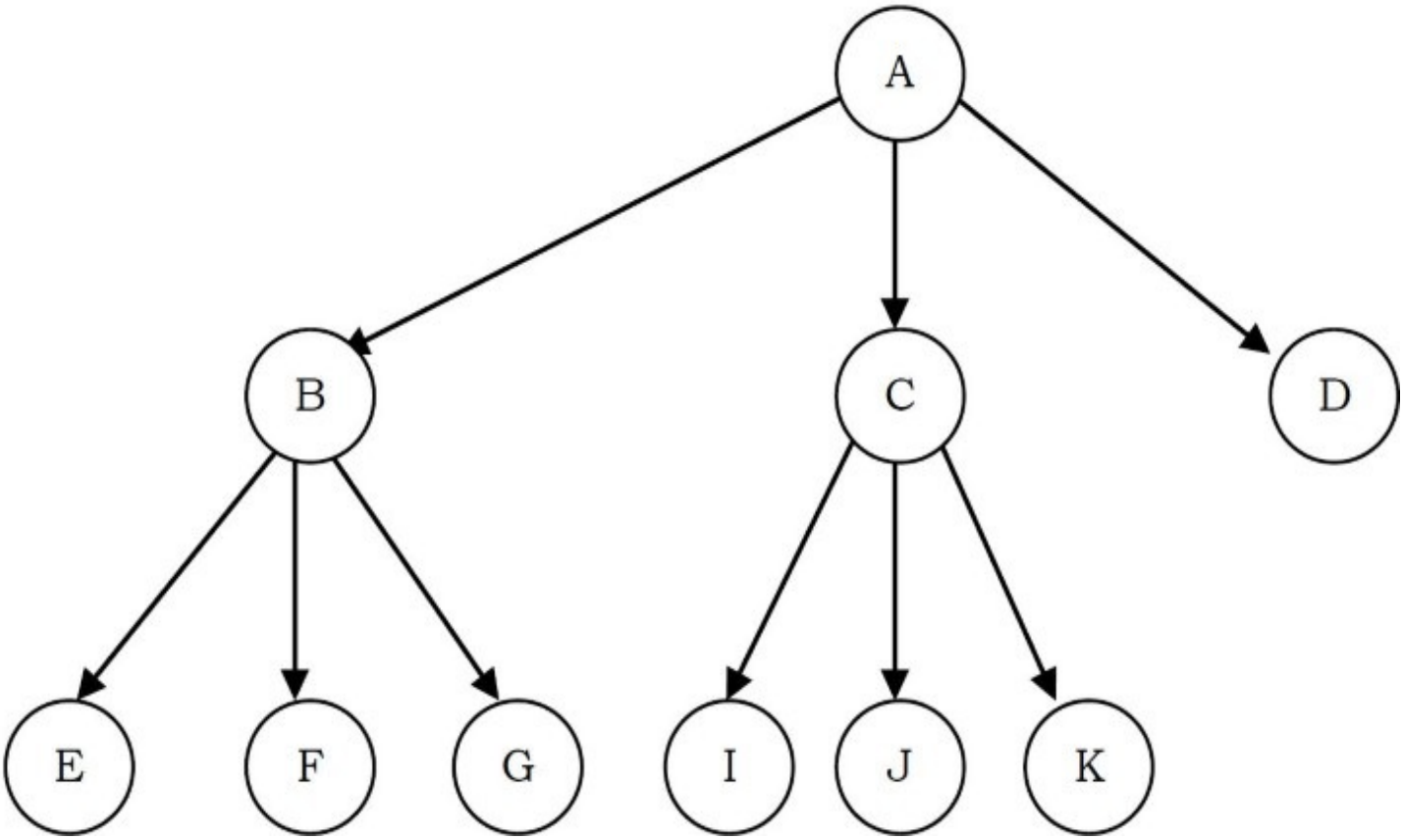
Two trees *root*1 and *root*2 are quasi-isomorphic if *root*1 can be transformed into *root*2 by swapping the left and right children of some of the nodes of *root*1. Data in the nodes are not important in determining quasi-isomorphism; only the shape is important. The trees below are quasi-isomorphic 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 example below is 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. The 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.

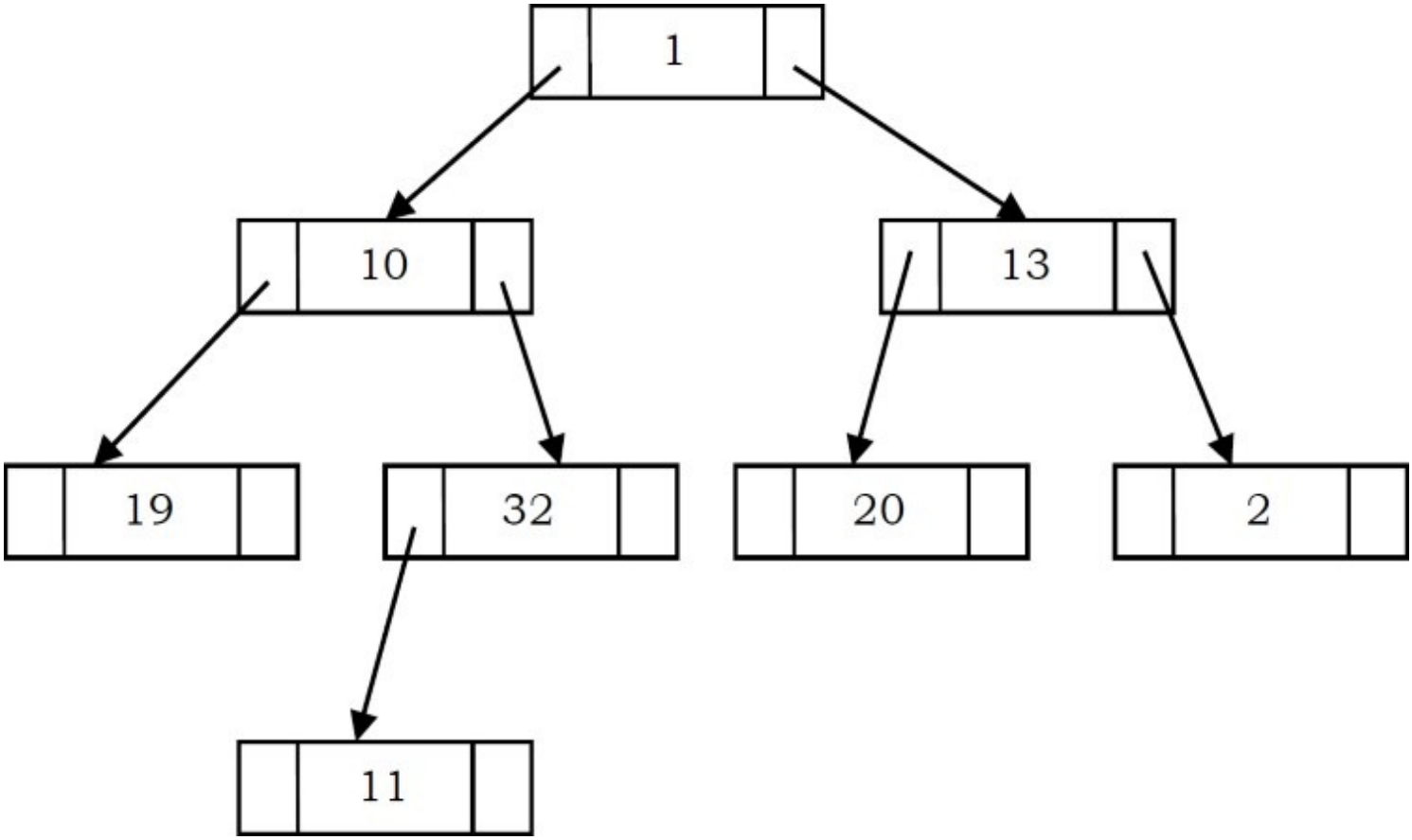
6.8 Threaded Binary Tree Traversals (Stack or Queue-less Traversals)

In earlier sections we have seen that, *preorder, inorder and postorder* binary tree traversals used stacks and *level order* traversals used queues as an auxiliary data structure. In this section we will discuss new traversal algorithms which do not need both stacks and queues. 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 the previous traversals carefully, stack/ queue is required because we have to record the current position in order to move to the 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 want to store some useful information in NULL

pointers. The next question is what to store?

The common convention is to 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 can call such binary trees *left threaded binary trees*.
- If we store successor information in NULL right pointers only, then we can call such binary trees *right threaded binary trees*.
- If we store predecessor information in NULL left pointers and successor information in NULL right pointers, then we can call such binary trees *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

Based on above discussion we get three representations for threaded binary trees.

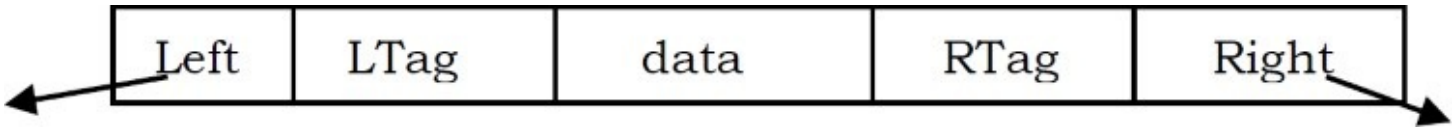
- *Preorder Threaded Binary Trees:* NULL left pointer will contain PreOrder predecessor information and NULL right pointer will contain PreOrder successor information.
- *Inorder Threaded Binary Trees:* NULL left pointer will contain InOrder predecessor information and NULL right pointer will contain 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 in each node, giving us, for threaded trees, nodes of the following form:



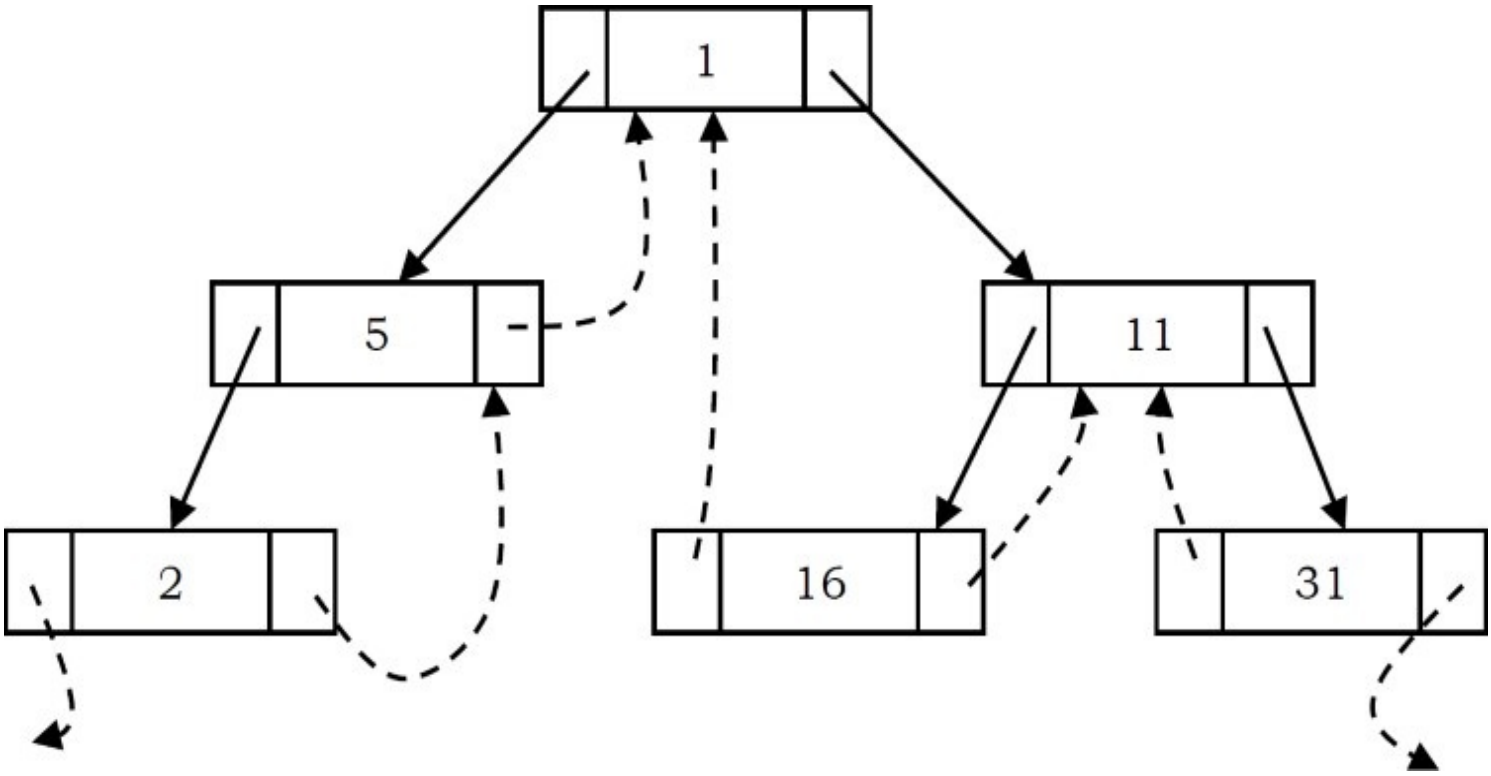

```
struct ThreadedBinaryTreeNode{
    struct ThreadedBinaryTreeNode *left;
    int LTag;
    int data;
    int RTag;
    struct ThreadedBinaryTreeNode *right;
};
```

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

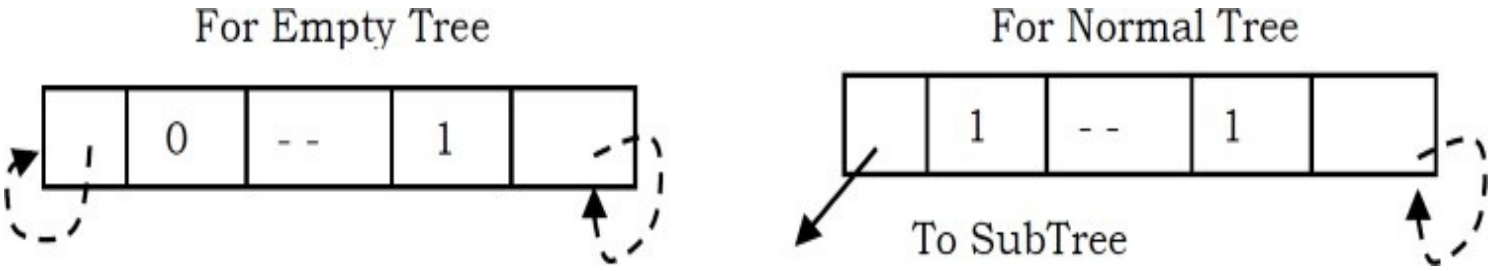
Note: Similarly, we can define preorder/postorder differences as well.

As an example, let us try representing a tree in inorder threaded binary tree form. The tree below shows what 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.

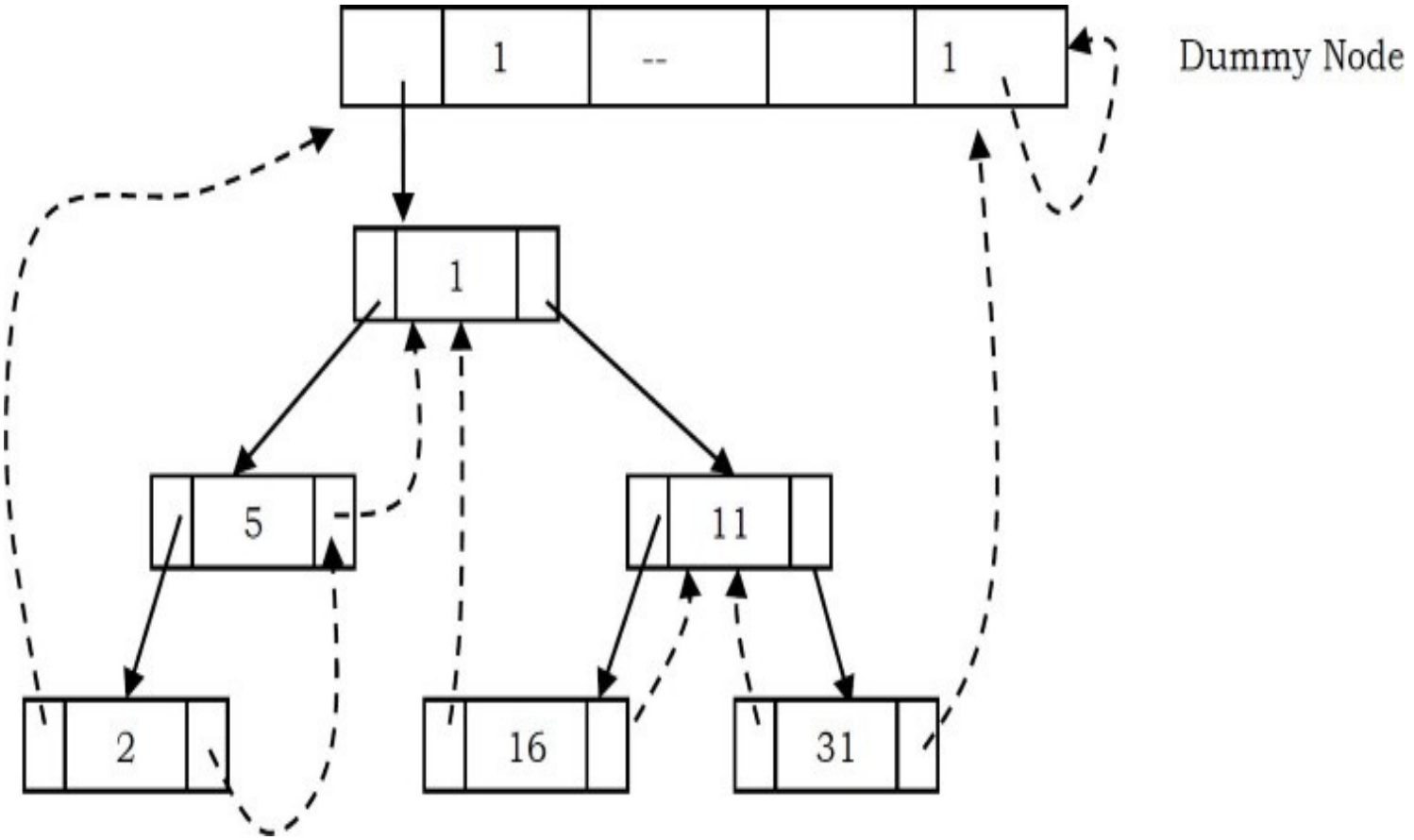


What should leftmost and rightmost pointers point to?

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);
        }
}
```

Alternative 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 in inorder traversal, start with *dummy* node and call PreorderSuccessorf) 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);
    }
}
```

Alternative coding:

```
void PreorderTraversal(struct ThreadedBinaryTreeNode *root) {
    struct ThreadedBinaryTreeNode *P = root;
    while(1){
        P = PreorderSuccessor(P);
        if(P == root) return;
        printf("%d",P→data);
    }
}
```

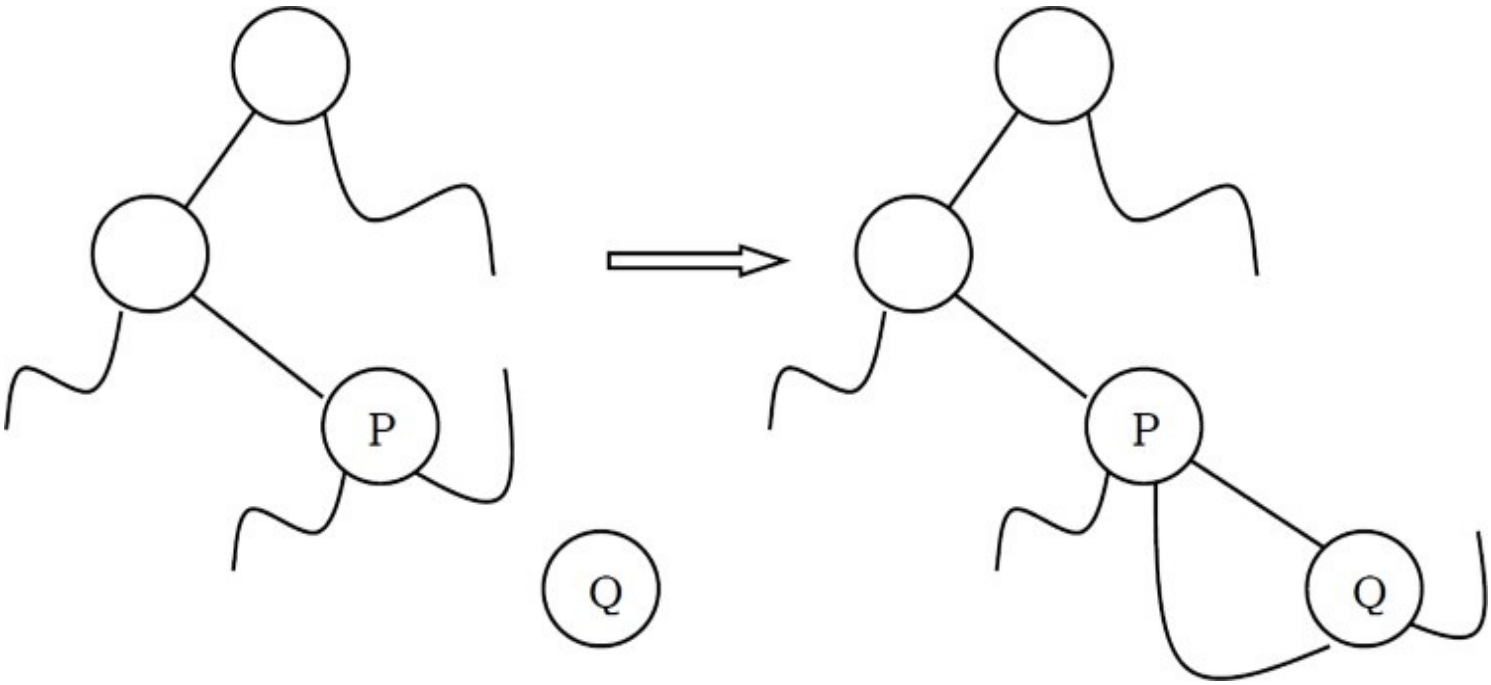
Time Complexity: O(*n*). Space Complexity: O(1).

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.

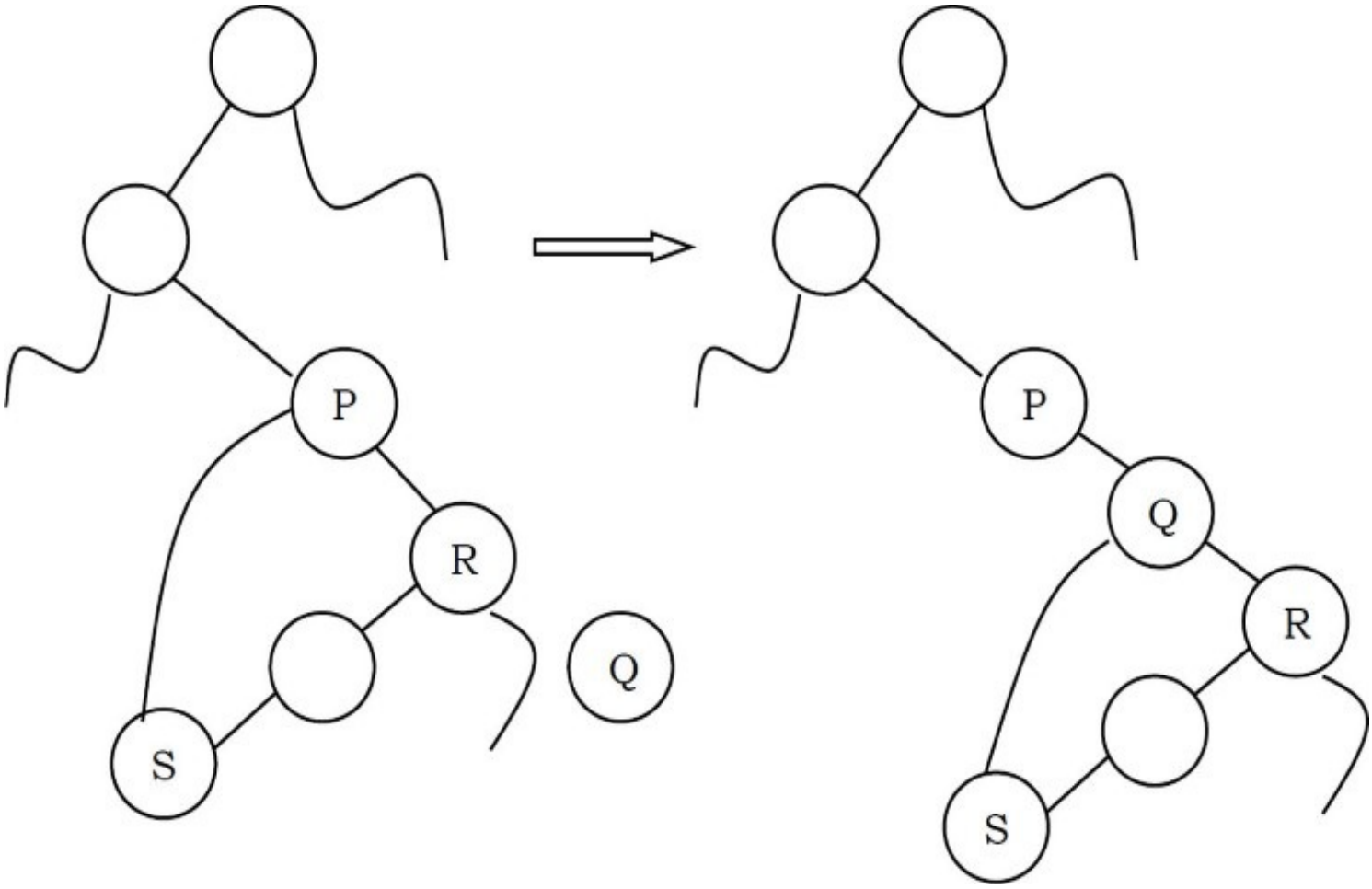
Insertion of Nodes in InOrder Threaded Binary Trees

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 have 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).

Threaded Binary Trees: Problems & Solutions

Problem-45 For a given binary tree (*n*ot 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, and thereafter its value is NULL. Since we are simply asking for the successor of the node we got the 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 *PreorderSuccessor(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;
}
```

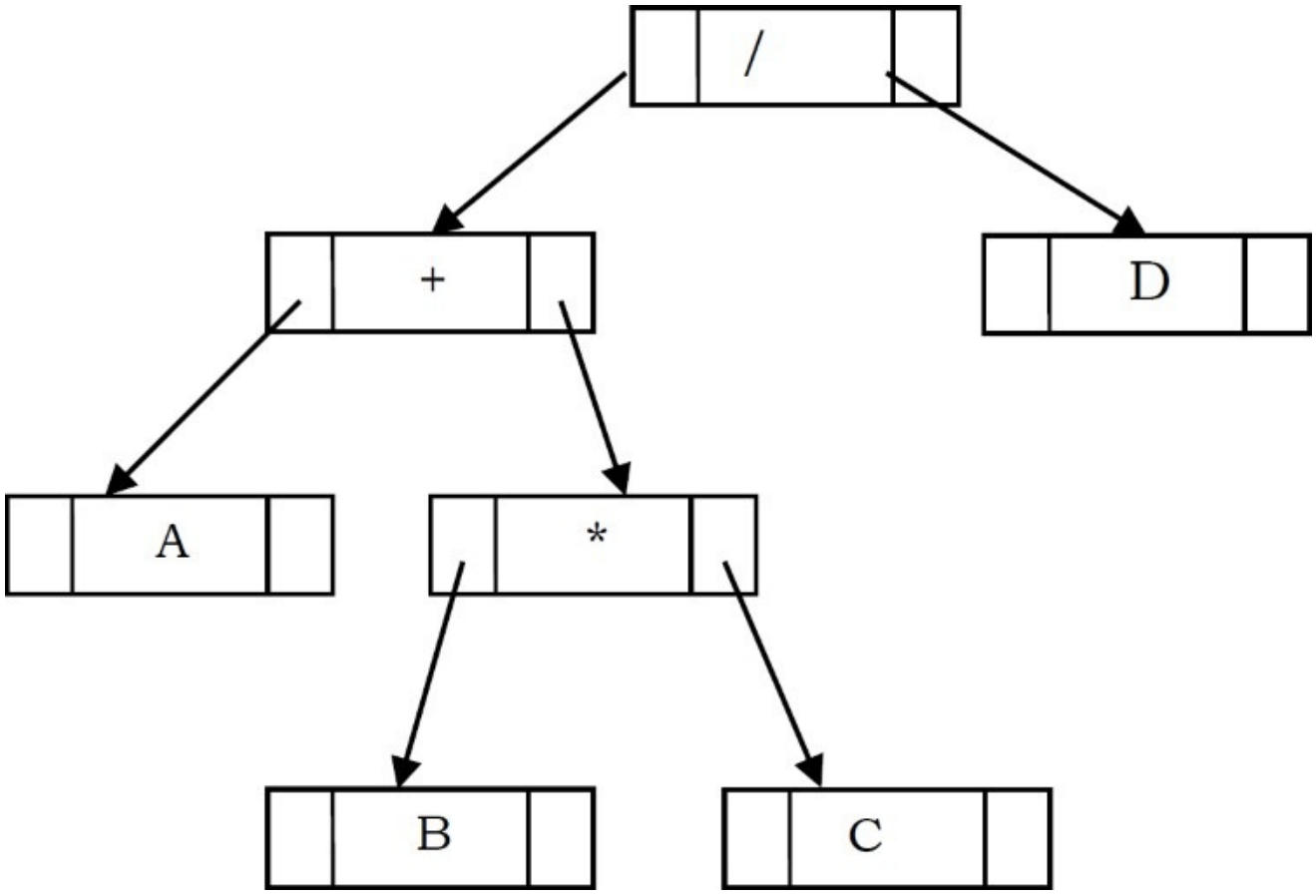
Problem-46 For a given binary tree (*n*ot threaded) how do we find the inorder successor?

Solution: Similar to the above discussion, we can find the inorder successor of a node as:

```
// In-order successor for an unthreaded binary tree
struct BinaryTreeNode *InorderSuccessor(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 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. An expression tree consists of binary expression. But for a u-nary operator, one subtree will be empty. The figure below shows a simple expression tree for (A + B * C) / D.

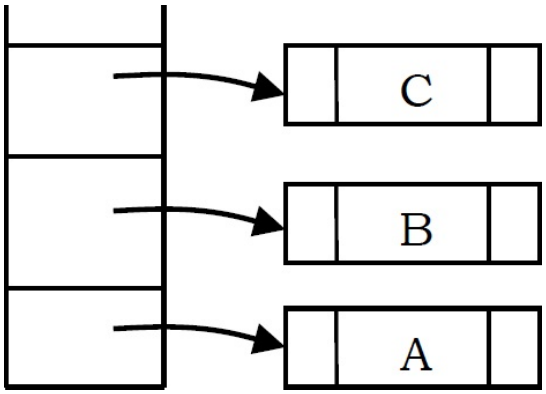


Algorithm for Building Expression Tree from Postfix Expression

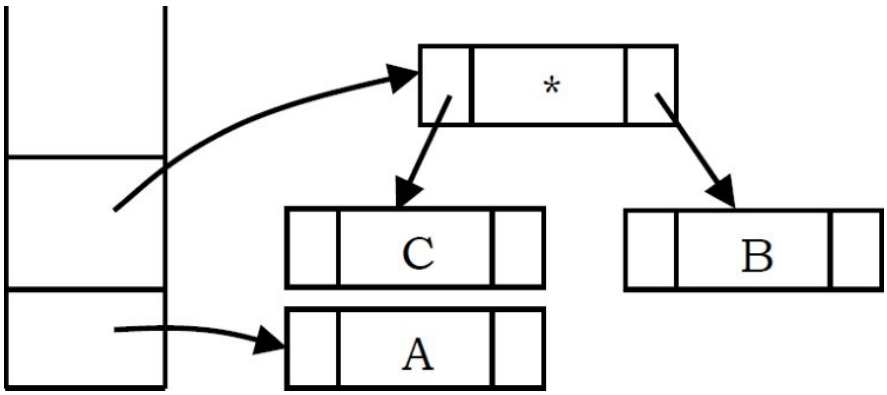
```
struct BinaryTreeNode *BuildExprTree(char postfixExpr[], int size){
    struct Stack *S = Stack(size);
    for (int i = 0; i< size; i++) {
        if(postfixExpr[i] is an operand) {
            struct BinaryTreeNode newNode = (struct BinaryTreeNode*)
                    malloc( sizeof (struct BinaryTreeNode));
            if(!newNode) {
                printf("Memory Error");
                return NULL;
            }
            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 NULL;
            }
            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.

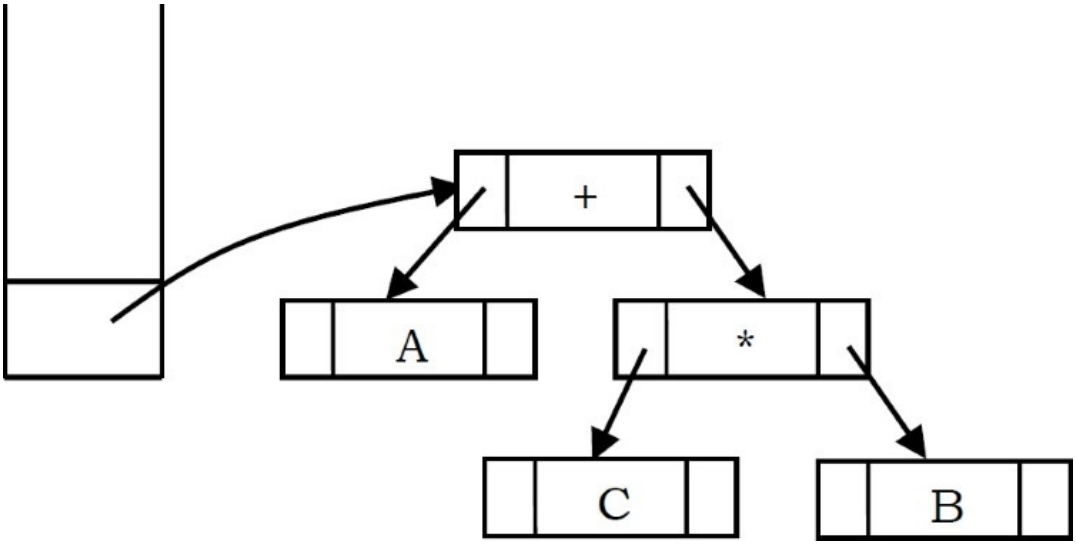
As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



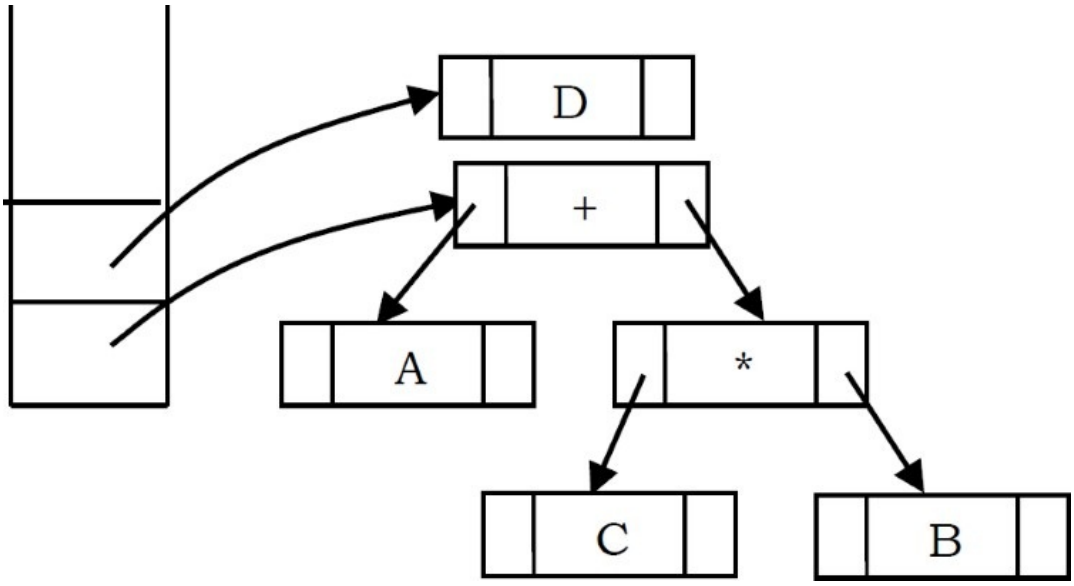
Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



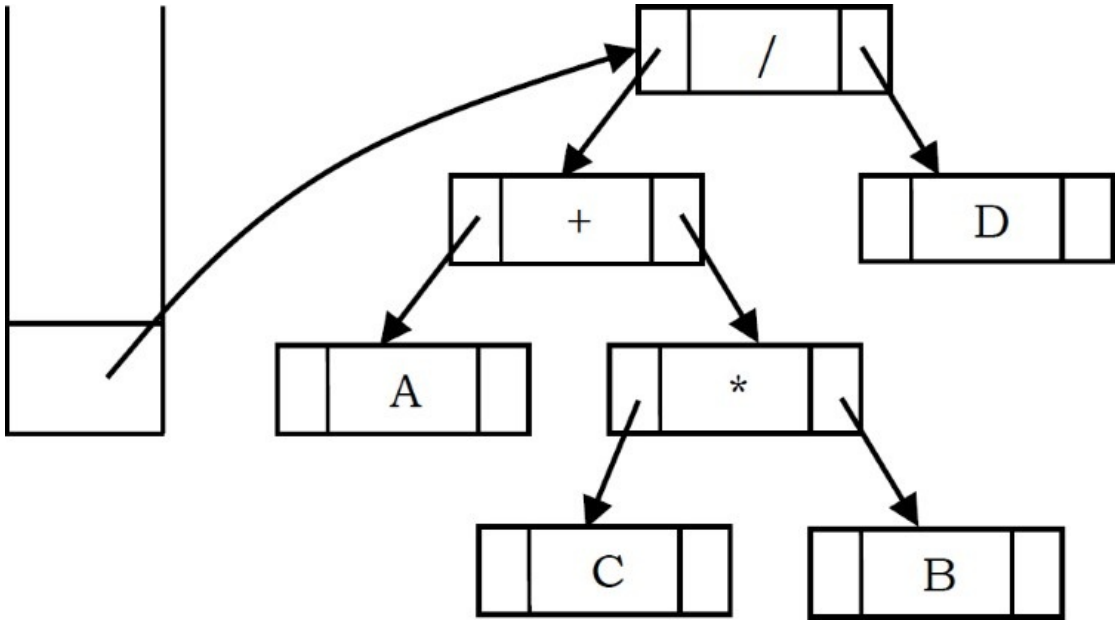
Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



Next, an operand 'D' is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



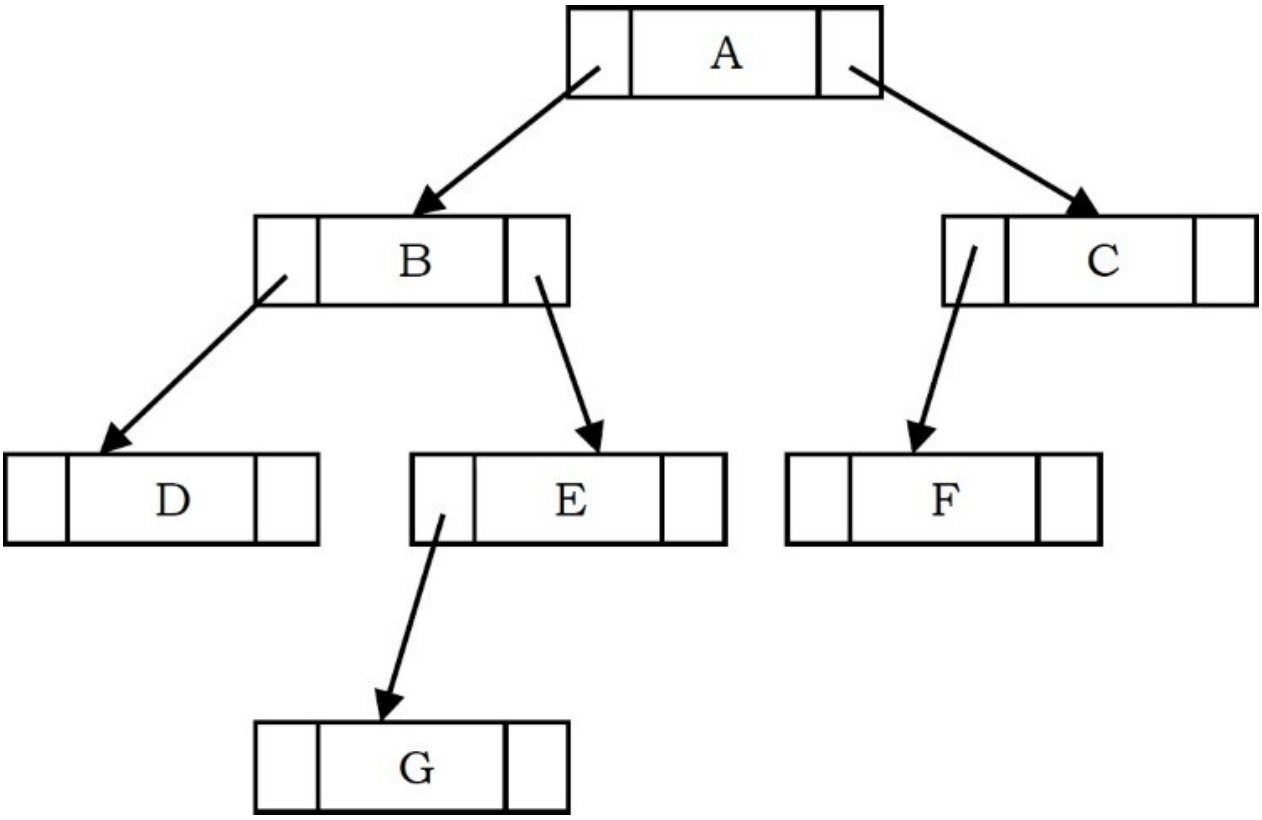
Finally, the last symbol (‘/’) is read, two trees are merged and a pointer to the final tree is left on the stack.



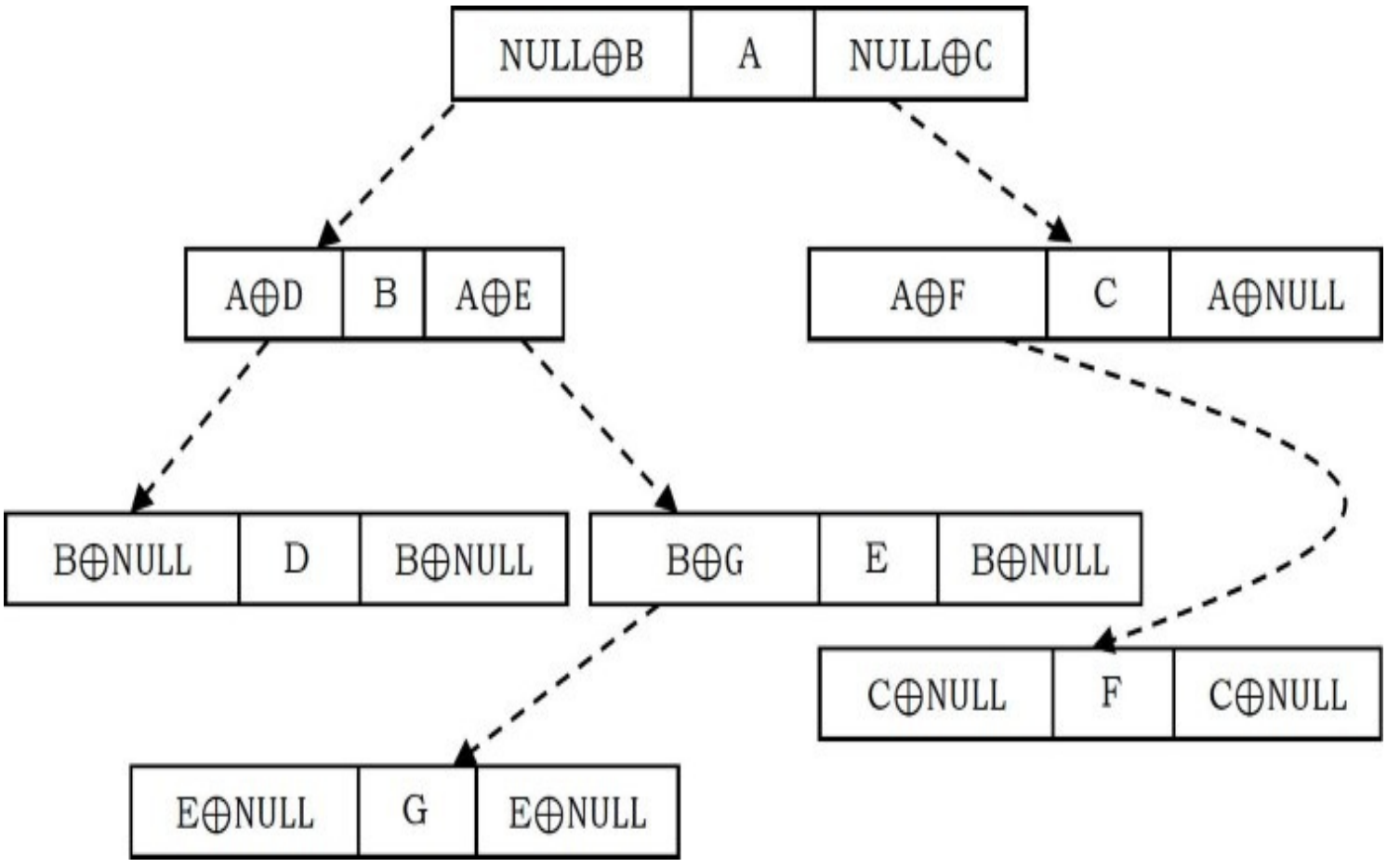
6.10 XOR Trees

This concept is 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 the 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 root's 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?

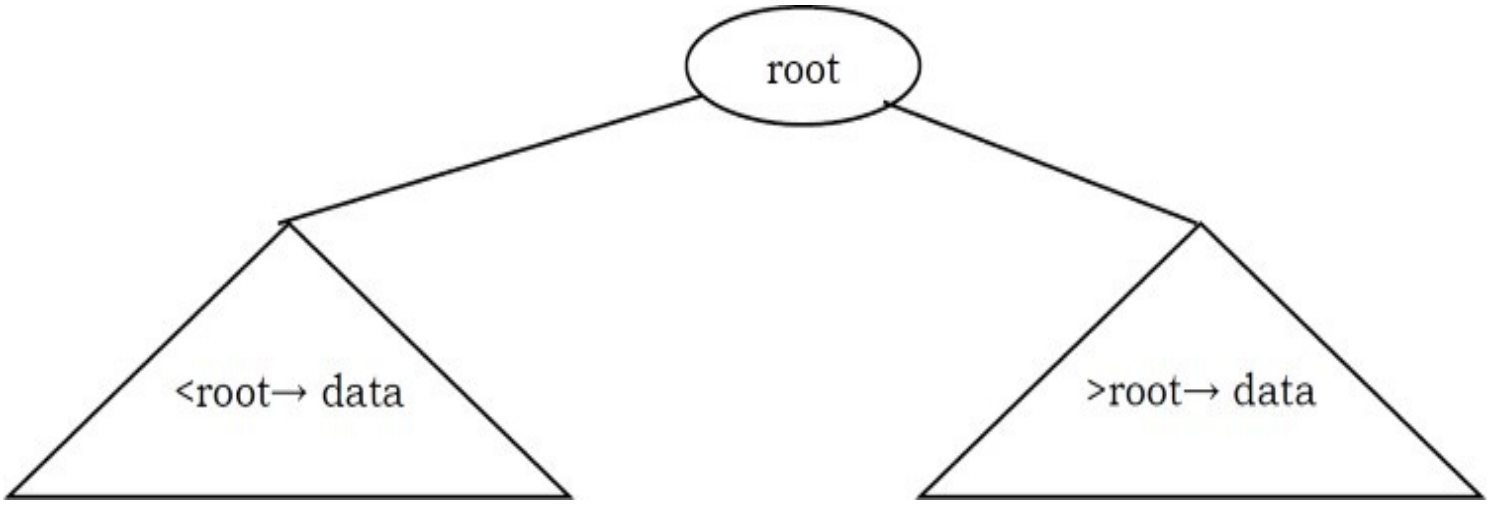
In previous sections we have discussed different tree representations and in all of them we did not impose any restriction on the nodes data. As a result, to search for an element we need to check both in left subtree and in right subtree. Due to this, the worst case complexity of search operation is 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*).

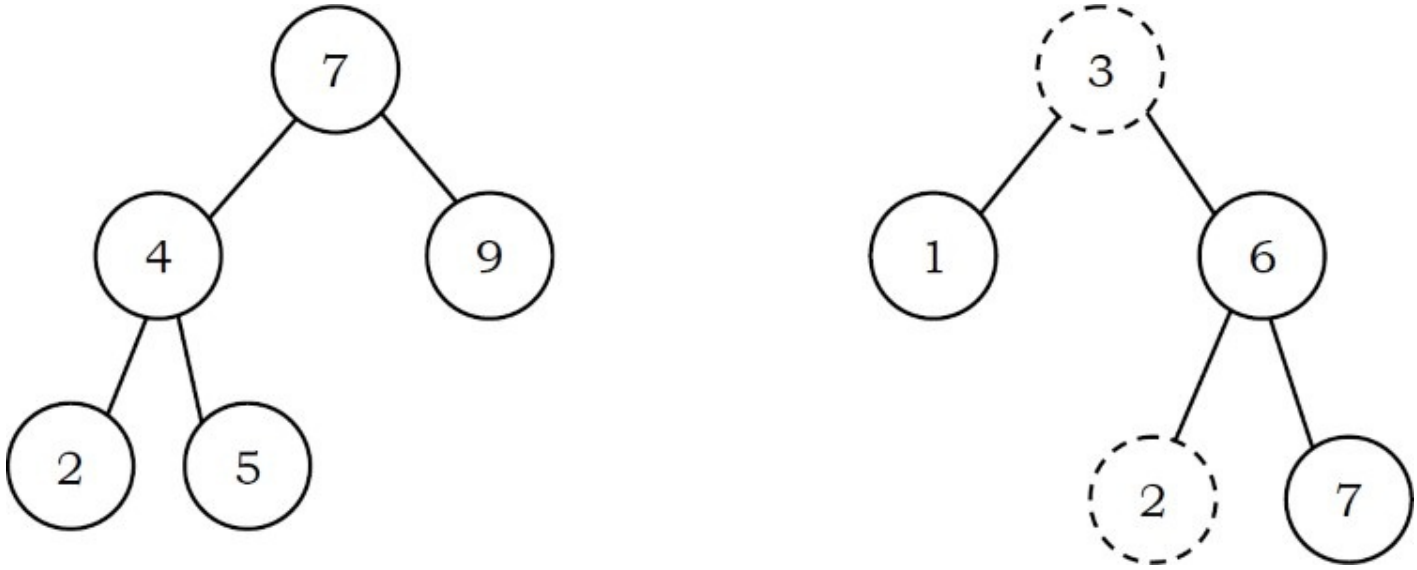
Binary Search Tree Property

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 the right tree is not a 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: Following are the main operations that are supported by binary search trees:

- 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 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, first we process left subtree, then root data, and finally we process right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.
- If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it. The same is the case with the right subtree.. Because of this, binary search trees take less time for searching an element than regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.
- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.
- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(lgn) worst-case time. If the tree is a linear chain of n nodes (skew-tree), however, the same operations takes O(n) worst-case time.

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 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 BST below, the minimum element is **4**.