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

Data Structures and Algorithms Made Easy in JAVA

Data Structure and Algorithmic Puzzles

CareerMonk Publications

Data Structures
And
Algorithms
Made Easy In JAVA

Data Structures and Algorithmic Puzzles

By
Narasimha Karumanchi

Acknowledgements

Mother and *Father*, it is impossible to thank you adequately for everything you have done, from loving me unconditionally to raising me in a stable household, where your persistent efforts and traditional values taught your children to celebrate and embrace life. I could not have asked for better parents or role-models. You showed me that anything is possible with faith, hard work and determination.

This book would not have been possible without the help of many people. I would like to express my gratitude to all of the people who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and design. In particular, I would like to thank the following individuals:

- *Mohan Mullapudi*, IIT Bombay, Architect, dataRPM Pvt. Ltd.
- *Navin Kumar Jaiswal*, Senior Consultant, Juniper Networks Inc.
- *A. Vamshi Krishna*, IIT Kanpur, Mentor Graphics Inc.
- *Kondrakunta Murali Krishna*, B-Tech., Technical Lead, HCL
- *Cathy Reed, BA, MA*, Copy Editor
- *Prof. Girish P. Saraph, Founder, Vegayan Systems, IIT Bombay*
- *Kishore Kumar Jinka*, IIT Bombay
- *Prof. Hsin – mu Tsai, National Taiwan University, Taiwan*
- *Prof. Chintapalli Sobhan Babu. IIT, Hyderabad*
- *Prof. Meda Sreenivasa Rao, JNTU, Hyderabad*

Last but not least, I would like to thank the *Directors* of *Guntur Vikas College, Prof. Y.V. Gopala Krishna Murthy* & *Prof. Ayub Khan* [*ACE Engineering Academy*], *T.R.C.Bose* [*Ex. Director* of *APTransco*], *Ch.Venkateswara Rao VNR Vignanajyothi* [*Engineering College, Hyderabad*], *Ch.Venkata Narasaiah* [*IPS*], *Yarapathineni Lakshmaiah* [*Manchikallu, Gurazala*], & *all our well – wishers* for helping me and my family during our studies.

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

Preface

Dear Reader,

Please hold on! I know many people typically do not read the Preface of a book. But I strongly recommend that you read this particular Preface.

It is not the main objective of this book to present you with the theorems and proofs on *data structures* and *algorithms*. I have followed a pattern of improving the problem solutions with different complexities (for each problem, you will find multiple solutions with different, and reduced, complexities). Basically, it's an enumeration of possible solutions. With this approach, even if you get a new question, it will show you a way to *think* about the possible solutions. You will find this book useful for interview preparation, competitive exams preparation, and campus interview preparations.

As a *job seeker*, if you read the complete book, I am sure you will be able to challenge the interviewers. If you read it as an *instructor*, it will help you to deliver lectures with an approach that is easy to follow, and as a result your students will appreciate the fact that they have opted for Computer Science / Information Technology as their degree.

This book is also useful for *Engineering degree students* and *Masters degree students* during their academic preparations. In all the chapters you will see that there is more emphasis on problems and their analysis rather than on theory. In each chapter, you will first read about the basic required theory, which is then followed by a section on problem sets. In total, there are approximately 700 algorithmic problems, all with solutions.

If you read the book as a *student* preparing for competitive exams for Computer Science / Information Technology, the content covers *all the required topics* in full detail. While writing this book, my main focus was to help students who are preparing for these exams.

In all the chapters you will see more emphasis on problems and analysis rather than on theory. In each chapter, you will first see the basic required theory followed by various problems.

For many problems, *multiple* solutions are provided with different levels of complexity. We start with the *brute force* solution and slowly move toward the *best solution* possible for that problem. For each problem, we endeavor to understand how much time the algorithm takes and how much memory the algorithm uses.

It is recommended that the reader does at least one *complete* reading of this book to gain a full understanding of all the topics that are covered. Then, in subsequent readings you can skip directly to any chapter to refer to a specific topic. Even though many readings have been done for the purpose of correcting errors, there could still be some minor typos in the book. If any are found, they will be updated at *www.CareerMonk.com*. You can monitor this site for any corrections and also for new problems and solutions. Also, please provide your valuable suggestions at: *Info@CareerMonk.com*.

I wish you all the best and I am confident that you will find this book useful.

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

Other Books by Narasimha Karumanchi


IT Interview Questions


Elements of Computer Networking


Data Structures and Algorithmic Thinking with Python


Data Structures and Algorithms Made Easy (C/C++)


Coding Interview Questions


Data Structures and Algorithms for GATE


Peeling Design Patterns

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 getting in to the definition of variables, let us relate them to an old mathematical equation. Many of us would have solved many mathematical equations since childhood. As an example, consider the equation below:

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

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.

```
public class newType {
   public int data1;
   public int data 2;
   private float data3;
   ...
   private char data;
   //Operations
}
```

1.3 Data Structure

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 (say, Linked Lists). *Examples:* Linked Lists, Stacks and Queues.
2) *Non – linear data structures:* Elements of this data structure are stored/accessed in a non-linear order. *Examples:* Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system-defined data types. We all know that, by default, all primitive data types (int, float, 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 a 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 are: creating the stack, push an element onto the stack, pop an element from the stack, finding the current top of the stack, finding the 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.

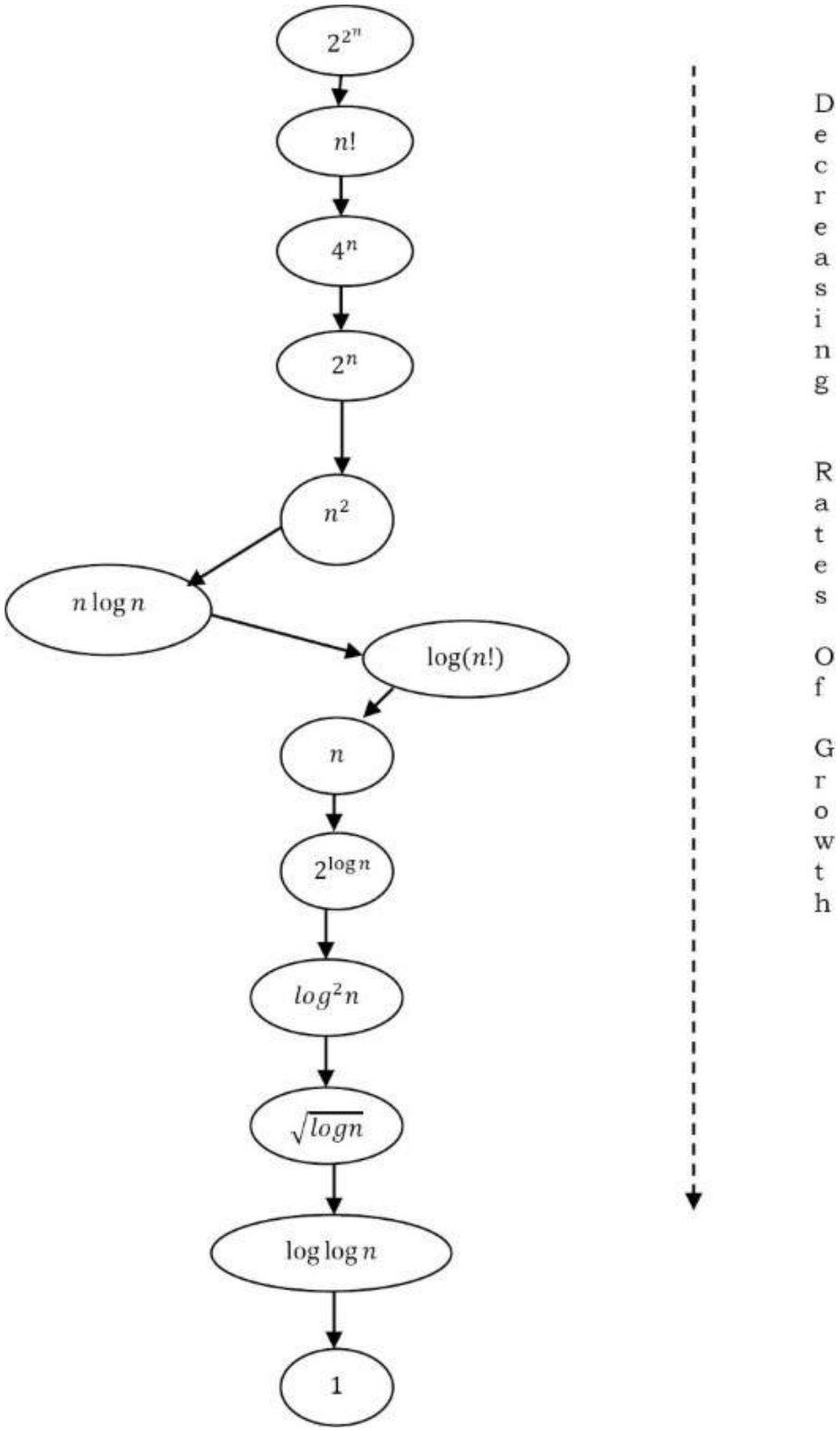
$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

1.11 Commonly used Rates of Growth

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

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



1.12 Types of Analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) 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.

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

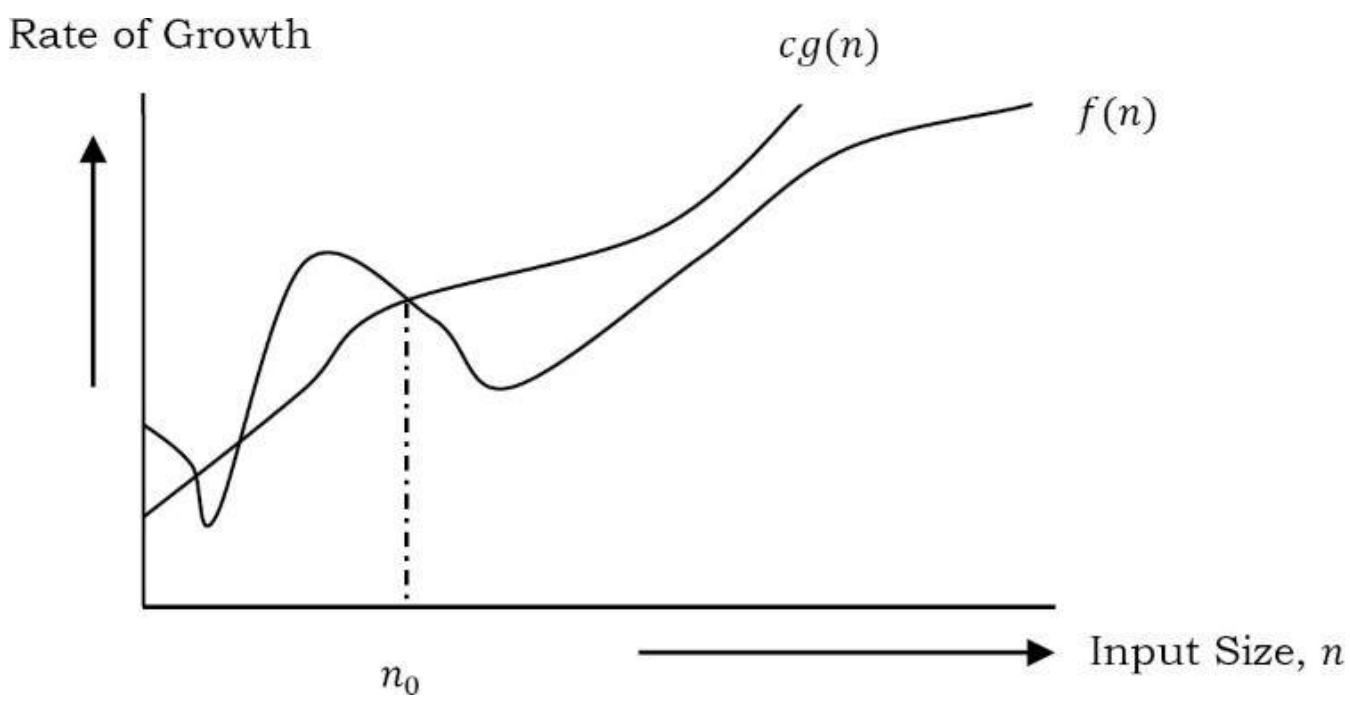
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

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

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.

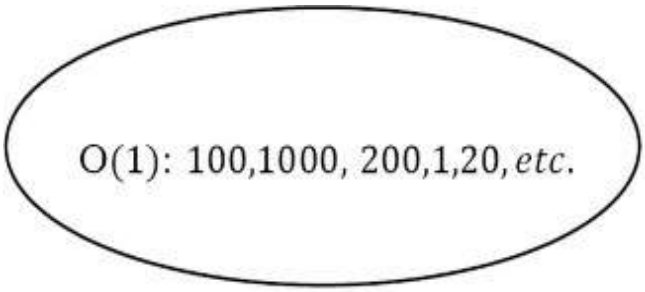
Let us see the O–notation with a little more detail. O–notation defined as $O(g(n)) = \{f(n)$*:* there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth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

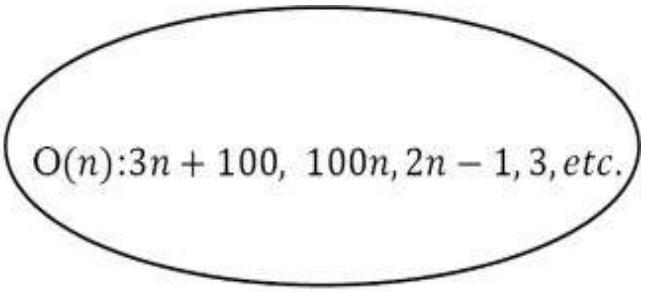
Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0, the rate of growth could be different. n_0 is called threshold for the given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

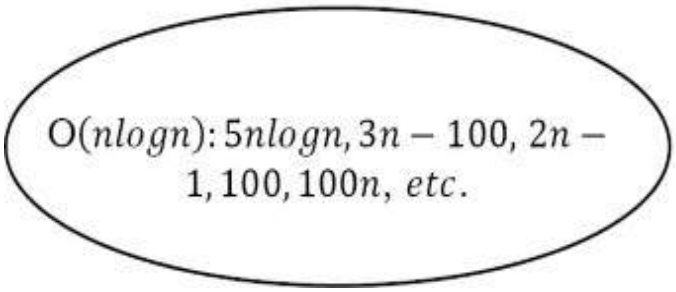
Big-O Visualization

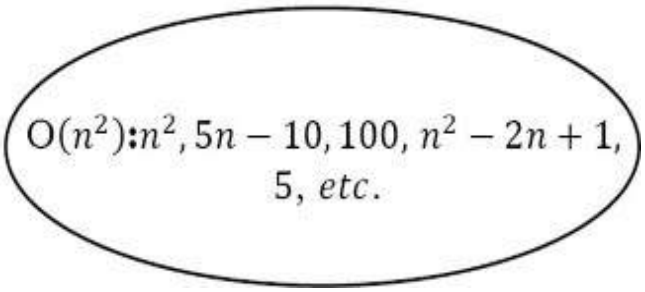
$O(g(n))$ is the set of functions with smaller or the same order of growth as $g(n)$. For example; $O(n^2)$ includes $O(1)$, $O(n)$, $O(nlogn)$, etc.

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: $3n + 8 \leq 4n$, for all $n \geq 8$
$\therefore 3n + 8 = O(n)$ 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 $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(n^3)$ 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 $n \geq 1$
$\therefore 410 = O(1)$ 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.

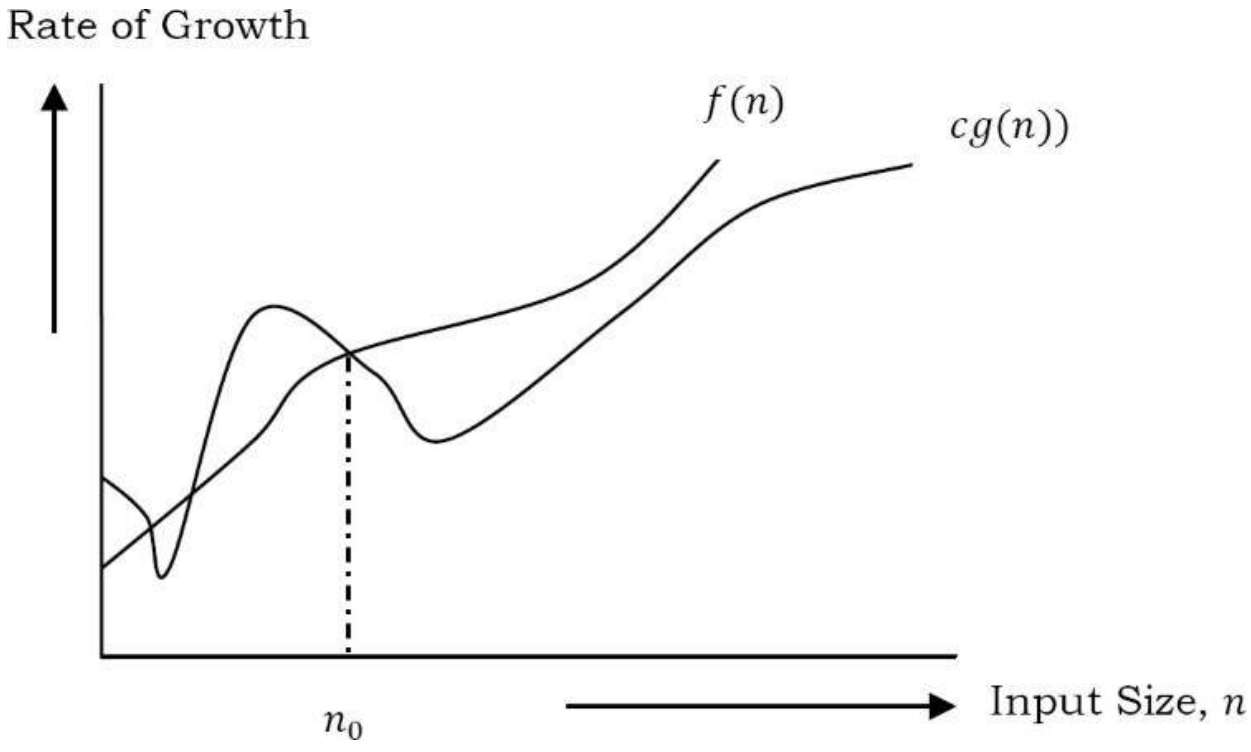
Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n$, for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.

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

1.15 Omega-Ω Notation

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 $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give 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: $\exists c, n_0$ 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: $\exists$ 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 $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

$\Rightarrow$ Contradiction: n cannot be smaller than a constant

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

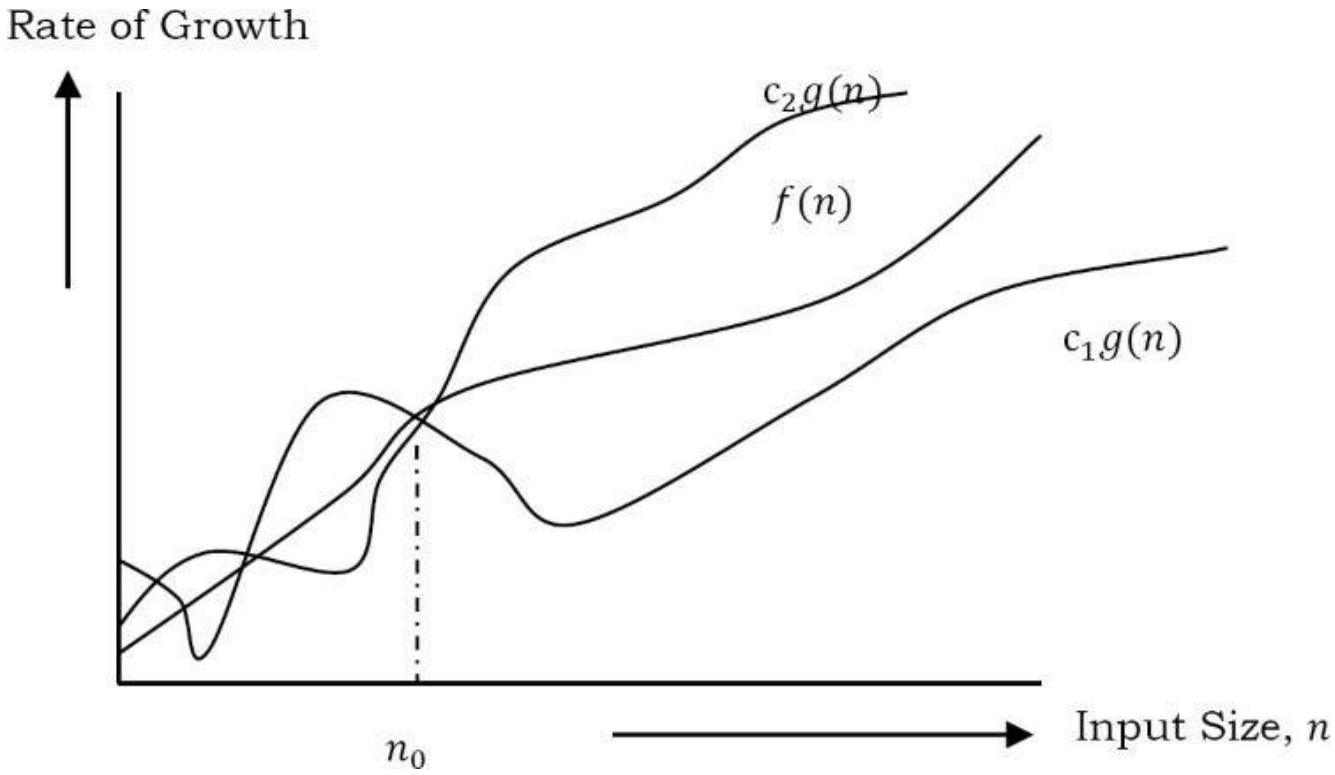
1.16 Theta-Θ Notation

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 $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

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* Θ(*g*(*n*)) = {*f*(*n*): there exist positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$}. *g*(*n*) is an asymptotic tight bound for *f*(*n*). Θ(*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: $\frac{n^2}{5} \le \frac{n^2}{2} - \frac{n}{2} \le n^2$, for all, $n \ge 2$

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

Example 2 Prove $n \ne \Theta(n^2)$

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

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

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

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

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

Example 4 Prove $n \ne \Theta(\log n)$

Solution: $c_1 log n \leq n \leq c_2 log n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ – 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 executed n times
      for (j=1; j<=n; j++)
            k = k+1; //constant time
}
```

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

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

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

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

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

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

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

5) **Logarithmic complexity:** An algorithm is O(*logn*) if it takes a constant time to cut the problem size by a fraction (usually by½). 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: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.24 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, $a > 0$, $b > 0$, $k \geq 0$, 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 $T(n) \leq cnlogn$:

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

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:

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

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 $n\sqrt{logn}$:

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

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 becomes 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.

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

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

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

.

.

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

This clearly shows that the complexity of this function is O(3^n).

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

Problem-22 Find the complexity of the below recurrence:

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

Solution: Let us try solving this function with substitution.

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

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

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

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

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

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

$$T(n) = 1$$

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

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

```
public void Function(int n) {
        int i=1, s=1;
        while( s <= n) {
                i++;
                s= s+i;
                System.out.println("*");
        }
}
```

Solution: Consider the comments in the below function:

```
public 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;
        System.out.println("*");
    }
}
```

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

```
public 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 $i^2 > n \Rightarrow T(n) = O(\sqrt{n})$. The reasoning is smae as that of Problem-23.

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

```
public 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++)
            for(k=1; k<=n; k= k * 2)
                count++;
}
```

Solution: Consider the comments in the following function.

```
public 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++)
            //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?

```
public 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.

```
public 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 given below.

```
public  void function( int n ) {
        if(n == 1) return;
        for(int i = 1 ; i <= n ; i + + ) {
                for(int j= 1 ; j <= n ; j + + ) {
                        System.out.println("*" );
                        break;
                }
        }
}
```

Solution: Consider the comments in the following function.

```
public  void 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 + + )        {
                  System.out.println("*" );
                  break;
            }
      }
}
```

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

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

```
public  void function( int n ) {
            if( n == 1 ) return;
            for(int i = 1 ; i <= n ; i + + )
                  for(int j = 1 ; j <= n ; j + + )
                        System.out.println("*" ) ;
            function( n-3 );
      }
```

Solution: Consider the comments in the function below:

```
public void 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
         System.out.println("*" ) ;
    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:

$$T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \leq k * n \text{ , where } k \text{ is a constant.}$$

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 $\Omega(logn)$.

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

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?

```
public  void function(n) {
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j+ = i )
                        System.out.println("*") ;
}
```

Solution: Consider the comments in the function below:

```
public void 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 )
            System.out.println("*") ;
}
```

In the above program, the 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.

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

Solution: Consider the comments in the function below:

```
public void 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) = Θ(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.

```
public void function(int n) {
        if(n <= 1) return;
        for (int i=1 ; i <= 3 ; i++ )
                function (n − 1).
}
```

Solution: Consider the comments in the below function:

```
public void 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 f whose code is given below. What is the running time of *function*, as a function of n?

```
public void function (int n) {
        if(n <= 1)  return;
        for(int i = 1; i < n; i + +)
                System.out.println("*");
        function ( 0.8n ) ;
}
```

Solution: Consider the comments in the below function:

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

The recurrence for this piece of code is $T(n) = T(.8n) + O(n) = T\left(\frac{4}{5}n\right) + O(n) = \frac{4}{5}T(n) + O(n)$. 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, $logn = mlog2 \Rightarrow m = logn$. Now, the given function becomes:

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

To make it simple we assume $S(m) = T(2^m) \Longrightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Longrightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$. Applying the master theorem format would result in $S(m) = O(mlogrn)$. 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 $S(m) = O\left(m^{log_2^2}\right) = O(m)$. Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the function given below.

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

Solution: Consider the comments in the below function:

```
public 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 function, 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 pseuedocode as a function of *n*.

```
public 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 pseuedocode and call running time of function(n) as $T(n)$.

```
public 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 pseudocode given below:

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

Solution: Consider the comments in the pseudocode given below:

```
temp = 1        // constant 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?

```
publicvoid  function(int n) {
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j * = 2 )
                        System.out.println("*");
}
```

Solution: Consider the comments in the function given below:

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

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

Problem-47 Running time of the following program?

```
public void function(int n) {
        for(int i = 1 ; i <= n/3 ; i + + )
                for(int j = 1 ; j <= n ; j += 4 )
                        System.out.println(" * ");
}
```

Solution: Consider the comments in the function given below:

```
public void 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)
            System.out.println(" * ");
}
```

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

Problem-48 Find the complexity of the below function:

```
public void function(int n) {
    if(n <=  1) return;
    if(n > 1) {
            System.out.println(" * ");
            function( n/2 );
            function( n/2 );
        }
}
```

Solution: Consider the comments in the function given below:

```
void function(int n) {
    if(n <= 1) return;                  //constant time
    if(n > 1) {
            System.out.println(" * ");      //constant time
            //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:

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

Solution:

```
public void 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 $\Sigma_{1 \le k \le n} O(n)$, where O($n$) stands for order n is:

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

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

Problem-51 Which of the following three claims are correct

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

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

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

Problem-52 Consider the following functions: $f(n) = 2^n$ $g(n) = n!$ $h(n) = n^{logn}$

Which of the following statements about the asymptotic 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) = Ω (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) = Ω($\sqrt{n}$)
(B) T(n) = O($\sqrt{n}$) and T(n) = Ω(1)
(C) T(n) = O(n) and T(n) = Ω($\sqrt{n}$)
(D) None of the above

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

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

```
public 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) $\Theta(n)$

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(0)=T(1)=1. Which one of the following is FALSE?

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

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

Problem-57 Find the complexity of the below function:

```
public void 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:

```
public void 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 complexity of the below function:

```
public void 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;
    }
}
}
```

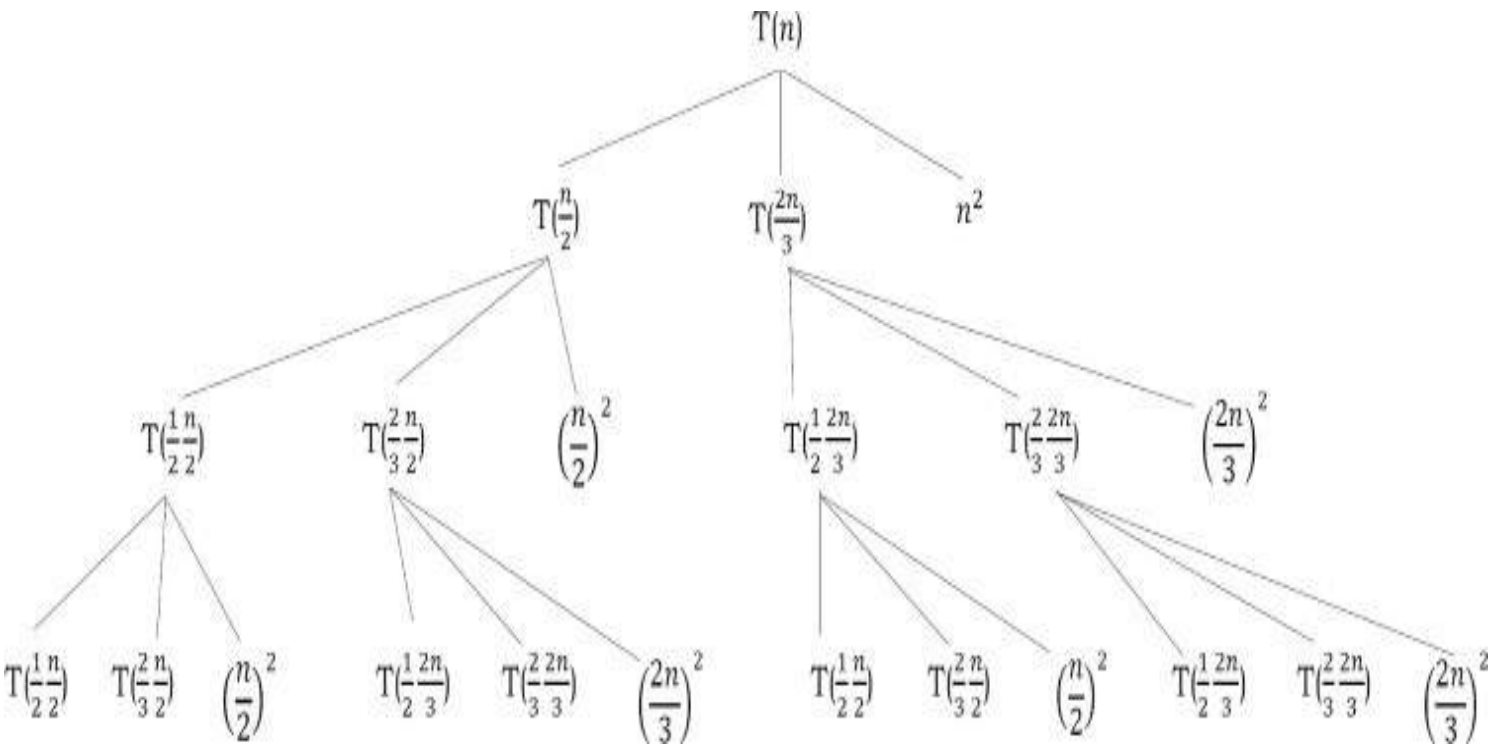
Solution: Consider the worst-case.

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

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 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 $c_1 n \leq T(i) \leq c_2 n$. Then,

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

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-63 Rank the following functions by order of growth: $(n+1)!, n!, 4^n, n \times 3^n, 3^n + n^2 + 20n, (\frac{3}{2})^n, 4n^2, 4^{lgn}, n^2 + 200, 20n + 500, 2^{lgn}, n^{2/3}, 1.$

Solution:

Function	Rate of Growth	
$(n+1)!$	$O(n!)$	
$n!$	$O(n!)$	
4^n	$O(4^n)$	
$n \times 3^n$	$O(n3^n)$	
$3^n + n^2 + 20n$	$O(3^n)$	
$(\frac{3}{2})^n$	$O((\frac{3}{2})^n)$	Decreasing rate of growths
$4n^2$	$O(n^2)$	
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-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 *cursive 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:

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

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* – *l*)!. In the recursive case, when *n* is greater than 1, the function calls itself to determine the value of (n –*l*)! 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:

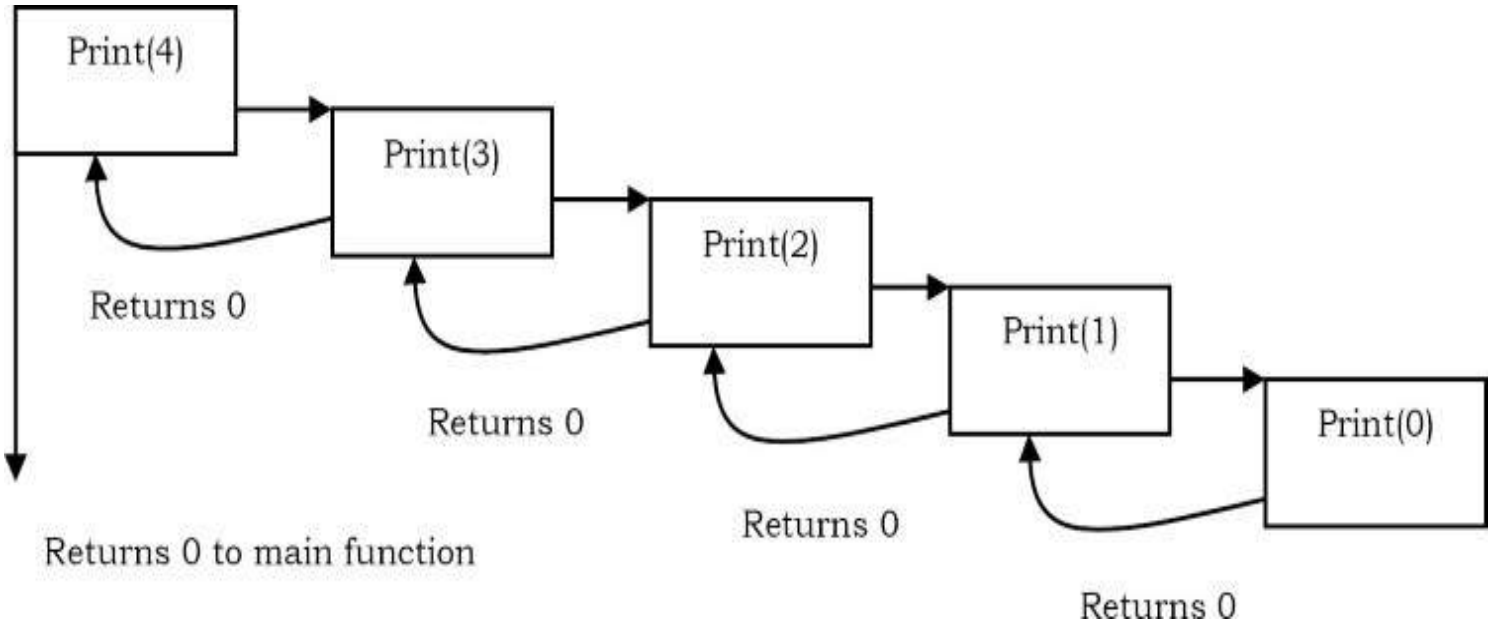
```
// calculates factorial of a positive integer
public int Factorial(int n) {
    // base cases: fact of 0 is 1
    if(n == 0)
        return 1;
    // recursive case: multiply n by (n – 1) factorial
    else
        return n*Factorial(n-1);
}
```

2.5 Recursion and Memory (Visualization)

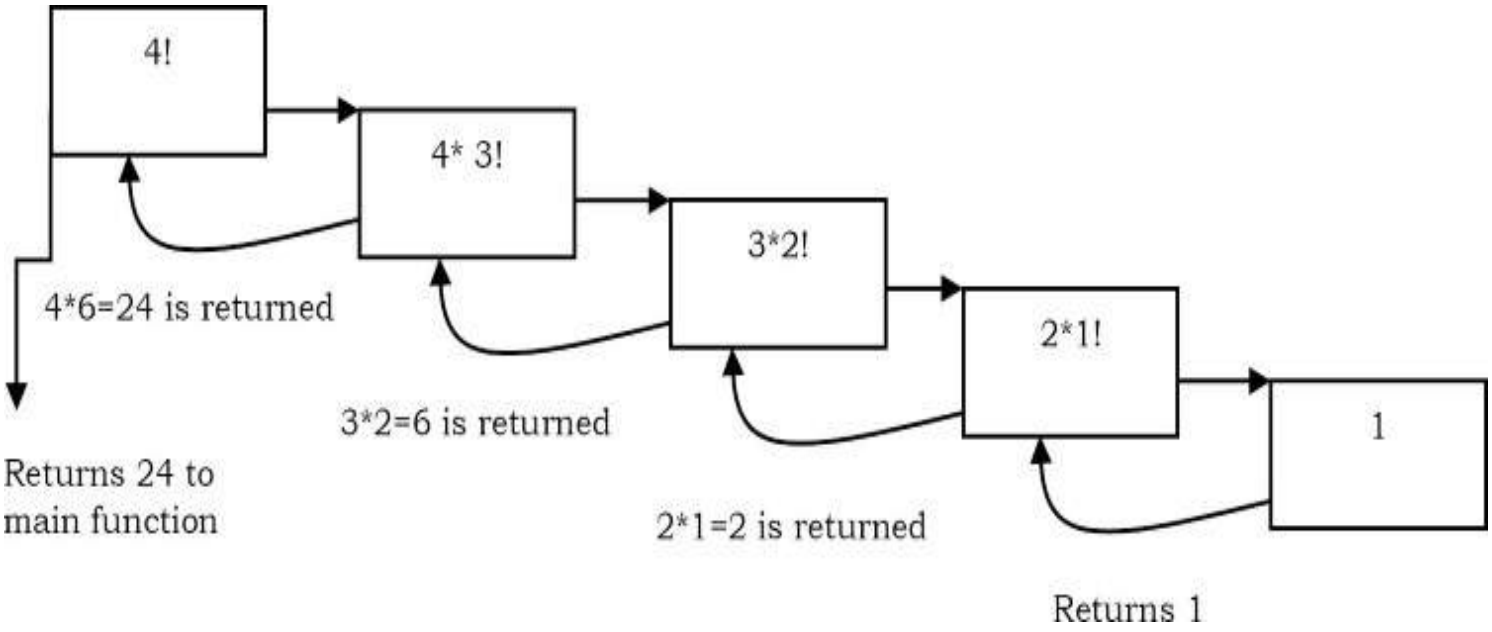
Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes time. For better understanding,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

```
public int Print(int n) {
    if( n = = 0)                    // this is the terminating base case
        return 0;
    else {
        System.out.println(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 (needs 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 any 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.

```
public void TowersOfHanoi(int n, char frompeg, char topeg, char auxpeg) {
    /* If only 1 disk, make the move and return */
    if(n==1) {
            System.out.println("Move disk 1 from peg " + frompeg + " to peg " + 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 */
     System.out.println("Move disk from peg" + frompeg + " to peg " + 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:

```
public int isArrayInSortedOrder(int[] A, int index){
    if(A.length() == 1)
        return 1;
    return (A[index -1] < A[index -2])?0:isArrayInSortedOrder(A, index -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
- The Knapsack Problem
- N-Queens 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:

```
public void Binary(int n) {
    if(n < 1 )
        System.out.println(A);              //Assume array A is a class 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 *System.out.println* 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):

```
public void k-string(int n, int k) {
    //process all k-ary strings of length m
    if(n < 1 )
        System.out.println(A);              //Assume array A is a class 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 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

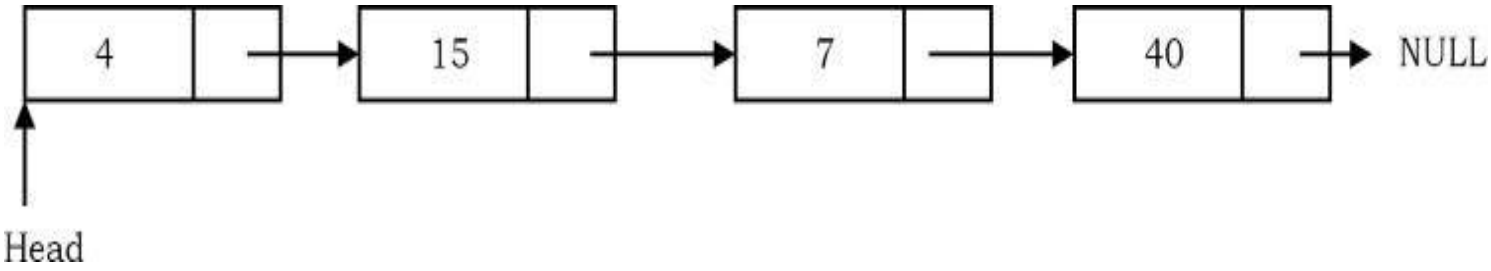
3

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

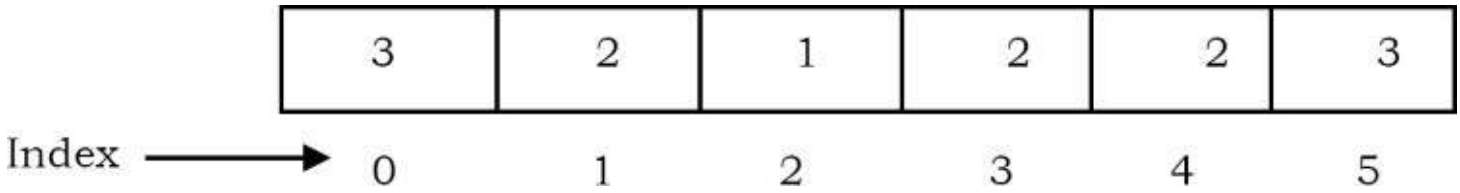
- Delete List: removes all elements of the list (disposees the list)
- Count: returns the number of elements in the list
- Find n^{th} node from the end of the list

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 *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 the size.

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.

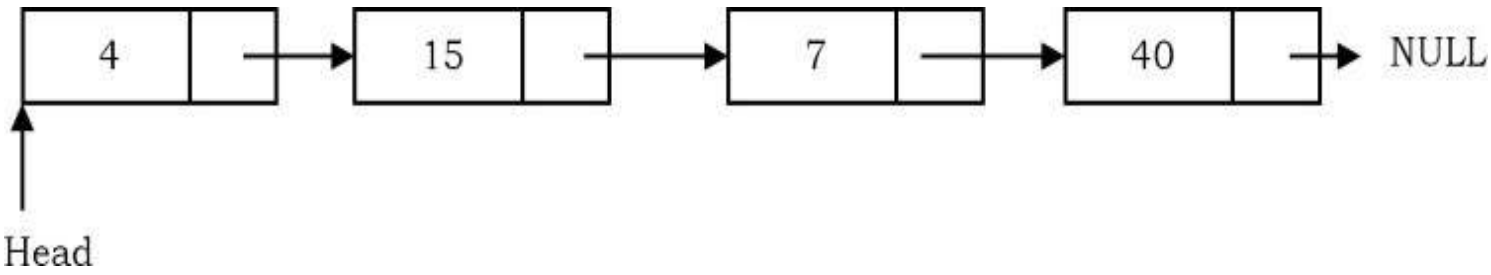
Finally, linked lists waste memory in terms of extra reference points.

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:

```
public class ListNode {
        private int data;
        private ListNode next;
        public ListNode(int data){
                this.data = data;
        }
        public void setData(int data){
                this.data = data;
        }
        public int getData(){
                return data;
        }
        public void setNext(ListNode next){
                this.next = next;
        }
        public ListNode getNext(){
                return this.next;
        }
}
```

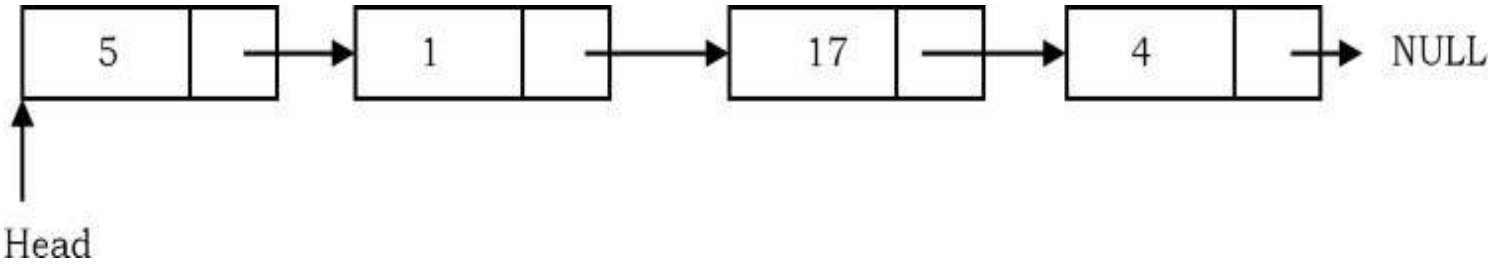
Basic Operations on a List

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

Traversing the Linked List

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

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



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

```
public int ListLength(ListNode headNode) {
        int length = 0;
        ListNode currentNode = headNode;
        while(currentNode != null){
                length++;
                currentNode = currentNode.getNext();
        }
        return length;
}
```

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:

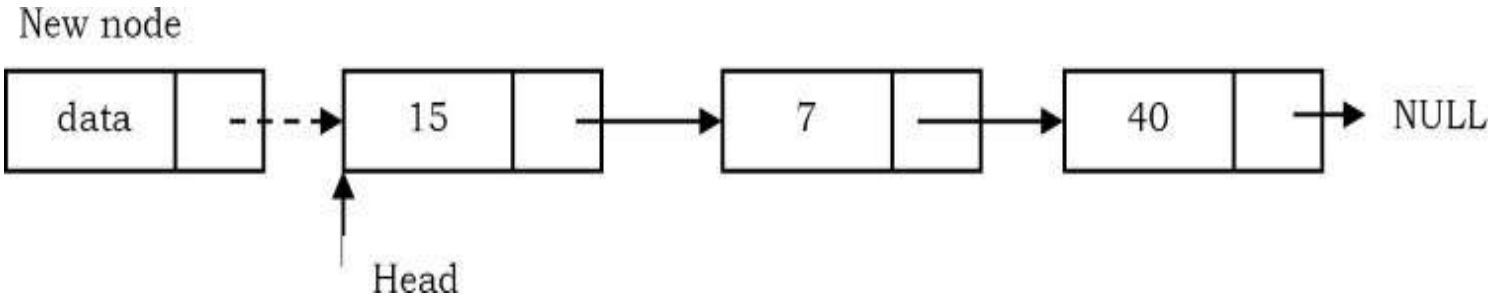
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

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*.

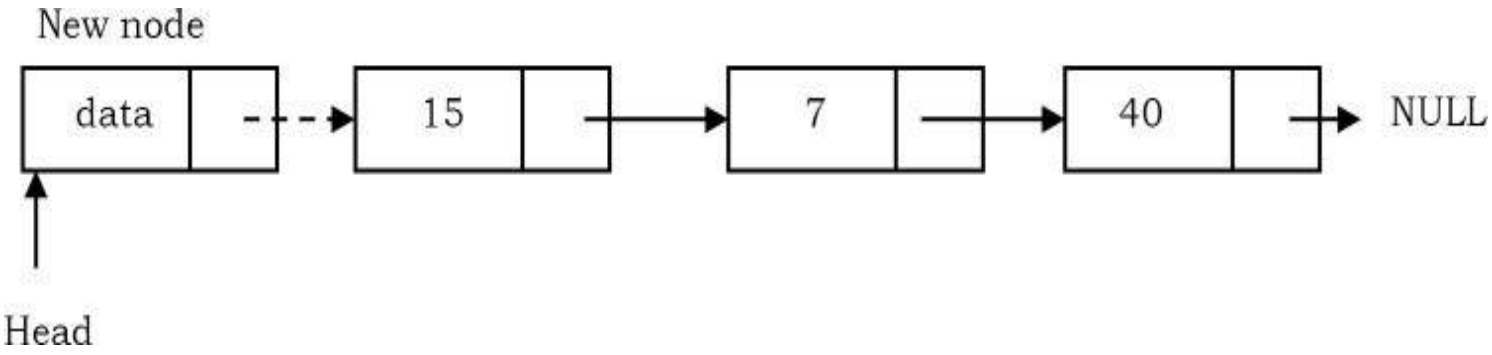
Inserting a Node in Singly Linked List at the Beginning

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

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



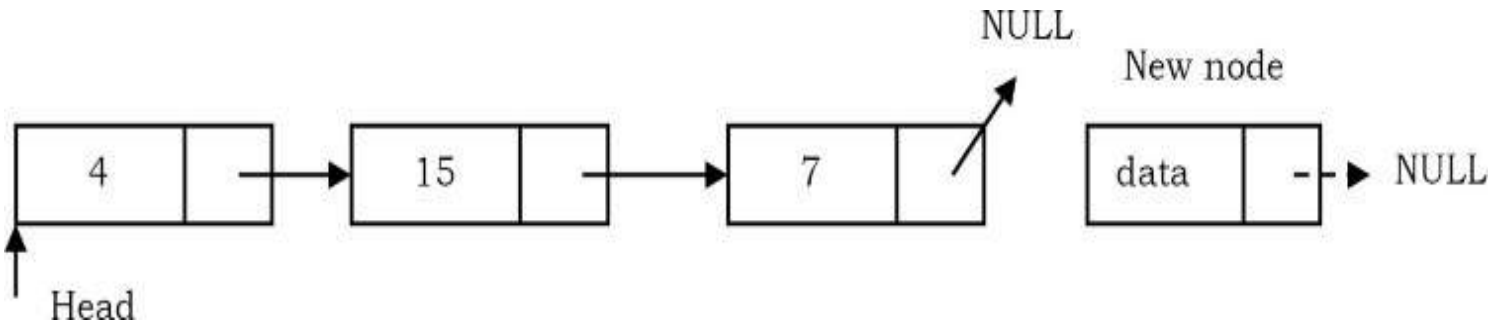
- Update head pointer to point to the new node.



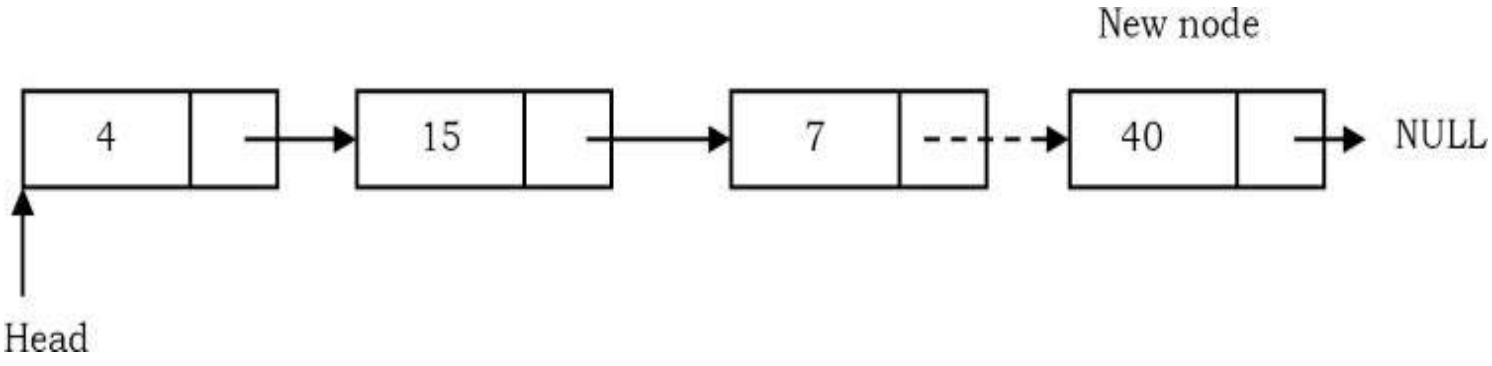
Inserting a Node in Singly Linked List at the Ending

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

- New nodes next pointer points to NULL.



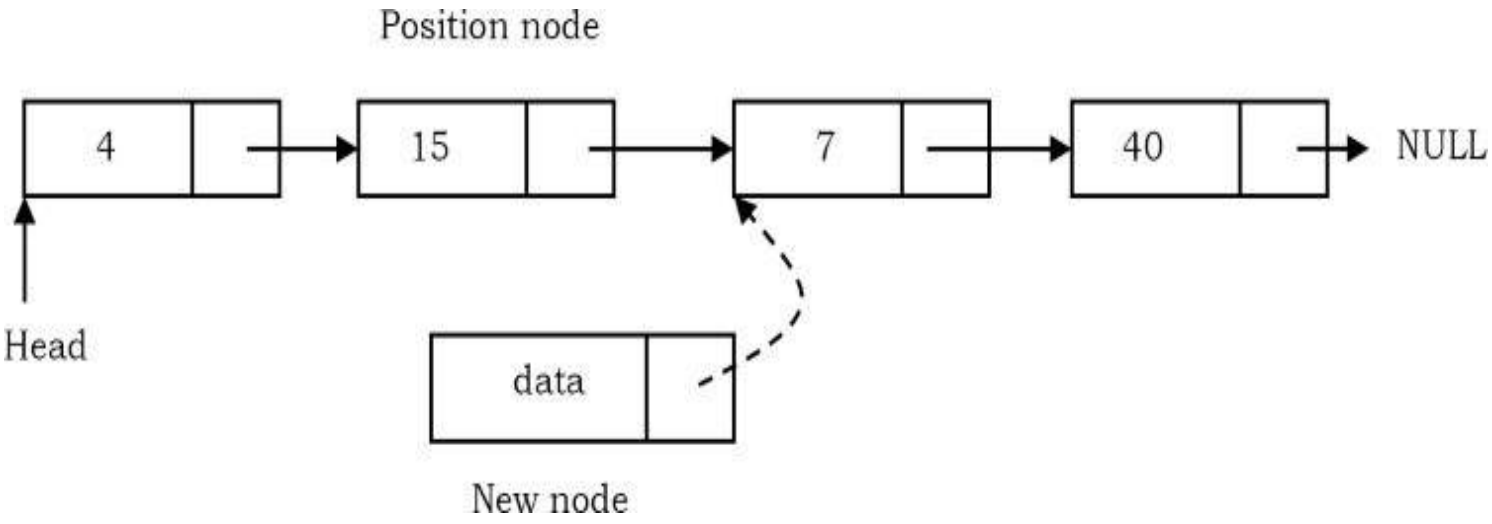
- Last nodes next pointer points to the new node.



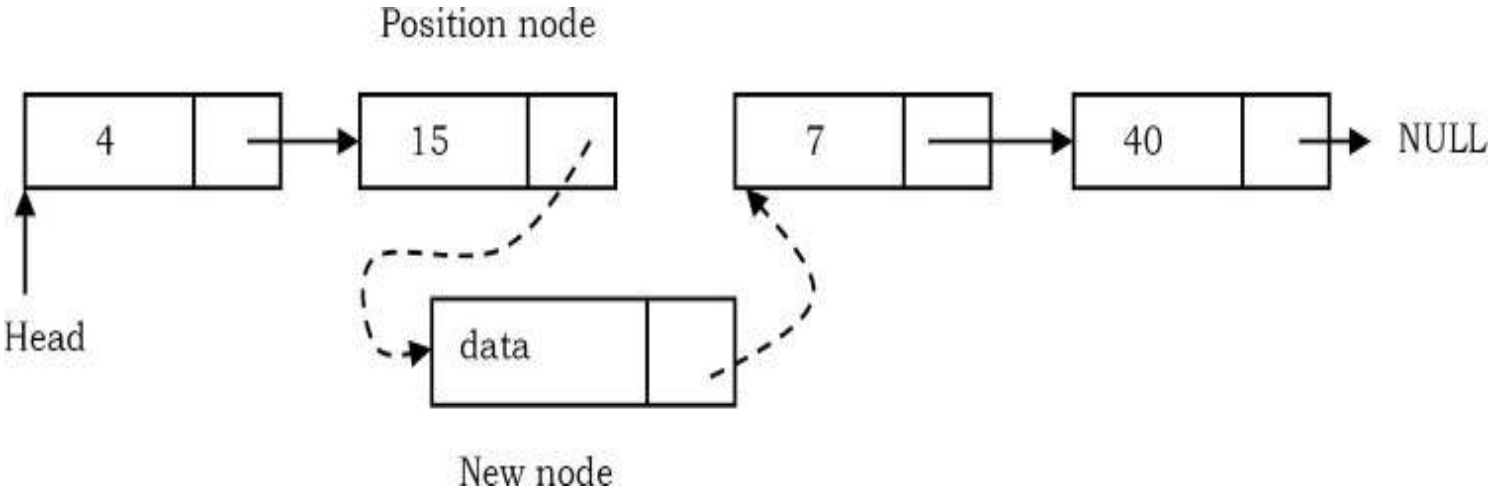
Inserting a Node in Singly Linked List at the Middle

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

- If we want to add an element at position 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.



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

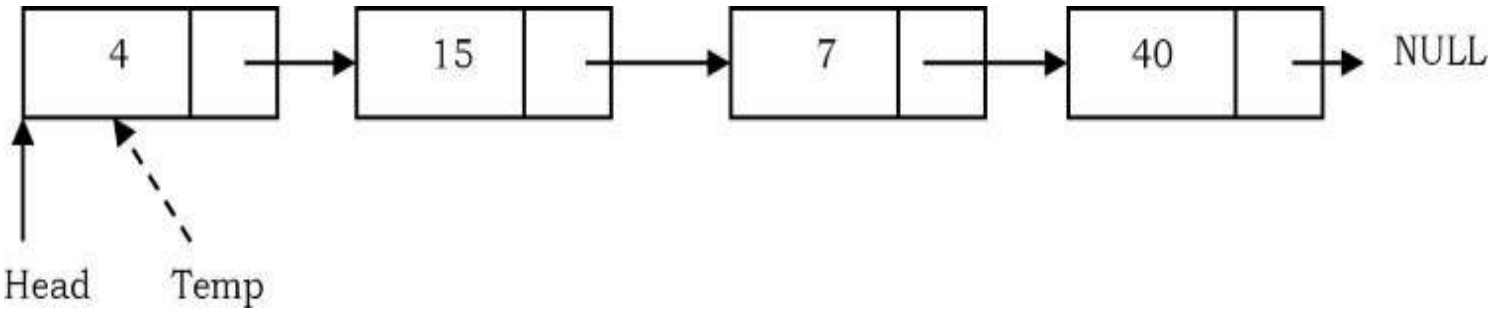
Similar to insertion, here we also have three cases.

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

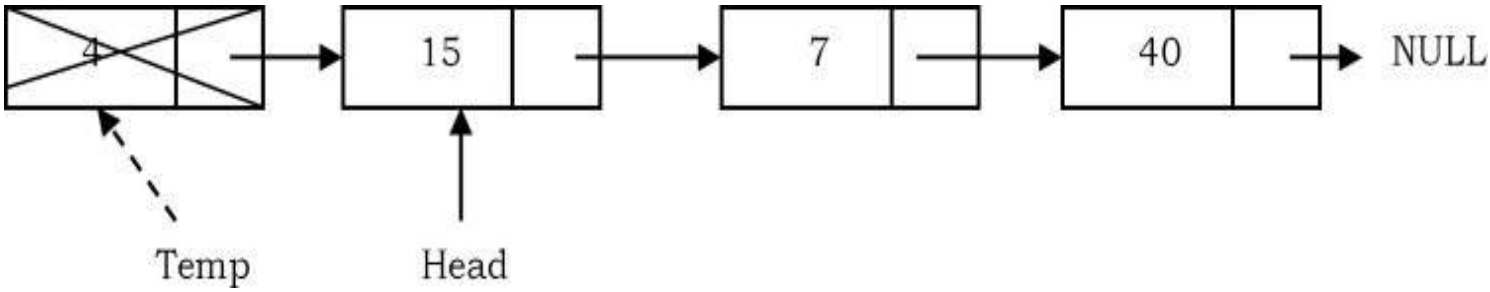
Deleting the First Node in Singly Linked List

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.



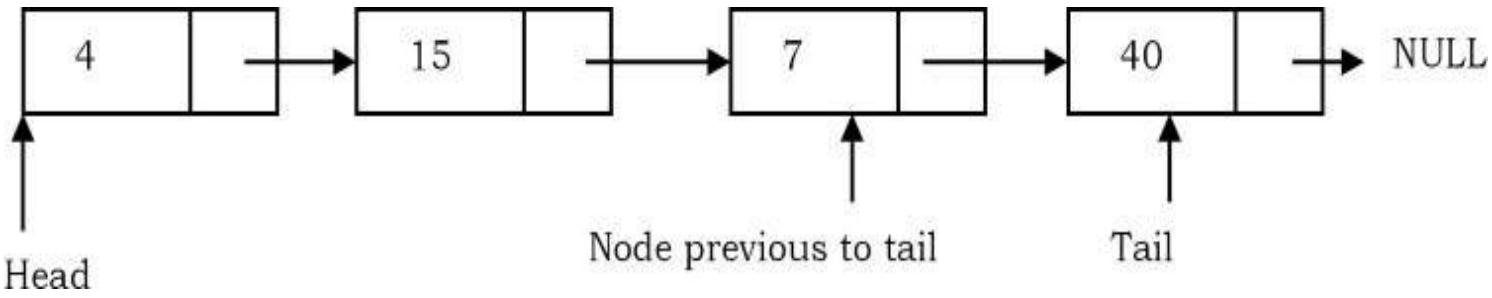
- Now, move the head nodes pointer to the next node and dispose of the temporary node.



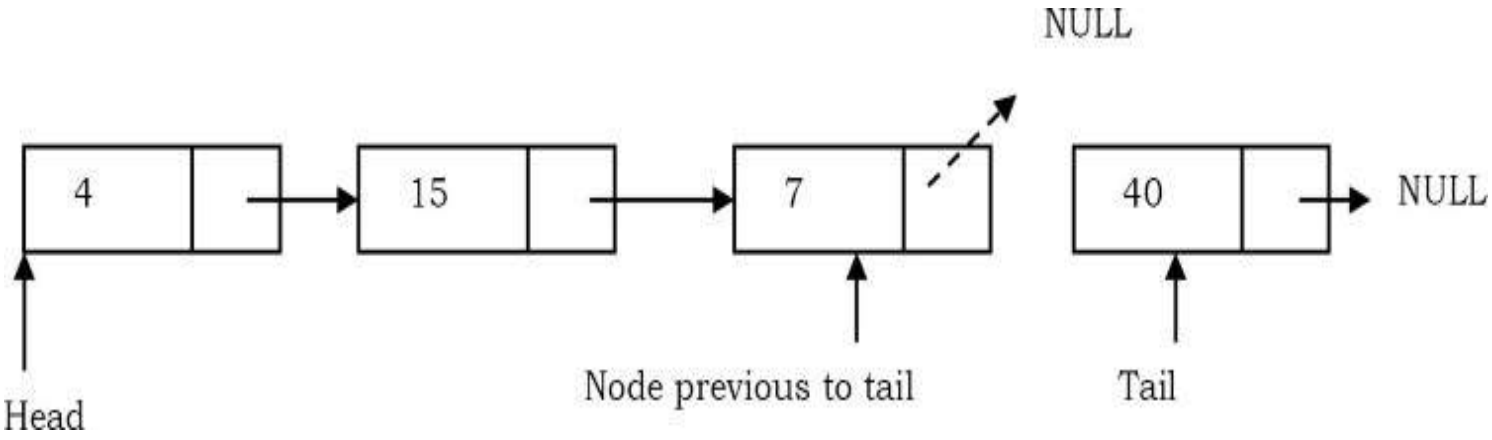
Deleting the Last node in Singly Linked List

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:

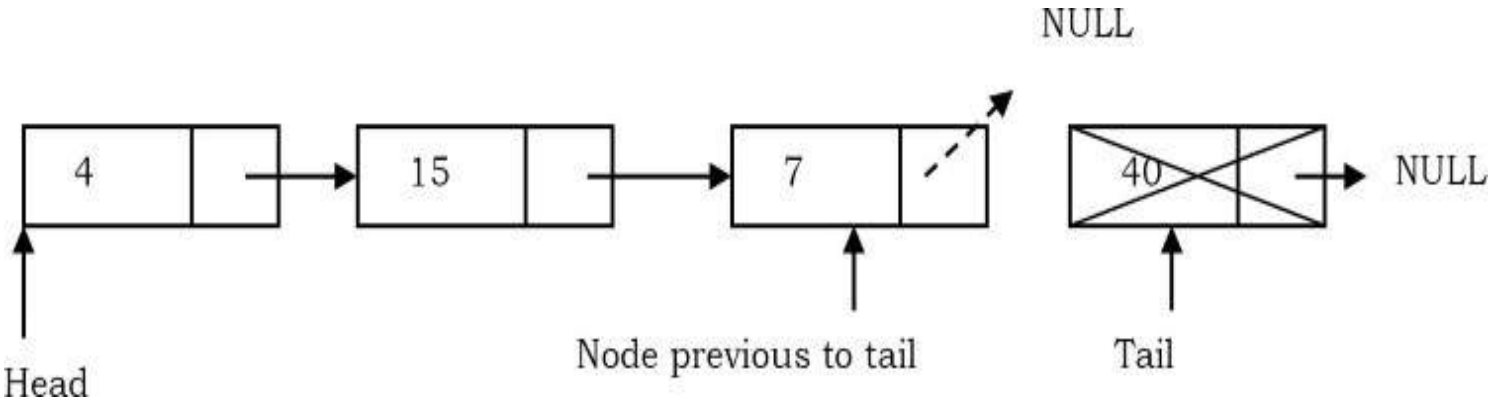
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.



- Update previous node's next pointer with NULL.



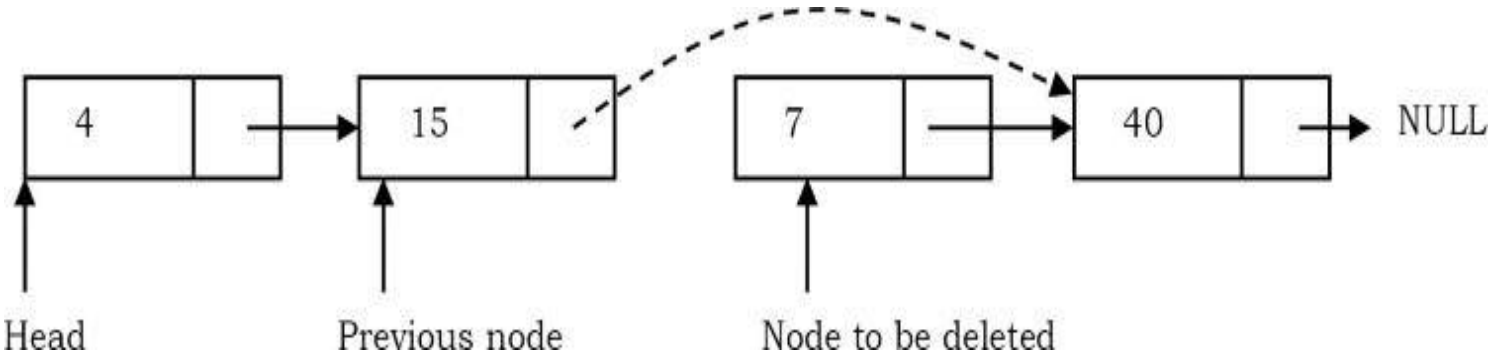
- Dispose of the tail node.



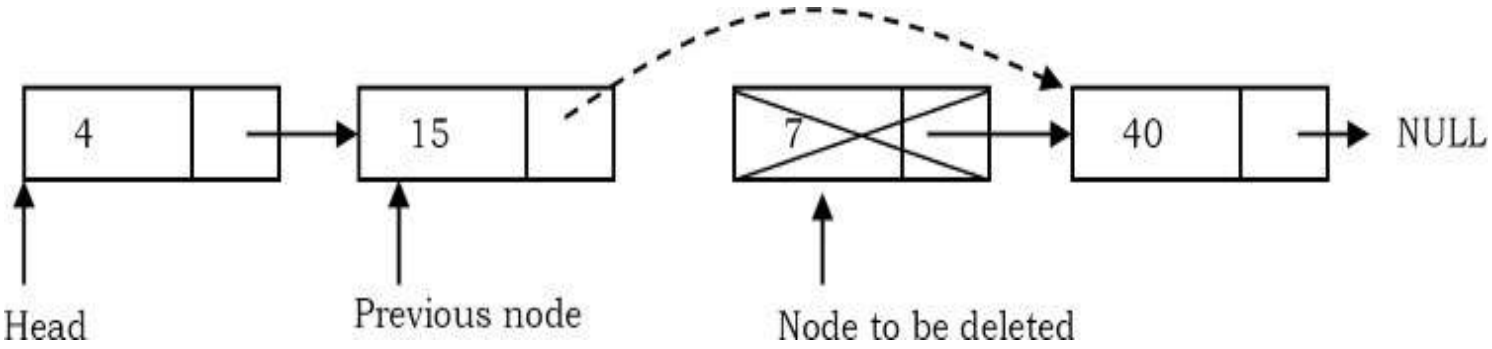
Deleting an Intermediate Node in Singly Linked List

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.



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). Since, we are creating only one temporary variable.

Deleting Singly Linked List

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

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

Implementation

```
public class LinkedList {
    // This class has a default constructor:
    public LinkedList() {
        length = 0;
    }
    // This is the only field of the class.  It holds the head of the list
    ListNode head;
    // Length of the linked list
    private int length = 0;
    // Return the first node in the list
    public synchronized ListNode getHead() {
        return head;
    }
    // Insert a node at the beginning of the list
    public synchronized void insertAtBegin(ListNode node) {
        node.setNext(head);
        head = node;
        length ++;
    }
    // Insert a node at the end of the list
    public synchronized void insertAtEnd(ListNode node) {
        if (head == null)
            head = node;
        else {
            ListNode p, q;
```

```
                for(p = head; (q = p.getNext()) != null; p = q);
                        p.setNext(node);
        }
        length ++;
}
// Add a new value to the list at a given position.
// All values at that position to the end move over to make room.
public void insert(int data, int position) {
        // fix the position
        if (position < 0) {
                position = 0;
        }
        if (position > length) {
                position = length;
        }
        // if the list is empty, make it be the only element
        if (head == null) {
                head = new ListNode(data);
        }
        // if adding at the front of the list...
        else if (position == 0) {
                ListNode temp = new ListNode(data);
                temp.next = head;
                head = temp;
        }
        // else find the correct position and insert
        else {
                ListNode temp = head;
                for (int i=1; i<position; i+=1) {
                        temp = temp.getNext();
                }
                ListNode newNode = new ListNode(data);
                newNode.next = temp.next;
                temp.setNext(newNode);
        }
        // the list is now one value longer
        length += 1;
}
// Remove and return the node at the head of the list
public synchronized ListNode removeFromBegin() {
        ListNode node = head;
        if (node != null) {
                head = node.getNext();
                node.setNext(null);
        }
        return node;
}
// Remove and return the node at the end of the list
public synchronized ListNode removeFromEnd() {
        if (head == null)
                return null;
        ListNode p = head, q = null, next = head.getNext();
        if (next == null) {
                head = null;
                return p;
        }
        while((next = p.getNext()) != null) {
                q = p;
                p = next;
        }
        q.setNext(null);
        return p;
}
```

```
// Remove a node matching the specified node from the list.
// Use equals() instead of == to test for a matched node.
public synchronized void removeMatched(ListNode node) {
        if (head == null)
                return;
        if (node.equals(head)) {
                head = head.getNext();
                return;
        }
        ListNode p = head, q = null;
        while((q = p.getNext()) != null) {
                if (node.equals(q)) {
                        p.setNext(q.getNext());
                        return;
                }
                p = q;
        }
}
// Remove the value at a given position.
// If the position is less than 0, remove the value at position 0.
// If the position is greater than 0, remove the value at the last position.
public void remove(int position) {
        // fix position
        if (position < 0) {
                position = 0;
        }
        if (position >= length) {
                position = length-1;
        }
        // if nothing in the list, do nothing
        if (head == null)
                return;
        // if removing the head element...
        if (position == 0) {
                head = head.getNext();
        }
        // else advance to the correct position and remove
        else {
                ListNode temp = head;
                for (int i=1; i<position; i+=1) {
                        temp = temp.getNext();
                }
                temp.setNext(temp.getNext().getNext());
        }
        // reduce the length of the list
        length -= 1;
}
// Return a string representation of this collection, in the form ["str1","str2",...].
public String toString() {
        String result = "[";
        if (head == null) {
                return result+"]";
        }
        result = result + head.getData();
        ListNode temp = head.getNext();;
        while (temp != null) {
                result = result + "," + temp.getData();
                temp = temp.getNext();
        }
        return result + "]";
}
```

```
    // Return the current length of the list.
    public int length() {
        return length;
    }
    // Find the position of the first value that is equal to a given value.
    // The equals method us used to determine equality.
    public int getPosition(int data) {
        // go looking for the data
        ListNode temp = head;
        int pos = 0;
        while (temp != null) {
            if (temp.getData() == data) {
                // return the position if found
                return pos;
            }
            pos += 1;
            temp = temp.getNext();
        }
        // else return some large value
        return Integer.MIN_VALUE;
    }
    // Remove everything from the list.
    public void clearList(){
        head = null;
        length = 0;
    }
}
```

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:

```
public class DLLNode {
        private int data;
        private DLLNode prev;
        private DLLNode next;
        public DLLNode(int data) {
                this.data = data;
                prev = null;
                next = null;
        }
        public DLLNode(int data, DLLNode prev, DLLNode next) {
                this.data = data;
                this.prev = prev;
                this.next = next;
        }
        public int getData() {
                return data;
        }
        public void setData(int data) {
                this.data = data;
        }
        public DLLNode getPrev() {
                return prev;
        }
        public DLLNode getNext() {
                return next;
        }
        public void setPrev(DLLNode where) {
                prev = where;
        }
        public void setNext(DLLNode where) {
                next = where;
        }
}
```

Doubly Linked List Insertion

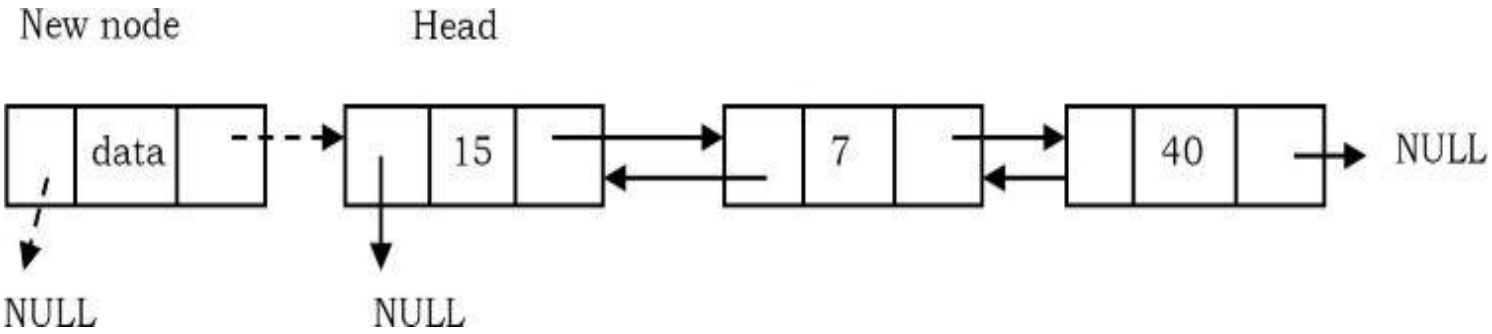
Insertion into a doubly-linked list has three cases (same as a singly linked list).

- 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.

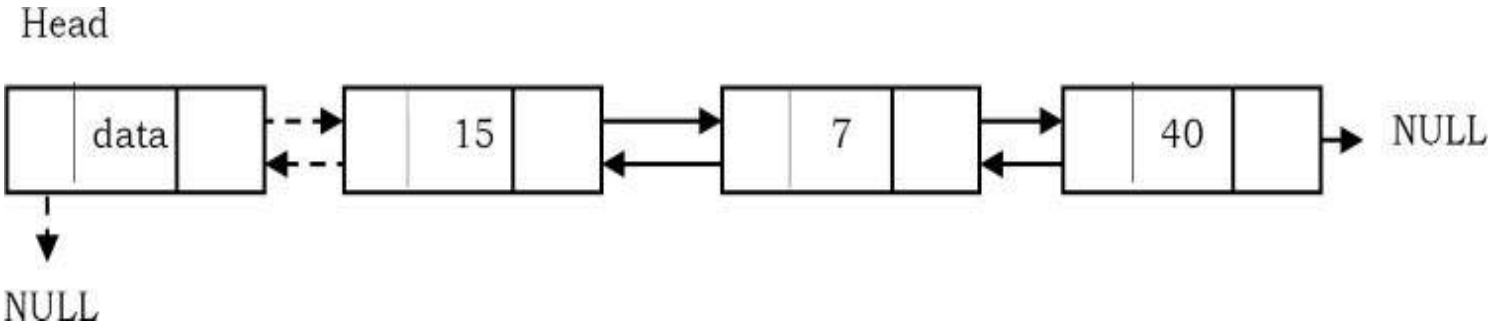
Inserting a Node in Doubly Linked List at the Beginning

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.



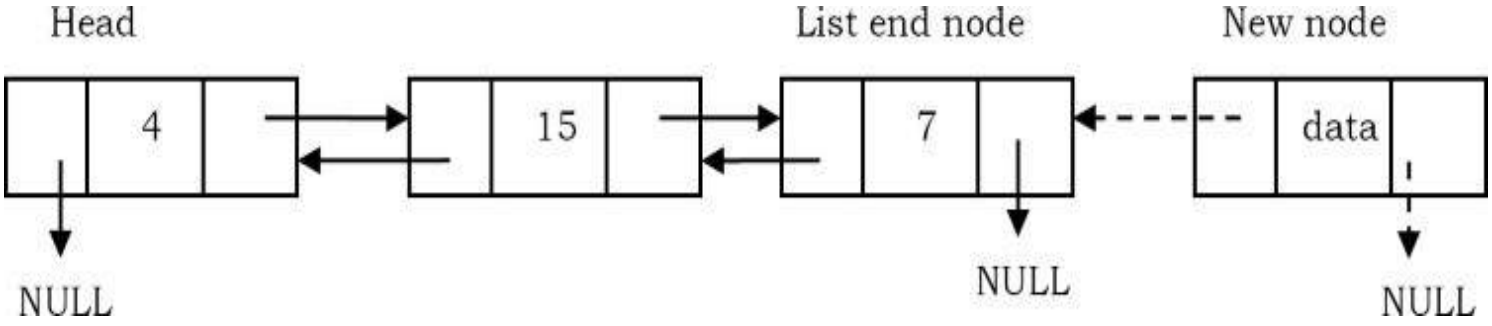
- Update head node's left pointer to point to the new node and make new node as head.



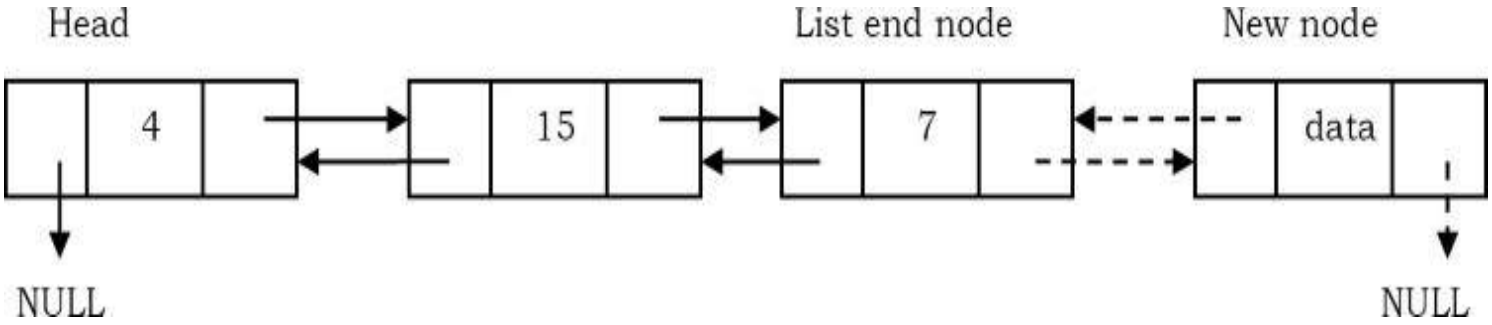
Inserting a Node in Doubly Linked List at the Ending

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

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



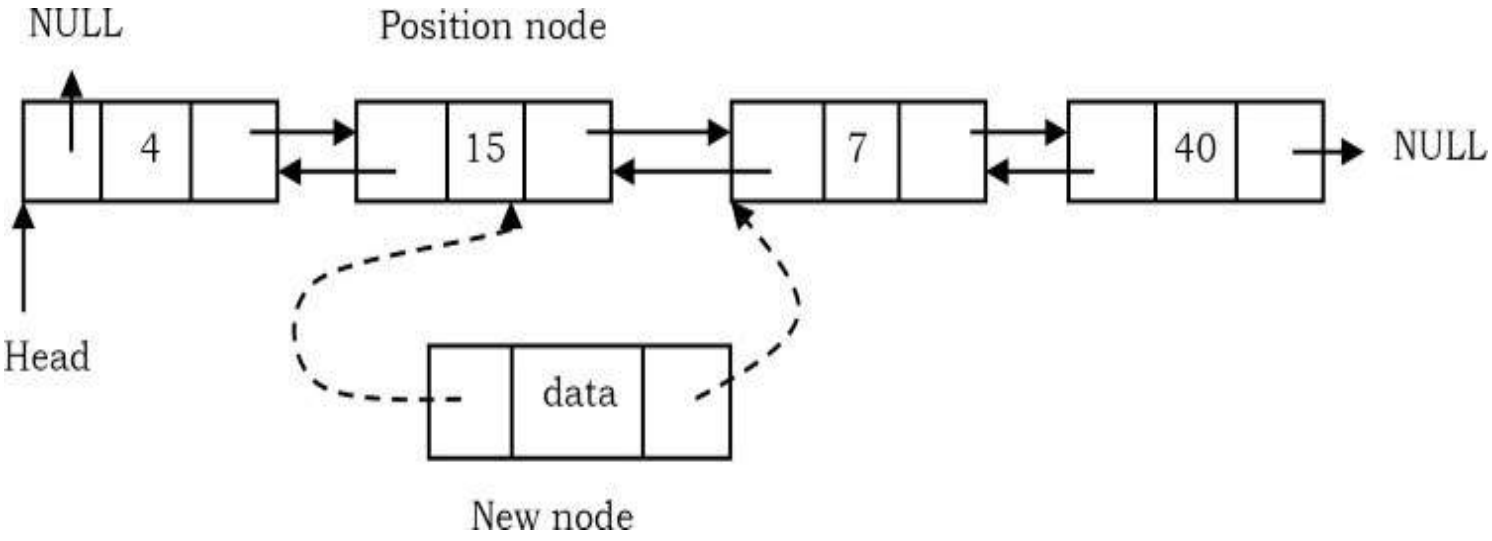
- Update right pointer of last node to point to new node.



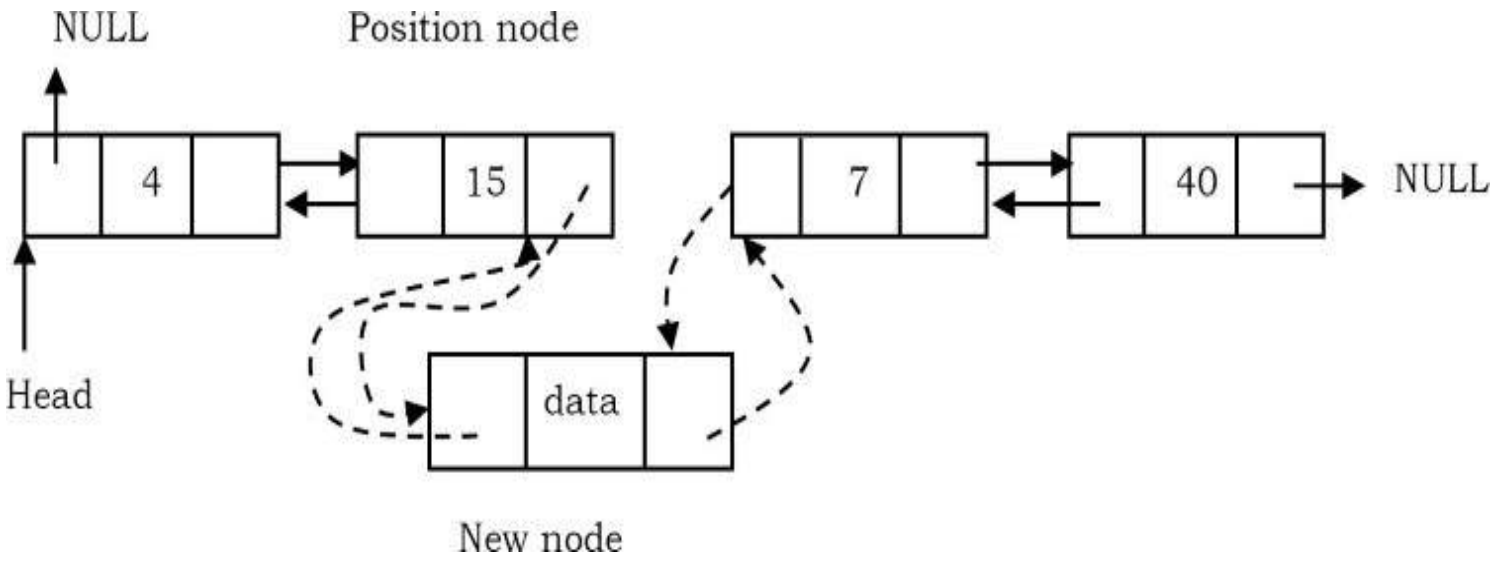
Inserting a Node in Doubly Linked List in the Middle

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.



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

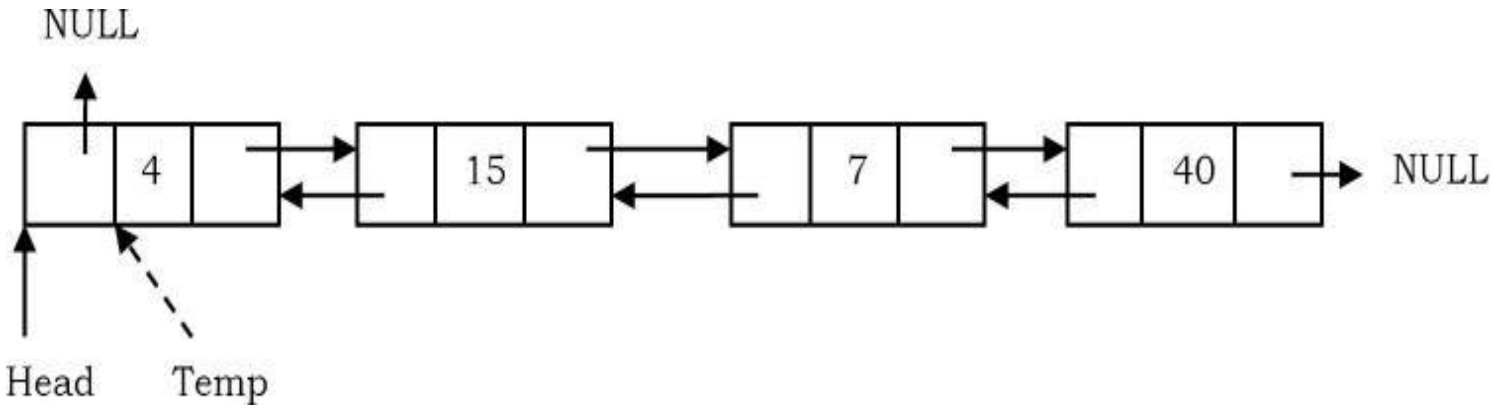
Similar to singly linked list deletion, here we have three cases:

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

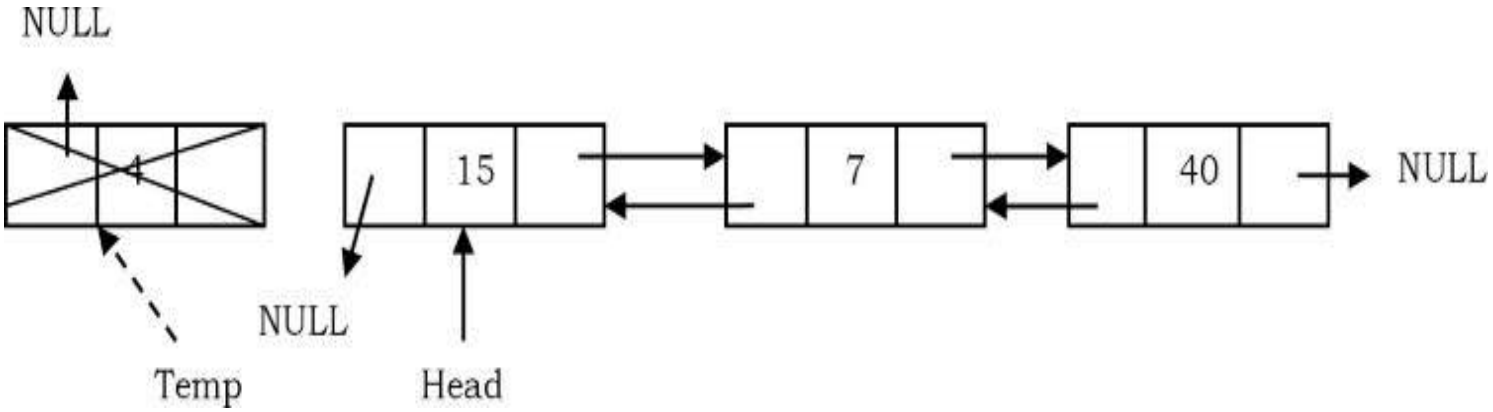
Deleting the First Node in Doubly Linked List

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.



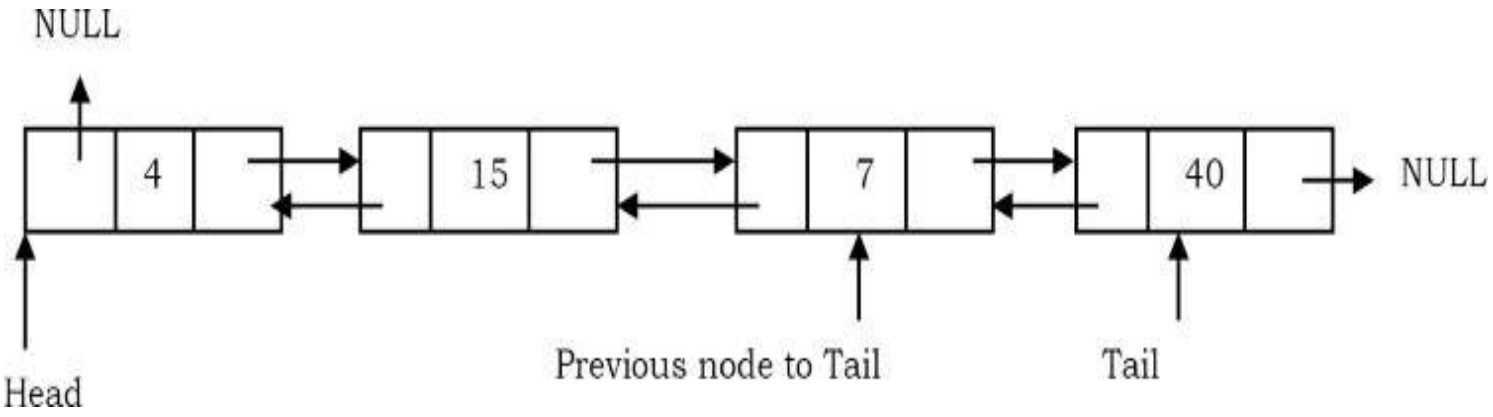
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL and dispose of the temporary node.



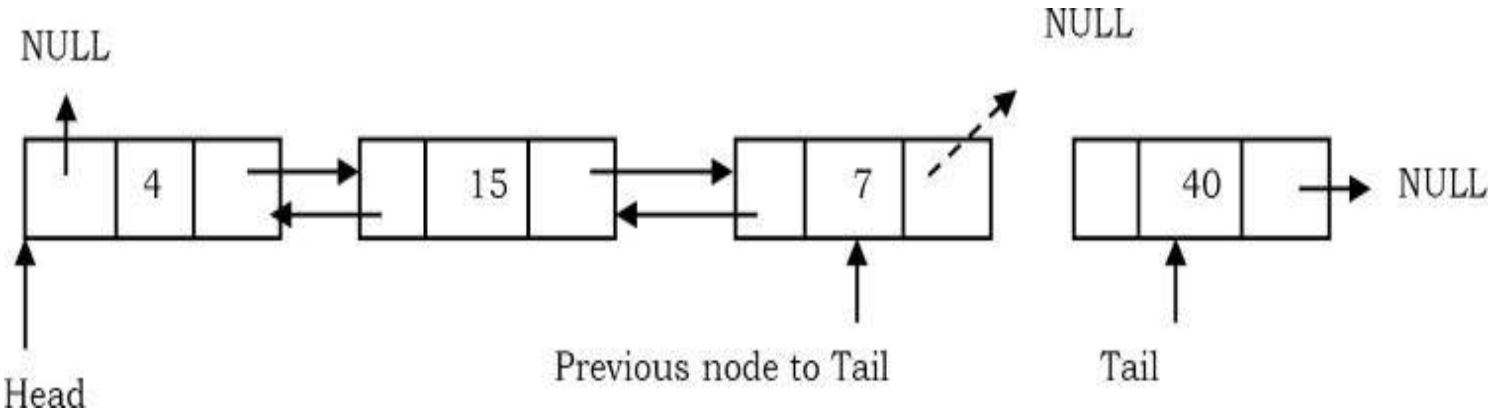
Deleting the Last Node in Doubly Linked List

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:

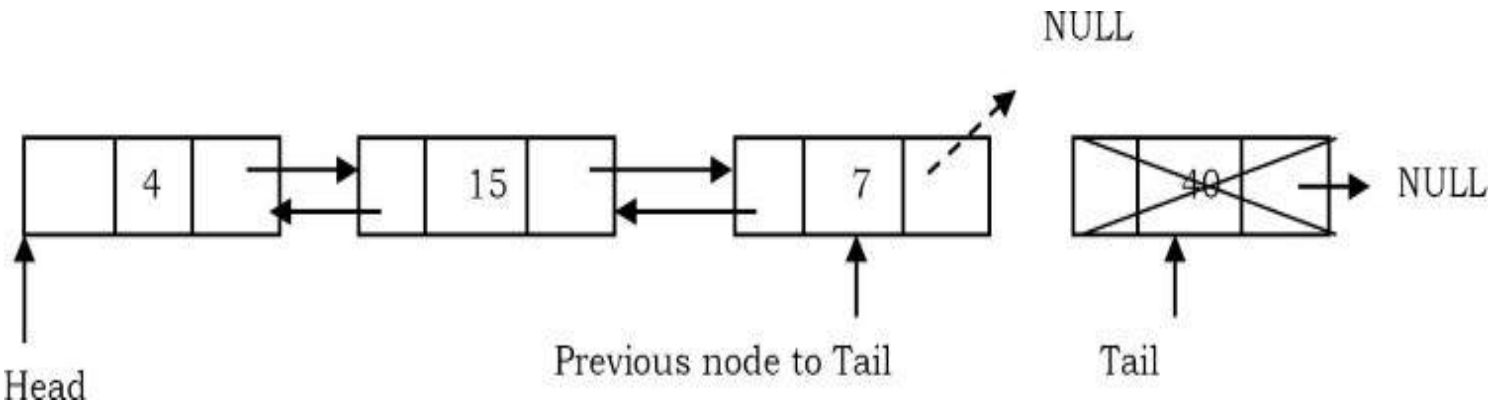
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.



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



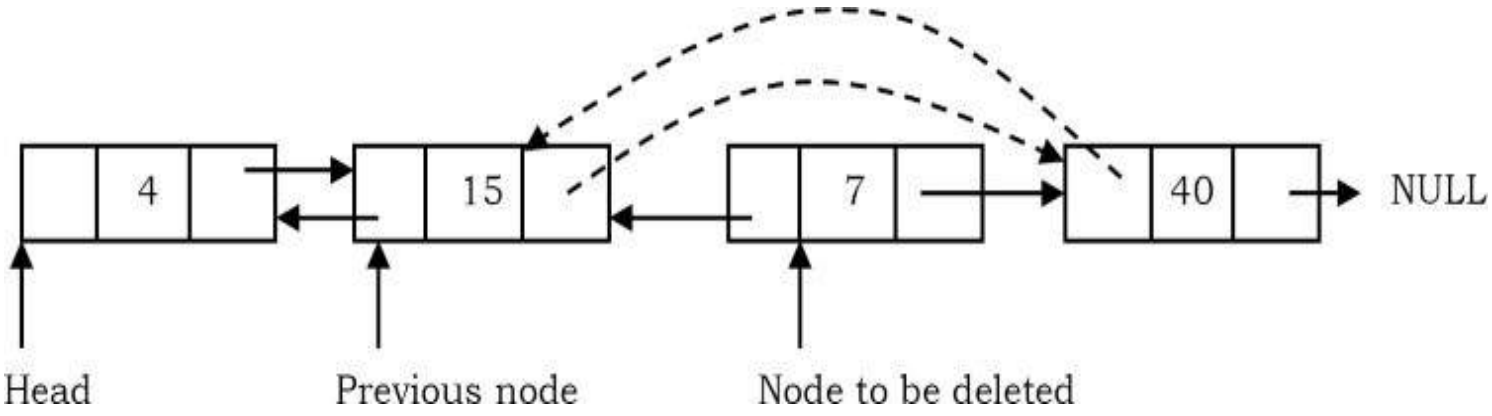
- Dispose of the tail node.



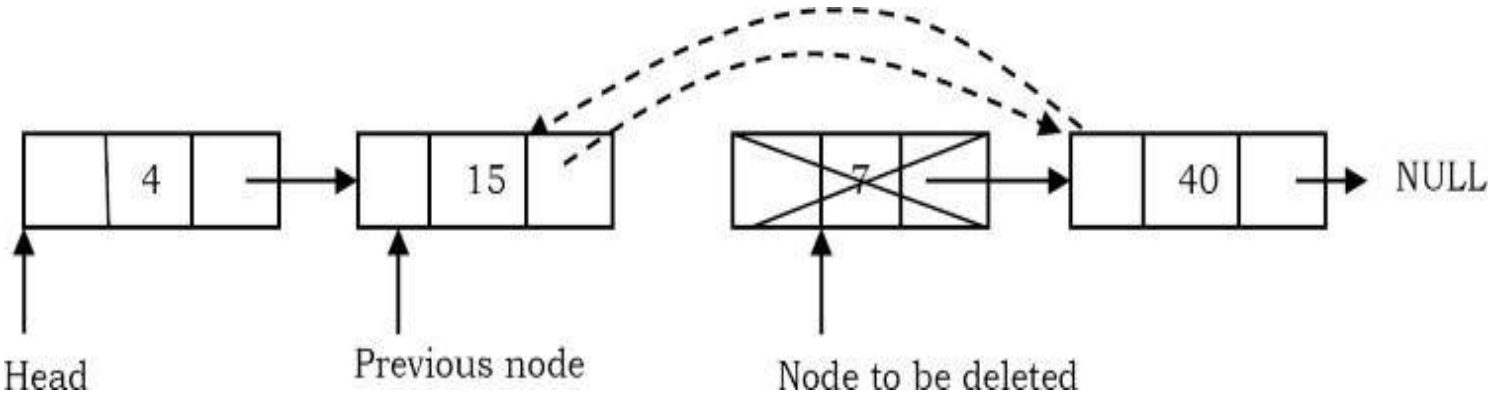
Deleting an Intermediate Node in Doubly Linked List

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.



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

Implementation

```
public class DoublyLinkedList{
	// properties
	private DLLNode head;
	private DLLNode tail;
	private int length;
	// Create a new empty list.
	public DoublyLinkedList() {
		head = new DLLNode(Integer.MIN_VALUE,null,null);
		tail = new DLLNode(Integer.MIN_VALUE, head, null);
		head.setNext(tail);
		length = 0;
	}
	// Get the value at a given position.
	public int get(int position) {
		return Integer.MIN_VALUE;
	}
	// Find the position of the head value that is equal to a given value.
	// The equals method us used to determine equality.
	public int getPosition(int data) {
		// go looking for the data
		DLLNode temp = head;
		int pos = 0;
		while (temp != null) {
			if (temp.getData() == data) {
				// return the position if found
				return pos;
			}
			pos += 1;
			temp = temp.getNext();
		}
		// else return some large value
		return Integer.MIN_VALUE;
	}
	// Return the current length of the DLL.
	public int length() {
		return length;
	}
	// Add a new value to the front of the list.
	public void insert(int newValue) {
```

```
            DLLNode newNode = new DLLNode(newValue, null, head.getNext());
            newNode.getNext().setPrev(newNode);
            head = newNode;
            length += 1;
    }
    // Add a new value to the list at a given position.
    // All values at that position to the end move over to make room.
    public void insert(int data, int position) {
            // fix the position
            if (position < 0) {
                    position = 0;
            }
            if (position > length) {
                    position = length;
            }
            // if the list is empty, make it be the only element
            if (head == null) {
                    head = new DLLNode(data);
                    tail = head;
            }
            // if adding at the front of the list...
            else if (position == 0) {
                    DLLNode temp = new DLLNode(data);
                    temp.next = head;
                    head = temp;
            }
            // else find the correct position and insert
            else {
                    DLLNode temp = head;
                    for (int i=1; i<position; i+=1) {
                            temp = temp.getNext();
                    }
                    DLLNode newNode = new DLLNode(data);
                    newNode.next = temp.next;
                    newNode.prev = temp;
                    newNode.next.prev = newNode;
                    temp.next = newNode;
            }
            // the list is now one value longer
            length += 1;
    }
    // Add a new value to the rear of the list.
    public void insertTail(int newValue) {
            DLLNode newNode = new DLLNode(newValue,tail.getPrev(),tail);
            newNode.getPrev().setNext(newNode);
            tail.setPrev(newNode);
            length += 1;
    }
    // Remove the value at a given position.
    // If the position is less than 0, remove the value at position 0.
    // If the position is greater than 0, remove the value at the last position.
    public void remove(int position) {
            // fix position
            if (position < 0) {
                    position = 0;
            }
            if (position >= length) {
                    position = length-1;
            }
            // if nothing in the list, do nothing
            if (head == null)
```

```
                return;
        // if removing the head element...
        if (position == 0) {
                head = head.getNext();
                if (head == null)
                        tail = null;
        }
        // else advance to the correct position and remove
        else {
                DLLNode temp = head;
                for (int i=1; i<position; i+=1) {
                        temp = temp.getNext();
                }
                temp.getNext().setPrev(temp.getPrev());
                temp.getPrev().setNext(temp.getNext());
        }
        // reduce the length of the list
        length -= 1;
}
// Remove a node matching the specified node from the list.
// Use equals() instead of == to test for a matched node.
public synchronized void removeMatched(DLLNode node) {
        if (head == null) return;
        if (node.equals(head)) {
                head = head.getNext();
                if (head == null)
                    tail = null;
                return;
        }
        DLLNode p = head;
        while(p != null) {
                if (node.equals(p)) {
                        p.prev.next = p.next;
                        p.next.prev = p.prev;
                        return;
                }
        }
}
// Remove the head value from the list. If the list is empty, do nothing.
public int removeHead() {
        if (length == 0)
           return Integer.MIN_VALUE;
        DLLNode save = head.getNext();
        head.setNext(save.getNext());
        save.getNext().setPrev(head);
        length -= 1;
        return save.getData();
}
// Remove the tail value from the list. If the list is empty, do nothing.
public int removeTail() {
        if (length == 0)
            return Integer.MIN_VALUE;
        DLLNode save = tail.getPrev();
        tail.setPrev(save.getPrev());
        save.getPrev().setNext(tail);
        length -= 1;
        return save.getData();
}
// Return a string representation of this collection, in the form: ["str1","str2",...].
public String toString() {
        String result = "[]";
        if (length == 0)
```

```
            return result;
        result = "[" + head.getNext().getData();
        DLLNode temp = head.getNext().getNext();
        while (temp != tail) {
            result += "," + temp.getData();
            temp =  temp.getNext();
        }
        return result + "]";
    }
    // Remove everything from the DLL.
    public void clearList(){
        head = null;
        tail = null;
        length = 0;
    }
}
```

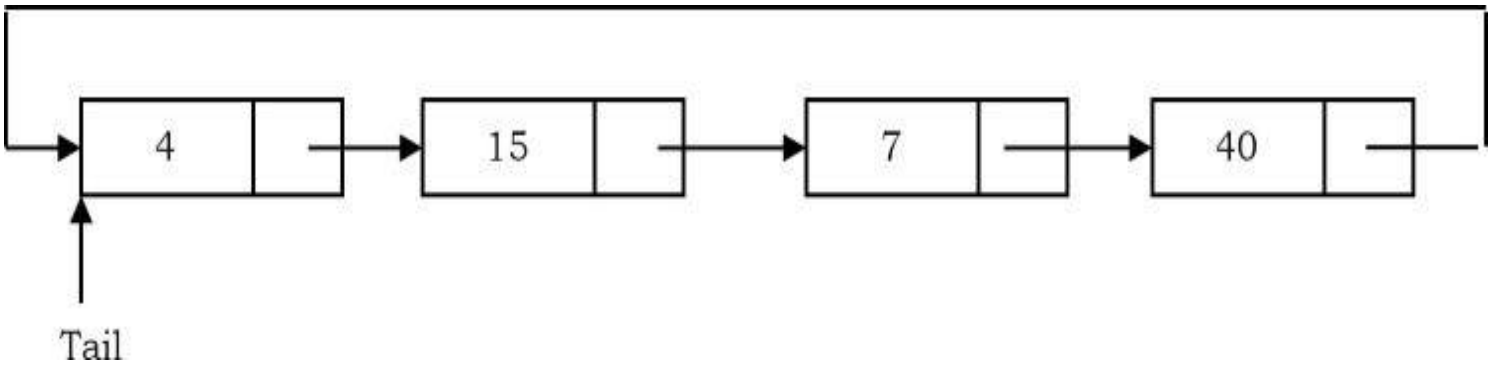
3.8 Circular Linked Lists

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). 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). For readability let us assume that the class name of circular linked list is CLLNode.

Counting nodes in a Circular Linked List

The circular list is accessible through the node marked *head*. (also called tail). 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.

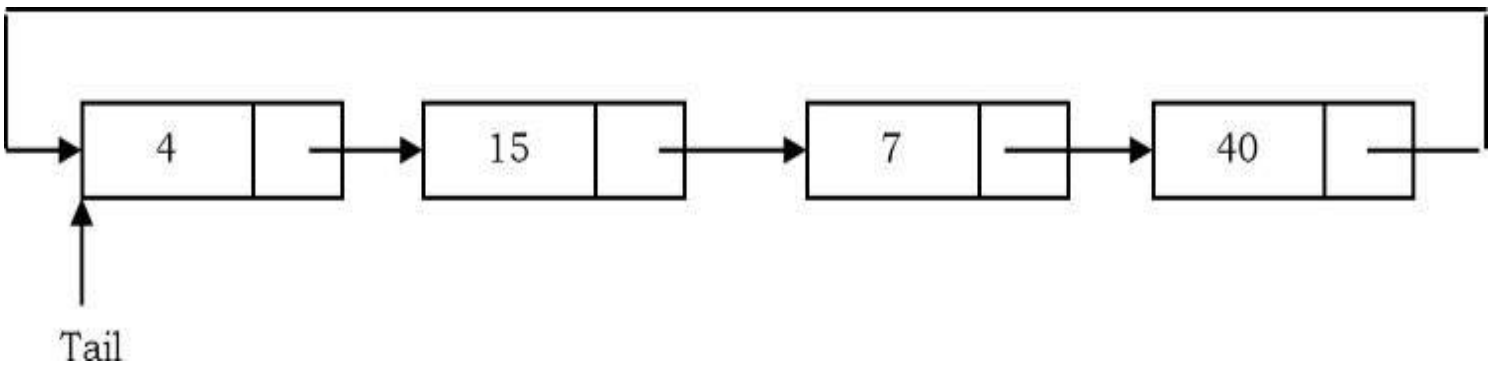



```
public int CircularListLength(CLLNode tail){
        int length = 0;
        CLLNode currentNode = tail.getNext();
        while(currentNode != tail){
                length++;
                currentNode = currentNode.getNext();
        }
        return length;
}
```

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

Printing the contents of a circular list

We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node 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.



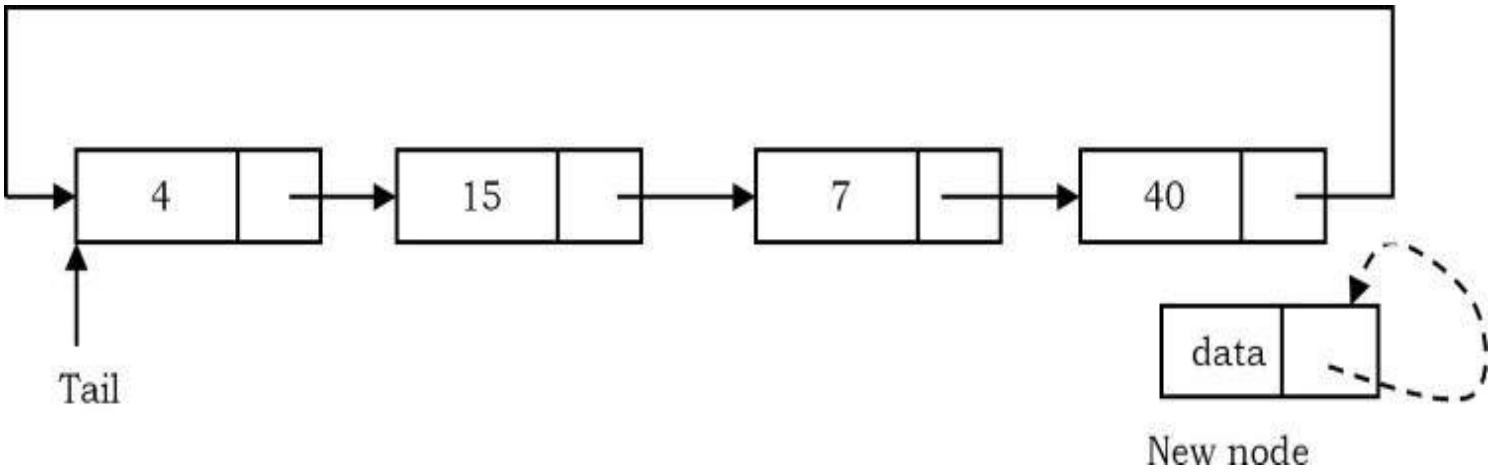
```
public void PrintCircularListData(CLLNode tail){
        CLLNode CLLNode = tail.getNext();
        while(CLLNode != tail){
                System.out.print(CLLNode.getData()+"->");
                CLLNode = CLLNode.getNext();
        }
        System.out.println("(" + CLLNode.getData() + ")headNode");
}
```

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

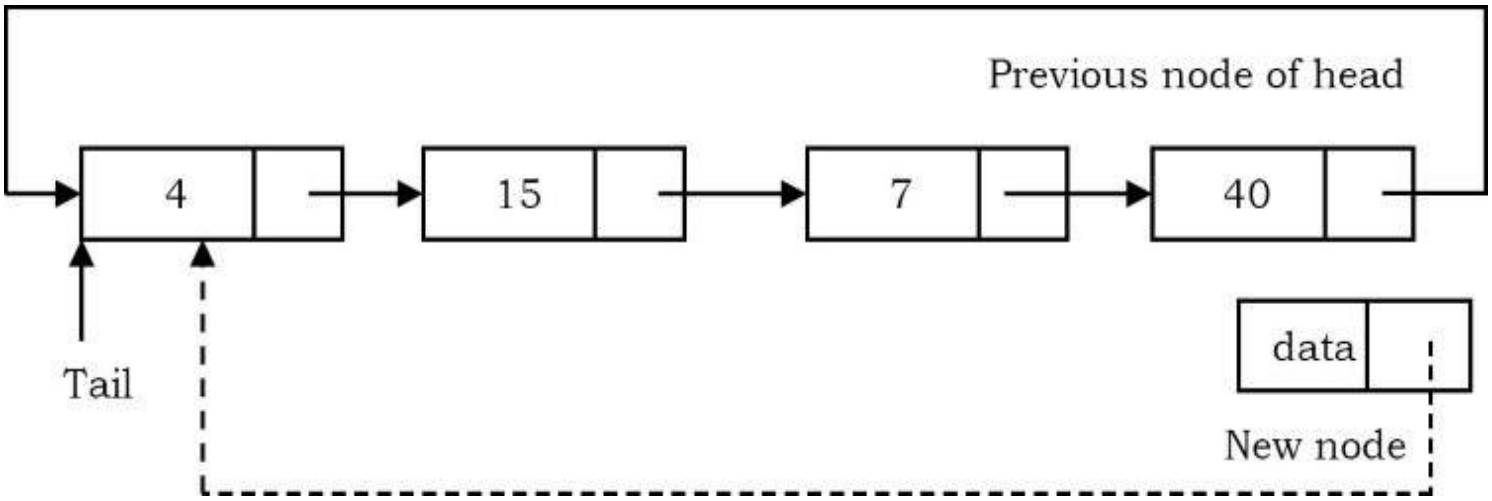
Inserting a Node at the End of a Circular Linked List

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.

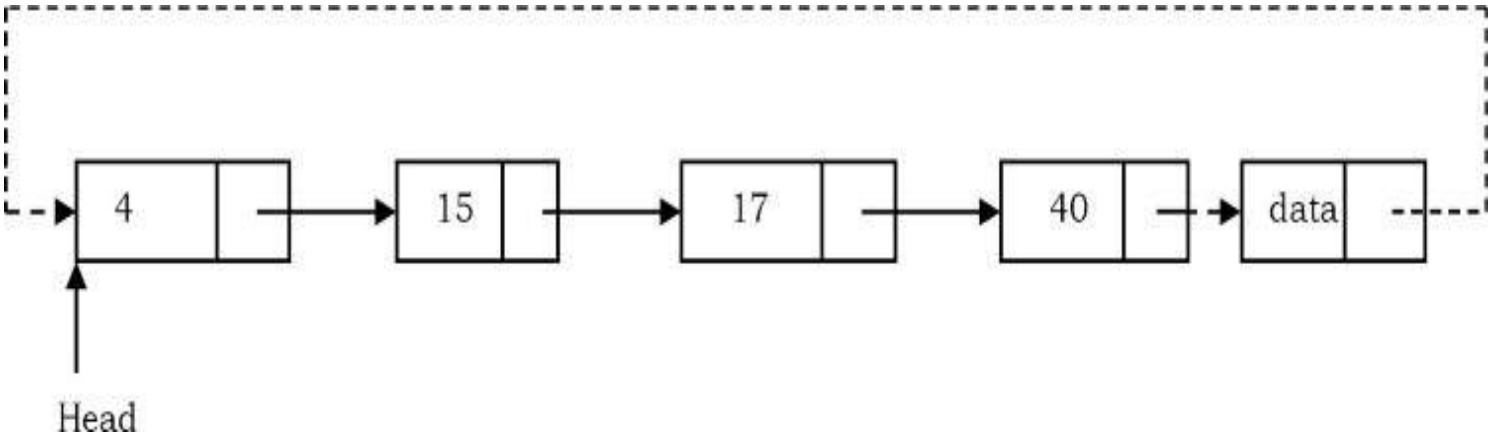
- Create a new node and initially keep its next pointer pointing to itself.



- 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.

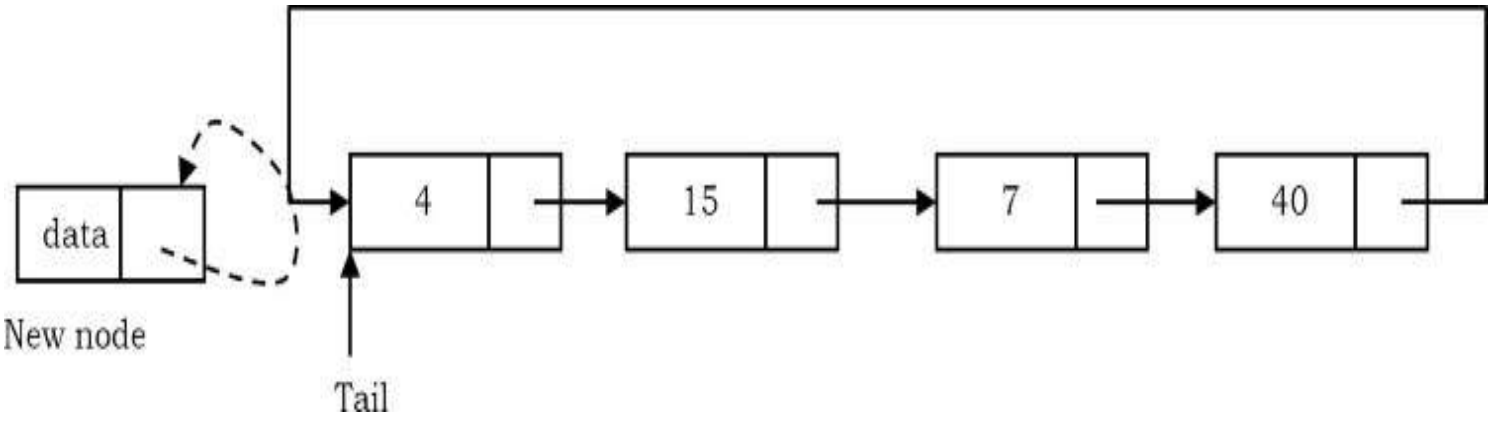


Time Complexity: O(*n*), for scanning the complete list of size *n*. Space Complexity: O(1), for temporary variable.

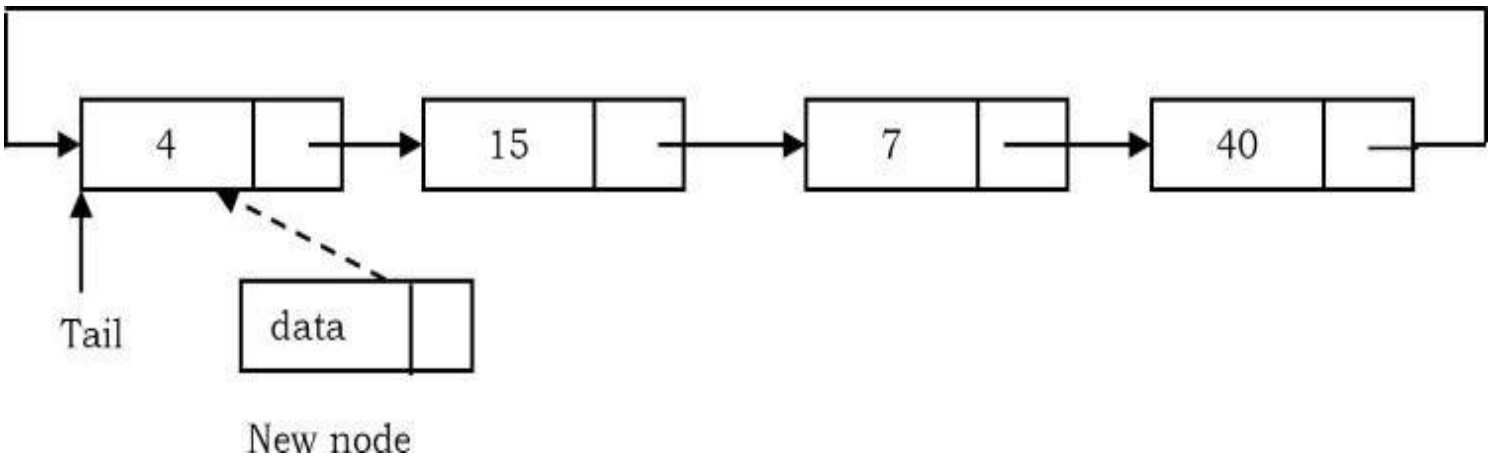
Inserting a Node at the front of a Circular Linked List

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:

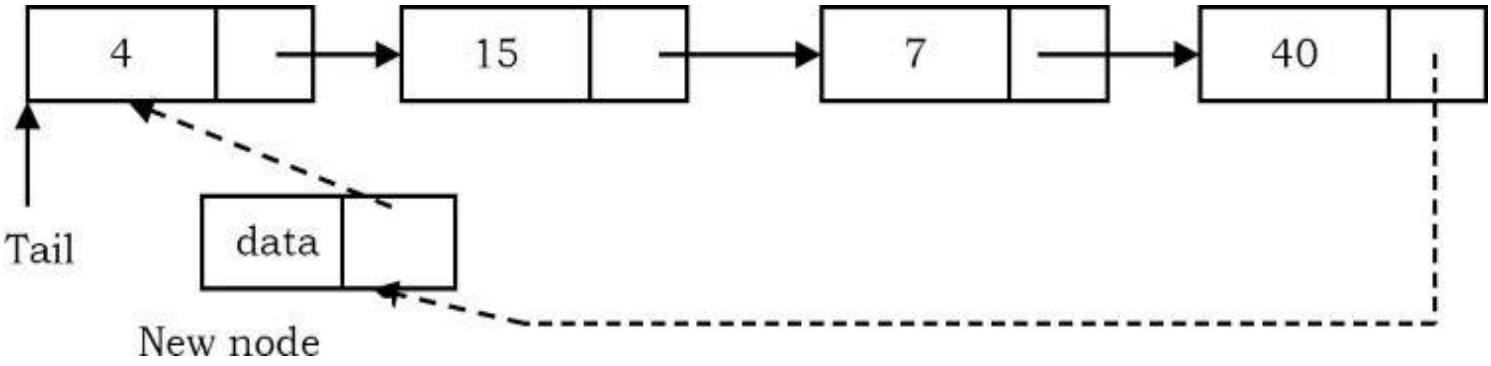
- Create a new node and initially keep its next pointer pointing to itself.



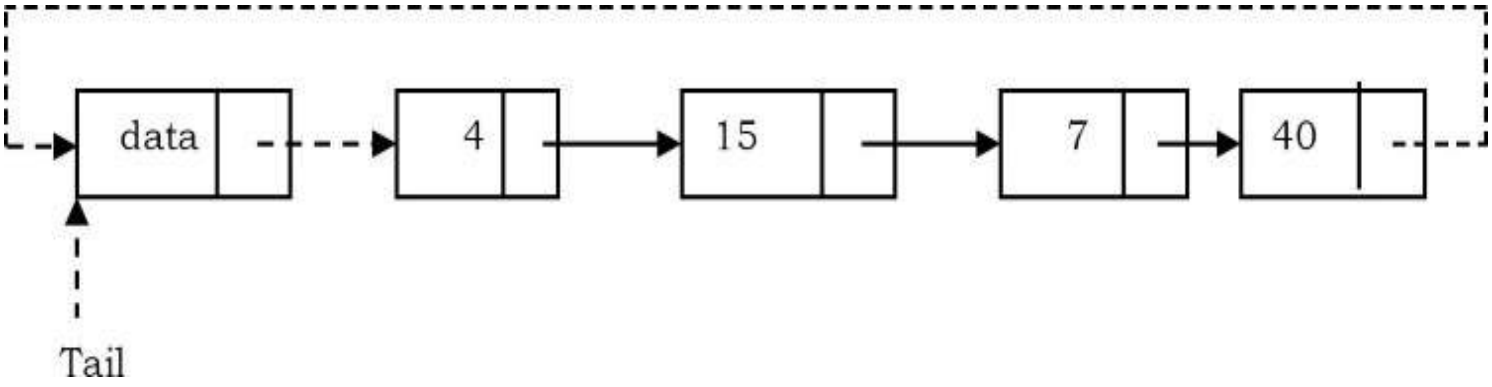
- Update the next pointer of the new node with the head node and also traverse the list until the tail. That means in a circular list we should stop at the node which is its previous node in the list.



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



- Make the new node as the head.

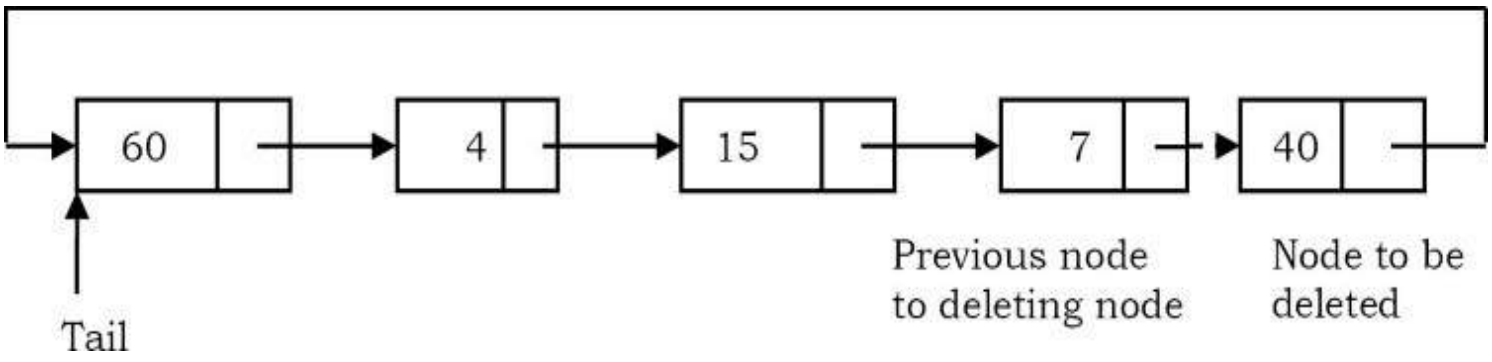


Time Complexity: O(*n*), for scanning the complete list of size *n*. Space Complexity: O(1), for temporary variable.

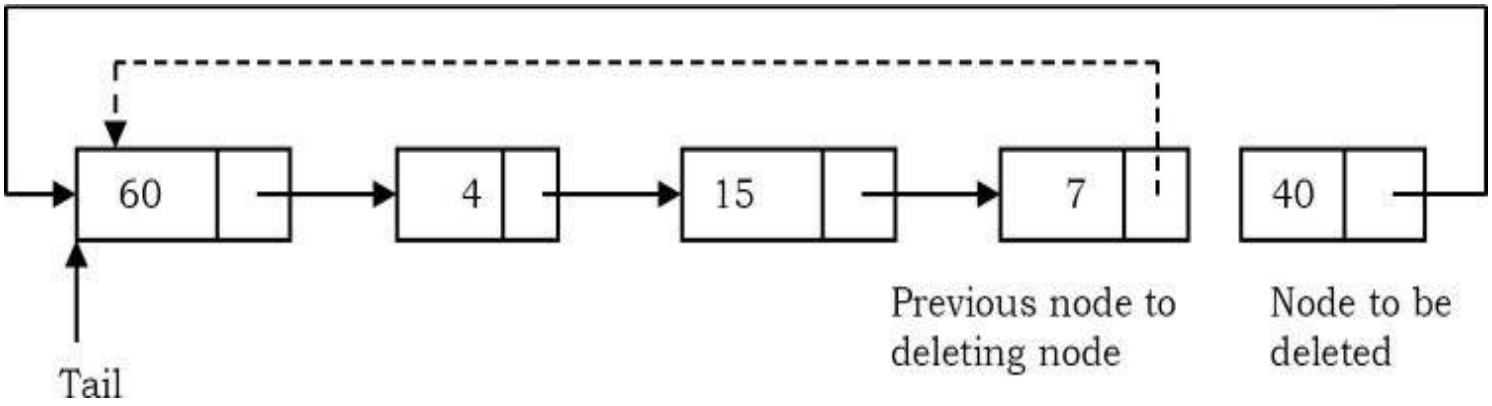
Deleting the last node in a Circular List

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

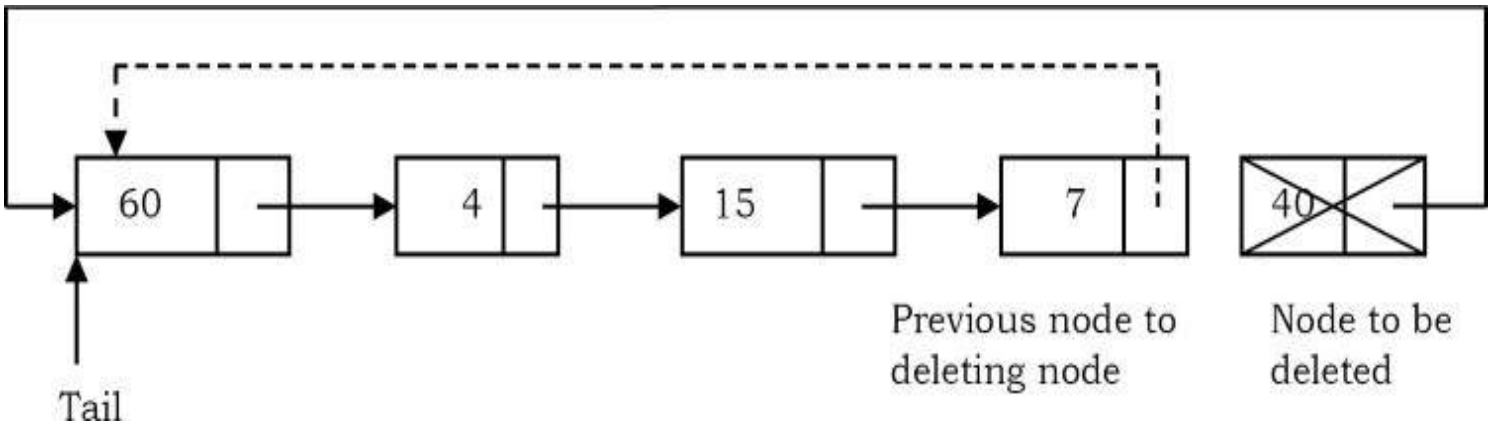
- Traverse the list and find the tail node and its previous node.



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



- Dispose of the tail node.

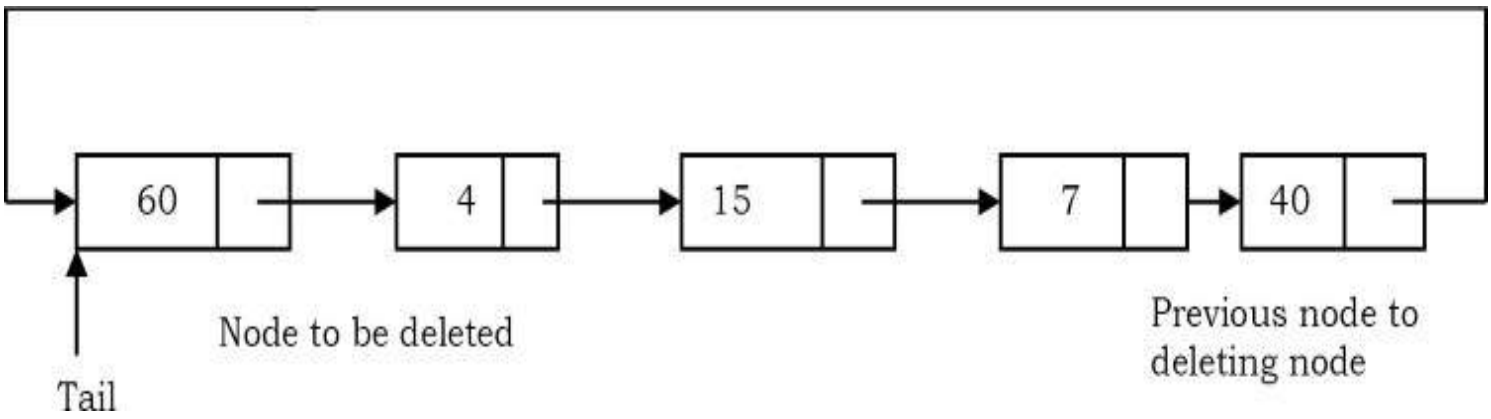


Time Complexity: O(*n*), for scanning the complete list of size *n*. Space Complexity: O(1), for a temporary variable.

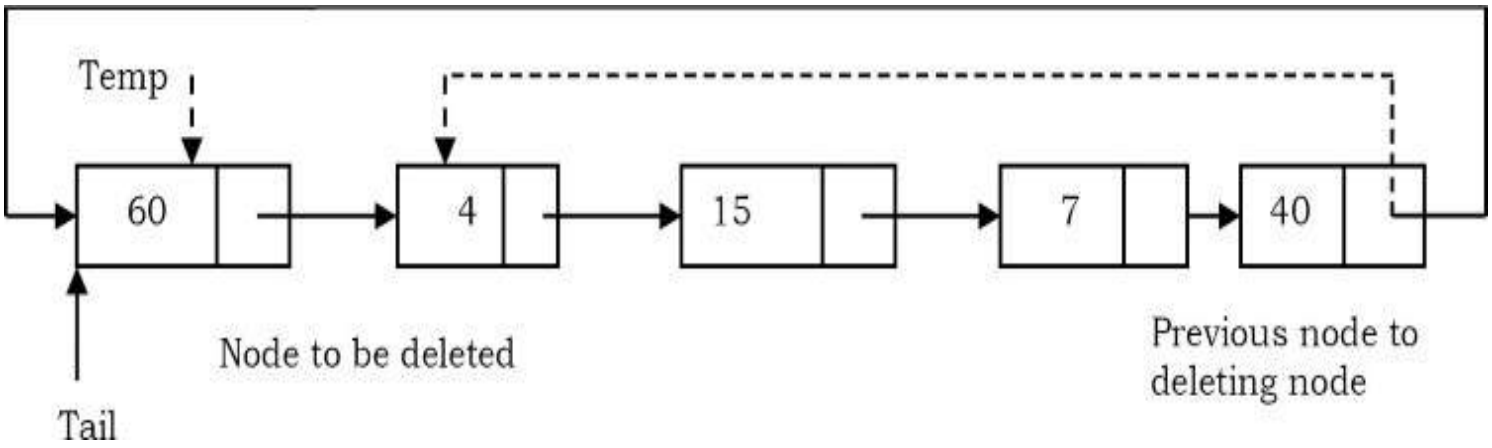
Deleting the First Node in a Circular List

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

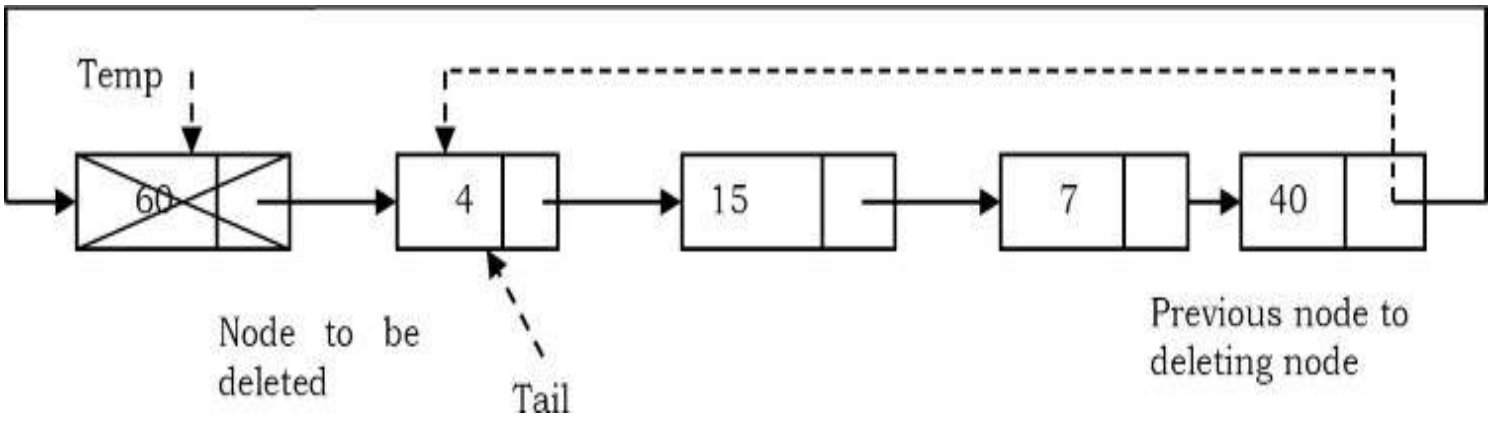
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- 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).



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.

Implementation

```
public class CircularLinkedList{
	protected CLLNode tail;
	protected int length;
	// Constructs a new circular list
	public CircularLinkedList(){
		tail = null;
		length = 0;
	}
	// Adds data to beginning of list.
	public void add(int data){
		addToHead(data);
	}
	// Adds element to head of list
	public void addToHead(int data){
		CLLNode temp = new CLLNode(data);
		if (tail == null) { // first data added
		   tail = temp;
		   tail.setNext(tail);
		} else { // element exists in list
		   temp.setNext(tail.getNext());
		   tail.setNext(temp);
		}
		length++;
	}
	// Adds element to tail of list
	public void addToTail(int data){
		// new entry:
		addToHead(data);
		tail = tail.getNext();
	}
	// Returns data at head of list
	public int peek(){
		return tail.getNext().getData();
	}
	// Returns data at tail of list
	public int tailPeek(){
		return tail.getData();
	}
	// Returns and removes data from head of list
	public int removeFromHead(){
		CLLNode temp = tail.getNext(); // ie. the head of the list
		if (tail == tail.getNext()) {
			tail = null;
		} else {
			tail.setNext(temp.getNext());
			temp.setNext(null); // helps clean things up; temp is free
		}
		length--;
		return temp.getData();
	}
	// Returns and removes data from tail of list
	public int removeFromTail(){
		if (isEmpty()){
			return Integer.MIN_VALUE;
```

```
            }
            CLLNode finger = tail;
            while (finger.getNext() != tail) {
                    finger = finger.getNext();
            }
            // finger now points to second-to-last data
            CLLNode temp = tail;
            if (finger == tail) {
                    tail = null;
            } else {
                    finger.setNext(tail.getNext());
                    tail = finger;
            }
            length--;
            return temp.getData();
    }
    // Returns true if list contains data, else false
    public boolean contains(int data){
            if (tail == null)
                return false;
            CLLNode finger;
            finger = tail.getNext();
            while (finger != tail && (!(finger.getData() == data))){
                finger = finger.getNext();
            }
            return finger.getData() == data;
    }
    // Removes and returns element equal to data, or null
    public int remove(int data){
            if (tail == null) return Integer.MIN_VALUE;
            CLLNode finger = tail.getNext();
            CLLNode previous = tail;
            int compares;
            for (compares = 0; compares < length && (!(finger.getData() == data)); compares++) {
                    previous = finger;
                    finger = finger.getNext();
            }
            if (finger.getData() == data) {
                    // an example of the pigeon-hole principle
                    if (tail == tail.getNext()) {
                            tail = null; }
                    else {
                            if (finger == tail)
                                    tail = previous;
                            previous.setNext(previous.getNext().getNext());
                    }
                    // finger data free
                    finger.setNext(null);       // to keep things disconnected
                    length--;                   // fewer elements
                    return finger.getData();
            }
            else return Integer.MIN_VALUE;
    }
    // Return the current length of the CLL.
    public int size(){
            return length;
    }
    // Return the current length of the CLL.
    public int length() {
            return length;
    }
    // Returns true if no elements in list
```

```
    public boolean isEmpty(){
        return tail == null;
    }
    // Remove everything from the CLL.
    public void clear(){
        length = 0;
        tail = null;
    }
    // Return a string representation of this collection, in the form: ["str1","str2",...].
    public String toString(){
        String result = "[";
        if (tail == null) {
            return result+"]";
        }
        result = result + tail.getData();
        CLLNode temp = tail.getNext();
        while (temp != tail) {
            result = result + "," + temp.getData();
            temp = temp.getNext();
        }
        return result + "]";
    }
}
```

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

```
public class DLLNode {
    private int data;
    private DLLNode next;
    private DLLNode previous;
    .....
}
```

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.

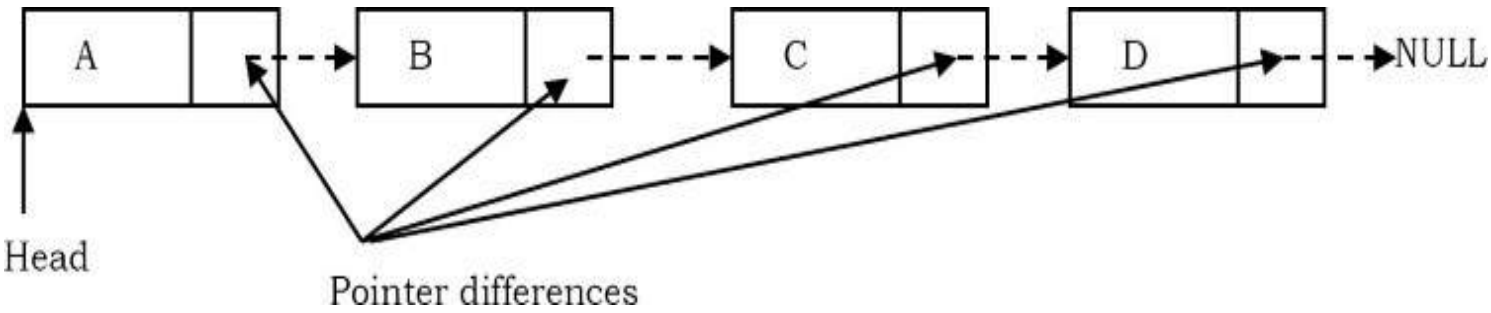
New Node Definition

```
public class ListNode {
    private int data;
    private 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.

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

The *ptrdiff* of the start node (head node) is the ⊕ of NULL and *next* node (next 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,

$$(B \oplus D) \oplus D = B \text{ (since, } D \oplus D=0)$$

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

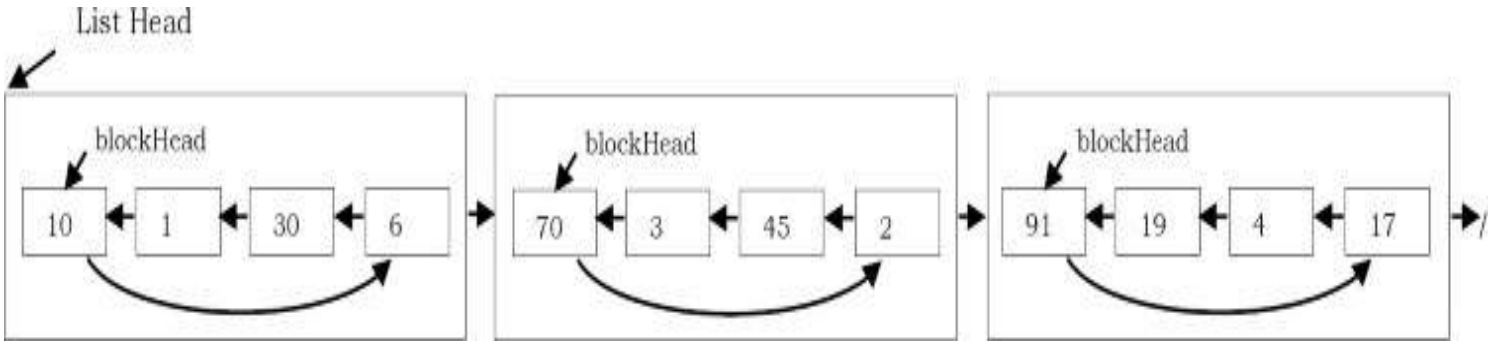
$$(B \oplus D) \oplus B = D \text{ (since, } B \oplus B=0)$$

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.



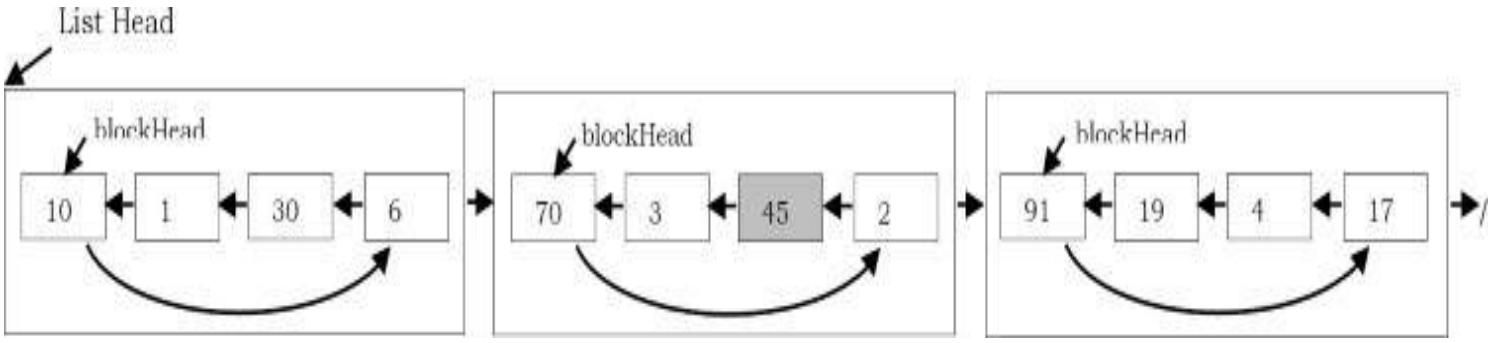
Assume that there will be no more than n elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly $\lfloor\sqrt{n}\rfloor$ elements. Thus, there will be no more than $\lfloor\sqrt{n}\rfloor$ blocks at any time.

Searching for an element in Unrolled Linked Lists

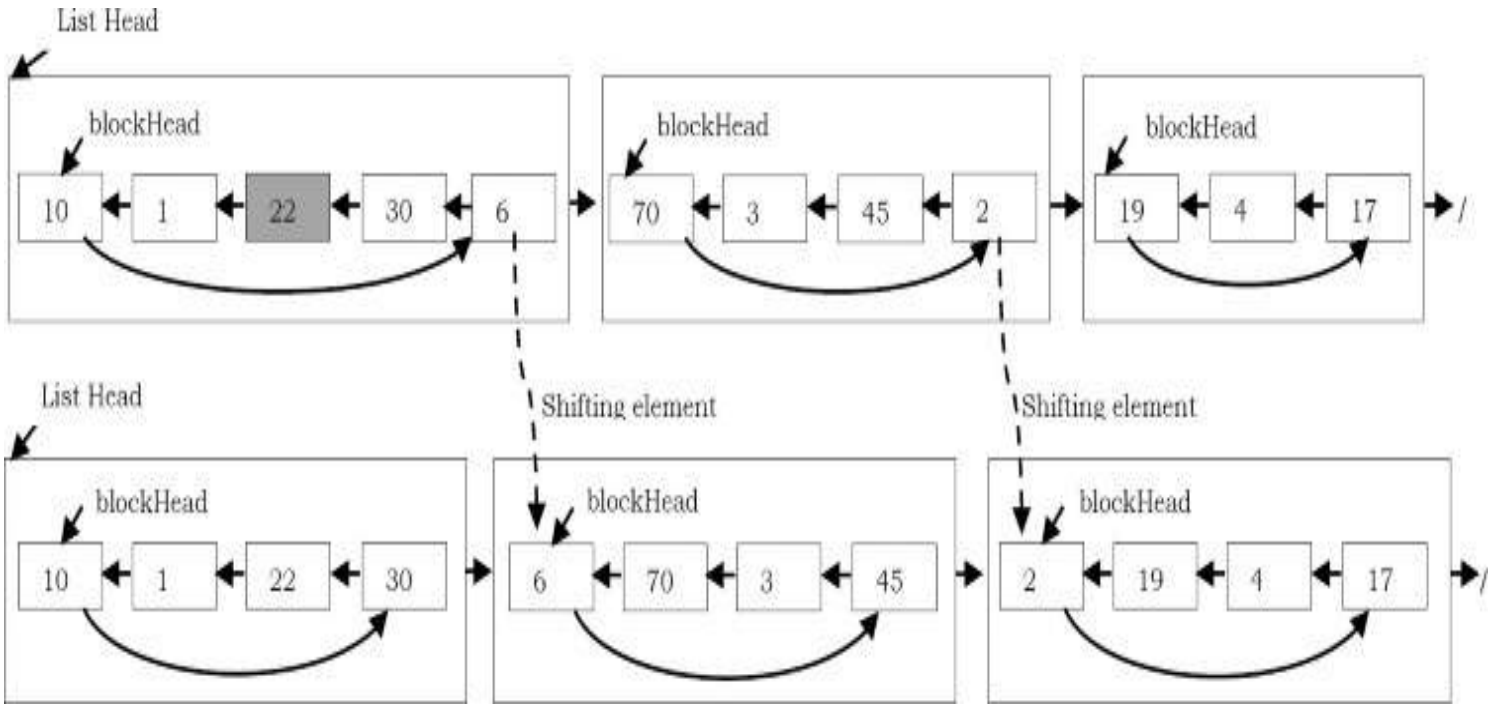
In unrolled linked lists, we can find the k^{th} element in O($\sqrt{n}$):

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 $\lceil\sqrt{n}\rceil$)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



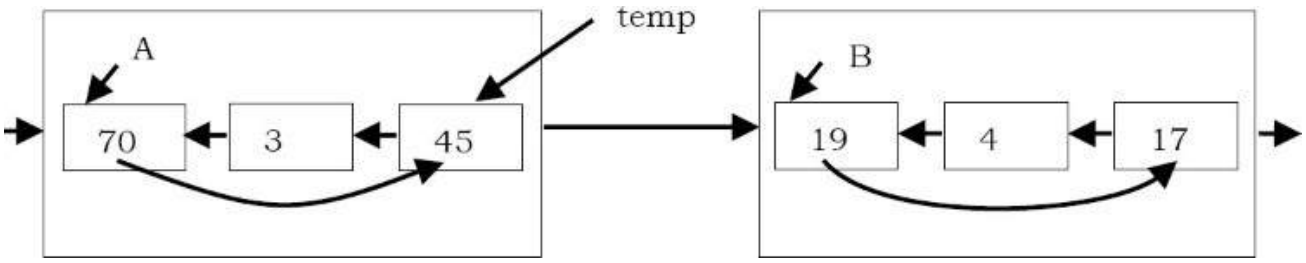
When inserting a node, we have to re-arrange the nodes in the unrolled linked list to maintain the properties previously mentioned, that each block contains $\lceil\sqrt{n}\rceil$ nodes. Suppose that we insert a node x after the i^{th} node, and x should be placed in the j^{th} block. Nodes in the j^{th} 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

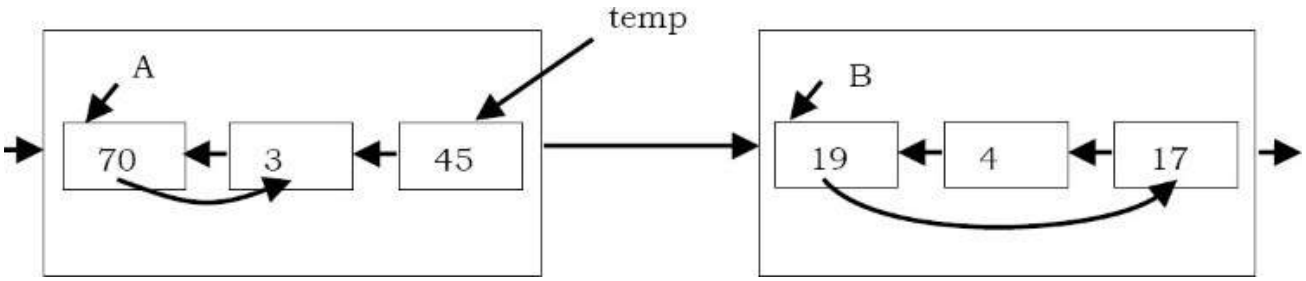
Note that each *shift* operation, which includes removing a node from the tail of the circular linked list in a block and inserting a node to the head of the circular linked list in the block after, takes

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.

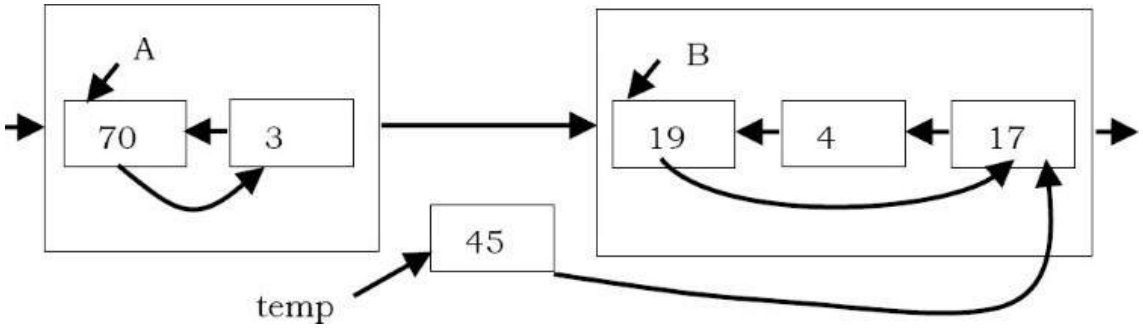
1. A temporary pointer is needed to store the tail of *A*.



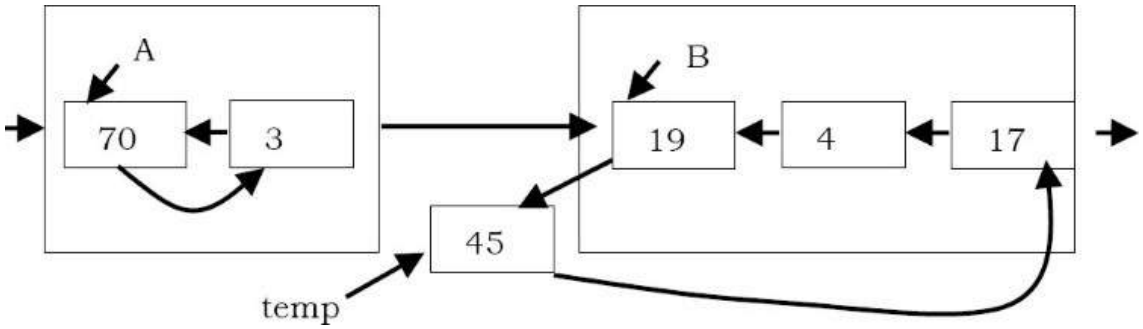
2. In block *A*, move the next pointer of the head node to point to the second-to-last node, so that the tail node of *A* can be removed.



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

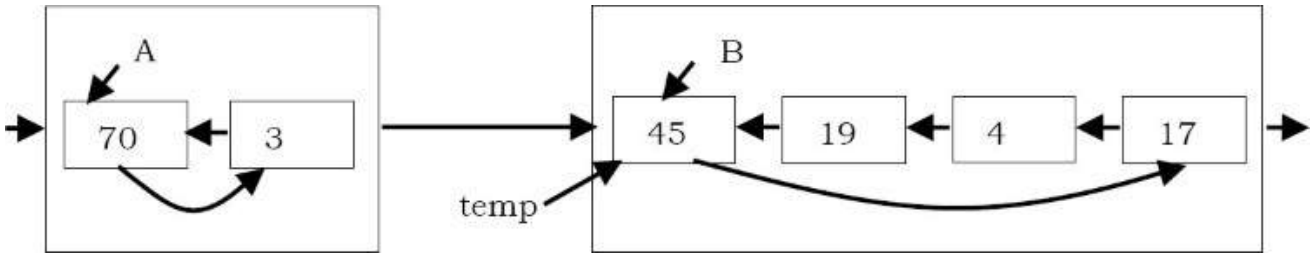


4. Let the next pointer of the head node of *B* point to the node temp points to.

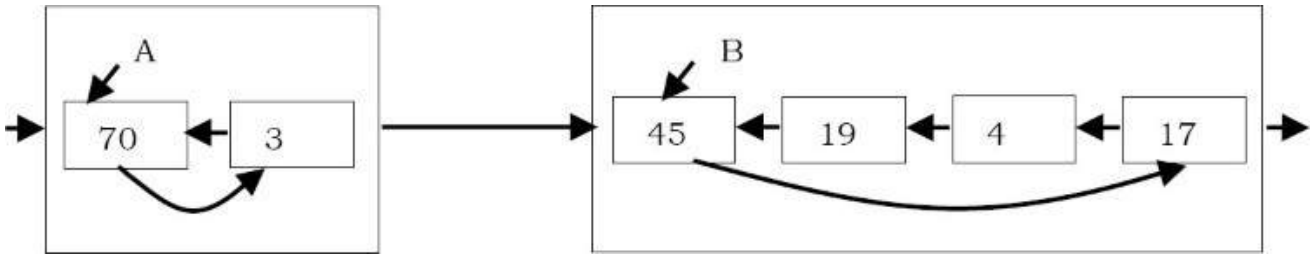


5. Finally, set the head pointer of *B* to point to the node *temp* points to. Now the node temp

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 Doubly 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.

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 1K 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:

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 1K 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

```
public class UnrolledLinkedList<E> extends AbstractList<E> implements List<E>, Serializable {
   //The maximum number of elements that can be stored in a single node.
   private int nodeCapacity;
   //The current size of this list.
   private int size = 0;
   //The first node of this list.
   private ListNode firstNode;
   //The last node of this list.
   private ListNode lastNode;
   //Constructs an empty list with the specified capacity
   public UnrolledLinkedList(int nodeCapacity) throws IllegalArgumentException {
      if (nodeCapacity < 8) {
         throw new IllegalArgumentException("nodeCapacity < 8");
      }
      this.nodeCapacity = nodeCapacity;
      firstNode = new ListNode();
      lastNode = firstNode;
   }
   public UnrolledLinkedList() {
      this(16);
   }
   public int size() {
```

```
    return size;
}
public boolean isEmpty() {
    return (size == 0);
}
//Returns true if this list contains the specified element.
public boolean contains(Object o) {
    return (indexOf(o) != -1);
}
public Iterator<E> iterator() {
    return new ULLIterator(firstNode, 0, 0);
}
//Appends the specified element to the end of this list.
public boolean add(E e) {
    insertIntoNode(lastNode, lastNode.numElements, e);
    return true;
}
//Removes the first occurrence of the specified element from this list,
public boolean remove(Object o) {
    int index = 0;
    ListNode node = firstNode;
    if (o == null) {
        while (node != null) {
            for (int ptr = 0; ptr < node.numElements; ptr++) {
                if (node.elements[ptr] == null) {
                    removeFromNode(node, ptr);
                    return true;
                }
            }
            index += node.numElements;
            node = node.next;
        }
    } else {
        while (node != null) {
            for (int ptr = 0; ptr < node.numElements; ptr++) {
                if (o.equals(node.elements[ptr])) {
                    removeFromNode(node, ptr);
                    return true;
                }
            }
            index += node.numElements;
            node = node.next;
        }
    }
    return false;
}
//Removes all of the elements from this list.
public void clear() {
    ListNode node = firstNode.next;
    while (node != null) {
        ListNode next = node.next;
        node.next = null;
        node.previous = null;
        node.elements = null;
        node = next;
    }
    lastNode = firstNode;
    for (int ptr = 0; ptr < firstNode.numElements; ptr++) {
        firstNode.elements[ptr] = null;
    }
    firstNode.numElements = 0;
    firstNode.next = null;
    size = 0;
}
```

```
//Returns the element at the specified position in this list.
public E get(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
        node = lastNode;
        p = size;
        while ((p -= node.numElements) > index) {
            node = node.previous;
        }
    }
    return (E) node.elements[index - p];
}
//Replaces the element at the specified position in this list with the specified element.
public E set(int index, E element) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    E el = null;
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
        node = lastNode;
        p = size;
        while ((p -= node.numElements) > index) {
            node = node.previous;
        }
    }
    el = (E) node.elements[index - p];
    node.elements[index - p] = element;
    return el;

}
//Inserts the specified element at the specified position in this list.
//Shifts the element currently at that position (if any) and any
//subsequent elements to the right (adds one to their indices).
public void add(int index, E element) throws IndexOutOfBoundsException {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
```

```
            node = lastNode;
            p = size;
            while ((p -= node.numElements) > index) {
                node = node.previous;
            }
        }
        insertIntoNode(node, index - p, element);
    }
    //Removes the element at the specified position in this list.
    //Shifts any subsequent elements to the left (subtracts one from their indices).
    public E remove(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        E element = null;
        ListNode node;
        int p = 0;
        if (size - index > index) {
            node = firstNode;
            while (p <= index - node.numElements) {
                p += node.numElements;
                node = node.next;
            }
        } else {
            node = lastNode;
            p = size;
            while ((p -= node.numElements) > index) {
                node = node.previous;
            }
        }
        element = (E) node.elements[index - p];
        removeFromNode(node, index - p);
        return element;
    }
    private static final long serialVersionUID = -6740523091030452111L;
    private class ListNode {
        ListNode next;
        ListNode previous;
        int numElements = 0;
        Object[] elements;
        ListNode() {
            elements = new Object[nodeCapacity];
        }
    }
    private class ULLIterator implements ListIterator<E> {
        ListNode currentNode;
        int ptr;
        int index;
        private int expectedModCount = modCount;
        ULLIterator(ListNode node, int ptr, int index) {
            this.currentNode = node;
            this.ptr = ptr;
            this.index = index;
        }
        public boolean hasNext() {
            return (index < size - 1)
        }
        public E next() {
            ptr++;
            if (ptr >= currentNode.numElements) {
                if (currentNode.next != null) {
                    currentNode = currentNode.next;
                    ptr = 0;
                } else {
```

```
                throw new NoSuchElementException();
            }
        }
        index++;
        checkForModification();
        return (E) currentNode.elements[ptr];
    }
    public boolean hasPrevious() {
        return (index > 0);
    }
    public E previous() {
        ptr--;
        if (ptr < 0) {
            if (currentNode.previous != null) {
                currentNode = currentNode.previous;
                ptr = currentNode.numElements - 1;
            } else {
                throw new NoSuchElementException();
            }
        }
        index--;
        checkForModification();
        return (E) currentNode.elements[ptr];
    }
    public int nextIndex() {
        return (index + 1);
    }
    public int previousIndex() {
        return (index - 1);
    }
    public void remove() {
        checkForModification();
        removeFromNode(currentNode, ptr);
    }
    public void set(E e) {
        checkForModification();
        currentNode.elements[ptr] = e;
    }
    public void add(E e) {
        checkForModification();
        insertIntoNode(currentNode, ptr + 1, e);
    }
    private void checkForModification() {
        if (modCount != expectedModCount) {
            throw new ConcurrentModificationException();
        }
    }
}
private void insertIntoNode(ListNode node, int ptr, E element) {
    // if the node is full
    if (node.numElements == nodeCapacity) {
        // create a new node
        ListNode newNode = new ListNode();
        // move half of the elements to the new node
        int elementsToMove = nodeCapacity / 2;
        int startIndex = nodeCapacity - elementsToMove;
        for (int i = 0; i < elementsToMove; i++) {
            newNode.elements[i] = node.elements[startIndex + i];
            node.elements[startIndex + i] = null;
        }
        node.numElements -= elementsToMove;
        newNode.numElements = elementsToMove;
        // insert the new node into the list
        newNode.next = node.next;
```

```
            newNode.previous = node;
            if (node.next != null) {
                node.next.previous = newNode;
            }
            node.next = newNode;
            if (node == lastNode) {
                lastNode = newNode;
            }
            // check whether the element should be inserted into
            // the original node or into the new node
            if (ptr > node.numElements) {
                node = newNode;
                ptr -= node.numElements;
            }
        }
        for (int i = node.numElements; i > ptr; i--) {
            node.elements[i] = node.elements[i - 1];
        }
        node.elements[ptr] = element;
        node.numElements++;
        size++;
        modCount++;
    }
    private void removeFromNode(ListNode node, int ptr) {
        node.numElements--;
        for (int i = ptr; i < node.numElements; i++) {
            node.elements[i] = node.elements[i + 1];
        }
        node.elements[node.numElements] = null;
        if (node.next != null && node.next.numElements + node.numElements <= nodeCapacity) {
            mergeWithNextNode(node);
        } else if (node.previous != null && node.previous.numElements + node.numElements <= nodeCapacity) {
            mergeWithNextNode(node.previous);
        }
        size--;
        modCount++;
    }
    //This method does merge the specified node with the next node.
    private void mergeWithNextNode(ListNode node) {
        ListNode next = node.next;
        for (int i = 0; i < next.numElements; i++) {
            node.elements[node.numElements + i] = next.elements[i];
            next.elements[i] = null;
        }
        node.numElements += next.numElements;
        if (next.next != null) {
            next.next.previous = node;
        }
        node.next = next.next.next;
        if (next == lastNode) {
            lastNode = node;
        }
    }
}
```

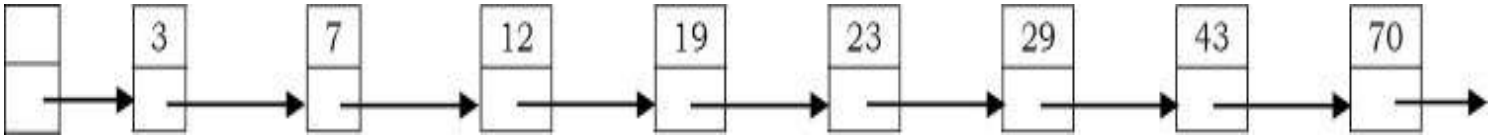
3.11 Skip Lists

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.

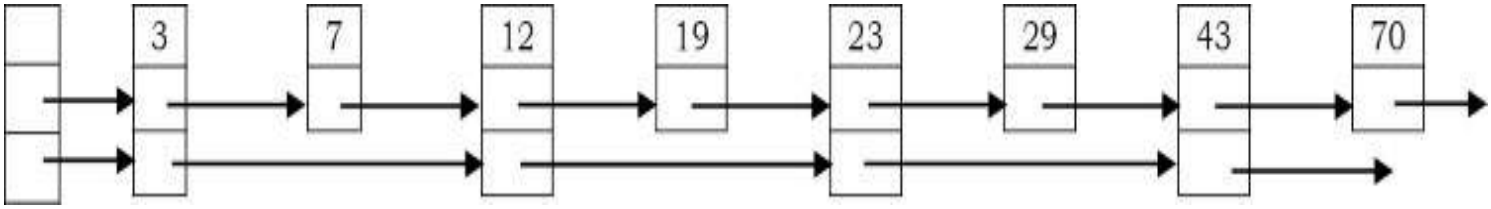
Skip list is a data structure that can be used as an alternative to balanced binary trees (refer to *Trees* chapter). As compared to a binary tree, skip lists allow quick search, insertion and deletion of elements. This is achieved by using probabilistic balancing rather than strictly enforce balancing. 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.

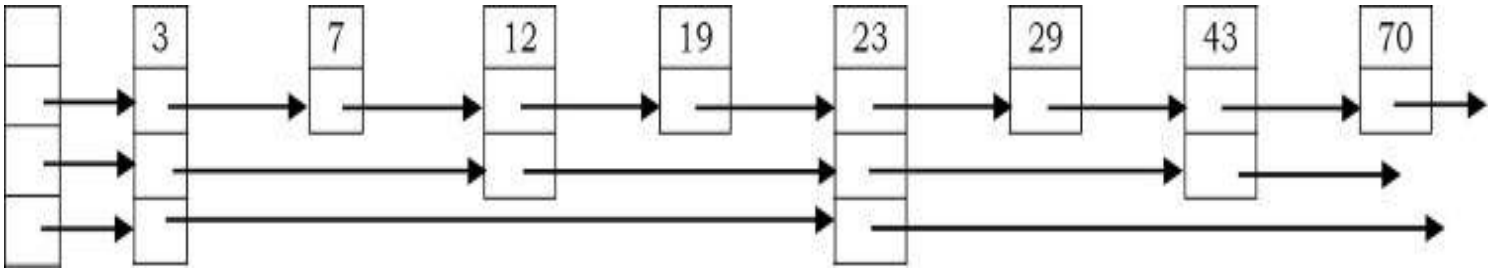
Skip Lists with One Level



Skip Lists with Two Level



Skip Lists with Three Levels



This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The Search operation returns the contents of the value associated with the desired key or failure if the key is not present. The Insert operation associates a specified key with a new value (inserting the key if it had not already been present). The Delete operation deletes the specified key. It is easy to support additional operations such as “find the minimum key” or “find the next key”.

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level i node has i forward pointers, indexed 1 through i. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant *MaxLevel*. The level of a list is the maximum level currently in the list (or 1 if the list is empty). The header of a list has forward pointers at levels one through MaxLevel. The forward pointers of the header at levels higher than the current maximum level of the list point to NULL.

Initialization

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialized so that the the level of the list is equal to 1 and all forward pointers of the list’s header point to NIL.

Search for an element

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice. A vector update is maintained so that when the search is complete (and we are ready to perform the splice), update[i] contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum

level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

Choosing a Random Level

Initially, we discussed a probability distribution where half of the nodes that have level i pointers also have level i+1 pointers. To get away from magic constants, we say that a fraction p of the nodes with level i pointers also have level i+1 pointers. (for our original discussion, p = 1/2). Levels are generated randomly by an algorithm. Levels are generated without reference to the number of elements in the list

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

```
import java.util.Random;
public class SkipList<T extends Comparable<T>, U>{
	private class Node{
		public T key;
		public U value;
		public long level;
		public Node next;
		public Node down;
		public Node(T key, U value, long level, Node next, Node down){
			this.key = key;
			this.value = value;
			this.level = level;
			this.next = next;
			this.down = down;
		}
	}
	private Node head;
	private Random _random;
	private long size;
	private double _p;
	private long level(){
		long level = 0;
		while (level <= size && _random.nextDouble() < _p) {
			level++;
		}
```

```
        return level;
    }
    public SkipList(){
        head = new Node(null, null, 0, null, null);
        _random = new Random();
        size = 0;
        _p = 0.5;
    }
    public void add(T key, U value){
        long level = level();
        if (level > head.level) {
            head = new Node(null, null, level, null, head);
        }
        Node cur = head;
        Node last = null;
        while (cur != null) {
            if (cur.next == null || cur.next.key.compareTo(key) > 0) {
                if (level >= cur.level) {
                    Node n = new Node(key, value, cur.level, cur.next, null);
                    if (last != null) {
                        last.down = n;
                    }
                    cur.next = n;
                    last = n;
                }
                cur = cur.down;
                continue;
            } else if (cur.next.key.equals(key)) {
                cur.next.value = value;
                return;
            }
            cur = cur.next;
        }
        size++;
    }
    public boolean containsKey(T key){
        return get(key) != null;
    }
    public U remove(T key){
        U value = null;
        Node cur = head;
        while (cur != null) {
            if (cur.next == null || cur.next.key.compareTo(key) >= 0) {
                if (cur.next != null && cur.next.key.equals(key)) {
                    value = cur.next.value;
                    cur.next = cur.next.next;
                }
                cur = cur.down;
                continue;
            }
            cur = cur.next;
        }
        size--;
        return value;
    }
    public U get(T key){
        Node cur = head;
        while (cur != null) {
            if (cur.next == null || cur.next.key.compareTo(key) > 0) {
                cur = cur.down;
                continue;
            } else if (cur.next.key.equals(key)) {
```

```
                    return cur.next.value;
                }
                cur = cur.next;
            }
            return null;
        }
    }
    public class SkipListsTest{
        public static void main(String [] args){
            SkipList s = new SkipList();
            s.add(1,100);
            System.out.println(s.get(1));
        }
    }
```

3.12 Linked Lists: Problems & Solutions

Problem-1 Implement Stack using Linked List

Solution: Refer to *Stacks* chapter.

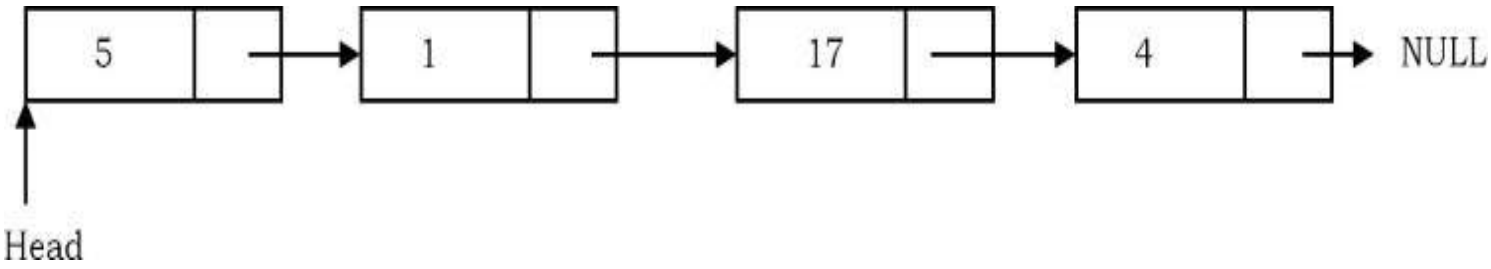
Problem-2 Find n^{th} node from the end of a Linked List.

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, Therefore, $T(m) = O(m)$. 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.

```
public ListNode  NthNodeFromEnd(ListNode head , int NthNode) {
      ListNode pTemp = head, pNthNode = null;
      for(int count =1; count< NthNode;count++) {
         if(pTemp != null)
            pTemp = pTemp.getNext();
      }
      while(pTemp!= null){
         if(pNthNode == null)
              pNthNode = head;
         else
              pNthNode = pNthNode.getNext();
         pTemp = pTemp.getNext();
      }
      if(pNthNode !=  null)
            return pNthNode;
      return null;
}
```

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

Problem-6 Can we solve Problem-5 with recursion?

Solution: We can use a global variable to track the post recursive call and when it is same as Nth', return the node.

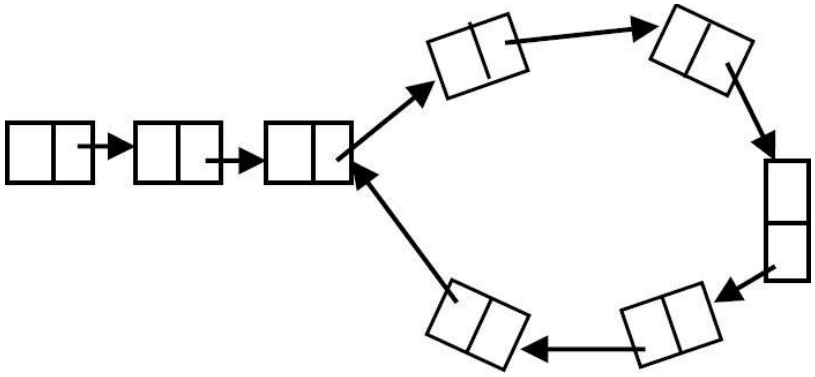
```
public ListNode NthNodeFromEnd(ListNode head, int Nth) {
   if(head != null) {
      NthNodeFromEnd(head.next, Nth);
      counter++;
      if(Nth ==counter) {
         return head;
      }
   }
   return null;
}
```

Time Complexity: O(*n*) for pre recursive calls and O(*n*) for post recursive calls, which is = O(2*n*) = O(*n*).
Space Complexity: O(n) for recursive stack.

Problem-7 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-8 Can we use the hashing technique for solving Problem-7?

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-9 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.

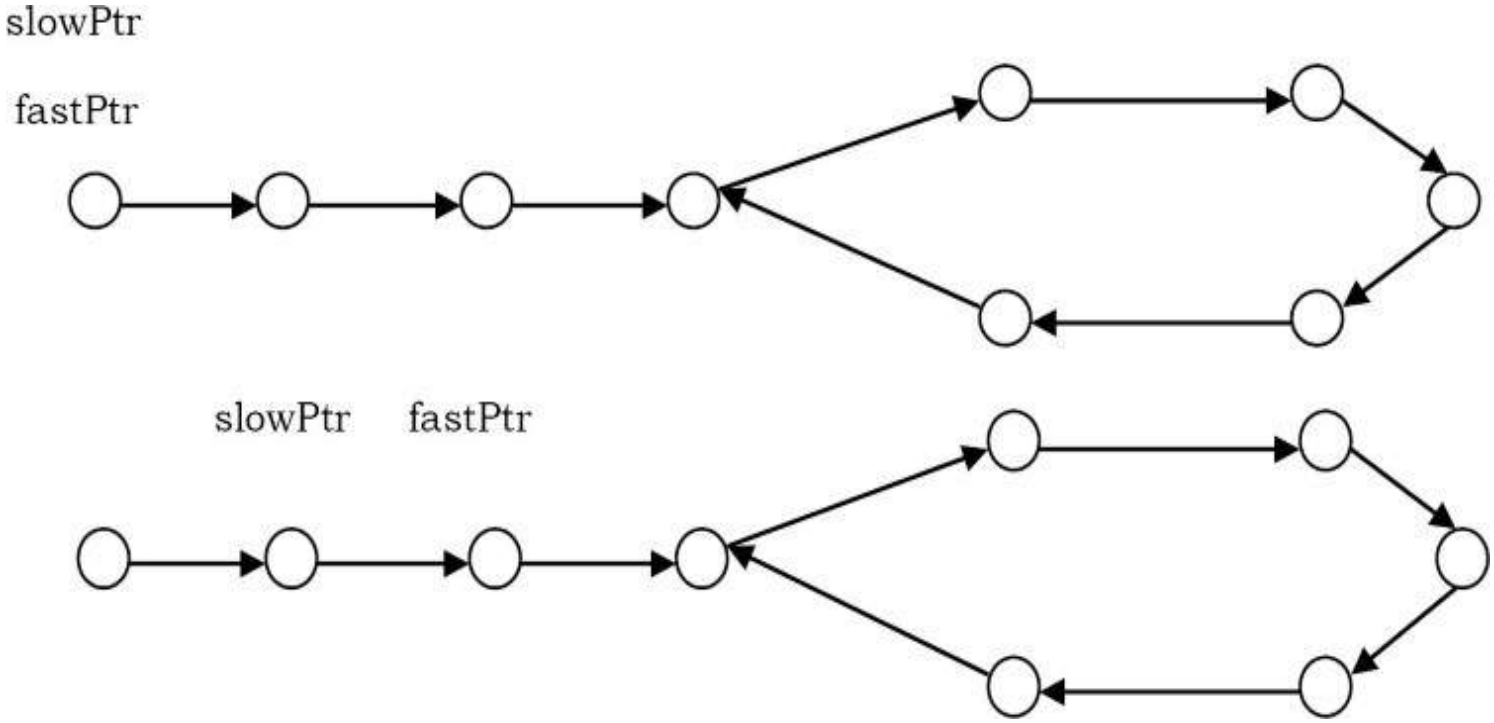
Problem-10 Can we solve the Problem-6 in O(n)?

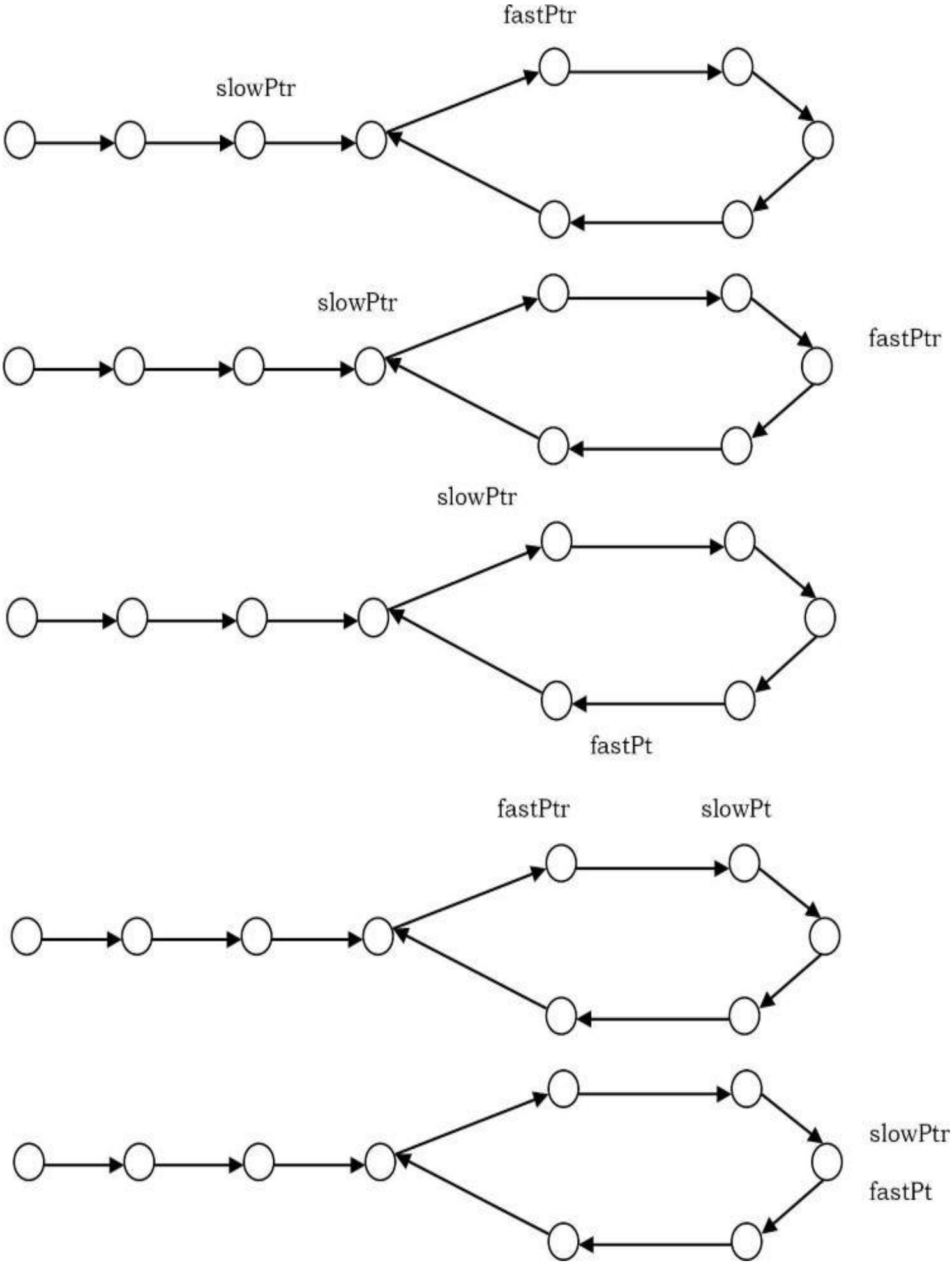
Solution: Yes. Efficient Approach (Memoryless Approach): This problem was solved by *Floyd*. The solution is named the Floyd cycle finding algorithm. It uses *two* pointers moving at different speeds to walk the linked list. Once they enter the loop they are expected to meet, which denotes that there 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
fastPt

```
private static boolean findIfLoopExistsUsingFloyds(ListNode head){
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    while (fastPtr != null && fastPtr.getNext() != null) {
        fastPtr = fastPtr.getNext().getNext();
        slowPtr = slowPtr.getNext();
        if (slowPtr == fastPtr)
            return true;
    }
    return false;
}
```

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

Problem-11 We are given a pointer to the first element of a linked list *L*. There are two possibilities for *L*, it either ends (snake) or its last element points back to one of the earlier elements in the list (snail). Give an algorithm that tests whether a given list L is a snake or a snail.

Solution: It is the same as Problem-6.

Problem-12 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-10. 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. Let *x* and y be travelers such that y is walking twice as fast as *x* (i.e. y = 2*x*). Further, let s be the place where *x* and y first started walking at the same time. Then when x and y meet again, the distance from s to the start of the loop is the exact same distance from the present meeting place of *x* and y to the start of the loop.

```
private static ListNode findBeginofLoop(ListNode head){
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    boolean loopExists = false;
    while (fastPtr != null && fastPtr.getNext() != null) {
        fastPtr = fastPtr.getNext().getNext();
        slowPtr = slowPtr.getNext();
        if (slowPtr == fastPtr) {
            loopExists = true;
            break;
        }
    }
    if (loopExists) {
        slowPtr = head;
        while (slowPtr != fastPtr) {
            slowPtr = slowPtr.getNext();
            fastPtr = fastPtr.getNext();
        }
        return fastPtr;
    } else
        return null;
}
```

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

Problem-13 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-14 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-15 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.

```
private static int findLengthOfTheLoop(ListNode head){
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    boolean loopExists = false;
    while (fastPtr != null && fastPtr.getNext() != null) {
        fastPtr = fastPtr.getNext().getNext();
        slowPtr = slowPtr.getNext();
        if (slowPtr == fastPtr) {
            loopExists = true;
            break;
        }
    }
    int length = 0;
    if (loopExists) {
        do {
            slowPtr = slowPtr.getNext();
            length++;
        } while (slowPtr != fastPtr);
    }
    return length;
}
```

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

Problem-16 Insert a node in a sorted linked list

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

```
public ListNode InsertInSortedList(ListNode head, ListNode newNode) {
        ListNode current = head;
        if(head == null) return newNode;
        // traverse the list until you find item bigger the new node value
        while (current != null && current.getData() <  newNode.getData()){
                temp = current;
                current = current.getNext();
        }
        //  insert the new node before the big item
        newNode.setNext(current);
        temp.setNext(newNode);
        return head;
}
```

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

Problem-17 Reverse a singly linked list.

Solution:

Iterative version:

```
public static ListNode reverseListIterative(ListNode head){
    //initially Current is head
    ListNode current = head;
    //initially previous is null
    ListNode prev = null;
    while (current != null) {
        //Save the next node
        ListNode next = current.getNext();
        //Make current node points to the previous
        current.setNext(prev);
        //update previous
        prev = current;
        //update current
        current = next;
    }
    return prev;
}
```

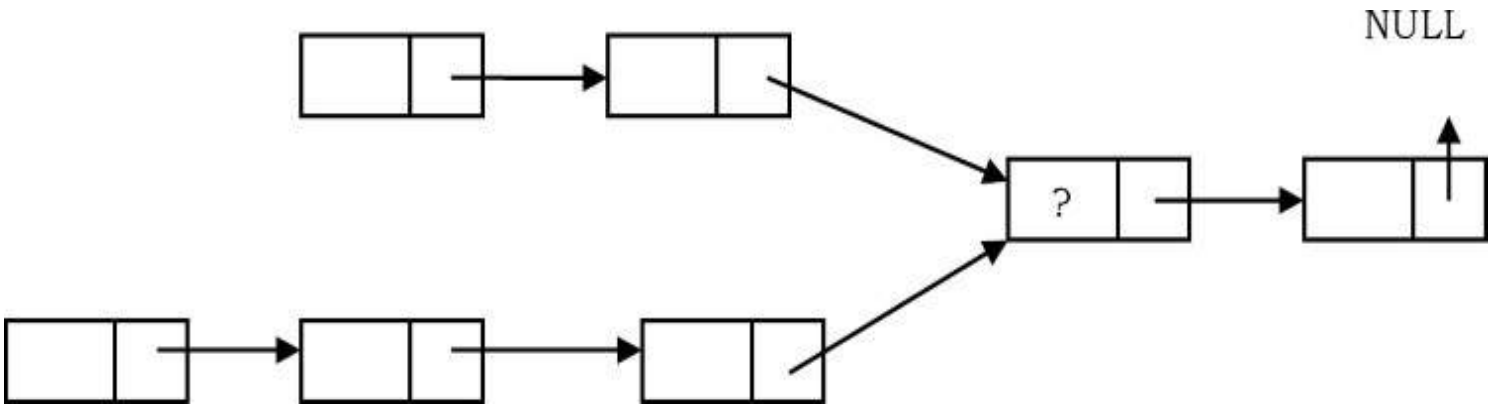
Time Complexity: O(*n*). Space Complexity: O(1).

Recursive version:

```
public static void reverseLinkedListRecursive(ListNode current, ListNode[] head){
    if (current == null) {
        return;
    }
    ListNode next = current.getNext();
    if (next == null) {
        head[0] = current;
        return;
    }
    reverseLinkedListRecursive(next, head);
    //Make next node points to current node
    next.setNext(current);
    //Remove existing link
    current.setNext(null);
}
```

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

Problem-18 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-19 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-20 Can we solve Problem-18 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-21 Can we use stacks for solving Problem-18?

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-22 Is there any other way of solving Problem-18?

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-23 Can we still think of finding an alternative solution for Problem-18?

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-24 Can we improve the complexity for the Problem-18?

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).

```
public static ListNode findIntersectingNode(ListNode list1, ListNode list2) {
    int L1=0, L2=0, diff=0;
    ListNode head1 = list1, head2 = list2;
    while(head1 != null) {
        L1++;
        head1 = head1.getNext();
    }
    while(head2 != null) {
        L2++;
        head2 = head2.getNext();
    }
    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.getNext();
    while(head1 != null && head2 != null) {
            if(head1 == head2)
                return head1.getData();
            head1= head1.getNext();
            head2= head2.getNext();
    }
    return null;
}
```

Problem-25 How will you find the middle of the linked list?

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

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

Problem-26 Can we improve the complexity of Problem-25?

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-27 Can we use the hash table for solving Problem-25?

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-28 Can we solve Problem-25 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 $[n/2]$.

```
public static ListNode findMiddle(ListNode head) {
    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.getNext() != null) {
        if(i == 0) {
            ptr1x = ptr1x.getNext(); //increment only the 1st pointer
            i=1;
        }
        else if( i == 1) {
            ptr1x = ptr1x.getNext(); //increment both pointers
            ptr2x = ptr2x.getNext();
            i = 0;
        }
    }
    return ptr2x;      //now return the ptr2 which points to the middle node
}
```

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

Problem-29 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
public static void printListFromEnd(ListNode head) {
    if(head == null)
        return;
    printListFromEnd(head.getNext());
    System.out.println(head.getData());
}
```

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

Problem-30 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.

```
public int IsLinkedListLengthEven(ListNode listHead) {
    while(listHead != null && listHead.getNext() != null)
        listHead = listHead.getNext().getNext();
    if(listHead == null) return 0;
    return 1;
}
```

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

Problem-31 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-32 Given two sorted Linked Lists, we need to merge them into the third list in sorted order.

Solution:

```
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    if(head1 == null)
        return head2;
    if(head2 == null)
        return head1;
    ListNode head = new ListNode(0);
    if(head1.data <= head2.data){
        head = head1;
        head.next = mergeTwoLists(head1.next, head2);
    }else{
        head = head2;
        head.next = mergeTwoLists(head2.next, head1);
    }
    return head;
}
```

Time Complexity – O(n).

Problem-33 Can we solve Problem-32 without recursion?

Solution:

```
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    ListNode head = new ListNode(0);
    ListNode curr = head;
    while(head1 != null && head2 != null){
        if(head1.data <= head2.data){
            curr.next = head1;
            head1 = head1.next;
        }else{
            curr.next = head2;
            head2 = head2.next;
        }
    }
    if(head1 != null)
        curr.next = head1;
    else if(head2 != null)
        curr.next = head2;
    return head.next;
}
```

Time Complexity – O(n).

Problem-34 Reverse the linked list in pairs. If you have a linked list that holds $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$, then after the function has been called the linked list would hold $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$.

Solution:

```
//Recursive Version
public static ListNode ReversePairRecursive(ListNode head) {
    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 version*/
public static ListNode  ReversePairIterative(ListNode  head) {
    ListNode  temp1 = null;
    ListNode  temp2 = null;
    while (head != null && head.next != null) {
        if (temp1 != null) {
            temp1.next.next = head.next;
        }
        temp1 = head.next;
        head.next = head.next.next;
        temp1.next = head;
        if (temp2 == null)
            temp2 = temp1;
        head = head.next;
    }
    return temp2;
}
```

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

Problem-35 Given a binary tree convert it to doubly linked list.

Solution: Refer *Trees* chapter.

Problem-36 How do we sort the Linked Lists?

Solution: Refer *Sorting* chapter.

Problem-37 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.

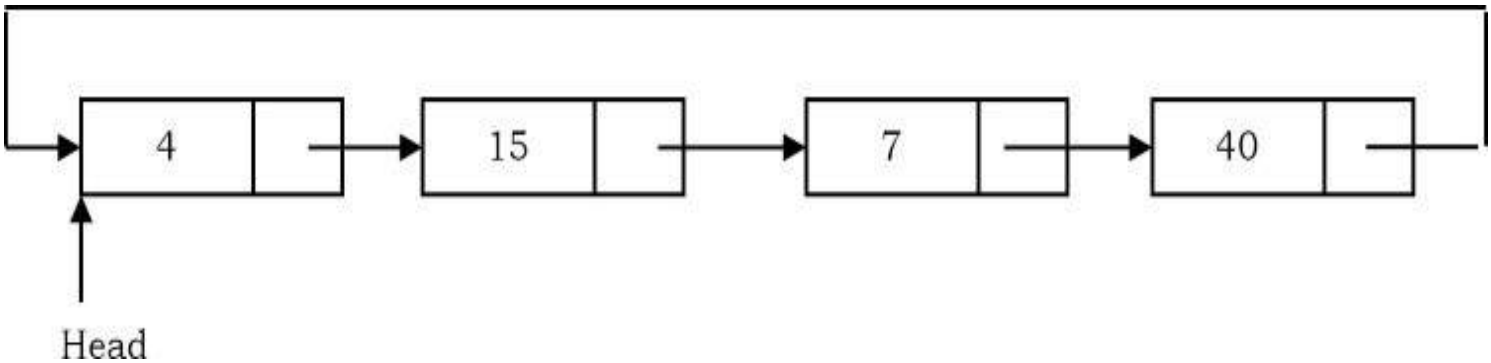
Problem-38 Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

Solution:

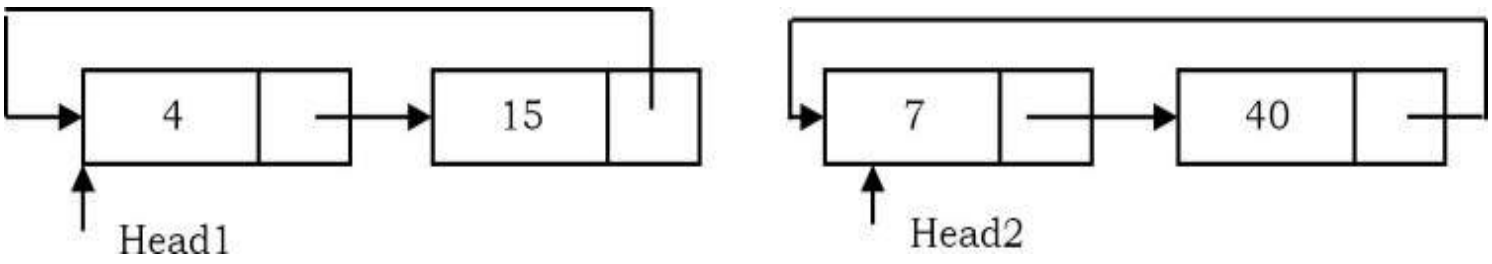
Algorithm

- 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:




```
public static void SplitList(ListNode head, ListNode head1, ListNode head2) {
    ListNode slowPtr = head, fastPtr = head;
    if(head == nlu) return;
    /* If there are odd nodes in the circular list then fastPtr.getNext() becomes
       head and for even nodes fastPtr.getNext().getNext() becomes head */
      while(fastPtr.getNext() != head && fastPtr.getNext().getNext() != head)    {
            fastPtr = fastPtr.getNext().getNext();
            slowPtr = slowPtr.getNext();
    }
    /* If there are even elements in list then move fastPtr */
    if(fastPtr.getNext().getNext() == head)
            fastPtr = fastPtr.getNext();
    /* Set the head pointer of first half */
    head1 = head;
    /* Set the head pointer of second half */
    if(head.getNext() != head)
            head2 = slowPtr.getNext();
    /* Make second half circular */
    fastPtr.setNext(slowPtr.getNext());
    /* Make first half circular */
    slowPtr.setNext(head);
}
```

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

Problem-39 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-40 Exchange the adjacent elements in a link list.

Solution:

```
public ListNode exchangeAdjacentNodes (ListNode head) {
    ListNode temp = new ListNode(0);
    temp.next = head;
    ListNode prev = temp, curr = head;
    while(curr != null && curr.next != null){
        ListNode tmp = curr.next.next;
        curr.next.next = prev.next;
        prev.next = curr.next;
        curr.next = tmp;
        prev = curr;
        curr = prev.next;
    }
    return temp.next;
}
```

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

Problem-41 For a given *K* value (*K* > 0) reverse blocks of if 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.

```
public static ListNode reverseKNodesRecursive(ListNode head, int k){
    ListNode current = head;
    ListNode next = null;
    ListNode prev = null;
    int count = k;
    //Reverse K nodes
    while (current != null && count > 0) {
        next = current.getNext();
        current.setNext(prev);
        prev = current;
        current = next;
        count--;
    }
    //Now next points to K+1 th node, returns the pointer to the head node
    if (next != null) {
        head.setNext(reverseKNodesRecursive(next, k));
    }
    //return head node
    return prev;
}
public static ListNode reverseKNodes(ListNode head, int k){
    //Start with head
    ListNode current = head;
    //last node after reverse
    ListNode prevTail = null;
    //first node before reverse
    ListNode prevCurrent = head;
    while (current != null) {
        //loop for reversing K nodes
        int count = k;
        ListNode tail = null;
        while (current != null && count > 0) {
            ListNode next = current.getNext();
            current.setNext(tail);
            tail = current;
            current = next;
            count--;
        }
        //reversed K Nodes
        if (prevTail != null) {
            //Link this set and previous set
            prevTail.setNext(tail);
        } else {
            //We just reversed first set of K nodes, update head point to the Kth Node
            head = tail;
        }
        //save the last node after reverse since we need to connect to the next set.
        prevTail = prevCurrent;
        //Save the current node, which will become the last node after reverse
        prevCurrent = current;
    }
    return head;
}
```

Problem-42 Can we solve Problem-39 with recursion?

Solution:

```
public static ListNode reverseKNodes(ListNode head, int k){
    //Start with head
    ListNode current = head;
    //last node after reverse
    ListNode prevTail = null;
    //first node before reverse
    ListNode prevCurrent = head;
    while (current != null) {
        //loop for reversing K nodes
        int count = k;
        ListNode tail = null;
        while (current != null && count > 0) {
            ListNode next = current.getNext();
            current.setNext(tail);
            tail = current;
            current = next;
            count--;
        }
        //reversed K Nodes
        if (prevTail != null) {
            //Link this set and previous set
            prevTail.setNext(tail);
        } else {
            //We just reversed first set of K nodes, update head point to the Kth Node
            head = tail;
        }
        //save the last node after reverse since we need to connect to the next set.
        prevTail = prevCurrent;
        //Save the current node, which will become the last node after reverse
        prevCurrent = current;
    }
    return head;
}
```

Problem-43 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.

Problem-44 **JosephusCircle:** N people have decided to elect a leader by arranging themselves in a circle and eliminating every M^{th} person around the circle, closing ranks as each person drops out. Find which person will be the last one remaining (with rank 1).

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

```
public ListNode  GetJosephusPosition(int N, int M) {
    ListNode p, q;
    // Create circular linked list containing all the players:
    p.setData(1);
    q = p;
    for (int i = 2; i <= N; ++i) {
            p = p.getNext();
            p.setData(i);
    }
    p.setNext(q);  // Close the circular linked list by having the last node point to the first.
    // Eliminate every M-th player as long as more than one player remains:
    for (int count = N; count > 1; --count) {
            for (int i = 0; i < M - 1; ++i)
                    p = p.getNext();
            p.setNext(p.getNext().getNext());  // Remove the eiminated player from the list.
    }
    System.out.println("Last player left standing (Josephus Position) is " + p.getData());
}
```

Problem-45 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 the hash table to associate newly created nodes with the instances of node in the given list.

Algorithm:

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

- Now for each node *X* in the original list we have a copy *Y* stored in our hash table. We scan the original list again and set the pointers building the new list.

```
public RandomListNode copyRandomList(RandomListNode head) {
    RandomListNode X = head, Y;
    Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode, RandomListNode>();
    while(X != null) {
        Y = new RandomListNode(X.label);
        Y.next = null;
        Y.random = null;
        map.put(X, Y);
        X = X.next;
    }
    X = head;
    while(X != null){
        Y = map.get(X);
        Y.next = map.get(X.next);
        Y.random = map.get(X.random);
        X = X.next;
    }
    return map.get(head);
}
```

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

Problem-46 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(*n*1*ogn*)

Solution: A.

Problem-47 **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.

```
public ListNode modularNodes(ListNode head, int k){
    ListNode modularNode;
    int i=0;
    if(k<=0)
        return null;
    for (;head!= null; head = head.getNext()){
        if(i%k == 0){
            modularNode = head;
        }
        i++;
    }
    return modularNode;
}
```

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

Problem-48 **Find modular node from the end:** Given a singly linked list, write a function to find the first element from the end 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 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.

```
public ListNode modularNodes(ListNode *head, int k){
    ListNode modularNode=null;
    int i=0;
    if(k<=0) return null;
    for (i=0; i < k; i++){
        if(head!=null)
            head = head.getNext();
        else
            return null;
    }
    while(head!= null)
        modularNode = modularNode.getNext();
        head = head.getNext();
    }
    return modularNode;
}
```

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

Problem-49 **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.

```
public ListNode fractionalNodes(ListNode head, int k){
    ListNode fractionalNode;
    int i=0;
    if(k<=0)
        return null;
    for (;head!= null; head = head.getNext()){
        if(i%k == 0){
            if(fractionalNode!=null)
                fractionalNode = head;
            else fractionalNode = fractionalNode.getNext();
        }
        i++;
    }
    return fractionalNode;
}
```

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

Problem-50 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-51 Consider the following Java program code, whose runtime F is a function of the input size *n*.

```
java.util.ArrayList<Integer> list = new java.util.ArrayList<Integer>();
for( int i = 0 ; i < n; i ++)
    list.add (0 , i ) ;
```

Which of the following is correct?

A) $F(n)=\Theta(n)$
B) $F(n)=\Theta(n^2)$
C) $F(n)=\Theta(n^3)$
D) $F(n)=\Theta(n^4)$
F) $F(n)=\Theta(nlogn)$

Solution: B. The operation list.add(0,i) on ArrayList has linear time complexity, with respect to the current size of the data structure. Therefore, overall we have quadratic time complexity.

Problem-52 Consider the following Java program code, whose runtime F is a function of the input size *n*.

```
java.util.ArrayList<Integer> list = new java.util.ArrayList<Integer>();
for(int i = 0 ; i < n; i ++)
    list.add ( i , i ) ;
for(int j = 0 ; j < n; j++)
    list.remove(n-j -1 );
```

Which of the following is correct?

A) $F(n)=\Theta(n)$
B) $F(n)=\Theta(n^2)$
C) $F(n)=\Theta(n^3)$
D) $F(n)=\Theta(n^4)$
F) $F(n)= \Theta(nlogn)$

Solution: A. Both loop bodies have constant time complexity because they operate the end of the ArrayList.

Problem-53 Consider the following Java program code, whose runtime F is a function of the input size *n*.

```
java.util.LinkedList<Integer> k = new java.util.LinkedList<Integer>();
for(int i = 0 ; i < n; i ++)
   for(int j = 0 ; j < n; j++)
        k.add(k.size()/2,j);
```

Which of the following is correct?

Solution: D. The LinkedList grows to quadratic size during the execution of the program fragment. Thus the body of the inner loop has quadratic complexity. The inner loop itself is executed n^2 times.

Problem-54 Given a singly linked list L: L_1-> L_2-> L_3...-> L_{n-1}-> L_n, reorder it to: L_1-> L_n-> L_2-> L_{n-1}.....

Solution: We divide the list in two parts for instance 1->2->3->4->5 will become 1->2->3 and 4->5, we have to reverse the second list and give a list that alternates both. The split is done with a slow and fast pointer. First solution (using a stack for reversing the list):

```
public class ReorderLists {
    public void reorderList(ListNode head) {
        if(head == null)
            return;
        ListNode slowPointer = head;
        ListNode fastPointer = head.next;
        while(fastPointer!=null && fastPointer.next !=null){
            fastPointer = fastPointer.next.next;
            slowPointer = slowPointer.next;
        }
        ListNode head2 = slowPointer.next;
        slowPointer.next = null;
        LinkedList<ListNode> queue = new LinkedList<ListNode>();
        while(head2!=null){
            ListNode temp = head2;
            head2 = head2.next;
            temp.next =null;
            queue.push(temp);
        }
        while(!queue.isEmpty()){
            ListNode temp = queue.pop();
            temp.next = head.next;
            head.next = temp;
            head = temp.next;
        }
    }
}
```

Alternative Approach:

```
public class ReorderLists {
    public void reorderList(ListNode head) {
        if(head == null)
            return;
        ListNode slowPointer = head;
        ListNode fastPointer = head.next;
        while(fastPointer!=null && fastPointer.next !=null){
            fastPointer = fastPointer.next.next;
            slowPointer = slowPointer.next;
        }
        ListNode head2 = slowPointer.next;
        slowPointer.next = null;
        head2= reverseList(head2);
        alternate (head, head2);
    }
    private ListNode reverseList(ListNode head){
        if (head == null)
            return null;
        ListNode reversedList = head;
        ListNode pointer = head.next;
        reversedList.next=null;
        while (pointer !=null){
            ListNode temp = pointer;
            pointer = pointer.next;
            temp.next = reversedList;
            reversedList = temp;
        }
        return reversedList;
    }
    private void alternate (ListNode head1, ListNode head2){
        ListNode pointer = head1;
        head1 = head1.next;
        boolean nextList1 = false;
        while(head1 != null || head2 != null){
            if((head2 != null && !nextList1) || (head1==null)){
                pointer.next = head2;
                head2 = head2.next;
                nextList1 = true;
                pointer = pointer.next;
            }
            else {
                pointer.next = head1;
                head1 = head1.next;
                nextList1 = false;
                pointer = pointer.next;
            }
        }
    }
}
```

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

Problem-55 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-56 How do you implement insertion sort for linked lists?

Solution:

```
public static ListNode insertionSortList(ListNode head) {
    if (head == null || head.next == null)
        return head;
    ListNode newHead = new ListNode(head.data);
    ListNode pointer = head.next;
    // loop through each element in the list
    while (pointer != null) {
        // insert this element to the new list
        ListNode innerPointer = newHead;
        ListNode next = pointer.next;
        if (pointer.data <= newHead.data) {
            ListNode oldHead = newHead;
            newHead = pointer;
            newHead.next = oldHead;
        } else {
            while (innerPointer.next != null) {
                if (pointer.data > innerPointer.data && pointer.data <= innerPointer.next.data) {
                    ListNode oldNext = innerPointer.next;
                    innerPointer.next = pointer;
                    pointer.next = oldNext;
                }
                innerPointer = innerPointer.next;
            }
            if (innerPointer.next == null && pointer.data > innerPointer.data) {
                innerPointer.next = pointer;
                pointer.next = null;
            }
        }
        // finally
        pointer = next;
    }
    return newHead;
}
```

Note: For details on insertion sort refer Sorting chapter.

Problem-57 Given a list, rotate the list to the right by k places, where k is non-negative. For example: Given 1_->2->3->4->5->NULL and k = 2, return 4->5->1->2->3->NULL.

Solution:

```
public ListNode rotateRight(ListNode head, int n) {
    if(head == null || head.next == null)
        return head;
    ListNode rotateStart = head, rotateEnd = head;
    while(n-- > 0){
        rotateEnd = rotateEnd.next;
        if(rotateEnd == null){
            rotateEnd = head;
        }
    }
    if(rotateStart == rotateEnd)
        return head;
    while(rotateEnd.next != null){
        rotateStart = rotateStart.next;
        rotateEnd = rotateEnd.next;
    }
    ListNode temp = rotateStart.next;
    rotateEnd.next = head;
    rotateStart.next = null;
    return temp;
}
```

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

Problem-58 You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list. For example with input: (3 -> 4 -> 3) + (5 -> 6 -> 4); the output should be 8 -> 0 -> 8.

Solution:

```
public ListNode addTwoNumbers(ListNode list1, ListNode list2) {
    if(list1 == null)
        return list2;
    if(list2 == null)
        return list1;
    ListNode head = new ListNode(0);
    ListNode cur = head;
    int advance = 0;
    while(list1 != null && list2 != null){
        int sum = list1.data + list2.data + advance;
        advance = sum / 10;
        sum = sum % 10;
        cur.next = new ListNode(sum);
        cur = cur.next;
        list1 = list1.next;
        list2 = list2.next;
    }
    if(list1 != null){
        if(advance != 0)
            cur.next = addTwoNumbers(list1, new ListNode(advance));
        else
            cur.next = list1;
    }else if(list2 != null){
        if(advance != 0)
            cur.next = addTwoNumbers(list2, new ListNode(advance));
        else
            cur.next = list2;
    }else if(advance != 0){
        cur.next = new ListNode(advance);
    }
    return head.next;
}
```

Problem-59 Given a linked list and a value K, partition it such that all nodes less than K come before nodes greater than or equal to K. You should preserve the original relative order of the nodes in each of the two partitions. For example, given 1->4->3->2->5->2 and K = 3, return 1->2->2->4->3->5.

Solution:

```
public ListNode partition(ListNode head, int K) {
    ListNode root = new ListNode(0);
    ListNode pivot = new ListNode(K);
    ListNode rootNext = root, pivotNext = pivot;
    ListNode currentNode = head;
    while(currentNode != null){
        ListNode next = currentNode.next;
        if(currentNode.data >= K){
            pivotNext.next = currentNode;
            pivotNext = currentNode;
            pivotNext.next = null;
        }else{
            rootNext.next = currentNode;
            rootNext = currentNode;
        }
        currentNode = next;
    }
    rootNext.next = pivot.next;
    return root.next;
}
```

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

Problem-60 Merge k sorted linked lists and return it as one sorted list.

Solution: Refer Priority Queues chapter.

Problem-66 Given a unordered linked list, how do you remove duplicates in it?

Solution:

```
public static void removeDuplicates2(ListNode head) {
        ListNode curr = head;
        if(curr == null || curr.getNext() == null) {
                return; // 0 or 1 nodes in the list so no duplicates
        }
        ListNode curr2;
        ListNode prev;
        while(curr != null) {
                curr2 = curr.getNext();
                prev = curr;
                while(curr2 != null) {
                        // check and see if curr and curr2 values are the same, if they are then delete curr2
                        if(curr.getData()==curr2.getData()) {
                                prev.setNext(curr2.getNext());
                        }
                        prev = curr2;
                        curr2 = curr2.getNext();
                }
                curr = curr.getNext();
        }
}
```

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

Problem-67 Can reduce the time complexity of Problem-61?

Solution: We can simply use hash table and check whether an element already exist.

```
// using a temporary buffer O(n)
public static void removeDuplicates(ListNode head) {
        Map<Integer, Boolean> mapper = new HashMap<Integer, Boolean>();
        ListNode curr = head;
        ListNode next;
        while (curr.getNext() != null) {
                next = curr.getNext();
                if(mapper.get(next.getData()) != null) {
                        // already seen it, so delete
                        curr.setNext(next.getNext());
                } else {
                        mapper.put(next.getData(), true);
                        curr = curr.getNext();
                }
        }
}
```

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

Problem-68 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-69 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 length of list1 and *n* is the length of list2. Space Complexity: O(1).

```
public static ListNode commonElement(ListNode list1, ListNode list2) {
        ListNode temp =  new ListNode(0);
        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

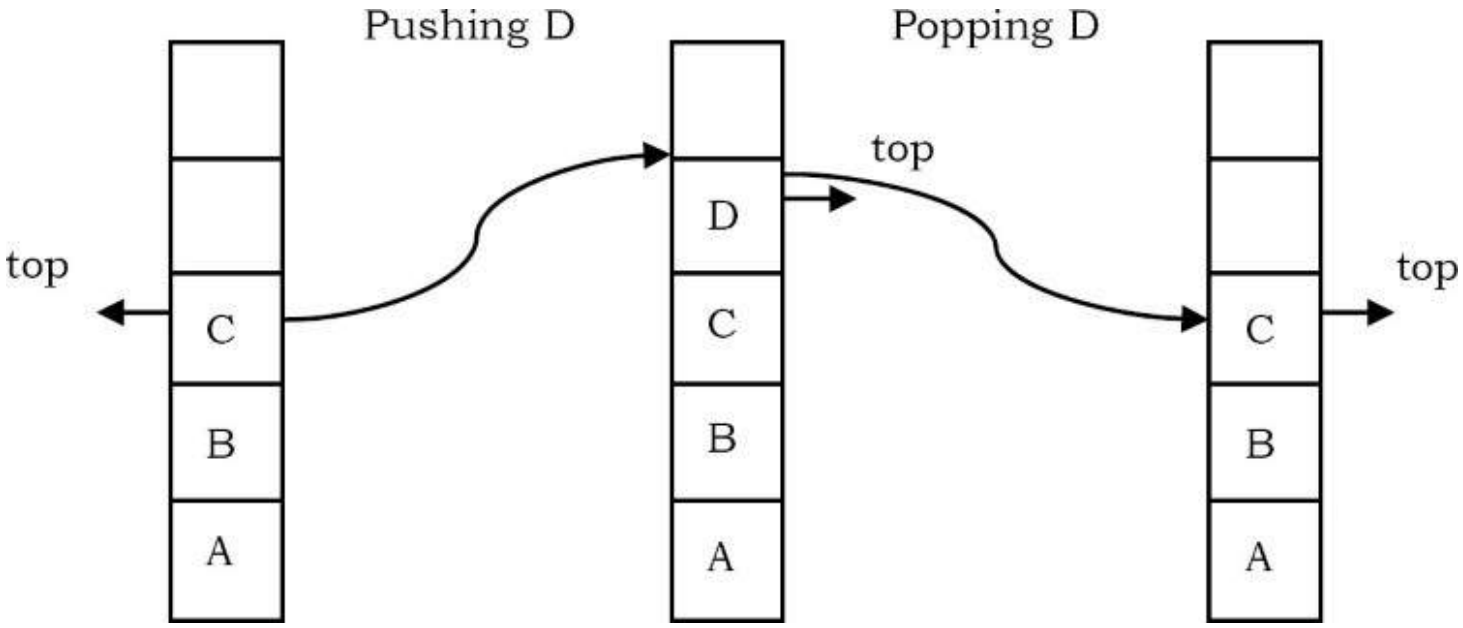
4.1 What is a Stack?

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

- void 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.

4.4 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.

4.5 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)

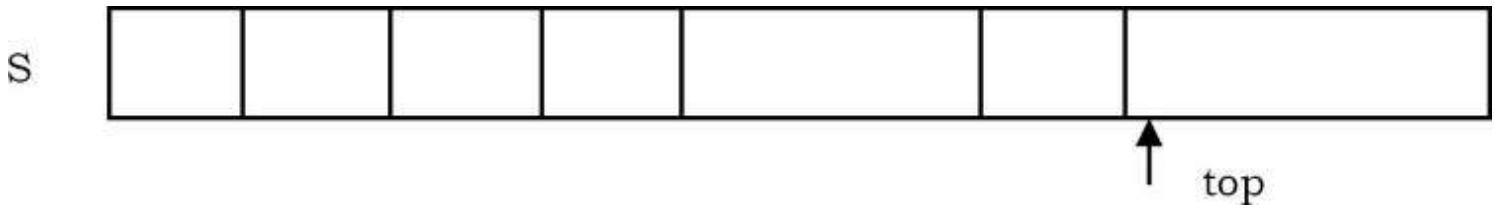
4.6 Implementation

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*.

```
public class FixedSizeArrayStack{
    // Length of the array used to implement the stack.
    protected int capacity;
    // Default array capacity.
    public static final int CAPACITY = 10;
    // Array used to implement the stack.
    protected int[] stackRep;
    // Index of the top element of the stack in the array.
    protected int top = -1;
    // Initializes the stack to use an array of default length.
    public FixedSizeArrayStack() {
            this(CAPACITY); // default capacity
    }
    // Initializes the stack to use an array of given length.
    public FixedSizeArrayStack(int cap) {
            capacity = cap;
            stackRep = new int[capacity]; // compiler may give warning, but this is ok
    }
    // Returns the number of elements in the stack. This method runs in O(1) time.
    public int size() {
            return (top + 1);
    }
    // Testes whether the stack is empty. This method runs in O(1) time.
    public boolean isEmpty() {
            return (top < 0);
    }
    // Inserts an element at the top of the stack. This method runs in O(1) time.
    public void push(int data) throws Exception {
            if (size() == capacity)
                    throw new Exception("Stack is full.");
            stackRep[++top] = data;
    }
    // Inspects the element at the top of the stack. This method runs in O(1) time.
    public int top() throws Exception {
            if (isEmpty())
                    throw new Exception("Stack is empty.");
            return stackRep[top];
    }
    // Removes the top element from the stack. This method runs in O(1) time.
    public int pop() throws Exception {
            int data;
            if (isEmpty())
               throw new Exception("Stack is empty.");
            data = stackRep[top];
            stackRep[top--] = Integer.MIN_VALUE;
            return data;
    }
    // Returns a string representation of the stack as a list of elements, with
    // the top element at the end: [ ... , prev, top ]. This method runs in O(n)
    // time, where n is the size of the stack.
    public String toString() {
            String s;
            s = "[";
            if (size() > 0)
                    s += stackRep[0];
            if (size() > 1)
                    for (int i = 1; i <= size() - 1; i++) {
                            s += ", " + stackRep[i];
                    }
            return 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 isEmpty()	O(1)
Time Complexity of isFullStack()	O(1)
Time Complexity of deleteStack()	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 $T(n)$ (number 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 $2n$ value (32). If we observe carefully, we are doing the doubling operation *logn* times.

Now, let us generalize the discussion. For n push operations we double the array size *logn* times. That means, we will have *logn* terms in 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 $O(1)$.

```
public class DynamicArrayStack{
    // Length of the array used to implement the stack.
    protected int capacity;
```

```
// Default array capacity.
public static final int CAPACITY = 16;     // power of 2
public static int MINCAPACITY=1<<15;   // power of 2
// Array used to implement the stack.
protected int[] stackRep;
// Index of the top element of the stack in the array.
protected int top = -1;
// Initializes the stack to use an array of default length.
public DynamicArrayStack() {
        this(CAPACITY);                    // default capacity
}
// Initializes the stack to use an array of given length.
public DynamicArrayStack(int cap) {
        capacity = cap;
        stackRep = new int[capacity];     // compiler may give warning, but this is ok
}
// Returns the number of elements in the stack. This method runs in O(1) time.
public int size() {
        return (top + 1);
}
// Testes whether the stack is empty. This method runs in O(1) time.
public boolean isEmpty() {
        return (top < 0);
}
// Inserts an element at the top of the stack. This method runs in O(1) time.
public void push(int data) throws Exception {
        if (size() == capacity)
            expand();
        stackRep[++top] = data;
}
private void expand() {
        int length = size();
        int[] newstack=new int[length<<1];
        System.arraycopy(stackRep,0,newstack,0,length);
        stackRep=newstack;
        this.capacity = this.capacity<<1;
}
// dynamic array operation: shrinks to 1/2 if more than than 3/4 empty
private void shrink() {
        int length = top + 1;
        if(length<=MINCAPACITY || top<<2 >= length)
            return;
        length=length + (top<<1); // still means shrink to at 1/2 or less of the heap
        if(top<MINCAPACITY) length = MINCAPACITY;
        int[] newstack=new int[length];
        System.arraycopy(stackRep,0,newstack,0,top+1);
        stackRep=newstack;
        this.capacity = length;
}
// Inspects the element at the top of the stack. This method runs in O(1) time.
public int top() throws Exception {
        if (isEmpty())
            throw new Exception("Stack is empty.");
        return stackRep[top];
}
// Removes the top element from the stack. This method runs in O(1) time.
public int pop() throws Exception {
        int data;
        if (isEmpty())
                throw new Exception("Stack is empty.");
        data = stackRep[top];
        stackRep[top--] = Integer.MIN_VALUE; // dereference S[top] for garbage collection.
        shrink();
        return data;
```

```
    }
    // Returns a string representation of the stack as a list of elements, with
    // the top element at the end: [ ... , prev, top ]. This method runs in O(n)
    // time, where n is the size of the stack.
    public String toString() {
        String s;
        s = "[";
        if (size() > 0)
            s += stackRep[0];
        if (size() > 1)
            for (int i = 1; i <= size() - 1; i++) {
                s += ", " + stackRep[i];
            }
        return s + "]";
    }
}
```

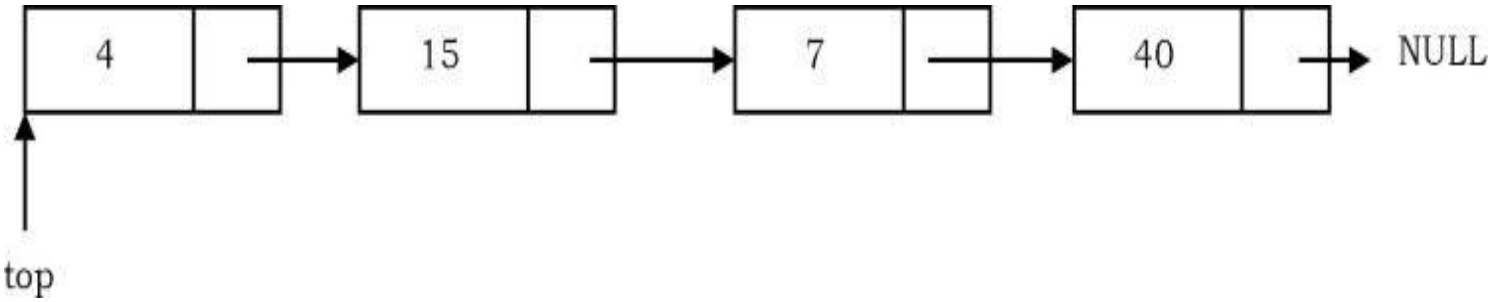
Performance: Let *n* be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of create Stack: DynArrayStack ()	O(1)
Time Complexity of push()	O(1) (Average)
Time Complexity of pop()	O(1)
Time Complexity of top()	O(1)
Time Complexity of isEmpty()	O(1)
Time Complexity of isStackFull ()	O(1)
Time Complexity of deleteStack()	O(1)

Note: Too many doublings may cause memory overflow exception.

Linked List Implementation

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


4
15
7
40
NULL
top

```
public class LinkedStack<T>{
    private int length;                    // indicates the size of the linked list
    private ListNode top;
    //  Constructor: Creates an empty stack.
    public LinkedStack()  {
            length = 0;
            top = null;
    }
    //  Adds the specified data to the top of this stack.
    public void push (int data)  {
            ListNode temp = new ListNode (data);
            temp.setNext(top);
            top = temp;
            length++;
    }
    //  Removes the data at the top of this stack and returns a
    //  reference to it. Throws an EmptyStackException if the stack
    //  is empty.
    public int pop() throws EmptyStackException{
            if (isEmpty())
                    throw new EmptyStackException();
            int result = top.getdata();
            top = top.getNext();
            length--;
            return result;
    }
    // Returns a reference to the data at the top of this stack.
    // The data is not removed from the stack.  Throws an
    // EmptyStackException if the stack is empty.
    public int peek() throws EmptyStackException{
            if (isEmpty())
                    throw new EmptyStackException();
            return top.getdata();
    }
    // Returns true if this stack is empty and false otherwise.
    public boolean isEmpty(){
            return (length == 0);
    }
    // Returns the number of elements in the stack.
    public int size(){
            return length;
    }
    // Returns a string representation of this stack.
    public String toString(){
            String result = "";
            ListNode current = top;
            while (current != null){
                    result = result + current.toString() + "\n";
                    current = current.getNext();
            }
            return result;
    }
}
```

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 create Stack: DynArrayStack()	O(1)
Time Complexity of push()	O(1) (Average)
Time Complexity of pop()	O(1)
Time Complexity of top()	O(1)
Time Complexity of isEmpty()	O(1)
Time Complexity of deleteStack()	O(n)

4.7 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 reasoning, refer to the *Implementation* section.

Comparing Array Implementation & 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.

4.8 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

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].

```
public boolean isValidSymbolPattern(String s) {
    Stack<Character> stk = new Stack<Character>();
    if(s == null || s.length() == 0)
        return true;
    for(int i = 0; i < s.length(); i++){
        if(s.charAt(i) == ')'){
            if(!stk.isEmpty() && stk.peek() == '(')
                stk.pop();
            else
                return false;
        }else if(s.charAt(i) == ']'){
            if(!stk.isEmpty() && stk.peek() == '[')
                stk.pop();
            else
                return false;
        }else if(s.charAt(i) == '}'){
            if(!stk.isEmpty() && stk.peek() == '{')
                stk.pop();
            else
                return false;
        }else{
            stk.push(s.charAt(i));
        }
    }
    if(stk.isEmpty())
        return true;
    else
        return false;
}
```

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

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

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

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

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

A
+AB
++AB-CD

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

A
AB+
AB+CD-+

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

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

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

Algorithm

a) Create a stack
b) for each character t in the input stream

```
if(t is an operand)  output t
else if(t is a right parenthesis)
        Pop and output tokens until a left parenthesis is popped (but not output)
else // t is an operator or left parenthesis
        pop and output tokens until one of lower priority than t is encountered or a left  parenthesis
        is encountered or the stack is empty
        Push t
```

c) pop and output tokens until the stack is empty

For better understanding let us trace out 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 For a given array with *n* symbols how many stack permutations are possible?

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

Problem-4 Discuss postfix evaluation using stacks?

Solution:

Algorithm:

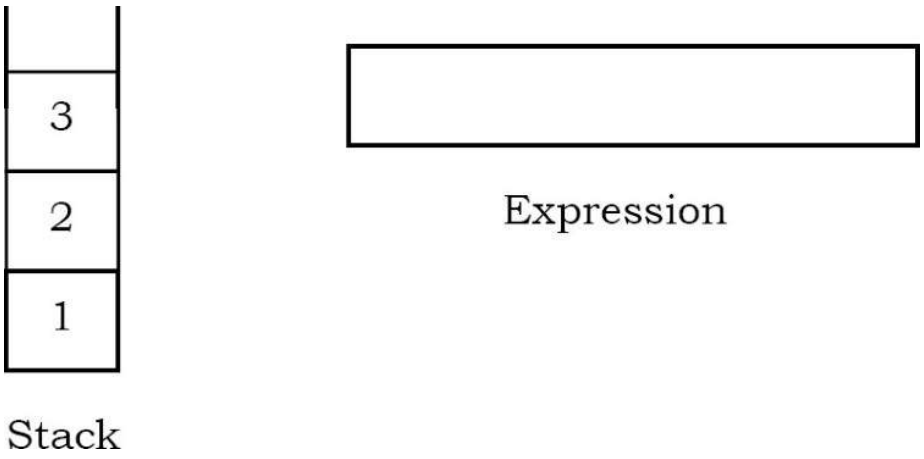
1 Scan the Postfix string from left to right.
2 Initialize an empty stack.
3 Repeat steps 4 and 5 till all the characters are scanned.
4 If the scanned character is an operand, push it onto the stack.
5 If the scanned character is an operator, and if the operator is a unary operator, then

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.

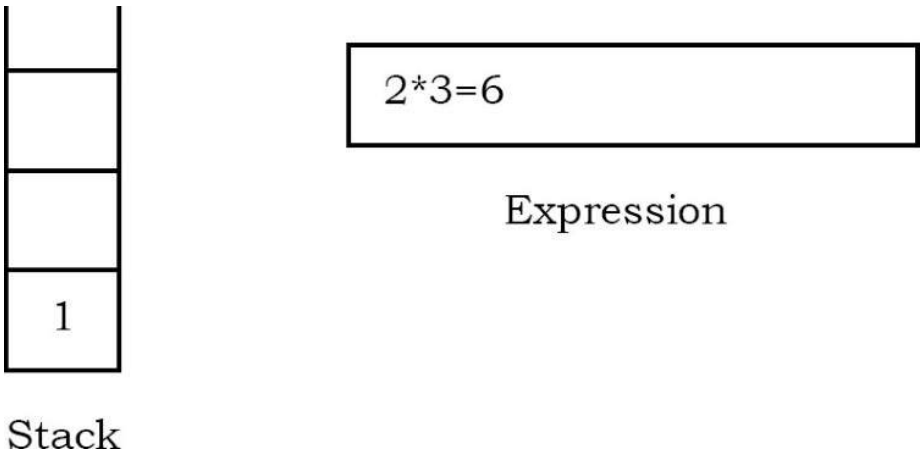
6 After all characters are scanned, we will have only one element in the stack.

7 Return top of the stack as result.

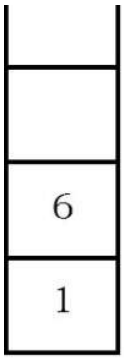
Example: Let us see how the above algorithm works using an example. Assume that the postfix string is 123*+5-. Initially the stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. They will be pushed into the stack in that order.



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.

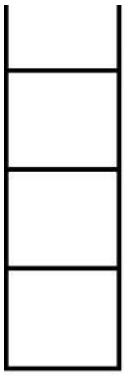


Stack

Expression

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.

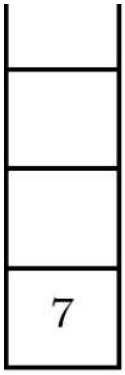
The second operand will be the first element that is popped.



Stack

Expression

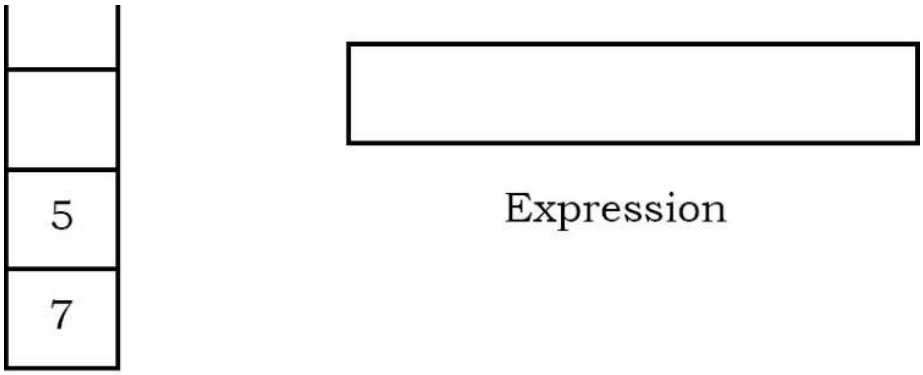
The value of the expression (1+6) that has been evaluated (7) is pushed into the stack.



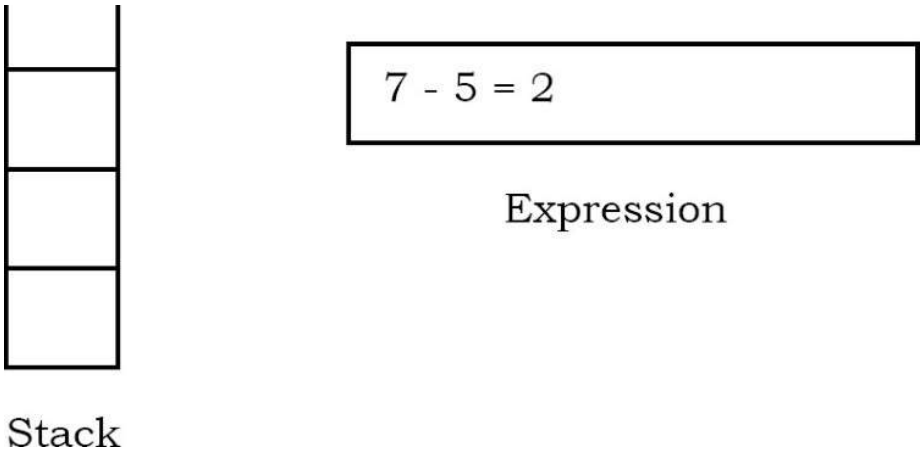
Stack

Expression

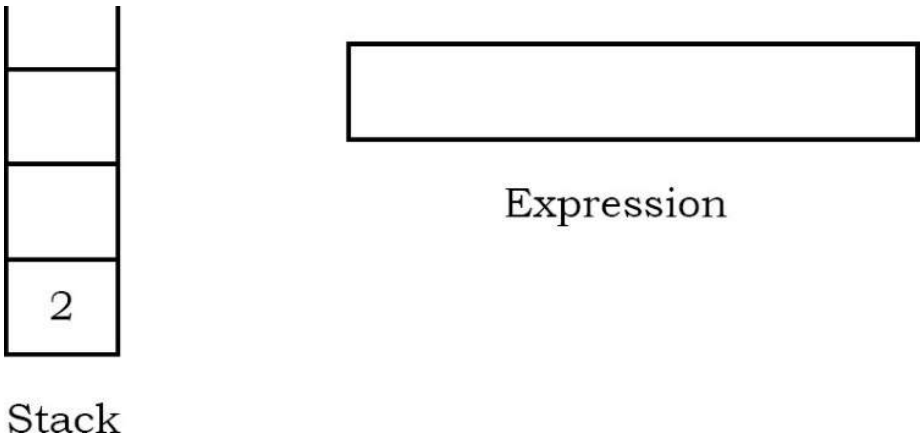
The next character scanned is “5”, which is added to the stack.



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

```
public int expressionEvaluation(String[] tokens) {
    Stack<Integer> s = new Stack<Integer>();
    for(String token : tokens){
        if(token.equals("+")){
            int op1 = s.pop();
            int op2 = s.pop();
            int res = op1+op2;
            s.push(res);
        }else if(token.equals("-")){
            int op1 = s.pop();
            int op2 = s.pop();
            int res = op2-op1;
            s.push(res);
        }else if(token.equals("*")){
            int op1 = s.pop();
            int op2 = s.pop();
            int res = op1 * op2;
            s.push(res);
        }else if(token.equals("/")){
            int op1 = s.pop();
            int op2 = s.pop();
            int res = op2 / op1;
            s.push(res);
        }else{
            s.push(Integer.parseInt(token));
        }
    }
    return s.pop();
}
```

Problem-5 Can we evaluate the infix expression with stacks in one pass?

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

Algorithm

1) Create an empty operator stack
2) Create an empty operand stack
3) For each token in the input string
    a. Get the next token in the infix string
    b. If next token is an operand, place it on the operand stack
    c. If next token is an operator

i. Evaluate the operator (next op)

4) While operator stack is not empty, pop operator and operands (left and right), evaluate left operator right and push result onto operand stack
5) Pop result from operator stack

Problem-6 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.

```
public class AdvancedStack implements Stack{
    private Stack elementStack = new LLStack();
    private Stack minStack = new LLStack();
    public void push(int data){
        elementStack.push(data);
        if(minStack.isEmpty() || (Integer)minStack.top() >= (Integer)data){
            minStack.push(data);
        }else{
            minStack.push(minStack.top());
        }
    }
    public int pop(){
        if(elementStack.isEmpty()) return null;
        minStack.pop();
        return elementStack.pop();
    }
    public int getMinimum(){
        return minStack.top();
    }
    public int top(){
        return elementStack.top();
    }
    public boolean isEmpty(){
        return elementStack.isEmpty();
    }
}
```

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-7 For Problem-6 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.

```
public class AdvancedStack  implements Stack{
    private Stack elementStack = new LLStack();
    private Stack minStack = new LLStack();
    public void Push(int data){
        elementStack.push(data);
        if(minStack.isEmpty() || (Integer)minStack.top() >= (Integer)data){
            minStack.push(data);
        }
    }
    public int Pop(){
        if(elementStack.isEmpty())
            return null;
        Integer minTop = (Integer) minStack.top();
        Integer elementTop = (Integer) elementStack.top();
        if(minTop.intValue() == elementTop.intValue())
            minStack.pop();
        return elementStack.pop();
    }
    public int GetMinimum(){
        return minStack.top();
    }
    public int Top(){
        return elementStack.top();
    }
    public boolean isEmpty(){
        return elementStack.isEmpty();
    }
}
```

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-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.

```
public int isPalindrome(String inputStr) {
    int i=0, j = inputStr.length;
    while(i < j && A[i] == A[j]) {
        i++;
        j--;
    }
    if(i < j ) {
        System.out.println("Not a Palindrome");
        return 0;
    }
    else {
        System.out.println("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 to *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.

```
public boolean isPalindrome(String inputStr){
    char inputChar[] = inputStr.toCharArray();
    Stack s = new LLStack();
    int i=0;
    while(inputChar[i] != 'X'){
        s.push(inputChar[i]);
        i++;
    }
    i++;
    while(i<inputChar.length){
        if(s.isEmpty()) return false;
        if(inputChar[i] != ((Character)s.pop()).charValue()) return false;
        i++;
    }
    return true;
}
```

Time Complexity: O(n). Space Complexity: O(n/2) ≈ O(n).

Problem-11 Given a stack, how to reverse the contents of the stack using only stack operations (push and 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.

```
public class StackReversal {
    public static void reverseStack(Stack stack){
        if(stack.isEmpty()) return;
        int temp = stack.pop();
        reverseStack(stack);
        insertAtBottom(stack, temp);
    }
    private static void insertAtBottom(Stack stack , int data){
        if(stack.isEmpty()){
            stack.push(data);
            return;
        }
        int temp = stack.pop();
        insertAtBottom(stack, data);
        stack.push(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.

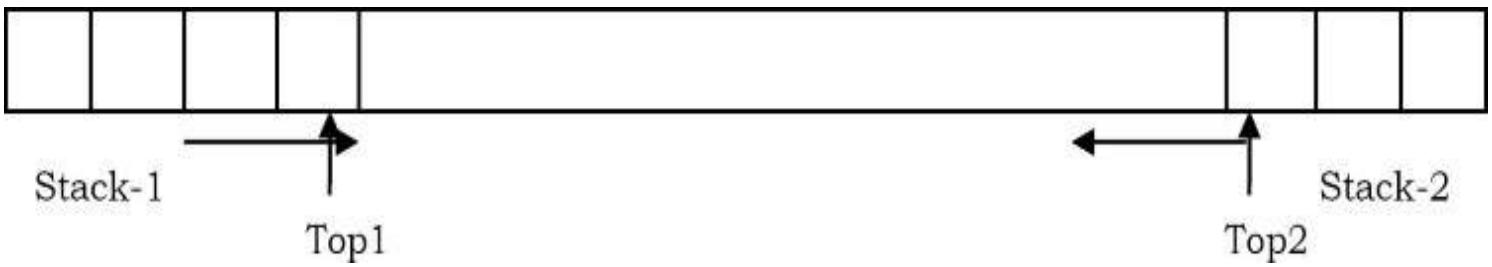
Solution: Refer *Queues* chapter.

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.

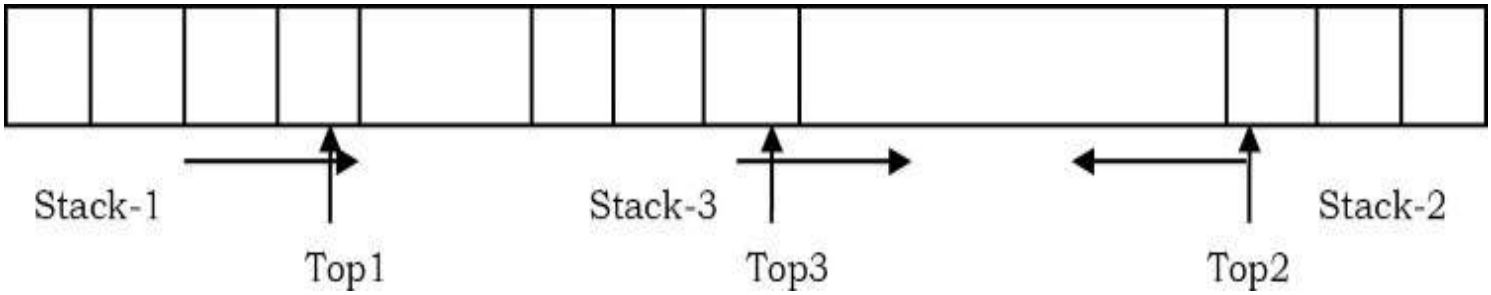
- If we want to push an element into the first stack then put the element at the left index.
- Similarly, if we want to push an element into the second stack then put the element at the right index.
- The first stack grows towards the right, and the second stack grows towards the left.

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

```
public class ArrayWithTwoStacks{
	private int[] dataArray;
	private int size, topOne,  topTwo;
	public ArrayWithTwoStacks(int size){
		if(size<2) throw new IllegalStateException("size < 2 is no persmissible");
		dataArray = new int[size];
		this.size = size;
		topOne = -1;
		topTwo = size;
	}
	public void push(int stackId, int data){
		if(topTwo == topOne+1) throw new StackOverflowException("Array is full");
		if(stackId == 1){
			dataArray[++topOne] = data;
		}else if(stackId == 2){
			dataArray[--topTwo] = data;
		}else return;
	}
	public int pop(int stackId){
		if(stackId == 1){
			if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
			int toPop = dataArray[topOne];
			dataArray[topOne--] = null;
			return toPop;
		}else if(stackId == 2){
			if(topTwo == this.size) throw new EmptyStackException("Second Stack is Empty");
			int toPop = dataArray[topTwo];
			dataArray[topTwo++] = null;
			return toPop;
		}else return null;
	}
	public int top(int stackId){
		if(stackId == 1){
			if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
				return dataArray[topOne];
			}else if(stackId == 2){
				if(topTwo == this.size)
					throw new EmptyStackException("Second Stack is Empty");
				return dataArray[topTwo];
			}else return null;
	}
	public boolean isEmpty(int stackId){
		if(stackId == 1){
			return topOne == -1;
		}else if(stackId == 2){
			return topTwo == this.size;
		}else return true;
	}
}
```

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 (Top 1): 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).
- op 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).

One Possible Implementation

```
public class ArrayWithThreeStacks {
	private int[] dataArray;
	private int size, topOne, topTwo, baseThree, topThree;
	public ArrayWithThreeStacks(int size){
		if(size<3) throw new IllegalStateException("Size < 3 is no persmissible");
		dataArray = new int[size];
		this.size = size;
		topOne = -1;
		topTwo = size;
		baseThree = size/2;
		topThree = baseThree;
	}
	public void push(int stackId, int data){
		if(stackId == 1){
			if(topOne+1 == baseThree){
				if(stack3IsRightShiftable()){
					shiftStack3ToRight();
					dataArray[++topOne] = data;
				}else throw new StackOverflowException("Stack1 has reached max limit");
			}else dataArray[++topOne] = data;
		}else if(stackId == 2){
			if(topTwo-1 == topThree){
				if(stack3IsLeftShiftable()){
					shiftStack3ToLeft();
					dataArray[--topTwo] = data;
				}else throw new StackOverflowException("Stack2 has reached max limit");
			}else dataArray[--topTwo] = data;
		}else if(stackId == 3){
			if(topTwo-1 == topThree){
				if(stack3IsLeftShiftable()){
					shiftStack3ToLeft();
					dataArray[++topThree] = data;
				}else throw new StackOverflowException("Stack3 has reached max limit");
			}else dataArray[++topThree] = data;
		}else return;
	}
	public int pop(int stackId){
		if(stackId == 1){
			if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
			int toPop = dataArray[topOne];
			dataArray[topOne--] = null;
			return toPop;
		}else if(stackId == 2){
```

```
                if(topTwo == this.size) throw new EmptyStackException("Second Stack is Empty");
                int toPop = dataArray[topTwo];
                dataArray[topTwo++] = null;
                return toPop;
            }else if(stackId == 3){
                if(topThree == this.size && dataArray[topThree] == null)
                    throw new EmptyStackException("Third Stack is Empty");
                int toPop = dataArray[topThree];
                if(topThree > baseThree) dataArray[topThree--] = null;
                if(topThree == baseThree) dataArray[topThree] = null;
                return toPop;
            }else return null;
    }
    public int top(int stackId){
            if(stackId == 1){
                if(topOne == -1) throw new EmptyStackException("First Stack is Empty");
                    return dataArray[topOne];
                }else if(stackId == 2){
                    if(topTwo == this.size)
                                throw new EmptyStackException("Second Stack is Empty");
                    return dataArray[topTwo];
                }else if(stackId == 3){
                    if(topThree == baseThree && dataArray[baseThree] == null)
                        throw new EmptyStackException("Third Stack is Empty");
                    return dataArray[topThree];
                }else return null;
    }
    public boolean isEmpty(int stackId){
            if(stackId == 1){
                    return topOne == -1;
            }else if(stackId == 2){
                return topTwo == this.size;
            }else if(stackId == 3){
                return (topThree == baseThree) && (dataArray[baseThree] == null);
            }else return true;
    }
    private void shiftStack3ToLeft() {
            for(int i=baseThree-1; i<=topThree-1;i++){
                dataArray[i] = dataArray[i+1];
            }
            dataArray[topThree--] = null;
            baseThree--;
    }
    private boolean stack3IsLeftShiftable() {
            if(topOne+1 < baseThree){
                return true;
            }
            return false;
    }
    private void shiftStack3ToRight() {
            for(int i=topThree+1; i>=baseThree+1;i--){
                dataArray[i] = dataArray[i-1];
            }
            dataArray[baseThree++] = null;
            topThree++;
    }
    private boolean stack3IsRightShiftable() {
            if(topThree+1 < topTwo){
                return true;
            }
            return false;
    }
}
```

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

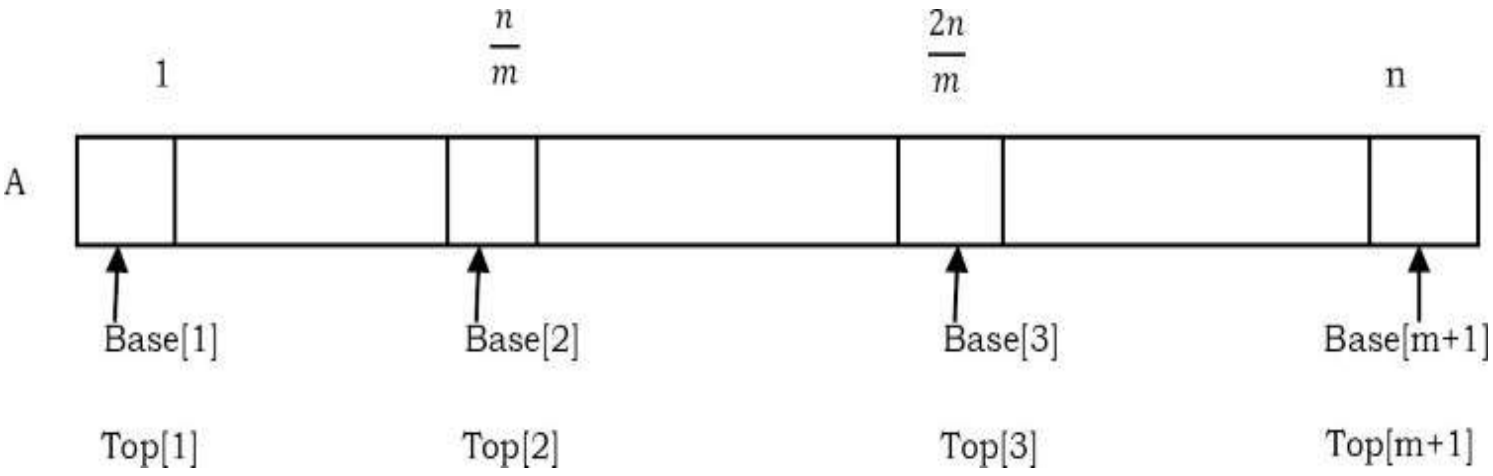
Solution: Yes. When either the left stack (which grows to the right) or the right stack (which grows to the left) bumps into the middle stack, we need to shift the entire middle stack to make room. The same happens if a push on the middle stack causes it to bump into the right stack.

To solve the above-mentioned problem (number 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.

```
public 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 we need to check is stack empty case.

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

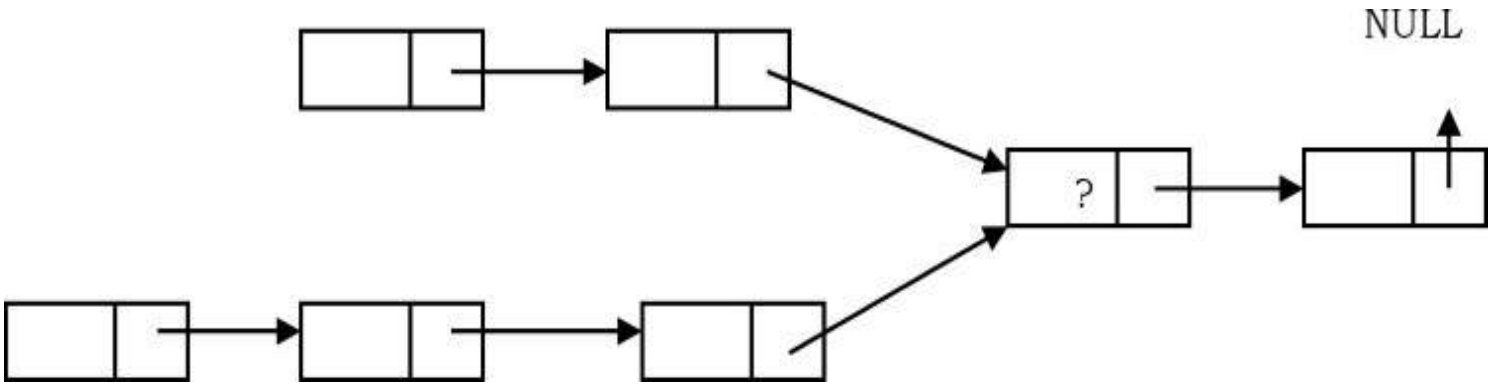
Time Complexity: O(1). Space Complexity: O(1).

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? (If a permutation is possible give the order string of operations.

Solution: SSSXXSSXSXXX outputs 325641. 154623 cannot be output as 2 is pushed much

before 3 so can appear only after 3 is output.

Problem-19 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-20 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* + *K* instead of doubling?

Solution: Let us assume that the initial stack size is 0. For simplicity let us assume that *K* = 10. For inserting the element we create a new array whose size is 0 + 10 = 10. Similarly, after 10 elements we again create a new array whose size is 10 + 10 = 20 and this process continues at values: 30, 40 ... That means, for a given *n* value, we are creating the new arrays at: $\frac{n}{10}, \frac{n}{20}, \frac{n}{30}, \frac{n}{40}$ ... The total number of copy operations 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-21 Given a string containing *n S's* and *n X's* where 5 indicates a push operation and Vindicates 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?

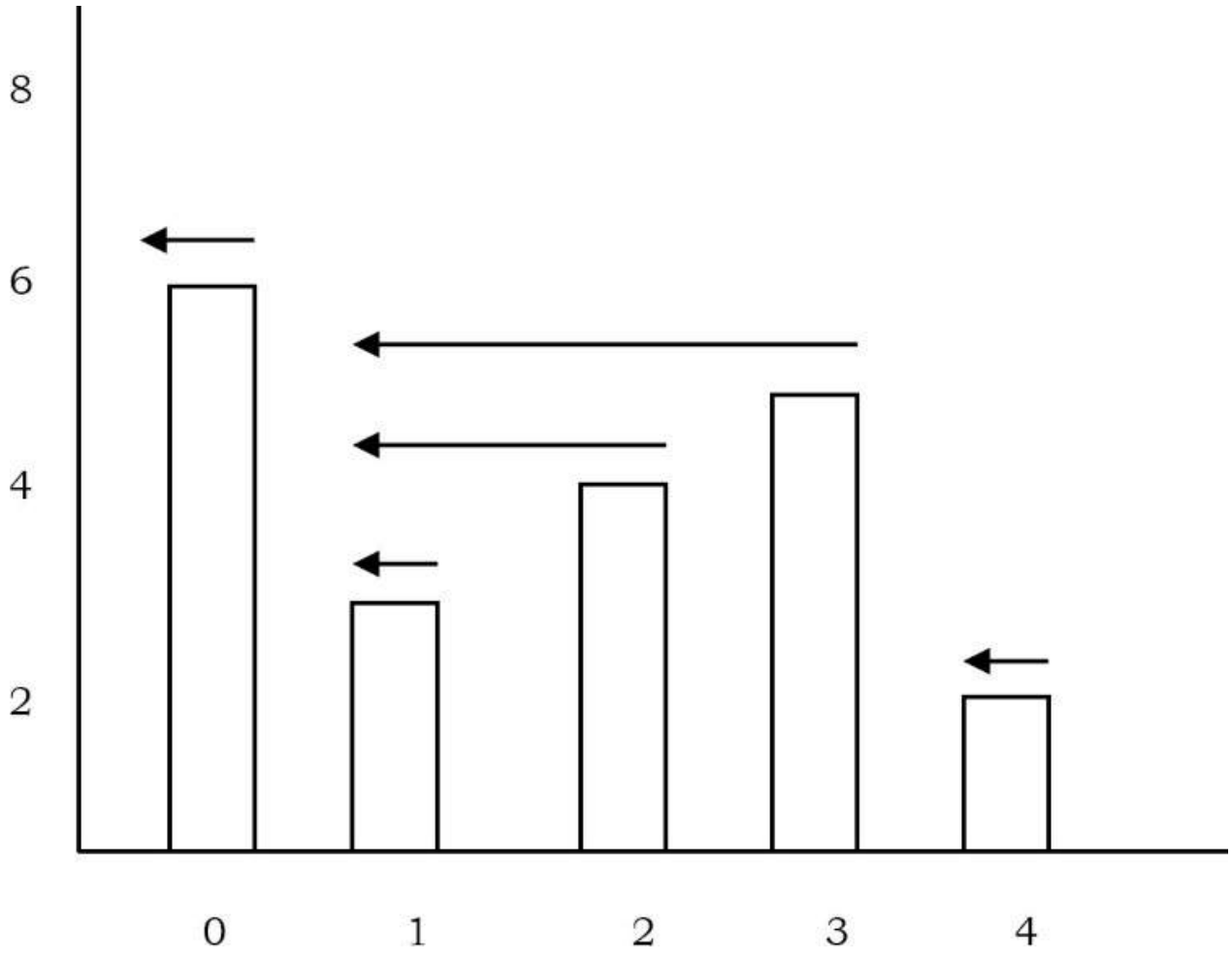
Solution: Given a string of length 2*n*, we wish to check whether the given string of operations is permissible or not with respect to its functioning on a stack. The only restricted operation is pop whose prior requirement is that the stack should not be empty. So while traversing the string from left 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 (5) should be greater than the number of pop operations (*X*).

Problem-22 **Finding of Spans:** Given an array A the span $S[i]$ of $A[i]$ is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ and such that $A[j] \leq A[j + 1]$?
Another 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: 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.

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



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.

```
public int[] FindingSpans(int[] inputArray){
    int[] spans = new int[inputArray.length];
    for(int i=0;i<inputArray.length;i++){
        int span = 1;
        int j=i-1;
        while(j>=0 && inputArray[j]<=inputArray[j+1]){
            span++;
            j--;
        }
        spans[i] = span;
    }
    return spans;
}
```

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

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

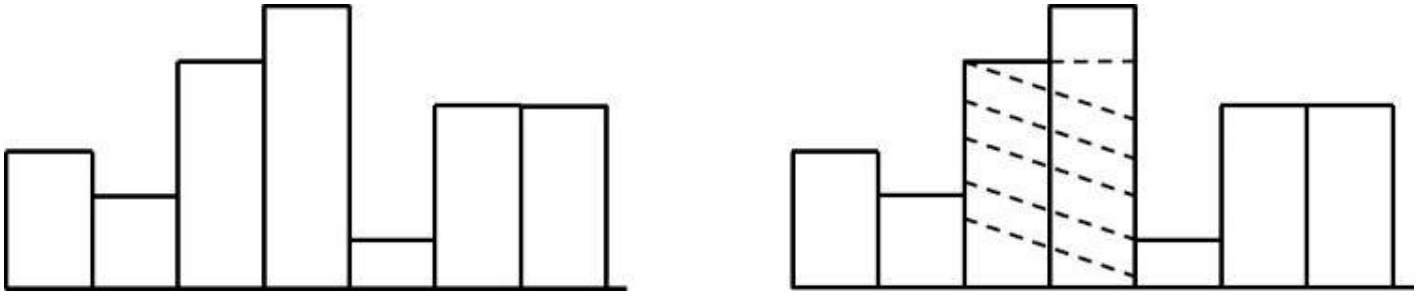
Solution: From the example above, we can see that span $S[i]$ on day i can be easily calculated if we know the closest day preceding i, such that the price is greater on that day than the price on day i. Let us call such a day as P. If such a day exists then the span is now defined as $S[i] = i - P$.

```
public int[] FindingSpans(int[] inputArray){
    int[] spans = new int[inputArray.length];
    Stack stack = new LLStack();
    int p = 0;
    for(int i=0;i<inputArray.length;i++){
        while (!stack.isEmpty() && inputArray[i] > inputArray[(Integer) stack.top()])
            stack.pop();
        if( stack.isEmpty())
            p = -1;
        else    p = (Integer) stack.top();
        spans[i] = i - p;
        stack.push(i);
    }
    return spans;
}
```

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 **Error! Reference source not found.,** can we improve the time complexity?

Solution: Linear search using a stack of incomplete subproblems: There are many ways of solving this problem. *Judge* has given a nice algorithm for this problem which is based on stack. We process the elements in left-to-right order and maintain a stack of information about started but yet unfinished sub histograms. If the stack is empty, open a new subproblem by pushing the element onto the stack. Otherwise compare it to the element on top of the stack. If the new one is greater we again push it. If the new one is equal we skip it. In all these cases, we continue with the next new element.

If the new one is less, we finish the topmost subproblem by updating the maximum area with respect to the element at the top of the stack. Then, we discard the element at the top, and repeat the procedure keeping the current new element. This way, all subproblems are finished 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 subproblems by updating the maximum area with respect to the elements at the top.

```
public class MaxRectangleAreaInHistrogram {
   public int MaxRectangleArea(int[] A) {
      Stack<Integer> s = new Stack<Integer>();
      if(A == null || A.length == 0)
         return 0;
      // Initialize max area
      int maxArea = 0;
      int i = 0;
      // run through all bars of given histogram
      while(i < A.length) {
         // If current bar is higher than the bar of the stack peek, push it to stack.
         if(s.empty() || A[s.peek()] <= A[i])
            s.push(i++);
         else {
            // if current bar is lower than the stack peek,
            // calculate area of rectangle with stack top as the smallest bar.
            // 'i' is 'right index' for the top and element before top in stack is 'left index'
            int top = s.pop();
            // calculate the area with A[top] stack as smallest bar and update maxArea if needed
            maxArea = Math.max(maxArea, A[top] * (s.empty() ? i : i - s.peek() - 1));
         }
      }
      // Now pop the remaining bars from stack and calculate area with every popped bar as the smallest bar.
      while(!s.isEmpty()) {
         int top = s.pop();
         maxArea = Math.max(maxArea, A[top] * (s.empty() ? i : i - s.peek() - 1));
      }
      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 Give an algorithm for sorting a stack in ascending order. We should not make any assumptions about how the stack is implemented.

Solution:

```
public static Stack<Integer> sort(Stack<Integer> stk) {
   Stack<Integer> rstk = new Stack<Integer>();
   while(!stk.isEmpty()){
      int tmp = stk.pop();
      while(!rstk.isEmpty() && rstk.peek() > tmp){
         stk.push(rstk.pop());
      }
      rstk.push(tmp);
   }
   return rstk;
}
```

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

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 an array of numbers, recursively remove adjacent duplicate numbers. The output array should not have any adjacent duplicates.

Input: 1,5,6, 8,8,8,0,1,1,0,6,5 *Output*: 1	*Input*: 1,9,6, 8,8,8,0,1,1,0,6,5} *Output*: 1, 9, 5

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

```
public class RemoveAdjacentDuplicates {
    public int removeAdjacentDuplicates(int []A){
        int stkptr=-1;
        int i=0;
        while (i<A.length){
            if (stkptr == -1 || A[stkptr]!=A[i]){
                stkptr++;
                A[stkptr]=A[i];
                i++;
            }else {
                while(i < A.length&& A[stkptr]==A[i])
                    i++;
                stkptr--;
            }
        }
        return stkptr;
    }
}
public class TestRemoveAdjacentDuplicates {
    public static void main(String[] args) {
        RemoveAdjacentDuplicates obj = new RemoveAdjacentDuplicates();
        int[] A = {1,5,6, 8,8,8,0,1,1,0,6,5};
        int index = obj.removeAdjacentDuplicates(A);
        for (int i = 0; i <= index; i++) {
            System.out.print(" " + A[i]);
        }
    }
}
```

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

Problem-28 If the stack gets too high, it might overbalance. There-fore, in real life, we would likely start a new stack when the previous stack exceeds some threshold. Implement a data structure that mimics this and composed of several stacks, and should create a new stack once the previous one exceeds capacity. push() and pop() of this class should behave identically to a regular stack.

Solution: Follow the comments in code below.

```
class StackForStackSets {
    private int top = -1;
    private int[] arr;
    // Maximum size of stack
    private int capacity;
    StackForStackSets(int capacity){
        this.capacity = capacity;
        arr = new int[capacity];
    }
    public void push(int v){
        arr[++top] = v;
    }
    public int pop(){
        return arr[top--];
    }
    // if the stack is at capacity
    public Boolean isAtCapacity(){
        return capacity == top + 1;
    }
    //return the size of the stack
    public int size(){
        return top+1;
    }
    public String toString(){
        String s = "";
        int index = top;
        while(index >= 0){
            s += "[" + arr[index--] + "]"+ " --> ";
        }
        return s;
    }
}
public class StackSets{
    // Number of elements for each stack
    private int threshold;
    private ArrayList<StackForStackSets> listOfStacks = new  ArrayList<StackForStackSets>();
    StackSets(int threshold) {
        this.threshold = threshold;
    }
    //get the last stack
    public StackForStackSets getLastStack(){
        int size = listOfStacks.size();
        if(size <= 0)
                return null;
        else return listOfStacks.get(size - 1);
    }
```

```
    //get the nth stack
    public StackForStackSets getNthStack(int n){
        System.out.println(n);
        int size = listOfStacks.size();
        if(size <= 0)
            return null;
        else return listOfStacks.get(n - 1);
    }
    //push value
    public void push(int value){
        StackForStackSets lastStack = this.getLastStack();
        if(lastStack == null){
           lastStack = new StackForStackSets(threshold);
           lastStack.push(value);
           listOfStacks.add(lastStack);
        }else {
           if( !lastStack.isAtCapacity())
               lastStack.push(value);
           else {
               StackForStackSets newLastStack = new StackForStackSets(threshold);
               newLastStack.push(value);
               listOfStacks.add(newLastStack);
           }
        }
    }
    // the pop
    public int pop(){
        StackForStackSets lastStack = this.getLastStack();
        int v = lastStack.pop();
        if(lastStack.size() == 0) listOfStacks.remove(listOfStacks.size() - 1);
        return v;
    }
    //pop from the nth stack
    public int pop(int nth){
        StackForStackSets nthStack = this.getNthStack(nth);
         int v = nthStack.pop();
         if(nthStack.size() == 0) listOfStacks.remove(listOfStacks.size() - 1);
         return v;
    }
    public String toString(){
        String s = "";
        for(int i = 0; i < listOfStacks.size(); i++){
            StackForStackSets stack = listOfStacks.get(i);
           s = "Stack "+ i + ": " + stack.toString() + s;
        }
        return s;
    }
    public static void main(String[] args){
        StackSets stacks = new StackSets(3);
        stacks.push(10); stacks.push(9); stacks.push(8);
        stacks.push(7); stacks.push(6); stacks.push(5);
        stacks.push(4); stacks.push(3); stacks.push(2);
        System.out.println("Popping " + stacks.pop(2));
        System.out.println("Popping from stack 1" + stacks.pop(1));
        System.out.println("Popping " + stacks.pop(3));
        System.out.println("Popping " + stacks.pop(2));
        System.out.println("the stack is: " + stacks);
    }
}
```