Contents

List of Figures

List of Tables

Chapter 1

Introduction to Data Structure and Algorithm

1.1 Introduction

- The concept of programming has been started with nonstructured, linear programs, known as **spaghetti code**, in which the logic flow wound through the program like spaghetti on a plate.
- Next came the concept of **modular programming**, in which programs were organized in functions, each of which still used a linear coding technique.
- In the 1970s, the basic principles of **structured programming** were formulated by computer scientists such as Edsger Dijkstra and Niklaus Wirth.

Atomic and Composite Data:

- **Atomic Data**
  - Atomic data are data that consist of a single piece of information; that is, they cannot be divided into other meaningful pieces of data.
  - For example, the integer 4562 may be considered a single integer value. Of course, we can decompose it into digits, but the decomposed digits do not have the same characteristics of the original integer; they are four single digit integers ranging from 0 to 9.
  - In some languages atomic data are known as scalar data because of their numeric properties.
- **Composit Data**
  - Composite data can be broken out into subfields that have meaning.

  - As an example of a composite data item, consider your telephone number. A telephone number actually has three different parts. First, there is the area code.
  - Then what you can consider to be your phone number is actually two different data items, a prefix consisting of a three-digit exchange and the number within the exchange consisting of four digits

Data Type:

- A **data type** consists of two parts: a set of data and the operations that can be performed on the data.
- Thus we see that the integer type consists of values (whole numbers in some defined range) and operations (add, subtract, multiply, divide, and any other operations appropriate for the data.

Abstract Data Type:

- Abbreviated ADT. The specification of a data type within some language, independent of an implementation.
- The interface for the ADT is defined in terms of a type and a set of operations on that type.
- The behavior of each operation is determined by its inputs and outputs.
- An ADT does not specify how the data type is implemented.
- These implementation details are hidden from the user of the ADT and protected from outside access, a concept referred to as encapsulation.

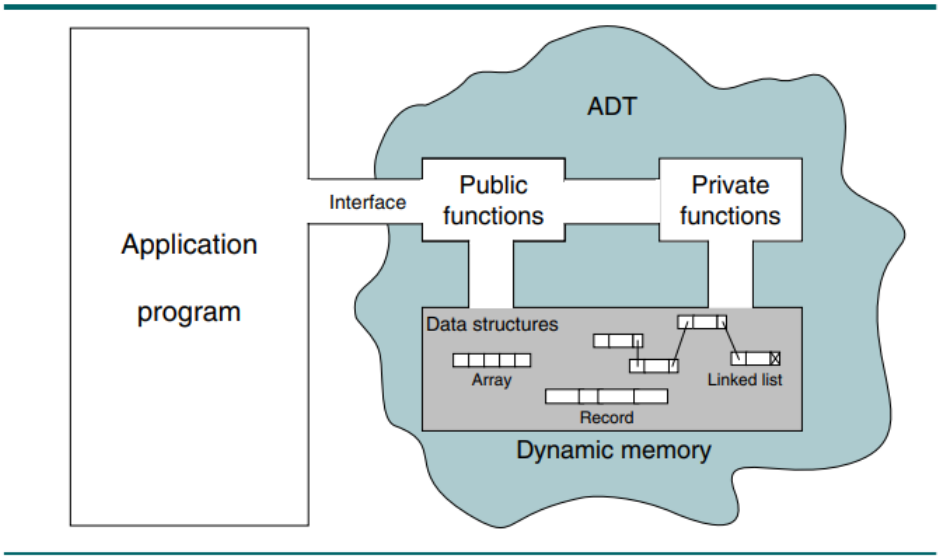

Figure 1.1: Abstract Data Model

Data Structure:

- **Data structure** is a systematic way of organizing and accessing data.
- It is an aggregation of atomic and composite data into a set with defined relationships.
- In this definition structure means a set of rules that holds the data together.

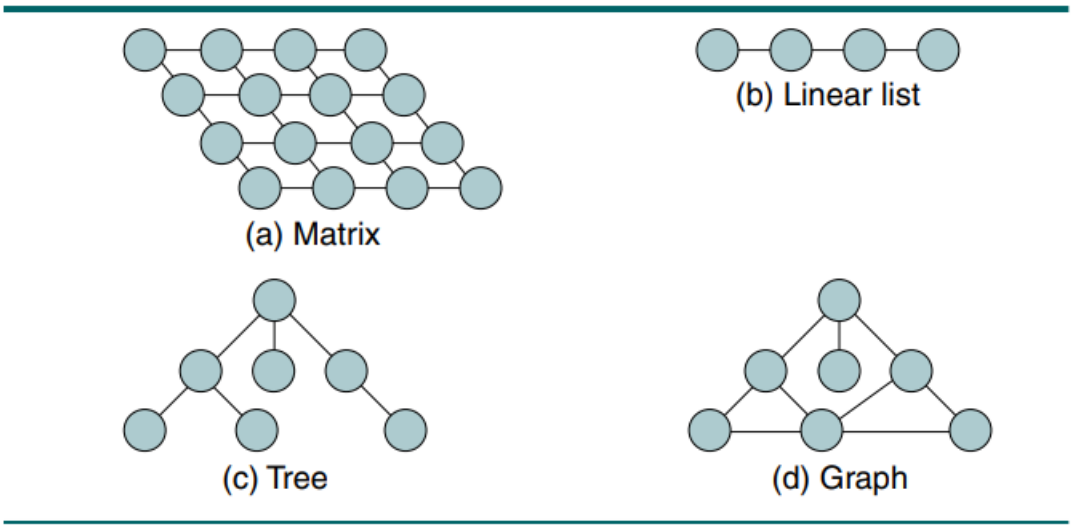

Figure 1.2: Some Data Structures

Algorithm:

- An **Algorithm** is a step-by-step procedure for performing some task in a finite amount of time.
- An algorithm is a computer program that describes a set of steps applied over a set of input to produce a set of output.
- The most important properties of an algorithm are:
    1. **Correctness:** The algorithm should be correct. It should be able to process all the given inputs and provide correct output.
    2. **Efficiency:** The algorithm should be efficient in solving problems. Efficiency is measured in two parameters. First is **Time-Complexity**, how quick result is provided by an algorithm. Second is **Space-Complexity**, how much RAM or memory that an algorithm is going to consume to give desired result.

1.2 Experimental Studies

- The most common way to study the efficiency of an algorithm is to implement it and experiment by running the program on various test inputs while recording the time spent during each execution.

- A simple mechanism for collecting such running times in Java is based on use of the *currentTimeMillis* method of the *System* class.

```
long startTime = System.currentTimeMillis( ); // record the starting time
 (run the algorithm)
long endTime = System.currentTimeMillis( ); // record the ending time
long elapsed = endTime - startTime; // compute the elapsed time
```

- However, the measured times reported by currentTimeMillis will vary greatly from machine to machine, and may likely vary from trial to trial, even on the same machine.
- This is because many processes share use of a computer's **central processing unit (or CPU)** and memory system; therefore, the elapsed time will depend on what other processes are running on the computer when a test is performed.
- As a tangible example of experimental analysis, we consider two algorithms for constructing long strings in Java. Our goal will be to have a method, with a calling signature such as repeat('*', 40), that produces a string composed of 40 asterisks:
  '**'
- The first algorithm we consider performs repeated string concatenation, based on the + operator. The second algorithm relies on Java's StringBuilder class, and is implemented as method repeat2. It is implemented as method repeat1 in below Code Fragment..

```
// Uses repeated concatenation to compose a String with n copies of
   character c.
public static String repeat1(char c, int n) {
   String answer = "";
   for (int j=0; j < n; j++)
      answer += c;
   return answer;
}

// Uses StringBuilder to compose a String with n copies of character c.
public static String repeat2(char c, int n) {
     StringBuilder sb = new StringBuilder( );
     for (int j=0; j < n; j++)
        sb.append(c);
     return sb.toString( );
}
```

- The executed trials to compose strings of increasing lengths to explore the relationship between the running time and the string length. The results of our experiments are shown as follows:

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Figure 1.3: Results of timing experiment on the methods from Code Fragment

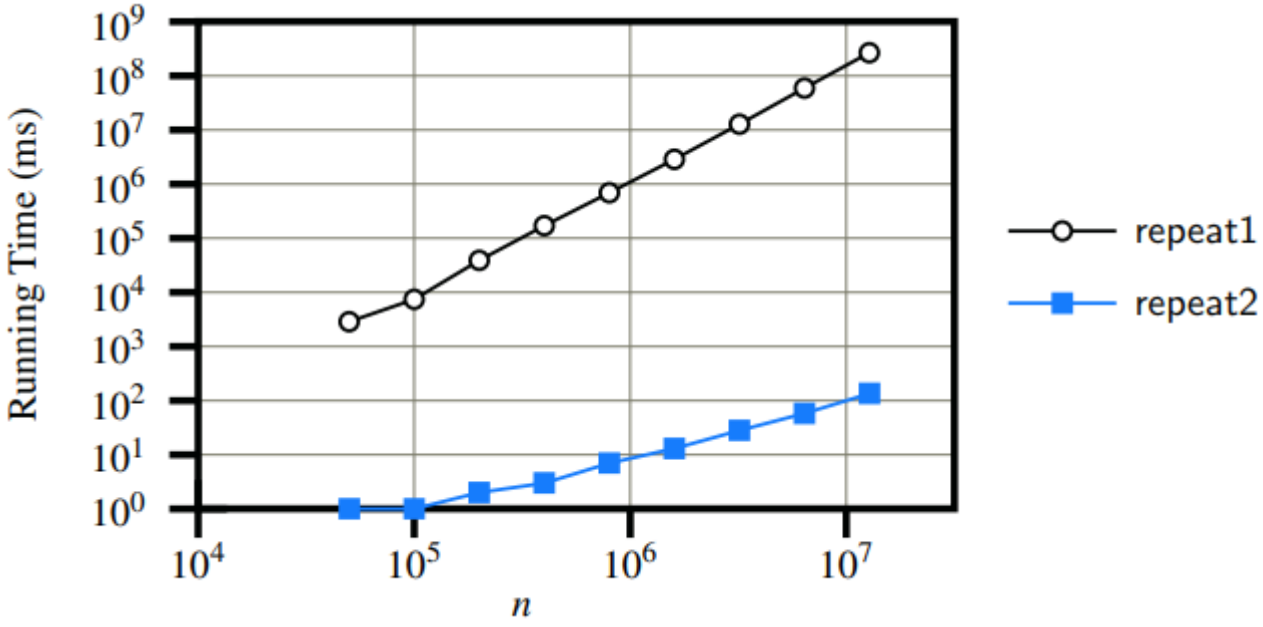

Figure 1.4: Chart of the results of the timing experiment

The most striking outcome of these experiments is how much faster the repeat2 algorithm is relative to repeat1.

Challenges of Experimental Analysis: There are three major limitations to their use for algorithm analysis:

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

1.2.1 Moving Beyond Experimental Analysis

The main goal is to develop an approach to analyzing the efficiency of algorithms that:

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high-level description of the algorithm without need for implementation.
3. Takes into account all possible inputs

Counting Primitive Operations To analyze the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm (either in the form of an actual code fragment, or language-independent pseudocode). We define a set of primitive operations such as the following:

- Assigning a value to a variable
- Following an object reference
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of an array by index
- Calling a method
- Returning from a method

Measuring Operations as a Function of Input Size

- To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function f(n) that characterizes the number of primitive operations that are performed as a function of the input size n.

Focusing on the Worst-Case Input

- An algorithm may run faster on some inputs than it does on others of the same size.
- Thus, we may wish to express the running time of an algorithm as the function of the input size obtained by taking the average over all possible inputs of the same size.
- Unfortunately, such an average-case analysis is typically quite challenging.

1.3 The Seven Functions used for Algorithm Analysis

1.3.1 The Constant Function

1.3.2 The Logarithm Function

1.3.3 The Linear Function

1.3.4 The N-Log-N Function

1.3.5 The Quadratic Function

1.3.6 The Cubic Function and Other Polynomials

1.3.7 The Exponential Function

1.4 Algorithm Efficiency

- When comparing two different algorithms that solve the same problem, you often find that one algorithm is an order of magnitude more efficient than the other.
- In this case, it only makes sense that you be able to recognize and choose the more efficient algorithm.
- As we study specific examples, we generally discuss the algorithm's efficiency as a function of the number of elements to be processed. The general format is

f(n) = efficiency

1.4.1 Linear Loops

We want to know how many times the body of the loop is repeated in the following code

```
for (i = 0; i < 1000; i++)
   application code
```

- Assuming i is an integer, the answer is 1000 times. The number of iterations is directly proportional to the loop factor 1000.
- The higher the factor, the higher the number of loops.
- Because the efficiency is directly proportional to the number of iterations, it is given as

  $$\mathbf{f(n) = n}$$

- However, the answer is not always as straightforward as it is in the above example. For instance, consider the following loop. How many times is the body repeated in this loop? Here the answer is 500 times.

  ```
  for (i = 0; i < 1000; i += 2)
     application code
  ```

- In this example the number of iterations is half the loop factor. Once again, however, the higher the factor, the higher the number of loops. The efficiency of this loop is proportional to half the factor, which makes it

  $$\mathbf{f(n) = n/2}$$

1.4.2 Logarithmic Loops

In a logarithmic loop, the controlling variable is multiplied or divided in each iteration. How many times is the body of the loops repeated in the following program segments?

```
for (i = 0; i < 1000; i *= 2)
   application code
```

```
for (i = 0; i < 1000; i *= 2)
   application code
```

Generalizing the analysis, we can say that the iterations in loops that multiply or divide are determined by the following formula:

f(n) = logn

1.4.3 Nested Loops

- Loops that contain loops are known as nested loops.
- When we analyze nested loops, we must determine how many iterations each loop completes.
- The total is then the product of the number of iterations in the inner loop and the number of iterations in the outer loop.

  ITERATIONS = OUTER LOOP ITERATIONS X INNER LOOP ITERATIONS

We now look at three nested loops: linear logarithmic, quadratic, and dependent quadratic.

Chapter 2

Introduction to LinkedList

2.1 Introduction

- Link list is a linear list of linked elements. Like arrays, linked list represents another linear data structure.
- In arrays, there is always a fixed relationship between the addresses of two consecutive elements as all the items of an array must be stored contiguously.
- However, note that this contiguity requirement makes expansion or contraction of the size of the array difficult.
- In linked list, data items may be scattered arbitrarily all over the memory, but we can achieve fast insertion and deletion in a dynamic situation.

Limitations of Array In many applications, the array is not suitable as it has some drawback. The drawbacks of Array are listed below:

- The maximum size of the array needs to be predicted beforehand. One cannot change the size of the array after allocating memory, but, many applications require resizing.
- Most of the space in the array is wasted when programmer allocates arrays with large size. On the other hand, when program ever needs to process more than the specify size then the code breaks.
- Storage of the array must be available contiguously. Required storage not always immediately available.
- Insertion and deletion operation may be very slow. The worst case occurs when the first element is to be deleted or inserted. Almost all the elements of the array need to be moved.
- Joining and splitting of two or more arrays is difficult.

2.2 LinkedList

Definition: A linked list is a linear ordered collection of finite homogeneous data elements called node, where the linear order is maintained by means of links or pointers.

- A linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node".
- A linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence.
- The list structure is created by the use node object and connect all its nodes together like the links in a chain.

Property	Array	Linked
Storage	Storage of the array must be available contiguously.	Storage need not be contiguous.
Memory utilization	The size of the array needs to be predicted beforehand because memory allocation is done in advance. Not necessary for storing addresses of any element.	Memory of linked list is not pre-allocated, memory is allocated whenever it is required. Extra memory space is necessary for storing addresses of the next node
Change of size	The array size is fixed, extend or shrink not possible during the execution of a program	Linked list may extend or shrink during the execution of a program
Insertion/ deletion	Insertion/deletion operations are slow, half of the elements are required to move on an average	Insertion/deletion operations are performed very fast, in a constant amount of time
Searching	Linear searching, binary searching, interpolation searching are possible	Binary searching, interpolation searching not possible, only linear searching is possible
Access element	Fast access to any element in a constant amount of time.	To access any element in a linked list, traversing is required.
Joining/ splitting	Joining and splitting of the two arrays is difficult.	Joining and splitting of two linked list is very easy.

2.2.1 Advantages of linked list

Linked lists have many advantages. Some of the very important advantages are:

1. Linked list are dynamic data structures. That is, they can extend or shrink during the execution of a program.

2. Storage need not be contiguous.
3. Efficient memory utilization. Here memory is not pre-allocated. Memory is allocated whenever it is required.
4. Insertion or deletion is easy and efficient, may be done very fast, in a constant amount of times, ndependent of the size of the list.
5. The joining of two linked lists can be done by assigning pointer of the second linked list in the last node of the first linked list.

2.2.2 Types of Linked list

There are different types of linked list. We can put linked lists into the following four types:

1. Singly linked list
2. Circular Linked list
3. Doubly Linked list
4. Circular Doubly linked list

2.3 Singly linked list

An element in a linked list is specially termed as a node. In a singly linked list, each node consists of two fields:

1. **data** field that contains the actual information of the element.
2. **link** or **next** field, contains the address of the next node in the list.
3. **head**, the linked list instance must keep a reference to the first node of the list, known as the head.

Figure 2.1: Single node

- A "data" field to store whatever element type the list holds for the user, and a "link or next" field, which is a pointer, used to link one node to the next node.
- **head:** The linked list instance must keep a reference to the first node of the list, known as the **head**.

- **tail:** The last node of the list is known as the **tail**.
- The tail of a list can be found by traversing the linked list— starting at the head and moving from one node to another by following each node's next reference.
- We can identify the tail as the node having **null** as its next reference. This process is also known as **link hopping** or **pointer hopping**.

Figure 2.2: Single LinkedList

2.4 Operations on Singly linked list

Operations supported by a singly linked list are as follows:

- **Createlist:** This operation creates a linked list.
- **Traverse:** This operation traverse/visit all the elements of the linked list exactly once
- **Insertion:** This operation inserts an element to the linked list
- **Deletion:** This operation removes an element from the linked list
- **Searching:** This operation performs linear searching for a key value in the linked list
- **Reverse:** This operation performs the reverse of the linked list
- **Merging:** This operation performs merging of two linked lists in a single linked list

2.4.1 Create List

The following algorithm creates a node and appends it at the end of the existing list.

- *'Head'* is a pointer which holds the address of the HEADER of the linked list.
- *'item'* is the data or value of the new node.
- *next* is a field which holds the address of the new node
- 'Temp' is a temporary node.

Algorithm 2.4.1 create (Head, item)

```
    [Create new Node]
 1: Create the new Node
 2: Set Node.data = item
 3: Set Node.next = NULL
    [Check for empty List]
 4: if Head = NULL then
        Set Head = Node
 5: else
        Set Temp = Head
 6:     while Temp.next ≠ null do
            Set Temp = Temp.next
 7:     end while
 8:     Temp.next = Node
 9: end if
10: Return Head
```

- In the abov ealgorithm **Node** represent a new node of the Linked list.
- At first creates a **Node** object and assign memory**[Statement-1]**.
- Assign the value or **item** to the **data** part of the **Node [Statement-2]**
- Store NULL in the **next** part of the **Node**. **[Statement-3]**.

2.4.1.1 Java Program to Create a new Node and Linked List

In java we can created a class named Node to represent a node object. The class has two members -

- the first is the **data** of integer type.It can be any data value that we want to store in the Node object.
- the other is the Node class type reference **next** which referes to the memory of the next node in the linked list.
- So when we represent any node for the linked list, it refers to an object of **Node class** type.

```
class Node {
   public int data;
   public Node next;
}
```

The Node class is used in the **createNode(Node head, int item)** method. It can be called repeatedly for creating the nodes.

- The **node,temp, head** are are Node class objects.

```
public static Node create(Node head, int item) {
   Node node=new Node();
   Node temp=new Node();
   node.data=item;
   node.next=null;
   if(head==null)
   head=node;
   else {
      temp=head;
      while(temp.next != null) {
         temp=temp.next;
      }
      Temp.next=node;
   }
   return head;
}
```

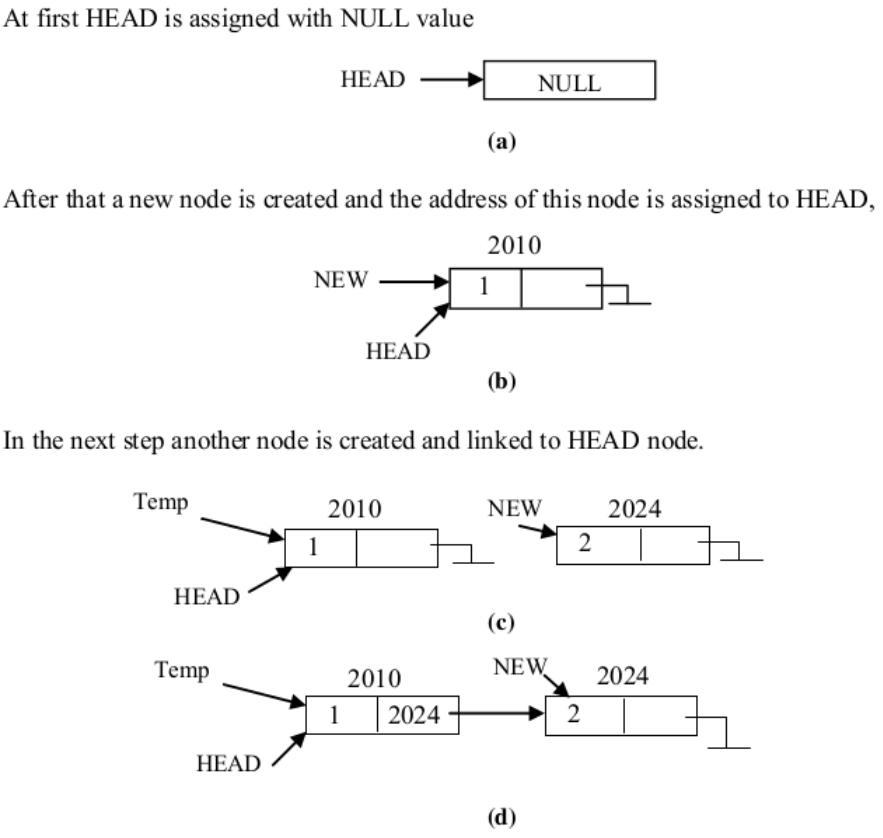

Figure 2.3: Create Single LinkedList

2.4.2 Traversing / Display a Linked List

This algorithm traverses a linked list and prints the data part of each node of the linked list.

- The '_HEAD' is a pointer which points to the starting node of the linked list
- '_Temp' is a temporary pointer to traverse the list.

Algorithm 2.4.2 Traverse (Head)

1: **if** Head = NULL **then**
   Print: "The Linked list is empty"
   Return Head
2: **end if**
3: Temp = Head
4: **while** $Temp \neq NULL$ **do**
   a) print: Temp.data
   b) Set Temp = Temp.next
5: **end while**
6: Return Head

Java method to traverse a LinkedList

```
public static Node traverse(Node head) {
   Node temp;
   if(head==null) {
      System.out.println("Empty LinkedList");
      return head;
   }
   else {
      System.out.println("Data of the Linked List:");
      Temp=head;
      while(Temp != null) {
         System.out.print(Temp.data+" ");
         Temp=temp.next;
      }
   }
   return head;
}
```

Complete Java Program to Create and Traverse a Linked List

Compete java code to create and Traverse a single LinkedList. It use a Node class for the node, two methods to create and display nodes of the LinkedList

```
package insertnode;
import java.util.Scanner;
//Class for the contents of a node
class Node{
   int data;
   Node next;
}
class LinkedList{
   //Method t create a new node by using the Node class object
   public static Node create(Node head) {
      Scanner sc=new Scanner(System.in);
      char choice;
      int nodeCount=1;
      do {
         Node node=new Node();
         Node temp;
         System.out.println("Enter data for the new node");
         node.data=sc.nextInt();
         node.next=null;
         if(head==null)
            head=node;
         else {
            temp=head;
            while(temp.next != null) {
               temp=temp.next;
            }
            temp.next=node;
         }
         System.out.println(nodeCount+" node created");
         System.out.println("Would you like to enter another node(y|n)");
         choice=sc.next().charAt(0);
         if(choice=='y' || choice=='Y')
         nodeCount++;
      }while(choice=='y' || choice=='Y');
      return head;
   }
   //Method to display the linked list
   public static void display(Node head) {
      Node temp;
```

```
      temp=head;
      if(head==null)
      System.out.println("Empty linked list");
      else {
         System.out.println("The elements of the linked list are:");
         while(temp != null) {
            System.out.print(temp.data+"-->");
            temp=temp.next;
         }
      }
   }
}
public class LinkedListInsertDriver {
   public static void main(String[] args) {
      Node head=null;
      head=LinkedList.create(head);
      LinkedList.display(head);
      head=LinkedList.insertBegin(head);
      LinkedList.display(head);
   }
}
```

Example 2.4.1 ***Java program to create a generic LinkedList. Insert node at begin end and at any position***

```
class Node{
   int data;
   Node next;
   Node(int data){
      this.data=data;
      next=null;
   }
}
class LinkedList{
   Node head=null;
   Node tail=null;
   // Insert at end by using head and tail
   void insertAtEnd(int item) {
      Node node=new Node(item);
      if(head==null) {
         head=node;
      }
      else {
         tail.next=node;
      }
      tail=node;
   }
   /*
   void insertAtEnd(int item) {
      Node node=new Node(item);
      Node temp=null;
      if(head==null) {
         head=node;
      }
      else {
         temp=head;
         while(temp.next!=null) {
            temp=temp.next;
         }
         temp.next=node;
         //tail=node;
      }
   }*/
   int size() {
      Node temp=head;
```

```
    int count=0;
    while(temp!=null) {
        count++;
        temp=temp.next;
    }
    return count;
}
void insertAtBegin(int item) {
    Node node=new Node(item);
    if(head==null) {
        head=node;
        //tail=node;
    }
    else {
        node.next=head;
        head=node;
    }
}
void insertAt(int pos,int item) {
    Node node=new Node(item);
    Node temp=head;
    if(pos==1) {
        insertAtBegin(item);
        return;
    }
    else if(pos==size()) {
        insertAtEnd(item);
        return;
    }
    for(int i=1;i<=pos-1;i++) {
        temp=temp.next;
    }
        node.next=temp.next;
        temp.next=node;
    }
int getAt(int pos) {
    Node temp=head;
    for(int i=1;i<=pos;i++) {
        temp=temp.next;
    }
    return temp.data;
}
void display() {
```

```
      Node temp=head;
      while(temp!=null) {
         System.out.print(temp.data+" --> ");
         temp=temp.next;
      }
      System.out.println();
   }
}
public class LinkedListDriver {
   public static void main(String[] args) {
      LinkedList ll=new LinkedList();
      ll.insertAtEnd(4);
      ll.insertAtEnd(8);
      ll.insertAtEnd(12);
      ll.insertAtEnd(5);
      ll.insertAtBegin(15);
      ll.insertAt(2, 100);
      ll.display();
      System.out.println("Size="+ll.size());

   }

}
```

Java program to create node and assign value using constructor. Link the nodes manually and use different methods to traverse the node of the linked list

```
package simplecreate;
class Node{
   int data;
   Node next;
   Node(){}
   Node(int data){
      this.data=data;
   }
}
public class DisplayLinkedList {
   //Method to find the length of the linked list
   public static int length(Node head) {
      int count=0;
      while(head!=null) {
         count++;
         head=head.next;
```

```
        }
        return count;
    }
    //Method to display the linked list
    public static void display(Node head) {
        Node temp=head;
        if(head==null)
            return;
        while(temp!=null) {
            System.out.print(temp.data+"-->");
            temp=temp.next;
        }
    }
    //Method for the recursive display of the linked list
    public static void recDisplay(Node head) {
        if(head==null)
            return;
        System.out.print(head.data+"-->");
        recDisplay(head.next);
    }
    //Method to reverse display of the linked list using recursive display
    public static void reverseDisplay(Node head) {
        if(head==null)
            return;
        reverseDisplay(head.next);
        System.out.print(head.data+"-->");
    }
public static void main(String[] args) {
    Node n1=new Node(5);
    Node n2=new Node(3);
    Node n3=new Node(2);
    Node n4=new Node(7);
    Node n5=new Node(10);

    //Connect the nodes
    n1.next=n2;
    n2.next=n3;
    n3.next=n4;
    n4.next=n5;

    //Find length of the lined list
    System.out.println("Length of the linked list = "+length(n1));
```

```
   //Display the Linked list using for loop
   System.out.println("\nDisplay the LinkedList using for loop ");
   Node temp=n1;
   for(int i=0;i<5;i++) {
      System.out.print(temp.data+"-->");
      temp=temp.next;
   }

   //Display the Linked list using display method
   System.out.println("\nDisplay the Linked list using display method");
   display(n1);

   //Display the Linked list using recurssion
   System.out.println("\nDisplay the Linked list using recurssion");
   recDisplay(n1);

   System.out.println("\nReverse Display the Linked list using recurssion");
   reverseDisplay(n1);
   }

}
```

2.4.3 Insertion in Singly Linked List

Insertion operation in a singly linked list can be done in different ways using position.

- Insertion at beginning.
- Insertion at end.
- Insertion at any index.

2.4.3.1 Insertion of a node at first position of a singly linked list

In the following algorithm insertion of node at beginning position is described. The '_HEAD' is a pointer which points to the starting node of the linked list. Node points to the new node.

Algorithm 2.4.3 insertAtBegin (Head, item)

[Create new Node]
1: Allocate memory for the new Node node
2: Set Node.data = item
3: Set Node.next = Head
[Make the HEADER to point to the new Node]
4: Set Head=Node
5: Return Head

The following diagrams explain the insertion operation at the beginning of a singly linked list. At first HEAD points to the first node of the list containing two nodes.

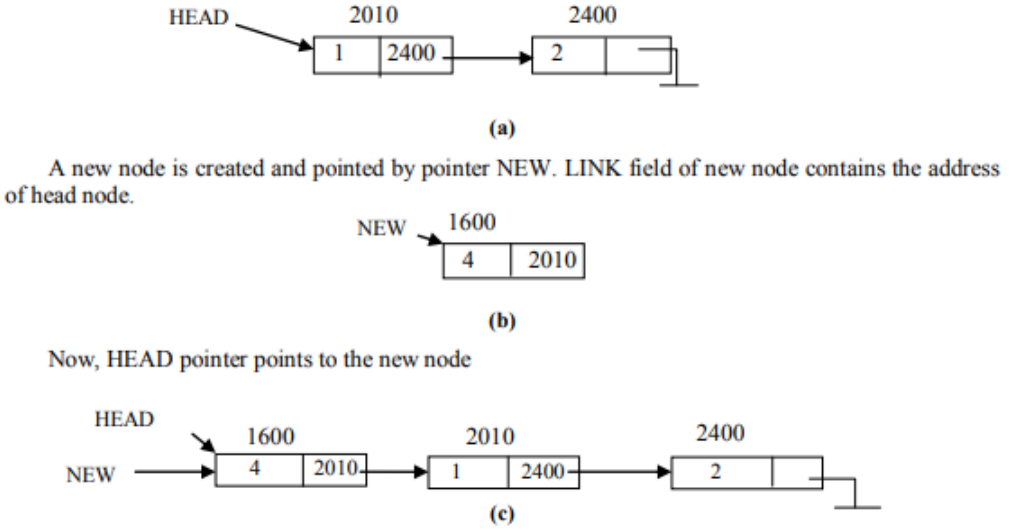

Figure 2.4: Node insertion in a single linked list at beginning

Java method to insert a node at the beginning of a linked list

```
public static Node insertAtBeg(Node head) {
   Scanner sc=new Scanner(System.in);
   Node node=new Node();
   System.out.println("Enter data for new node:");
   node.data=sc.nextInt();
   node.next=head;
   head=node;
   return head;
}
```

2.4.3.2 Insertion of a node at the end of a singly linked list

Algorithm 2.4.4 insertAtEnd (Head, item)

[Create new Node]

1: Allocate memory for the new Node node
2: Set Node.data = item
3: Set Node.next = NULL
4: Set Temp=Head [to make Temp to point to the first node]
5: **while** $Temp.next \neq null$ **do**
Set Temp = Temp.next
6: **end while**
7: Set Temp.next=Node
8: Return Head

Java method to insert a new node at the end of a linked list

```
public static Node insertAtEnd(Node head) {
   Scanner sc=new Scanner(System.in);
   Node node=new Node();
   System.out.println("Enter data for new node:");
   node.data=sc.nextInt();
   node.next=null;
   Node temp=head;
   while(temp.next != null) {
      temp=temp.next;
   }
   temp.next=node;
   return head;
}
```

2.4.3.3 Insertion of a node at any index position of a single linked list

```
public static int size(Node head) {
   Node temp=head;
   int count=0;
   while(temp != null) {
      count++;
      temp=temp.next;
   }
   return count;
}
public static Node insertAtIndex(Node head) {
   Scanner sc=new Scanner(System.in);
   Node temp=head;
   Node node=new Node();

   //Create new node and assign required data
   System.out.println("\nEnter data for new node:");
   node.data=sc.next();
   int length=size(head);

   //Specify the index position to insert the new node
   System.out.println("Enter the position to insert the new node");
   int indx=sc.nextInt();
   //If the index is at starting
   if(indx==0)
      head=insertAtBeg(head);
   //If the index is at the end of the linked list
   else if(indx==length)
      head=insertAtEnd(head);
   //At any place except start and end of the list
   else {
      for(int i=0;i<indx-1;i++) {
         temp=temp.next;
      }
      node.next=temp.next;
      temp.next=node;
   }
   return head;
}
```

Complete java program to perform node creation and inserction operation in a single LinkedList

```
package insertany;
import java.util.Scanner;
class Node{
   int regdNo;
   float mark;
   Node next;
}
public class LinkedList {
   //Method to create a Linked List
   public static Node createNode(Node head) {
      Scanner sc=new Scanner(System.in);
      char choice;
      do {
         Node node=new Node();
         Node temp;
         System.out.println("Enter Regd.No.");
         node.regdNo=sc.nextInt();
         System.out.println("Enter mark:");
         node.mark=sc.nextFloat();
         if(head==null)
            head=node;
         else {
            temp=head;
            while(temp.next != null) {
               temp=temp.next;
            }
            temp.next=node;
         }
         System.out.println("Want to create one more node(y/n) ");
         choice=sc.next().charAt(0);
      }while(choice=='y' || choice=='Y');
      return head;
   }
   //Method to compute the size of the linked list
   public static int size(Node head) {
      Node temp=head;
      int count=0;
      while(temp != null) {
      count++;
      temp=temp.next;
```

```
    }
    return count;
  }
  //Method to display the Linked List
  public static void display(Node head) {
    Node temp=head;
    if(head==null) {
    System.out.println("Empty linked list");
    return;
    }
    while(temp != null) {
      System.out.print("("+temp.regdNo+", "+temp.mark+")-->");
      temp=temp.next;
    }
  }
  //Method to insert a new node at begining of the linked list
  public static Node insertAtBeg(Node head) {
    Scanner sc=new Scanner(System.in);
    Node node=new Node();
    System.out.println("Enter mark for new node:");
    node.mark=sc.nextFloat();
    System.out.println("Enter registration number for new node");
    node.regdNo=sc.nextInt();
    node.next=head;
    head=node;
    return head;
  }
  //Method to insert a new node at end of the linked list
  public static Node insertAtEnd(Node head) {
    Scanner sc=new Scanner(System.in);
    Node node=new Node();
    System.out.println("Enter mark for new node:");
    node.mark=sc.nextFloat();
    System.out.println("Enter registration number for new node");
    node.regdNo=sc.nextInt();
    node.next=null;
    Node temp=head;
    while(temp.next != null) {
      temp=temp.next;
    }
    temp.next=node;
    return head;
  }
```

```
   //Method to insert a new node at any index position of the linked list
   public static Node insertAtIndex(Node head) {
      Scanner sc=new Scanner(System.in);
      Node node=new Node();
      Node temp=head;
      System.out.println("\nEnter mark for new node:");
      node.mark=sc.nextFloat();
      System.out.println("Enter registration number for new node:");
      node.regdNo=sc.nextInt();
      int length=size(head);

      System.out.println("Enter the position to insert the new node");
      int indx=sc.nextInt();
      if(indx==0)
         head=insertAtBeg(head);
      else if(indx==length)
         head=insertAtEnd(head);
      else {
         for(int i=0;i<indx-1;i++) {
            temp=temp.next;
         }
         node.next=temp.next;
         temp.next=node;
      }
      return head;
   }
   public static void main(String[] args) {
      Node start=null;
      start=createNode(start);
      System.out.println("Size of linked list: "+size(start));
      display(start);
      start=insertAtIndex(start);
      display(start);
   }

}
```

2.4.4 Delete Node from Single Linked List

Deletion operation in a singly linked list can be done in different ways using position.

- Deletion from beginning.
- Deletion from end.
- Deletion in the middle.

2.4.4.1 Delete node from begining

```
public static Node deleteBegin(Node head) {
   if(head==null) {
      System.out.println("Empty list");
      return head;
   }
   System.out.println("Deleting: "+head.data);
   head=head.next;
   return head;
}
```

2.4.4.2 Delete from the End

Method-1

```
public static Node deleteEnd(Node head) {
   int length=size(head);
   Node temp=head;
   for(int i=0;i<length-2;i++) {
      temp=temp.next;
   }
   temp.next=temp.next.next;
   return head;
}
```

Method-2

```
public static Node deleteEnd2(Node head) {
   Node temp=head;
   Node prev=null;
   while(temp.next != null) {
      prev=temp;
      temp=temp.next;
   }
```

```
   prev.next=null;
   return head;
}
```

2.4.4.3 Delete By Position

Method-1

```
public static Node deleteAtAny(Node head) {
   Scanner sc=new Scanner(System.in);
   int len=size(head);
   Node temp=head;
   System.out.println("Enter the position to insert the new node");
   int index=sc.nextInt();
   if(index==0)
      head=deleteBegin(head);
   if(index==len)
      head=deleteEnd(head);
   else {
      for(int i=0;i<=len-1;i++) {
         temp=temp.next;
      }
      temp.next=temp.next.next;
   }
   return head;
}
```

2.4.5 Reverse a Linked List

```
public static Node reverse(Node head) {
   Node prev=new Node();
   Node temp=new Node();
   Node current=new Node();
   current=head;
   prev=null;
   while(current != null) {
      temp=current.next;
      current.next=prev;
      prev=current;
      current=temp;
   }
   head=prev;
```

```
   return head;
}
```

Example 2.4.2 ***Java Program to demonstrate the Operations on single LinkedList***

```
package insertop;

import java.util.Scanner;
class Node{
   int regdNo;
   float mark;
   Node next;
}
class LinkedList{

   //Method to create a Linked List
   public Node create(Node head) {
      Scanner sc=new Scanner(System.in);
      char choice;
      do {
         Node node=new Node();
         Node temp=null;
         System.out.println("Enter the Registration Number: ");
         node.regdNo=sc.nextInt();
         System.out.println("Enter the Mark: ");
         node.mark=sc.nextInt();
         node.next=null;
         if(head==null) {
            head=node;
         }
         else {
            temp=head;
            while(temp.next!=null) {
               temp=temp.next;
            }
            temp.next=node;
         }

         System.out.println("Do you want to add another node:");
         choice=sc.next().charAt(0);
      }while(choice=='Y' || choice=='y');
      return head;
   }
```

```
//Method to display the Linked List
public void display(Node head) {
   Node temp;
   temp=head;
   if(temp==null)
      System.out.println("Empty linked list");
   else {
      System.out.println("The elements of the linked list are:");
      while(temp != null) {
         System.out.print(temp.regdNo+"|"+temp.mark+"--> ");
         temp=temp.next;
      }
      System.out.print("null");
      System.out.println();
   }
}

//Method to insert node at the beginning of the Linked List
Node insertAtBegin(Node head) {
   Scanner sc=new Scanner(System.in);
   Node node=new Node();
   System.out.println("Enter the Registration Number: ");
   node.regdNo=sc.nextInt();
   System.out.println("Enter the Mark: ");
   node.mark=sc.nextInt();
   if(head==null) {
      head=node;
   }
   else {
      node.next=head;
      head=node;
   }
   return head;
}
//Method to insert node at the end of the Linked List
Node insertAtEnd(Node head) {
   Scanner sc=new Scanner(System.in);
   Node node=new Node();
   Node temp=null;
   System.out.println("Enter the Registration Number: ");
   node.regdNo=sc.nextInt();
   System.out.println("Enter the Mark: ");
```

```
        node.mark=sc.nextInt();
        node.next=null;
        if(head==null) {
            head=node;
        }
        else {
            temp=head;
            while(temp.next!=null) {
                temp=temp.next;
            }
            temp.next=node;
        }
        return head;
    }

    //Method to find the size of the linked list
    int size(Node head) {
        Node temp=head;
        int count=0;
        while(temp!=null) {
            count++;
            temp=temp.next;
        }
        return count;
    }

    //Insert node at any position
    Node insertAtIndex(Node head) {
        Scanner sc=new Scanner(System.in);
        Node node=new Node();
        Node temp=head;
        System.out.println("Enter the position to insert a node");
        int pos=sc.nextInt();
        if(pos==1) {
            head=insertAtBegin(head);
            return head;
        }
        else if(pos==size(head)) {
            head=insertAtEnd(head);
            return head;
        }
        else {
            for(int i=1;i<=pos-1;i++) {
```

```
            temp=temp.next;
        }
        System.out.println("Enter the Registration Number: ");
        node.regdNo=sc.nextInt();
        System.out.println("Enter the Mark: ");
        node.mark=sc.nextInt();
        node.next=temp.next;
        temp.next=node;
    }
    return head;
}

//Method to search the given registration number
void searchItem(Node head) {
    Scanner sc=new Scanner(System.in);
    System.out.println("Search the registration no.");
    int regNo=sc.nextInt();
    Node temp=head;
    int pos=0;
    while(temp!=null) {
        pos++;
        if(temp.regdNo==regNo) {
            System.out.println(temp.regdNo+" available at "+pos);
            return;
        }
        else
            temp=temp.next;
    }
    System.out.println("Registration No. not found");
    return;
}

//Method to reverse the Linked List
public Node reverse(Node head) {
    Node prev=null;
    Node temp=null;
    Node current=head;
    while(current != null) {
        temp=current.next;
        current.next=prev;
        prev=current;
        current=temp;
    }
```

```
    head=prev;
    return head;
  }

  public Node deleteBegin(Node head) {
    if(head==null) {
      System.out.println("Empty list");
      return head;
    }
    System.out.println("Deleting Regd. No: "+head.regdNo);
    head=head.next;
    return head;
  }

  public Node deleteEnd(Node head) {
    Node temp=head;
    Node prev=null;
    while(temp.next != null) {
      prev=temp;
      temp=temp.next;
    }
    prev.next=null;
    return head;
  }

  public Node deleteAtAny(Node head) {
    Scanner sc=new Scanner(System.in);
    int len=size(head);
    Node temp=head;
    System.out.println("Enter the position to delete node");
    int index=sc.nextInt();
    if(index==1)
      head=deleteBegin(head);
    if(index==len)
      head=deleteEnd(head);
    else {
      for(int i=1;i<=index-1;i++) {
        temp=temp.next;
      }
      temp.next=temp.next.next;
    }
    return head;
  }
```

```
}
public class LinkedListInsertDriver {
   public static void main(String[] args) {
      Scanner sc=new Scanner(System.in);
      LinkedList ll=new LinkedList();
      Node head=null;
      int option;
      while(true) {
         System.out.println("0. Exit");
         System.out.println("1. Create a Linked List");
         System.out.println("2. Display");
         System.out.println("3. Insert Node at begin");
         System.out.println("4. Insert Node at end");
         System.out.println("5. Insert Node at Index position");
         System.out.println("6. Search by Registration Number");
         System.out.println("7. Reverse Linked List");
         System.out.println("8. Delete node from begining of the List");
         System.out.println("9. Delete node from end of the List");
         System.out.println("10. Delete node at any index of List");

         System.out.println("\nEnter your choice:");
         option=sc.nextInt();
         switch(option) {
            case 0:
               System.out.println("System exit");
               System.exit(0);
            case 1:
               head=ll.create(head);
               break;
            case 2:
               ll.display(head);
               break;
            case 3:
               head=ll.insertAtBegin(head);
               break;
            case 4:
               head=ll.insertAtEnd(head);
               break;
            case 5:
               head=ll.insertAtIndex(head);
               break;
            case 6:
               ll.searchItem(head);
```

```
            break;
         case 7:
            head=ll.reverse(head);
            break;
         case 8:
            head=ll.deleteBegin(head);
            break;
         case 9:
            head=ll.deleteEnd(head);
            break;
         case 10:
            head=ll.deleteAtAny(head);
            break;
         default:
            System.out.println("Wrong choice");

      }
   }
}

}
```

2.5 Doubly LinkedList

2.5.1 Create Node in Doubly LinkedList

Algorithm 2.5.1 createNodeDL (Head)

[Create new Node]
1: Create the new Node
2: Set Node.data = input "item"
3: Set Node.next = null
4: Set Node.prev = null
[Check for empty List]
5: **if** Head = null **then**
Set Head = Node
6: **else**
Set Temp = Head
7: **while** $Temp.next \neq null$ **do**
Set Temp = Temp.next
8: **end while**
9: Temp.next = Node
10: Node.prev = Temp
11: **end if**
12: Return Head

Method 2.5.1 *Java method to create node in Doubley LinkedList*

```
public Node createNode(Head){
   Scanner sc=new Scanner(System.in);
   Node temp=null;
   Node node=new Node();
   System.out.println("Enter the item");
   node.data=sc.nextInt();
   if(head==null) {
      head=node;
   }
   else {
      temp=head;
      while(temp.next!=null) {
         temp=temp.next;
      }
      temp.next=node;
      node.prev=temp;
   }
}
```

2.5.2 Traverse Doubly LinkedList

Algorithm 2.5.2 TraverseDL (Head)

1: **if** Head = null **then**
   Print: "The Linked list is empty"
   Return Head
2: **end if**
3: Temp = Head
4: **while** $Temp \neq null$ **do**
   a) print: Temp.data
   b) Set Temp = Temp.next
5: **end while**
6: Return Head

Method 2.5.2 *Java method to traversen Doubley LinkedList*

```
void display(Node head) {
   if(head==null){
      System.out.print("Eplt LinkedList);
      return;
   }
   Node temp=head;
   while(temp!=null) {
      System.out.print(temp.data+" ");
      temp=temp.next;
   }
   System.out.println("null");
}
```

2.5.3 Fid size of Doubly LinkedList

Method 2.5.3 *Java Method to find the size of the Doubly linked list*

```
//Method to find the size of the doubly linked list
int size(Node head) {
   Node temp=head;
   int count=0;
   while(temp!=null) {
      count++;
      temp=temp.next;
   }
   return count;
}
```

2.5.4 Insert Node at Begin of the Doubly LinkedList

Method 2.5.4 *Java Method to Insert Node at Begin of the Doubly LinkedList*

```
Node insertAtBegin(Node head) {
   Scanner sc=new Scanner(System.in);
   System.out.println("Insert node at the begining");
   Node node=new Node();
   System.out.println("Enter the item");
   node.data=sc.nextInt();
   if(head==null) {
      head=node;
   }
   else {
      node.next=head;
      head=node;
   }
   return head;
}
```

2.5.5 Insert Node at End of the Dobly LinkedLIst

Method 2.5.5 *Java Method to Insert Node at End of the Dobly LinkedLIst*

```
Node insertAtEnd(Node head) {
   Scanner sc=new Scanner(System.in);
   System.out.println("Inserting node at end:");
   Node temp=null;
   Node node=new Node();
   System.out.println("Enter the item");
   node.data=sc.nextInt();
   if(head==null) {
      head=node;
   }
   else {
      temp=head;
      while(temp.next!=null) {
         temp=temp.next;
      }
      temp.next=node;
      node.prev=temp;
   }
   return head;
```

```
}
```

2.5.6 Insert Node at any position of the Doubly LinkedList

Method 2.5.6 *Java Method to Insert Node at any position of the Doubly LinkedList*

```
Node insertAtIndex(Node head) {
   Scanner sc=new Scanner(System.in);
   Node node=new Node();
   System.out.println("Enter the item");
   node.data=sc.nextInt();
   System.out.println("Enter the position to insert");
   int pos=sc.nextInt();
   Node temp=head;
   for(int i=1;i<pos;i++) {
      temp=temp.next;
   }
   temp.next=node;
   node.prev=temp;
   return head;
}
```

2.5.7 Delete Node from the Beginning of a Doubly LinkedList

Method 2.5.7 *Java Method to Delete Node from the Beginning of a Doubly LinkedList*

```
Node deleteBeginDL(Node head) {
   if(head==null) {
      System.out.println("Empty list");
      return head;
   }
   else {
      head=head.next;
      head.prev=null;
   }
   return head;
}
```

2.5.8 Delete Node from the End of a Doubly LinkedList

Method 2.5.8 *Java Method to Delete Node from the End of a Doubly LinkedList*

```
Node deleteEndDL(Node head) {
   Node temp=null;
   Node ptemp=null;
   if(head==null) {
      System.out.println("Empty list");
      return head;
   }
   else {
      temp=head;
      while(temp.next!=null) {
         ptemp=temp;
         temp=temp.next;
      }
      ptemp.next=null;
      temp.prev=null;
   }
   return head;
}
```

2.5.9 Delete Node from any position of a Doubly LinkedList

Method 2.5.9 *Java Method to Delete Node from any position of a Doubly LinkedList*

```
Node deleteAtIndexDL(Node head) {
   Scanner sc=new Scanner(System.in);
   Node temp=null;
   int index;
   if(head==null) {
      System.out.println("Empty list");
      return head;
   }
   else {
      System.out.println("Enter the position to delete:");
      index=sc.nextInt();
      temp=head;
      for(int i=1;i<index;i++) {
         temp=temp.next;
      }
      System.out.println("Deleting node at index: "+(index+1)+" data:
          "+temp.next.data);
      temp.next=temp.next.next;
      temp.next.prev=temp;
```

```
    }
    return head;
}
```

Example 2.5.1 *Java Program to demonstrate the Operations on single LinkedList*

```
package createdl_head;

import java.util.Scanner;

class Node{
    int data;
    Node prev;
    Node next;
    Node(){
        prev=next=null;
    }
}
class LinkedList{
    public Node createNodeDL(Node head) {
        //System.out.println("Creating linked list.....");
        Scanner sc=new Scanner(System.in);
        char choice;
        do {
            Node temp=null;
            Node node=new Node();
            System.out.println("Enter the item");
            node.data=sc.nextInt();
            if(head==null) {
                head=node;
            }
            else {
                temp=head;
                while(temp.next!=null) {
                    temp=temp.next;
                }
                temp.next=node;
                node.prev=temp;
            }
            System.out.println("Want to create a node:Y|y");
            choice=sc.next().charAt(0);
        }while(choice == 'Y' ||choice == 'y');
```

```
    return head;
  }
  void displayDL(Node head) {
    Node temp=head;
    while(temp!=null) {
      System.out.print(temp.data+" ");
      temp=temp.next;
    }
  }
  //Method to find the size of the doubly linked list
  int sizeDL(Node head) {
    Node temp=head;
    int count=0;
    while(temp!=null) {
      count++;
      temp=temp.next;
    }
    return count;
  }
  Node insertAtBeginDL(Node head) {
    Scanner sc=new Scanner(System.in);
    System.out.println("Insert node at the begining");
    Node node=new Node();
    System.out.println("Enter the item");
    node.data=sc.nextInt();
    if(head==null) {
      head=node;
    }
    else {
      node.next=head;
      head=node;
    }
    return head;
  }
  Node insertAtEndDL(Node head) {
    Scanner sc=new Scanner(System.in);
    System.out.println("Inserting node at end:");
    Node temp=null;
    Node node=new Node();
    System.out.println("Enter the item");
    node.data=sc.nextInt();
    if(head==null) {
      head=node;
```

```
    }
    else {
       temp=head;
       while(temp.next!=null) {
          temp=temp.next;
       }
       temp.next=node;
       node.prev=temp;
    }
    return head;
 }
 Node insertAtIndexDL(Node head) {
    Scanner sc=new Scanner(System.in);
    Node node=new Node();
    System.out.println("Enter the item");
    node.data=sc.nextInt();
    System.out.println("Enter the position to insert");
    int pos=sc.nextInt();
    Node temp=head;
    for(int i=1;i<pos-1;i++) {
       temp=temp.next;
    }
    node.next=temp.next;
    node.prev=temp;
    temp.next.prev=node;
    temp.next=node;
    return head;
 }
 Node deleteBeginDL(Node head) {
    if(head==null) {
       System.out.println("Empty list");
       return head;
    }
    else {
       System.out.println("Deleting first node: "+head.data);
       head=head.next;
       head.prev=null;
    }
    return head;
 }
 Node deleteEndDL(Node head) {
    Node temp=null;
    Node ptemp=null;
```

```
    if(head==null) {
       System.out.println("Empty list");
       return head;
    }
    else {
       temp=head;
       while(temp.next!=null) {
          ptemp=temp;
          temp=temp.next;
       }
       System.out.println("Deleting last node: "+temp.data);
       ptemp.next=null;
       temp.prev=null;
    }
    return head;
  }
  Node deleteAtIndexDL(Node head) {
    Scanner sc=new Scanner(System.in);
    Node temp=null;
    int index;
    if(head==null) {
       System.out.println("Empty list");
       return head;
    }
    else {
       System.out.println("Enter the position to delete:");
       index=sc.nextInt();
       temp=head;
       for(int i=1;i<index-1;i++) {
          temp=temp.next;
       }
       System.out.println("Deleting node at index: "+(index+1)+" data:
           "+temp.next.data);
       temp.next=temp.next.next;
       temp.next.prev=temp;
    }

    return head;
  }
}
public class DoubleLInkedLIstDriver {

  public static void main(String[] args) {
```

```
Scanner sc=new Scanner(System.in);
LinkedList ll=new LinkedList();
Node head=null;
int option;
while(true) {
   System.out.println("0. Exit");
   System.out.println("1. Create a Linked List");
   System.out.println("2. Display");
   System.out.println("3. Insert Node at begin");
   System.out.println("4. Insert Node at end");
   System.out.println("5. Insert Node at Index position");
   System.out.println("6. Delete node from begining of the List");
   System.out.println("7. Delete node from end of the List");
   System.out.println("8. Delete node from any position");
   System.out.println("7. Reverse Linked List");



   System.out.println("\nEnter your choice:");
   option=sc.nextInt();
   switch(option) {
   case 0:
      System.out.println("System exit");
      System.exit(0);
   case 1:
      head=ll.createNodeDL(head);
      break;
   case 2:
      ll.displayDL(head);
      break;
   case 3:
      head=ll.insertAtBeginDL(head);
      break;
   case 4:
      head=ll.insertAtEndDL(head);
      break;
   case 5:
      head=ll.insertAtIndexDL(head);
      break;
   case 6:
      head=ll.deleteBeginDL(head);
      break;
   case 7:
```

```
            head=ll.deleteEndDL(head);
            break;
         case 8:
            head=ll.deleteAtIndexDL(head);
            break;
         default:
            System.out.println("Wrong choice");
         }
      }


   }

}
```

Chapter 3

Stack

3.1 Introduction

- A stack is a one of the most important and useful non-primitive linear data structure in computer science.
- As all the insertion and deletion in a stack is done from the top of the stack, the lastly added element will be first to be removed from the stack.
- Real-life examples of the stack are a stack of books, a stack of plates, a stack of cards, a stack of coins, etc.

Definition: A stack is a sequential collection of objects that are inserted and removed according to the **last-in, first-out (LIFO)** principle.

Note: The most frequently accessed element in the stack is the top most elemental, whereas the least accessible element is the bottom of the stack.

3.2 Operation on Stack

The following tasks are performed by the top variable:

- To keep track, how many cells are used,
- Whether the stack is full or empty
- Insert new element in the stack
- Delete elements from the stack

The stack is an abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, we can implement a stack using either array or linked list. There are some basic operations that allow us to perform different actions on a stack.

- **Push:** Add an element to the top of a stack
- **Pop:** Remove an element from the top of a stack
- **IsEmpty:** Check if the stack is empty
- **IsFull:** Check if the stack is full
- **Peek:** Get the value of the top element without removing it

3.2.1 Working of Stack Data Structure

The operations on stack will be carried out as follows:

- A pointer called **TOP** is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to **-1** so that we can check if the stack is empty by comparing **TOP == -1**.
- On pushing an element, we increase the value of **TOP** and place the new element in the position pointed to by **TOP**
- On popping an element, we return the element pointed to by **TOP** and reduce its value.
- Before pushing, we check if the stack is already full
- Before popping, we check if the stack is already empty

3.3 Array Implementation of Stack

The array can be used to implement a stack of fixed size, therefore only fixed a number of data items can be pushed and popped.

- Consider the following Fig-**??**, the stack of size 4.
- Therefore, maximum only 4 data items can be inserted.
- The top index always keeps track of the last inserted element of the stack, which is the top of the stack.
- Initially, the top is initialized by -1 (for the zero-based array) when there are no items in the stack, i.e. stack is empty.

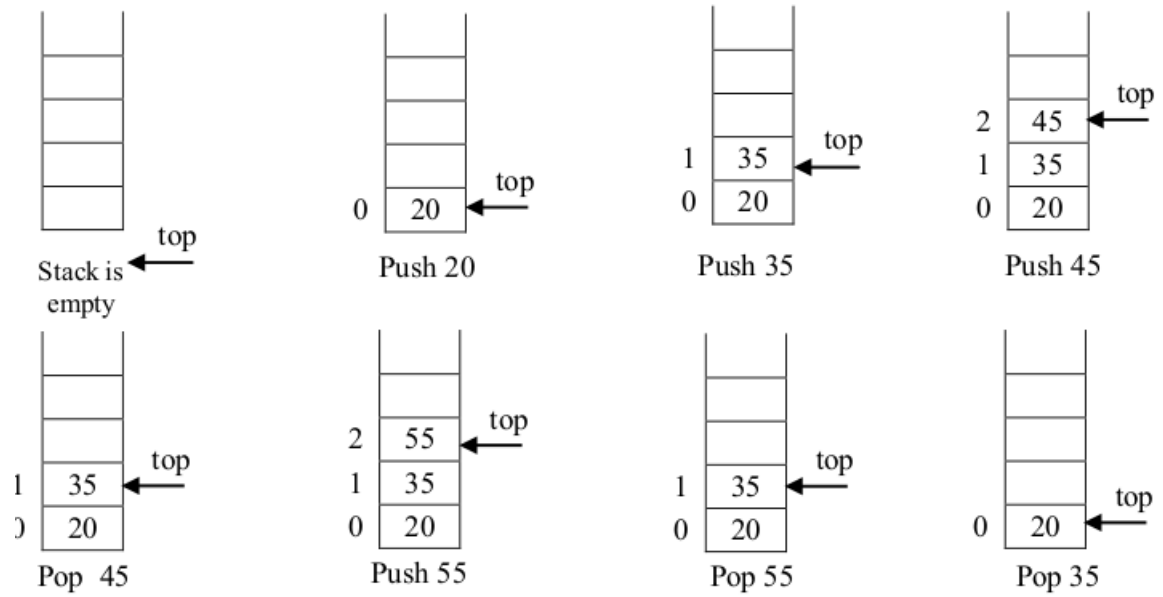

Figure 3.1: Stack Operations

Method 3.3.1 *Creating a Stack Class*

```
class Stack{
   Scanner sc=new Scanner(System.in);
   int top=-1;
   int n;
   int stackArr[];
   Stack() {
      System.out.println("Enter size of the stack array:");
      n=sc.nextInt();
      stackArr=new int[n];
   }
}
```

Method 3.3.2 *Method to Check stack is empty or not*

```
boolean isEmpty() {
   if(top==-1)
      return true;
   else
      return false;
}
```

Method 3.3.3 *Method to Check stack is full or not*

```
boolean isFull() {
```

```
   if(top==stackArr.length)
      return true;
   else
      return false;
}
```

Method 3.3.4 *Method to push element in the Stack*

```
void push(int item) {
   if(isFull()) {
      System.out.println("Stack is full(Overflow)");
      return;
   }
   else {
      stackArr[++top]=item;
   }
}
```

Method 3.3.5 *Method to pop element from the Stack*

```
void pop() {
   if(isEmpty()) {
      System.out.println("Stack is empty(Underflow)");
      return;
   }
   else {
      System.out.println("Pop element: "+stackArr[top--]);
   }
}
```

Method 3.3.6 *Method to Display the Stack*

```
void display() {
   for(int i=0;i<=top;i++) {
      System.out.print(stackArr[i]+" ");
   }
   System.out.println();
}
```

Program 3.3.1 *Complete Java Program For Stack Operation Using Array*

```
import java.util.Scanner;
class Stack{
```

```
Scanner sc=new Scanner(System.in);
int top=-1;
int n;
int stackArr[];
Stack() {
   System.out.println("Enter size of the stack array:");
   n=sc.nextInt();
   stackArr=new int[n];
}
boolean isEmpty() {
   if(top==-1)
      return true;
   else
      return false;
}
boolean isFull() {
   if(top==stackArr.length)
      return true;
   else
      return false;
}
void push(int item) {
   if(isFull()) {
      System.out.println("Stack is full(Overflow)");
      return;
   }
   else {
      stackArr[++top]=item;
   }
}
void pop() {
   if(isEmpty()) {
      System.out.println("Stack is empty(Underflow)");
      return;
   }
   else {
      System.out.println("Pop element: "+stackArr[top--]);
   }
}
void display() {
   for(int i=0;i<=top;i++) {
      System.out.print(stackArr[i]+" ");
   }
```

```
      System.out.println();
   }
}
public class StackDriver {
   public static void main(String[] args) {
      Stack st=new Stack();
      st.push(10);
      st.push(8);
      st.push(7);
      st.push(4);
      st.display();
      st.pop();
      st.display();
   }
}
```

The following program demonstrate the implementation operation using array where the push and pop method takes the array as argument.

Program 3.3.2 ***Java Program to demonstrate the Operations on Stack using array***

```
package arraystack;

import java.util.Scanner;
public class ArrayImpStack {
   static final int MAX=10; //Capacity of the stack
   static boolean isEmpty(int top) {
      if(top==-1)
         return true;
      else
         return false;
   }
   static boolean isFull(int top) {
      if(top==MAX-1)
         return true;
      else
         return false;
   }
   static int push(int S[],int top) {
      Scanner sc=new Scanner(System.in);
      if(isFull(top)) {
         System.out.println("Overflow");
      }
```

```
    else {
      System.out.println("Enter item:");
      int item=sc.nextInt();
      S[++top]=item;
    }
    return top;
  }
  static int pop(int S[], int top) {
    if(isEmpty(top))
      System.out.println("Underflow");
    else {
      System.out.println("pop the top: "+S[top--]);
    }
    return top;
  }
  static void display(int S[], int top) {
    for(int i=0;i<=top;i++) {
      System.out.print(S[i]+" ");
    }
    System.out.println();
  }
  public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    int stack[]=new int[MAX];
    int top=-1;
    while(true) {
      System.out.println();
      System.out.println("0. Exit");
      System.out.println("1. Push");
      System.out.println("2. Display");
      System.out.println("3. Pop");
      System.out.println("\nEnter your choice");
      int choice=sc.nextInt();
      switch(choice) {
        case 0:
          System.out.println("Exit from Stack Operation");
          System.exit(0);
        case 1:
          top=push(stack,top);
          break;
        case 2:
          System.out.println("The Stack elements are: ");
          display(stack, top);
```

```
                break;
            case 3:
                top=pop(stack, top);
                break;
            default:
                System.out.println("Choice is not matching.....");
                break;
        }
    }

  }

}
```

3.4 Linked List Implementation of Stack

- Another way to represent stack is by using the singly linked list, which is also known as Linked Stack.
- A linked list is a dynamic data structure and each element of a linked list is a node that contains a data and a next similar to the linked list.
- The next is a pointer to another node and the data is the value stored in the node.
- The linked list header acts as the top of the Stack.
- All push or pop operations are taking place at the front of the linked list.
- Each operation always changes the header of the linked list.
- When the stack is empty then HEAD is null. If the stack has at least one node, the first node is the top of the stack.
- In the push operation, it needs to add a new node to the front of the list.
- The pop operation removes the first node of the linked list when the stack is not empty.

Method 3.4.1 *Method to Create Node for Stack*

```
class Node{
   int data;
   Node next;
}
```

Method 3.4.2 ***Method for PUSH Operation***

```
class Stack{
   Node top=null;
   int size=0;
   public void push(int item) {
      Node node=new Node();
      node.data=item;
      node.next=top;
      top=node;
      size++;
   }
}
```

Method 3.4.3 ***Method for POP Operation***

```
void pop() {
   if(top==null) {
      System.out.println("Underflow: Stack is empty");
   }
   System.out.println("Delete top data: "+top.data);
   top=top.next;
}
```

Method 3.4.4 *Method for Display the Stack*

```
void display() {
   Node temp=top;
   while(temp != null) {
      System.out.print(temp.data);
      temp=temp.next;
      if(temp != null) {
         System.out.print("-->");
      }
   }
   System.out.println();
}
```

Program 3.4.1 *Complete Java Program for Linked List implementation of Stack*

```
class Node{
   int data;
   Node next;
}
class Stack{
   private Node top=null;
   private int size=0;
   public void push(int item) {
      Node node=new Node();
      node.data=item;
      node.next=top;
      top=node;
      size++;
   }
   void display() {
      Node temp=top;
      while(temp != null) {
         System.out.print(temp.data);
```

```
            temp=temp.next;
            if(temp != null) {
               System.out.print("-->");
            }
         }
         System.out.println();
      }
      int length() {
         return size;
      }
      void pop() {
         if(top==null) {
            System.out.println("Underflow: Stack is empty");
         }
         System.out.println("Delete top data: "+top.data);
         top=top.next;
      }
      int peep() {
         return top.data;
      }
}
public class LinkNodeStack {
      public static void main(String[] args) {
         Stack st=new Stack();
         st.push(5);
         st.push(4);
         st.push(7);
         st.push(3);
         st.push(9);
         st.push(8);
         st.display();
         System.out.println("TOP: "+st.peep());
         st.pop();
         System.out.println("TOP: "+st.peep());
         st.display();
      }
}
```

3.5 Evaluation of Arithmetic Expressions

- An expression is defined as a number of operands or data items combined with several operators.
- An Arithmetic expression consists of arithmetic operators and operands.
- There are three types of notations in an arithmetic expression.

  i Infix Notation

  ii Prefix Notation

  iii Postfix Notation

3.5.1 Infix Notation

- Most usually, in arithmetic expressions, the binary operator appears between its two operands. This is called infix notation.
- The general form of the infix notation is:

  Op1 operator Op2 where Op1 and Op2 are two operands

- **Example:** a + b

3.5.2 Prefix Notation

- In prefix notation, binary operators appear before its two operands. This notation is also known as Polish notation.
- In prefix notation, the operations that are to be performed is absolutely determined by the positions of the operators and operands in the expression.
- Therefore, parentheses are never used when writing expressions in prefix notation.
- The general form of the prefix notation is:

  operator Op1 Op2 where Op1 and Op2 are two operands

- **Example:** +ab

3.5.3 Postfix Notation

- In postfix notation, binary operators appear after its two operands. This notation is also known as Reverse Polish notation.
- In postfix notation, the operations are to be performed is absolutely determined by the positions of the operators and operands in the expression.
- Therefore, parentheses are never used when writing expressions in postfix notation. The general form of the postfix notation is:

  Op1 Op2 operator where Op1 and Op2 are two operands

- **Example:** ab+

3.6 Converting infix expression to postfix form

- The order of evaluation can be fixed by assigning a priority to each operator.
- The operators within parentheses having the highest priority will be evaluated first.
- When an expression has two operators with same priority then the expression is evaluated according to its associativity (left to right or right to left) order.

Operator	Description	Priority
+ -	Unary operator	5
^	Power operator	4
* / %	Multiplication, Division, Remainder	3
+ -	Addition, Subtraction	2
<<= >>= == !=	Relational operators	1

Figure 3.2: Stack Operations

- There is an algorithm to convert an infix expression to the equivalent postfix expression.
- A stack is used here to store operators rather than operand.
- The purpose of the stack is to reverse the order of the operators in the expression.

Algorithm converts an infix expression to the equivalent postfix expression.

Algorithm 3.6.1 POSTFIX (Q, P)

[*Q is a given infix expression and P is a postfix expression*]

1: Push "(" onto stack & add ")" to the end of Q.
2: Scan Q from left to right
3: **while** *Stack* $\neq$ *null* **do**
4: If the element is an operand then add it to P.
5: If the element is left parenthesis "(" then push it onto the stack.
6: If the element is an operator then:
a) Repeatedly pop from stack (until the element on top of the stack has higher or same precedence than the operator currently scanned) and add it to P.
b) Add the operator to stack.
7: If the element is a right parenthesis ")"then:
a) Repeatedly pop from stack and add to P each operator until a left parenthesis "(" is found
b) Pop the left parenthesis from the stack.
8: **end while**
9: Return P

Example 3.6.1 *Find the postfix expression of the following infix expression:* $A + B * C$

Add "(" in the Stack and ")" at the end of the expression. It is represented as Initial in the table and highlighted. The expression will be $A + B * C$ *)*

Sl. No.	Symbol Scanned	Stack	Postix Expression(P)
Initial		(	
1	A	(	A
2	+	(+	A
3	B	(+	AB
4	*	(+*	AB
5	C	(+*	ABC
6	)	Empty Stack	ABC*+

Example 3.6.2 *Find the postfix expression of the following infix expression:* $A + B * C/D - E$

Add "(" in the Stack and ")" at the end of the expression. It is represented as Initial in the table and highlighted. The expression will be $A + B * C/D - E$ *)*

Sl. No.	Symbol Scanned	Stack	Postix Expression(P)
Initial		(	
1	A	(	A
2	+	(+	A
3	B	(+	AB
4	*	(+*	AB
5	C	(+*	ABC
6	/	(+/	ABC*
7	D	(+/	ABC*D
8	-	(-	ABC*D/+
9	E	(-	ABC*D/+E
10	)	Empty Stack	ABC*D/+E-

Example 3.6.3 *Find the postfix expression of the following infix expression:*
$Q = A + (B * C - (D/E \uparrow F) * G) * H$

Serial Number	Symbol Scanned	Stack	Postfix Expression (P)
Initial		(	
1	A	(	A
2	+	(+	A
3	(	(+ (	A
4	B	(+ (	A B
5	*	(+ (*	A B
6	C	(+ (*	A B C
7	-	(+ (-	A B C *
8	(	(+ (- (	A B C *
9	D	(+ (- (	A B C * D
10	/	(+ (- (/	A B C * D
11	E	(+ (- (/	A B C * D E
12	↑	(+ (- (/ ↑	A B C * D E
13	F	(+ (- (/ ↑	A B C * D E F
14	)	(+ (-	A B C * D E F ↑ /
15	*	(+ (- *	A B C * D E F ↑ /
16	G	(+ (- *	A B C * D E F ↑ / G
17	)	(+	A B C * D E F ↑ / G * -
18	*	(+ *	A B C * D E F ↑ / G * -
19	H	(+ *	A B C * D E F ↑ / G * - H
20	)	STACK EMPTY	A B C * D E F ↑ / G * - H * +

Postfix expression A B C * D E F ↑ / G * - H * +

Figure 3.3: Infix to Postfix Convertion

3.6.1 Evaluation of a Postfix Expression

Algorithm 3.6.2 POSTFIX (P)

[*P is a postfix expression*]
1: Add a right parenthesis ")" at the end of P.
2: Read P from left to right and repeat step 3 and 4
3: **for each** element of P until the ")" is found.
4: **If** an operand is found, put it onto the stack.
5: **If** an operator # is found then
a) Pop the two top elements of the stack,
Where A is the top element and B is the next to top element
b) Evaluate R = B # A
[End of If]
[End of Loop]
6: Set Result equals to the top element on stack
7: Return

Example 3.6.4 *Find the value of following postfix expression:* $532 * 8 + *$

Serial Number	Symbol Scanned	Stack	Output
1	5	5	
2	3	5 3	
3	2	5 3 2	
4	*	5 6	
5	8	5 6 8	
6	+	5 14	
7	*	70	
			70

Figure 3.4: Postfix Evaluation

3.7 Questions

1. Convert following infix expression to postfix expression using Stack:

   (a) $((a + b)/d - ((e - f) + g)$

   (b) $A + (B * C - D/(E + F) * G) * H$

   (c) $12/3 * 6 + 6 - 6 + 8/2$

   (d) $10 + 3 - 2 - 8/2 * 6 - 7$

2. Convert following infix expression to postfix expression and evaluate the postfix expression using Stack

   (a) $12/3 * 6 + 6 - 6 + 8/2$

   (b) $10 + 3 - 2 - 8/2 * 6 - 7$

Chapter 4

Queue

4.1 Introduction

The queue is also another useful non-primitive linear data structure in computer science. A real-life example of the queue is line or sequence of people or vehicles awaiting their turn to be attended to or to proceed.

Definition: A queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue.

The element to enter the queue first will be deleted from the queue first. That is why a queue is called **First-In-First-Out (FIFO)** system.

4.1.1 Stack vs Queue

Stack	Queue
In the stack, items are inserted and deleted at the one end of the list	In the queue, items are inserted at one end (called rear) and deleted at another end (called the front)
Stack is Last-in-First-out system	The queue is the First-in-First-out system

4.2 Array Implementation of Queue

Method 4.2.1 *Method to insert element in a Queue*

```
class Queue{
   int front=-1;
   int rear=-1;
   int size=0;
   int arrQ[]=new int[10];
   void enque(int item) {
      if(size==arrQ.length-1) {
```

```
            System.out.println("Queue is full");
            return;
        }
        if(front==-1) {
            front=rear=0;
            arrQ[rear]=item;
        }
        else {
            arrQ[++rear]=item;
        }
        size++;
    }
}
```

Method 4.2.2 *Method to delete element from a Queue*

```
void dequeue() {
    int x=arrQ[front];
    System.out.println("Remving item: "+x);
    front++;
    size--;
}
```

Method 4.2.3 *Method to Display the Queue*

```
void displayQ() {
    for(int i=front;i<=rear;i++) {
        System.out.print(arrQ[i]+" ");
    }
    System.out.println();
}
```

Method 4.2.4 *Method to check empty of full status of the Queue*

```
boolean isEmpty() {
    if(size==0) return true;
    else return false;
}
boolean isFull(){
    if(size==arrQ.length) return true;
    else return false;
}
```

Program 4.2.1 *Complete Java Program for Operations on Queue*

```
import java.util.Scanner;
class Queue{
   int front=-1;
   int rear=-1;
   int size=0;
   int arrQ[];
   Queue() {
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter maximum number of elements");
      int n=sc.nextInt();
      arrQ=new int[n];
   }
   void enqueue(int item) {
      if(size==arrQ.length) {
         System.out.println("Queue is full");
         return;
      }
      if(front==-1) {
         front=rear=0;
         arrQ[rear]=item;
      }
      else {
         arrQ[++rear]=item;
      }
      size++;
   }
   void dequeue() {
      int x=arrQ[front];
      System.out.println("Removing item: "+x);
      front++;
      size--;
   }
   boolean isEmpty() {
      if(size==0) return true;
      else return false;
   }
   void displayQ() {
      for(int i=front;i<=rear;i++) {
         System.out.print(arrQ[i]+" ");
      }
      System.out.println();
```

```
   }
}
public class QueueArray {
   public static void main(String[] args) {
      Queue q=new Queue();
      System.out.println("size = "+q.size);
      q.enqueue(1);
      q.enqueue(2);
      q.enqueue(3);
      q.enqueue(4);
      q.displayQ();
      System.out.println("size = "+q.size);
      q.dequeue();
      q.displayQ();
      System.out.println("size = "+q.size);
   }
}
```

```
Output:
Enter maximum number of elements in the Queue
10
size = 0
1
2
3
4
size = 4
Removing item: 1
2
3
4
size = 3
```

Program 4.2.2

4.3 LinkedList Implementation of Queue

- Singly linked list can be used to represent a queue, which is also known as **Linked Queue**.
- In this representation, any number of data items can be inserted and deleted.
- The front and rear pointers always keep track of the first node and the last node in the linked list respectively.
- Initially, **front** and **rear** are initialized by null (i.e. front = rear = null), when there are no items in the queue, that means the queue is empty.
- The linked list header acts as the front of the queue.
- All deletion operations take place at the **front** of the list.
- All insertion operations take place at the end of the Linked queue.
- If the queue contains a single element then front and rear points to new node (i.e. front = rear = node).
- When a new item is inserted in the queue, a new node is inserted at the end of the linked list, the rear points to the new node.
- When an item is deleted from the queue, the node from the front of the queue is deleted.
- Now, if **front = null** then if we try to delete an item, it results in **underflow** condition. It indicates that the queue is empty and we cannot delete an item.
- Whenever the queue is found empty, we can reset the **front and rear by null**.

4.4 Comparisons of queer representation using linked list over the array

- The array is fixed size, therefore, a number of elements will be limited in the queue. Since linked list is dynamic and can be changed easily, so the number of elements can be changed.
- The front and rear node reference acting as a pointers in linked list consume additional memory compared to an array.
- In array implementation, sometimes dequeue operation not possible, although there are free slots. This drawback can be overcome in linked list representation.
- In array and linked list enqueue and dequeue operations can be done in $O(1)$.

4.5 Operations on Queue using Linked List

Method 4.5.1 *Enqueue operation*

```
void enqueue(int item) {
   Node node=new Node();
   node.data=item;
   node.next=null;
   if(rear==null)
      front=rear=node;
   else {
      rear.next=node;
      rear=node;
   }
}
```

Method 4.5.2 *Dequeue operation*

```
void enqueue(int item) {
   Node node=new Node();
   node.data=item;
   node.next=null;
   if(rear==null)
   front=rear=node;
   else {
   rear.next=node;
   rear=node;
   }
   size++;
}
```

Program 4.5.1 *Complete Java Program for LinkedList implementation of Queue*

```
class Node{
   int data;
   Node next;
}
class Queue{
   Node front=null; // It is head
   Node rear=null; // It is tail
   int size=0;
   boolean isEmpty() {
   if(size==0) return true;
   else return false;
```

```
    }
    void enqueue(int item) {
        Node node=new Node();
        node.data=item;
        node.next=null;
        if(rear==null)
            front=rear=node;
        else {
            rear.next=node;
            rear=node;
        }
        size++;
    }
    void dequeue() {
        if(isEmpty()) {
            System.out.println("Queue is empty");
            return;
        }
        int x=front.data;
        System.out.println("Removing item: "+x);
        front=front.next;
        size--;
    }
    void displayQ() {
        Node temp=front;
        if(size==0) {
            System.out.println("No element in the Queue");
            return;
        }
        while(temp != null) {
            System.out.print(temp.data+" ");
            temp=temp.next;
        }
        System.out.println();
    }
}
public class LinkedQueue {
    public static void main(String[] args) {
        Queue q=new Queue();
        q.displayQ();
        System.out.println("size = "+q.size);
        q.enqueue(1);
        q.enqueue(2);
```

```
        q.enqueue(3);
        q.enqueue(4);
        q.displayQ();
        System.out.println("size = "+q.size);
        q.dequeue();
        q.displayQ();
        System.out.println("size = "+q.size);
    }
}
```

```
Output:
No element in the Queue
size = 0
1 2 3 4
size = 4
Removing item: 1
2 3 4
size = 3
```

Appendix Title Here

Write your Appendix content here.