Exercise No: 11

DOUBLY LINKED LIST

AIM

Write a Java program for the following:

1) Create a doubly linked list of elements.
2) Delete a given element from the above list.
3) Display the contents of the list after deletion.

THEORY

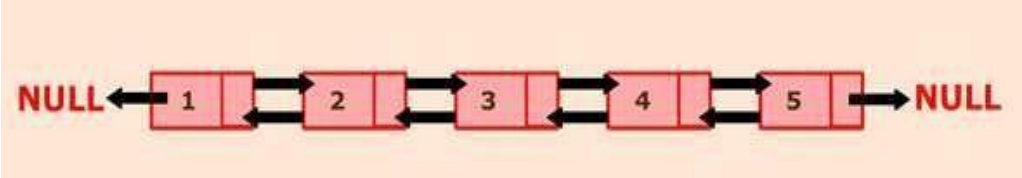
Doubly Linked List is a variation of the linked list. The linked list is a linear data structure which can be described as the collection of nodes. Nodes are connected through pointers. Each node contains two fields: data and pointer to the next field. The first node of the linked list is called the head, and the last node of the list is called the tail of the list.

One of the limitations of the singly linked list is that it can be traversed in only one direction that is forward. The doubly linked list has overcome this limitation by providing an additional pointer that point to the previous node. With the help of the previous pointer, the doubly linked list can be traversed in a backward direction thus making insertion and deletion operation easier to perform. So, a typical node in the doubly linked list consists of three fields:

Data represents the data value stored in the node.

Previous represents a pointer that points to the previous node.

Next represents a pointer that points to next node in the list.



Above picture represents a doubly linked list in which each node has two pointers to point to previous and next node respectively. Here, node 1 represents the head of the list. The previous pointer of the head node will always point to NULL. Next pointer of node one will point to node 2. Node 5 represents the tail of the list whose previous pointer will point to node 4, and next will point to **NULL**

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner.

ALGORITHM

Step 1: Define a Node class which represents a node in the list. It will have three properties: data, previous which will point to the previous node and next which will point to the next node.

Step 2: Define another class for creating a doubly linked list, and it has two nodes: head and tail. Initially, head and tail will point to null.

Step 3: addNode() will add node to the list:

a. It first checks whether the head is null, then it will insert the node as the head.
b. Both head and tail will point to a newly added node.
c. Head's previous pointer will point to null and tail's next pointer will point to null.
d. If the head is not null, the new node will be inserted at the end of the list such that new node's previous pointer will point to tail.
e. The new node will become the new tail. Tail's next pointer will point to null.

Step 4: display() will show all the nodes present in the list.

a. Define a new node 'current' that will point to the head.
b. Print current.data till current points to null.
c. Current will point to the next node in the list in each iteration.

PROGRAM

```
import java.util.Scanner;
class DoublyLinkedList
{
        private Node head;
        class Node
   {
                private int data;
                private Node left;
                private Node right;

                public Node(int data)
      {
                        this.data = data;
                        this.left = null;
                        this.right = null;
                }
        }

        public void insertAtEnd(int data)
   {
                Node temp = new Node(data);
                if(head == null)
      {
                        head = temp;
                }
                else
      {
                        Node ptr = head;
                        while(ptr.right != null)
        {
```

```
                    ptr = ptr.right;
                }
                ptr.right = temp;
                temp.left = ptr;
            }
            System.out.println(data+ " is inserted Successfully");
        }

        public void deleteAtFront()
    {

            if(head == null)
       {
                System.out.println("List is Empty");
            }
            else
       {
                int data = head.data;
                head = head.right;
                head.left = null;
                System.out.println(data+" is deleted from the list");
            }
        }
        public void display()
    {
            Node temp = head;
            if(head == null)
       {
                System.out.println("List Empty");
            }
            else
```

```
    {
                while(temp != null)
      {
                        System.out.print(temp.data+"\t");
                        temp = temp.right;
                }
                System.out.println();
        }
      }
}
class Test
{
      public static void main(String args[])
{
            DoublyLinkedList dl = new DoublyLinkedList();
            int ch = 0;
            while(ch != 4){
                  System.out.println("1.Insert\n2.Delete\n3.Display\n4.Exit");
                  Scanner sc = new Scanner(System.in);
                  ch = sc.nextInt();
                  sc.nextLine();

                  switch(ch){
                        case 1: System.out.println("Enter the element");
                                    int elt = sc.nextInt();
                                    sc.nextLine();
                                    dl.insertAtEnd(elt);
                                    break;
                        case 2:dl.deleteAtFront();
                             break;
                        case 3:dl.display();
```

```
                    break;
                case 4: break;
                default: System.out.println("Invalid Choice");
            }
        }
    }
}
```

OUTPUT

1. Insert
2. Delete
3. Display
4. Exit

>>>1

Enter the element

>>>5

5 is inserted Successfully

1. Insert
2. Delete
3. Display
4. Exit

>>>1

Enter the element

>>>10

10 is inserted Successfully

1. Insert
2. Delete
3. Display
4. Exit

>>>1

Enter the element

>>>15

15 is inserted Successfully

1.Insert

2.Delete

3.Display

4.Exit

>>>3

5 10 15

1. Insert

2.Delete

3.Display

4.Exit

>>>2

5 is deleted from the list

1.Insert

2. Delete

3.Display

4.Exit

>>>3

10 15

1. Insert

2.Delete

3.Display

4.Exit

>>>4

VIVA VOCE QUESTIONS

1. Explain Linked List in short?
2. How will you represent a linked list in a graphical view?
3. How many types of Linked List exist?
4. What are the main differences between the Linked List and Linear Array?
5. Explain the operations in doubly linked list?

Exercise No: 12

QUICK SORT

AIM

Write a Java program that implements Quick sort algorithm for sorting a list of names in ascending order

THEORY

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

ALGORITHM

QuickSortPivot()

Step 1: Choose the highest index value has pivot

Step 2: Take two variables to point left and right of the list excluding pivot

Step 3: left points to the low index

Step 4: right points to the high

Step 5: while value at left is less than pivot move right

Step 6: while value at right is greater than pivot move left

Step 7: if both step 5 and step 6 does not match swap left and right

Step 8: if left ≥ right, the point where they met is new pivot

QuickSort()

Step 1: Make the right-most index value pivot

Step 2: partition the array using pivot value

Step 3: quicksort left partition recursively

Step 4: quicksort right partition recursively

PROGRAM

```
class QuickSort
{
        public static int partition(String[] A,int p,int r)
   {
                String temp;
                String x = A[r];
                int i = p-1;
                for(int j = p; j <= r-1; j++)
      {
                        if(A[j].compareTo(x) <= 0)
         {
                                i = i+1;
                                temp = A[i];
                                A[i] = A[j];
                                A[j] = temp;
                        }
                }
                temp = A[i+1];
                A[i+1] = A[r];
```

```
                A[r] = temp;
                return i+1;
        }
        public static void quickSort(String [] A,int p,int r)
  {
                if(p < r)
    {
                        int q = partition(A,p,r);
                        quickSort(A,p,q-1);
                        quickSort(A,q+1,r);
                }
        }

        public static void main(String [] args)
  {
                String names[] ={"Sangeeth","Poornima", "Pratheesh","Vikas","Karthi","Abin"};
                quickSort(names,0,5);
                System.out.println("After QuickSort");
                for(int i=0;i<names.length;i++){
                        System.out.println(names[i]);
                }
        }
}
```

OUTPUT

Abin

Karthi

Poornima

Pratheesh

Sangeeth

Vikas

VIVA VOCE QUESTIONS

1. Explain divide and conquer technique?
2. What is the worst case time complexity of the Quick sort?
3. Quick sort is not stable. Justify?
4. What is pivot element?

Exercise No: 13

TRAFFIC LIGHT

AIM

Write a Java program that simulates a traffic light. The program lets the user select one of three lights: red, yellow, or green. When a radio button is selected, the light is turned on, and only one light can be on at a time. No light is on when the program starts.

THEORY

An applet is a special kind of Java program that runs in a Java enabled browser. This is the first Java program that can run over the network using the browser. Applet is typically embedded inside a web page and runs in the browser.

In other words, we can say that Applets are small Java applications that can be accessed on an Internet server, transported over Internet, and can be automatically installed and run as a part of a web document.

After a user receives an applet, the applet can produce a graphical user interface. It has limited access to resources so that it can run complex computations without introducing the risk of viruses or breaching data integrity.

Lifecycle of Java Applet

Following are the stages in Applet

1. Applet is initialized.
2. Applet is started
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Circles and Ellipses

The Graphics class does not contain any method for circles or ellipses. To draw an ellipse, use drawOval(). To fill an ellipse, use fillOval().

Syntax

void drawOval(int top, int left, int width, int height)

void fillOval(int top, int left, int width, int height)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by (top,left) and whose width and height are specified by width and height. To draw a circle, specify a square as the bounding rectangle i.e get height = width.

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

Components of Event Handling

Event handling has three main components,

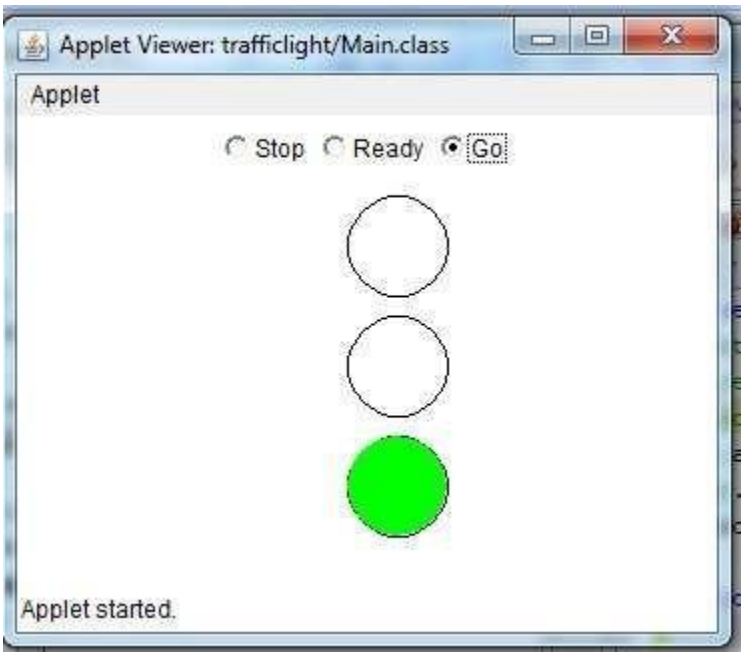
- **Events :** An event is a change in state of an object.
- **Events Source :** Event source is an object that generates an event.
- **Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

PROGRAM

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*<applet code="Signals" width=400 height=250></applet>*/
public class Signals extends Applet implements ItemListener
{
String msg="";
Checkbox stop,ready,go;
CheckboxGroup cbg;
public void init()
{
cbg = new CheckboxGroup();
stop = new Checkbox("Stop", cbg, false);
ready = new Checkbox("Ready", cbg, false);
go= new Checkbox("Go", cbg, false);
add(stop);
add(ready);
add(go);
stop.addItemListener(this);
ready.addItemListener(this);
go.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
repaint();
}
public void paint(Graphics g)
{
msg=cbg.getSelectedCheckbox().getLabel();
```

```
g.drawOval(165,40,50,50);
g.drawOval(165,100,50,50);
g.drawOval(165,160,50,50);
if(msg.equals("Stop"))
{
g.setColor(Color.red);
g.fillOval(165,40,50,50);
}
else if(msg.equals("Ready"))
{
g.setColor(Color.yellow);
g.fillOval(165,100,50,50);
}
else
{
g.setColor(Color.green);
g.fillOval(165,160,50,50);
}
}
}
```

OUTPUT



VIVA VOCE QUESTIONS

1. Explain delegation event model?
2. Explain event source, Listeners and event class?
3. List the methods in Listener interface?
4. Compare Swing and AWT?
5. Define components and containers?
6. Define applet?
7. Explain applet life cycle?

Exercise No: 14

SIMPLE CALCULATOR

AIM

Write a Java program that works as a simple calculator. Arrange Buttons for digits and the + - * % operations properly. Add a text field to display the result. Handle any possible exceptions like divide by zero. Use Java Swing.

THEORY

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener** - It is also known as event handler.Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now get executed and returns.

Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key,

selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

public void addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as **addKeyListener()**. For the mouse event listener, the method will be called as **addMouseMotionListener()**. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

Some listeners allow only one listener to register. Below is an example:

public void addTypeListener(TypeListener e2)

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

public void removeTypeListener(TypeListener e2?)

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the **MouseMotionListener** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

PROGRAM

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
public class calculator extends JFrame implements ActionListener
{
  JButton b10,b11,b12,b13,b14,b15;
  JButton b[]=new JButton[10];
   int i,r,n1,n2;
   JTextField res;
   char op;
  public calculator()
 {
    super("Calulator");
     setLayout(new BorderLayout());
     JPanel p=new JPanel();
     p.setLayout(new GridLayout(4,4));
```

```
for(int i=0;i<=9;i++)
{
  b[i]=new JButton(i+"");
  p.add(b[i]);
  b[i].addActionListener(this);
}
b10=new JButton("+");
p.add(b10);
b10.addActionListener(this);

b11=new JButton("-");
p.add(b11);
b11.addActionListener(this);

b12=new JButton("*");
p.add(b12);
b12.addActionListener(this);

b13=new JButton("/");
p.add(b13);
b13.addActionListener(this);

b14=new JButton("=");
p.add(b14);
b14.addActionListener(this);

b15=new JButton("C");
p.add(b15);
b15.addActionListener(this);

res=new JTextField(10);
```

```
    add(p,BorderLayout.CENTER);
    add(res,BorderLayout.NORTH);
    setVisible(true);
    setSize(200,200);
   }
public void actionPerformed(ActionEvent ae)
{
 JButton pb=(JButton)ae.getSource();
        if(pb==b15)
        {
         r=n1=n2=0;
         res.setText("");
        }
        else
                if(pb==b14)
                {
                 n2=Integer.parseInt(res.getText());
                 eval();
                 res.setText(""+r);
                }

                else
                {
                  boolean opf=false;
                  if(pb==b10)
                        { op='+';
                          opf=true;
                        }
                  if(pb==b11)
                        { op='-';opf=true;}
                  if(pb==b12)
```

```
                    { op='*';opf=true;}
                if(pb==b13)
                    { op='/';opf=true;}

                if(opf==false)
                {
                 for(i=0;i<10;i++)
                 {
                    if(pb==b[i])
                    {
               String t=res.getText();
               t+=i;
                    res.setText(t);
                    }
                 }
                }
                else
                {
                    n1=Integer.parseInt(res.getText());
                    res.setText("");
                }
               }
}
int eval()
{
        switch(op)
        {
        case  '+':  r=n1+n2;  break;
        case   '-':   r=n1-n2;   break;
        case  '*':  r=n1*n2;  break;
        case '/':  r=n1/n2; break;
```

```
        }
        return 0;
}

  public static void main(String arg[])
  {
     new calculator();
   }
}
```

OUTPUT

VIVA VOCE QUESTIONS

1. What is Java Swing?
2. What Are Heavyweight Components?
3. What is a Layout Manager?
4. What are the advantages of the Event-delegation Model?