Data Structure Lab Manual

First semester

2018-2019

Data Structure Lab Schedule

Data Structure Lab Schedule (First Semester 2018-2019)	
Week#	**Topic**
2	Object- Oriented Design
3+4	Graphical User Interface and Event-Driven Programming
5+6	Fundamental Data Structures
7	Recursion
8	**Midterm Exam**
9+10	Stack and Queues
11+12	BST
13	Graphics
14	AVL tree
15	**Final Exam**

Experiment No. 1

OOP Revision

Description:

Object-oriented programming (OOP) involves programming using objects. An object represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

- The state of an object (also known as its properties or attributes) is represented by data fields with their current values.
- The behavior of an object (also known as its actions) is defined by methods. To invoke a method on an object is to ask the object to perform an action

Objects of the same type are defined using a common class. A class is a template, blueprint, or contract that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class.

The aim of this lab is to conduct a revision of OOP and its related concepts.

The basic concepts of OOP: Object, Class and Interface, Inheritance, Abstraction, Polymorphism, Encapsulation, Overloading and overriding.

Suggested Exercises:

1) Design and implement GeometricShape class and classes of each geometric shape (Circle, Rectangle, Triangle, Line…etc.).
2) Apply the basic concepts on the geometric example (inheritance, polymorphism, abstraction, interfaces...etc.).
3) Override the equals method in the Object class.
4) Rewrite the Rectangle, Circle, and Triangle…etc. classes and implement the Comparable interface.

Experiment No. 2

GUI and Event handling

Description:

In this lab, we will introduce the basics of Java GUI programming and Event-Driven Programming.

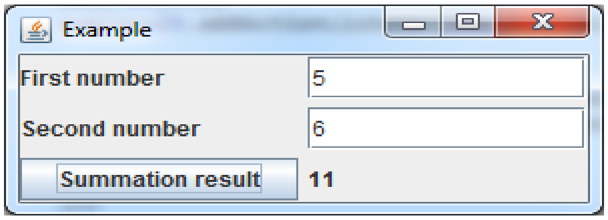
- There are current three sets of java APIs for graphics programming: AWT (abstract windowing toolkit), Swing and JavaFX.
    - AWT API was introduced in JDK 1.0. Most of the AWT components have become obsolete and should be replaced by newer Swing components.
    - Swing API, a much more comprehensive set of graphics libraries that enhances the AWT. Swing components depend less on the target platform and use less of the native GUI resource. For this reason, Swing components that don't rely on native GUI are referred to as lightweight components, and AWT components are referred to as heavyweight components.
    - The latest JavaFX, which was integrated into JDK 8, is meant to replace Swing.

  In this lab we will use Swing because it has been adopted in the theoretical course
- The GUI API contains classes that can be classified into three groups: component classes, container classes, and helper classes.
- The component classes, such as JButton, JLabel, and JTextField, are for creating the user interface. The container classes, such as JFrame, JPanel, and JApplet, are used to contain other components. The helper classes, such as Graphics, Color, Font, FontMetrics, and Dimension, are used to support GUI components.
- The basic components that will be used (JFrame,JPanle, JButton, JTextField and JLabels).
    - JFrame: To create a user interface, you need to create either a frame or an applet to hold the user-interface components.
    - JPanel: JPanels act as subcontainers to group user-interface components.

- The components will be arranged in the container depending on the layout of the container. There are many layouts manager as: FlowLayout, GridLayout, BorderLayout.

- Java uses a delegation-based model for event handling: a source object fires an event, and an object interested in the event handles it. The latter object is called a listener. In this lab we will deal with ActionListener, MouseListener, KeyListner and WindowsListner.

Example of designing a GUI



```
import javax.swing.*;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Example extends JFrame {
    JLabel num1 = new JLabel("First number");
    JLabel num2 = new JLabel("Second number");
    JButton sumresult = new JButton("Summation result");
    JLabel result = new JLabel();
    JTextField num1Input = new JTextField(8);
    JTextField num2Input = new JTextField(8);
    public Example() {
        setLayout(new GridLayout(3, 2, 5, 5));
        add(num1);
        add(num1Input);
        add(num2);
        add(num2Input);
        add(sumresult);
        add(result);
        sumresult.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                int x=Integer.parseInt(num1Input.getText());
                int y=Integer.parseInt(num2Input.getText());
                result.setText( x+ y + "");
            }
        });
    }
    public static void main(String[] args) {
        Example frame = new Example();
        frame.setTitle("Example");
        frame.setSize(300, 125);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Suggested Exercises:

In the following question use the layout managers and new components like JRadioButton, JCheckBox, JTextArea, JProgressBar…etc.

1) Design and implement a scientific calculator.
2) Design and implement Tic Tac Toe game.
3) Design and implement a program to calculate the area of any geometric shape.

Experiment No. 3

ArrayList

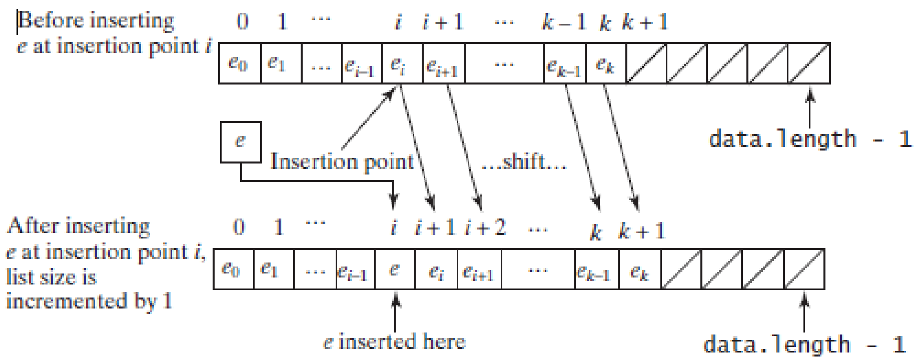
Description:

An array list is implemented using an array.

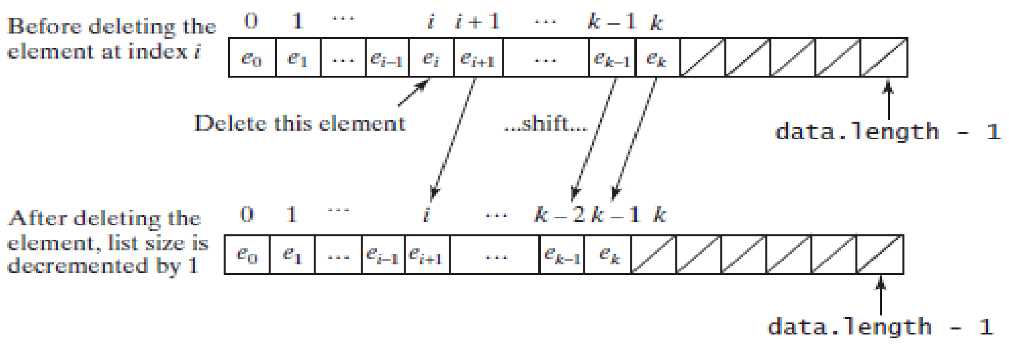
An array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use arrays to implement dynamic data structures.

Initially, an array, say data of E [] type, is created with a default size.

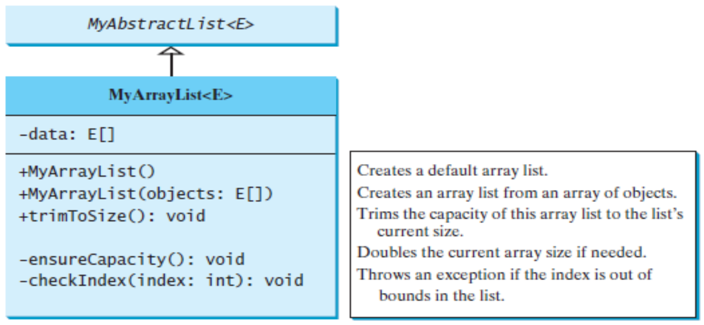
1) When inserting a new element into the array, first make sure that there is enough room in the array. If not, create a new array twice as large as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array. Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1, as shown in the following figure.



2) To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1, as shown in the following figure.



Implementation:


MyAbstractList<E>
MyArrayList<E>
-data: E[]
+MyArrayList()
+MyArrayList(objects: E[])
+trimToSize(): void
-ensureCapacity(): void
-checkIndex(index: int): void
Creates a default array list.
Creates an array list from an array of objects.
Trims the capacity of this array list to the list's current size.
Doubles the current array size if needed.
Throws an exception if the index is out of bounds in the list.

```java
public class MyArrayList<E> extends MyAbstractList<E> {
  public static final int INITIAL_CAPACITY = 16;
  private E[] data = (E[]) new Object[INITIAL_CAPACITY];

  /** Create a default list */
  public MyArrayList() {
  }

  /** Create a list from an array of objects */
  public MyArrayList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      add(objects[i]); // Warning: don't use super(objects)!
  }

  @Override /** Add a new element at the specified index */
  public void add(int index, E e) {
    ensureCapacity();

    // Move the elements to the right after the specified index
    for (int i = size - 1; i >= index; i--)
      data[i + 1] = data[i];

    // Insert new element to data[index]
    data[index] = e;

    // Increase size by 1
    size++;
  }

  /** Create a new larger array, double the current size + 1 */
  private void ensureCapacity() {
    if (size >= data.length) {
      E[] newData = (E[])(new Object[size * 2 + 1]);
      System.arraycopy(data, 0, newData, 0, size);
      data = newData;
    }
  }
```

```
public E remove(int index) {
  checkIndex(index);

  E e = data[index];

  // Shift data to the left
  for (int j = index; j < size - 1; j++)
    data[j] = data[j + 1];

  data[size - 1] = null; // This element is now null

  // Decrement size
  size--;

  return e;
}



public void clear() {
  data = (E[])new Object[INITIAL_CAPACITY];
  size = 0;
}

@Override /** Return true if this list contains the element */
public boolean contains(E e) {
  for (int i = 0; i < size; i++)
    if (e.equals(data[i])) return true;

  return false;
}

@Override /** Return the element at the specified index */
public E get(int index) {
  checkIndex(index);
  return data[index];
}
```

```
public E set(int index, E e) {
  checkIndex(index);
  E old = data[index];
  data[index] = e;
  return old;
}

@Override
public String toString() {
  StringBuilder result = new StringBuilder("[");

  for (int i = 0; i < size; i++) {
    result.append(data[i]);
    if (i < size - 1) result.append(", ");
  }

  return result.toString() + "]";
}

/** Trims the capacity to current size */
public void trimToSize() {
  if (size != data.length) {
    E[] newData = (E[])(new Object[size]);
    System.arraycopy(data, 0, newData, 0, size);
    data = newData;
  } // If size == capacity, no need to trim
}
```

<u>Suggested Exercises:</u>

Create the following methods in MyArrayList class

1) Method to swap two elements in an arraylist.
2) Method to revers an arraylist using recursion.
3) Implement the **rotate()** method that rotates the elements in cyclic order following is the method signature: **Static void rotate (MyArrayList** list**, int r):** Rotates the elements in the arraylist list by the specified distance **r.**
4) Write a method to merge two sorted ArrayLists.

Experiment No. 4

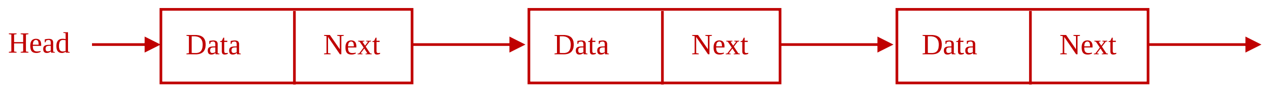
LinkedList

Description:

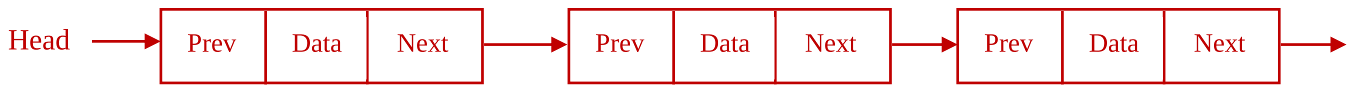
LinkedList class uses a linked structure to implement a dynamic list. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

There are many types of linked list:

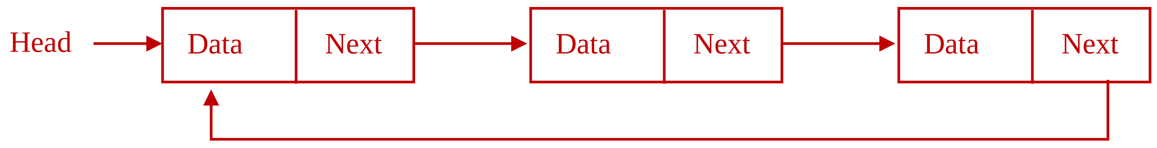
1- Single LinkedList: The most common type where each node has data and a pointer to the next node.
2- Double LinkedList: another additional pointer is added to point to the previous node enabling moving forward and backward.
3- Circular LinkedList: in this form the last node is linked to the first node forming a loop.



a) Single LinkedList



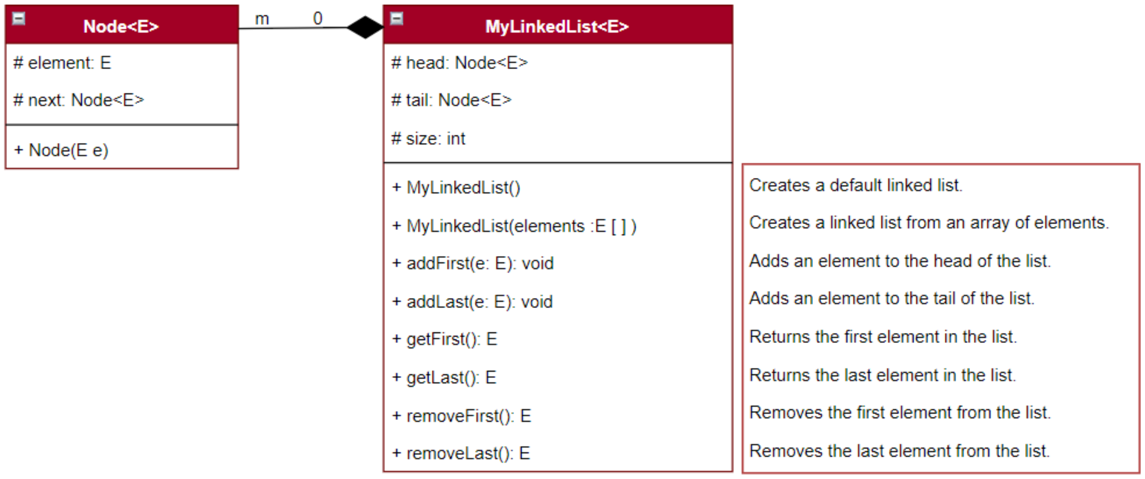
b) Double LinkedList



c) Circular LinkedList

Implementation:

In this lab, LinkedList class will be implemented in addition to different operation on LinkedList. First, class Node is implemented, where each object from this class holds an element in the LinkedList.

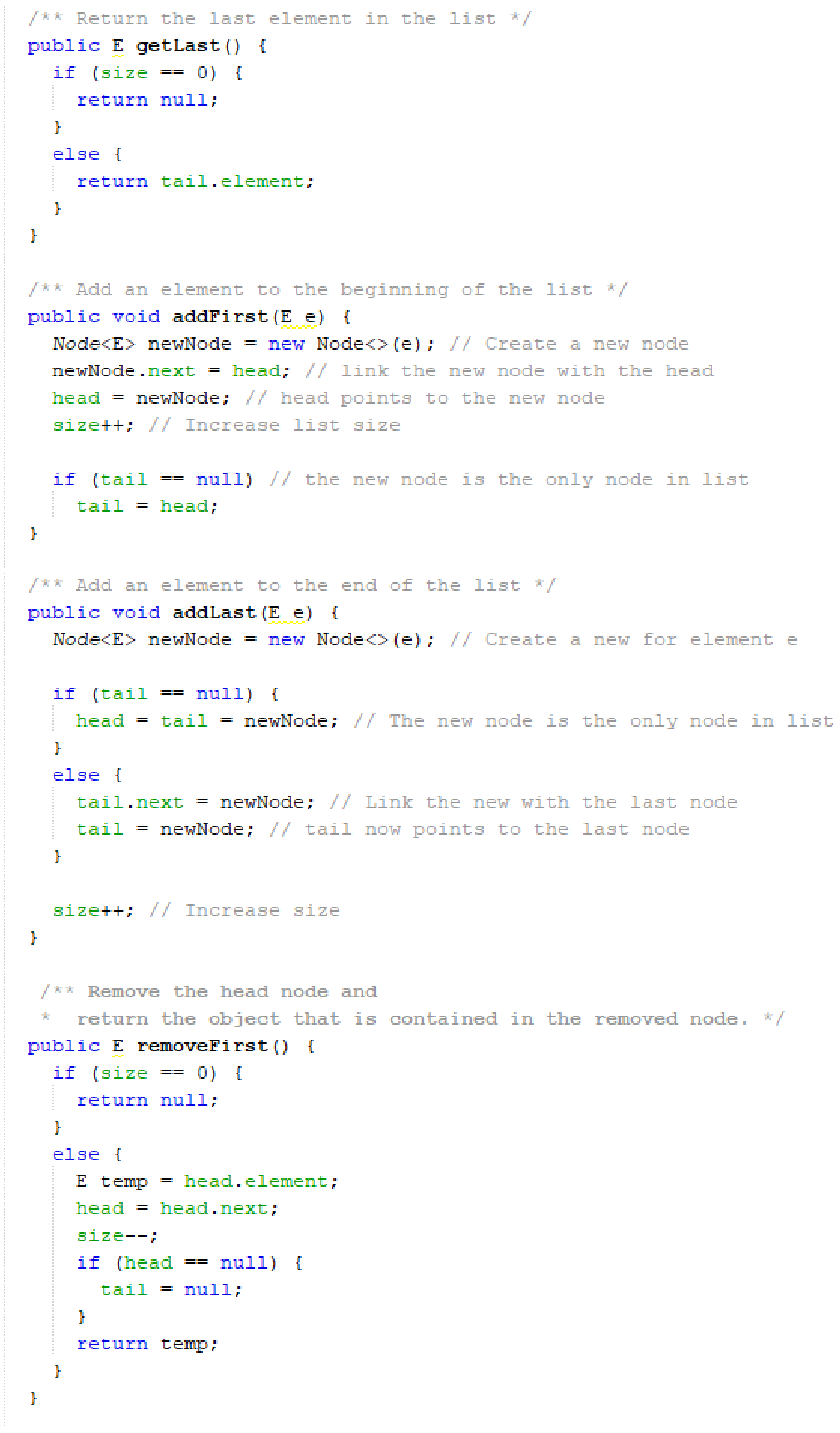

Node<E>
# element: E
# next: Node<E>
+ Node(E e)
m
0
MyLinkedList<E>
# head: Node<E>
# tail: Node<E>
# size: int
+ MyLinkedList()
+ MyLinkedList(elements :E [ ])
+ addFirst(e: E): void
+ addLast(e: E): void
+ getFirst(): E
+ getLast(): E
+ removeFirst(): E
+ removeLast(): E
Creates a default linked list.
Creates a linked list from an array of elements.
Adds an element to the head of the list.
Adds an element to the tail of the list.
Returns the first element in the list.
Returns the last element in the list.
Removes the first element from the list.
Removes the last element from the list.

a) Node Class implementation:

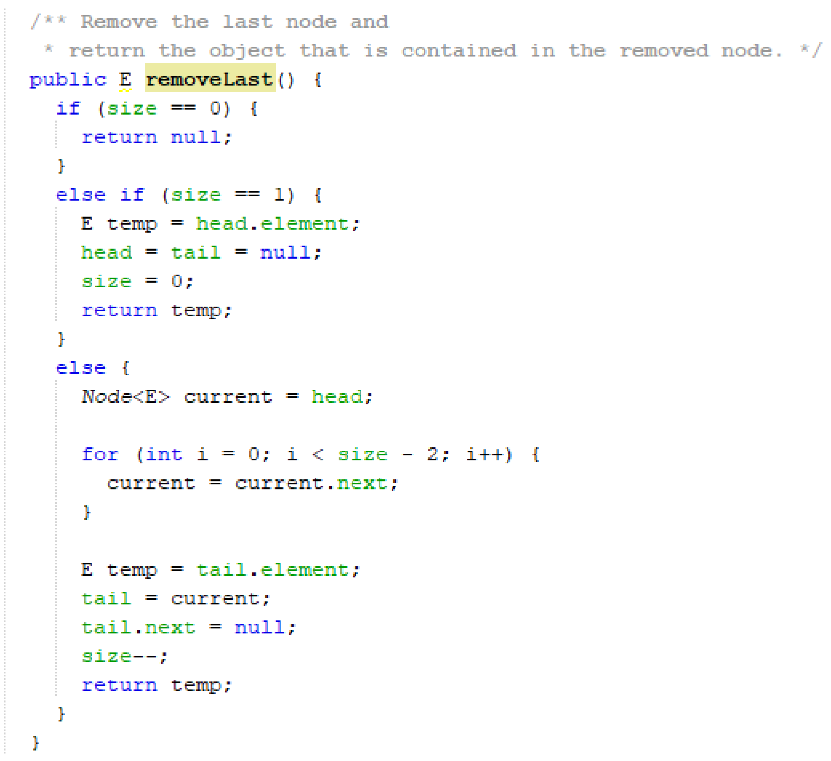
```
protected class Node<E> {

    protected E element;
    protected Node<E> next;

    public Node(E element) {
        this.element = element;
    }

}
```

b) MyLinkedList Class implementation:

```
public class MyLinkedList<E> {
  private Node<E> head, tail;
  private int size = 0; // Number of elements in the list

  /** Create an empty list */
  public MyLinkedList() {
  }

  /** Create a list from an array of objects */
  public MyLinkedList(E[] objects) {
    for (int i = 0; i < objects.length; i++)
      addLast(objects[i]);
  }

  /** Return the head element in the list */
  public E getFirst() {
    if (size == 0) {
      return null;
    }
    else {
      return head.element;
    }
  }
```



```java
/** Return the last element in the list */
public E getLast() {
  if (size == 0) {
    return null;
  }
  else {
    return tail.element;
  }
}

/** Add an element to the beginning of the list */
public void addFirst(E e) {
  Node<E> newNode = new Node<>(e); // Create a new node
  newNode.next = head; // link the new node with the head
  head = newNode; // head points to the new node
  size++; // Increase list size

  if (tail == null) // the new node is the only node in list
    tail = head;
}

/** Add an element to the end of the list */
public void addLast(E e) {
  Node<E> newNode = new Node<>(e); // Create a new for element e

  if (tail == null) {
    head = tail = newNode; // The new node is the only node in list
  }
  else {
    tail.next = newNode; // Link the new with the last node
    tail = newNode; // tail now points to the last node
  }

  size++; // Increase size
}

 /** Remove the head node and
 *  return the object that is contained in the removed node. */
public E removeFirst() {
  if (size == 0) {
    return null;
  }
  else {
    E temp = head.element;
    head = head.next;
    size--;
    if (head == null) {
      tail = null;
    }
    return temp;
  }
}
```



```
/** Remove the last node and
 * return the object that is contained in the removed node. */
public E removeLast() {
  if (size == 0) {
    return null;
  }
  else if (size == 1) {
    E temp = head.element;
    head = tail = null;
    size = 0;
    return temp;
  }
  else {
    Node<E> current = head;

    for (int i = 0; i < size - 2; i++) {
      current = current.next;
    }

    E temp = tail.element;
    tail = current;
    tail.next = null;
    size--;
    return temp;
  }
}
```

Suggested Exercises:

1) Implement add method to add element at specific index.
2) Implement remove method to remove element at specific index.
3) Implement reverse method to reverse linked list using recursion and iteration.
4) Implement the methods contains(E e), get(int index), indexOf(E e), lastIndexOf(E e), and set(int index, E e).
5) Write a test class to deal with Linked List.
6) Write a method to merge two sorted linked lists.
7) Implement the Double LinkedList class.
8) Implement the Circular LinkedList class.

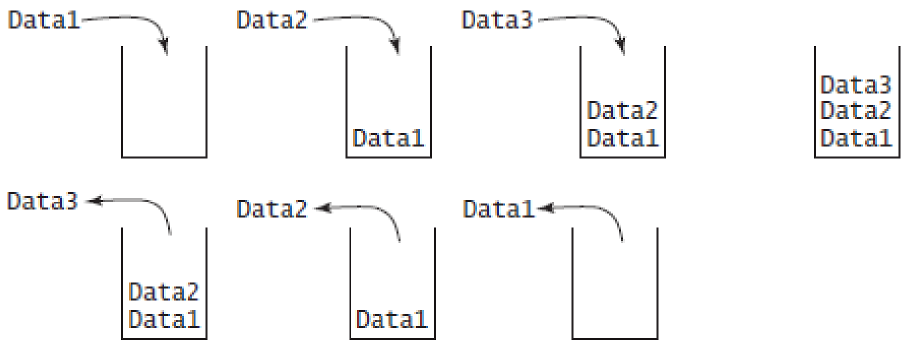
Experiment No. 5

Stacks

Description:

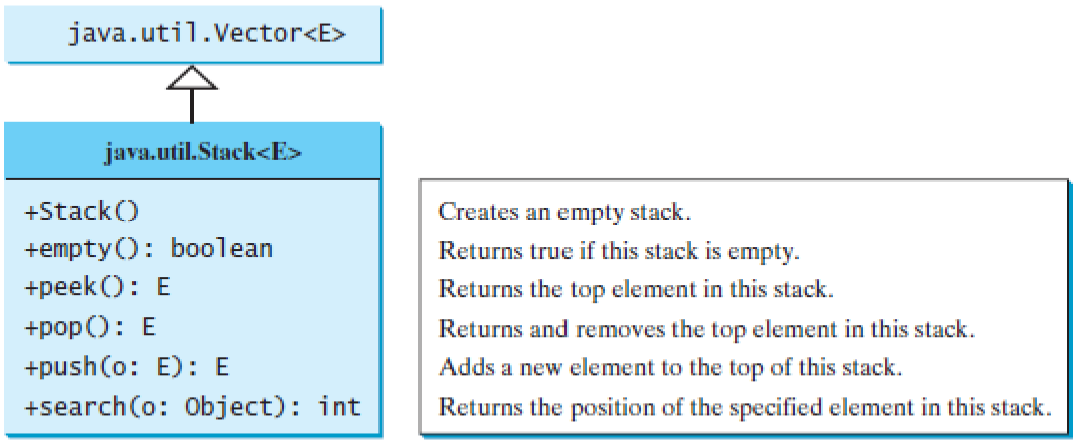
Stacks can be implemented using array lists and queues can be implemented using linked lists.

- A stack can be viewed as a special type of list whose elements are accessed, inserted, and deleted only from the end (top), as shown in the following figure.

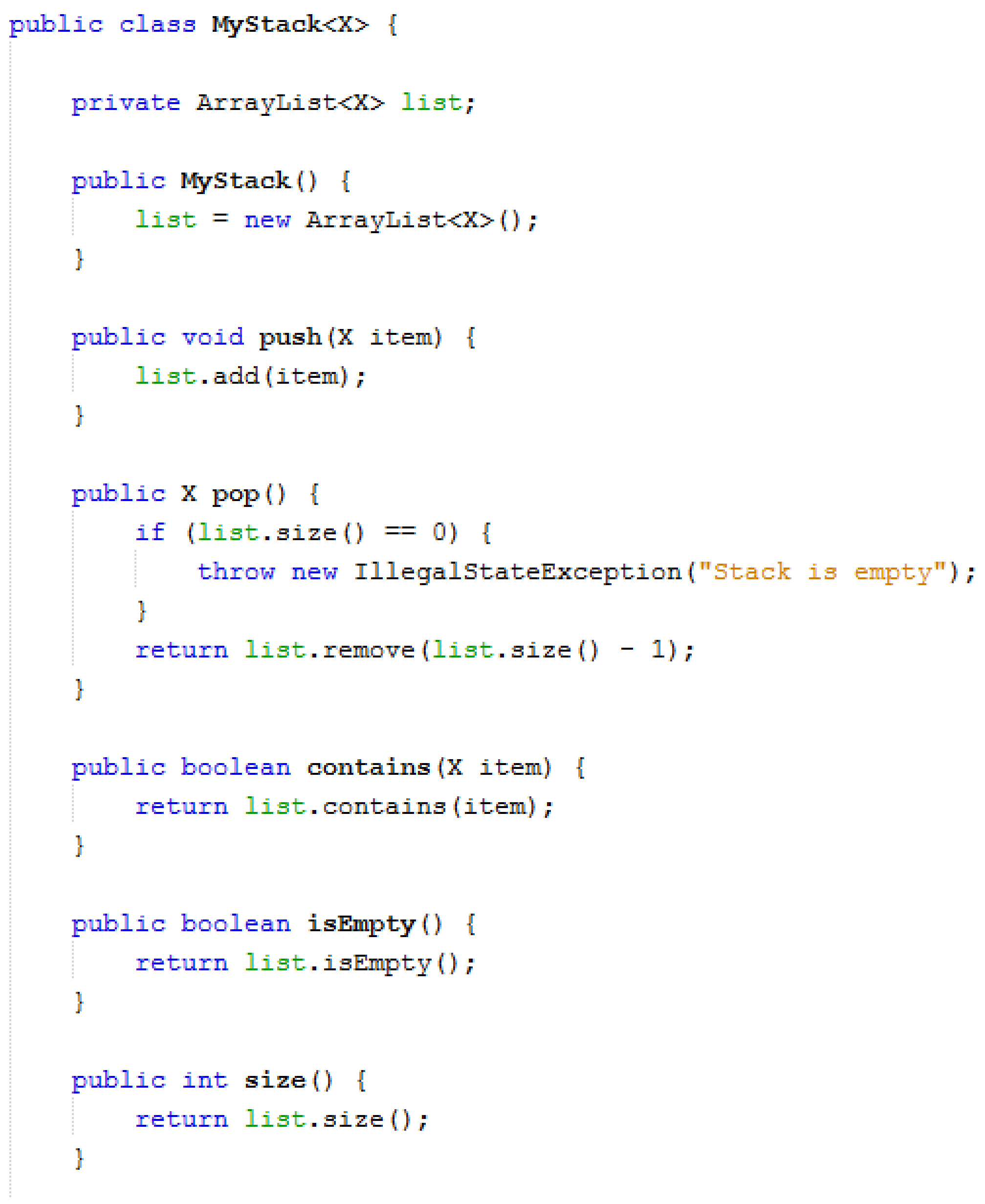


- A stack holds data in a last-in, first-out fashion.

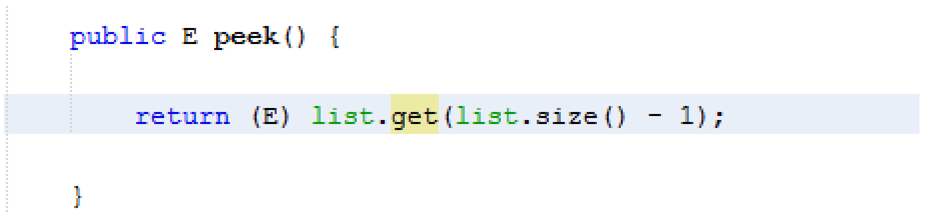
Implementation:



```
+Stack()                   Creates an empty stack.
+empty(): boolean          Returns true if this stack is empty.
+peek(): E                 Returns the top element in this stack.
+pop(): E                  Returns and removes the top element in this stack.
+push(o: E): E             Adds a new element to the top of this stack.
+search(o: Object): int    Returns the position of the specified element in this stack.
```



```java
public class MyStack<X> {

    private ArrayList<X> list;

    public MyStack() {
        list = new ArrayList<X>();
    }

    public void push(X item) {
        list.add(item);
    }

    public X pop() {
        if (list.size() == 0) {
            throw new IllegalStateException("Stack is empty");
        }
        return list.remove(list.size() - 1);
    }

    public boolean contains(X item) {
        return list.contains(item);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int size() {
        return list.size();
    }
```

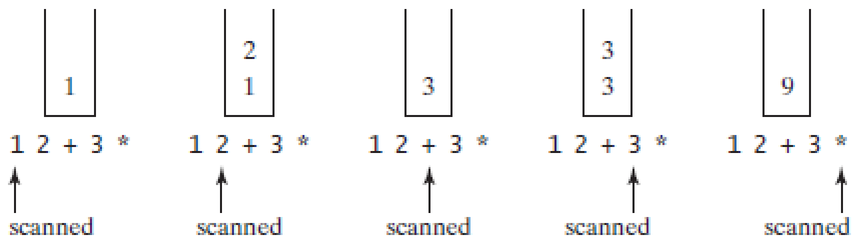


```java
    public E peek() {

        return (E) list.get(list.size() - 1);

    }
```

Suggested Exercises:

1.

(*Postfix notation*) Postfix notation is a way of writing expressions without using parentheses. For example, the expression `(1 + 2) * 3` would be written as `1 2 + 3 *`. A postfix expression is evaluated using a stack. Scan a postfix expression from left to right. A variable or constant is pushed into the stack. When an operator is encountered, apply the operator with the top two operands in the stack and replace the two operands with the result. The following diagram shows how to evaluate `1 2 + 3 *`.



Write a program to evaluate postfix expressions. Pass the expression as a command-line argument in one string.

2.

(*Convert infix to postfix*) Write a method that converts an infix expression into a postfix expression using the following header:

```
public static String infixToPostfix(String expression)
```

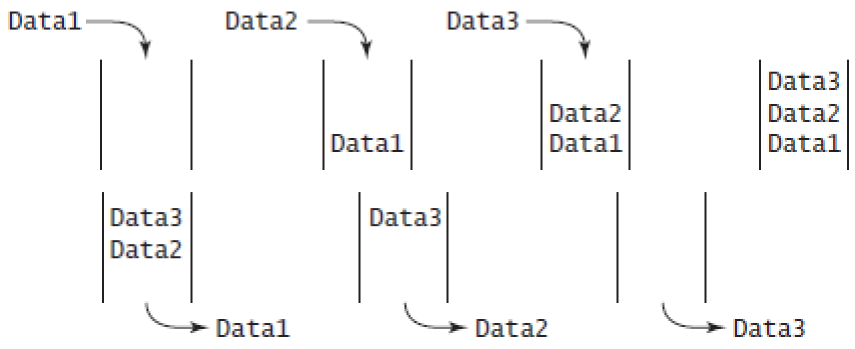
For example, the method should convert the infix expression `(1 + 2) * 3` to `1 2 + 3 *` and `2 * (1 + 3)` to `2 1 3 + *`.

Experiment No. 6

Queue

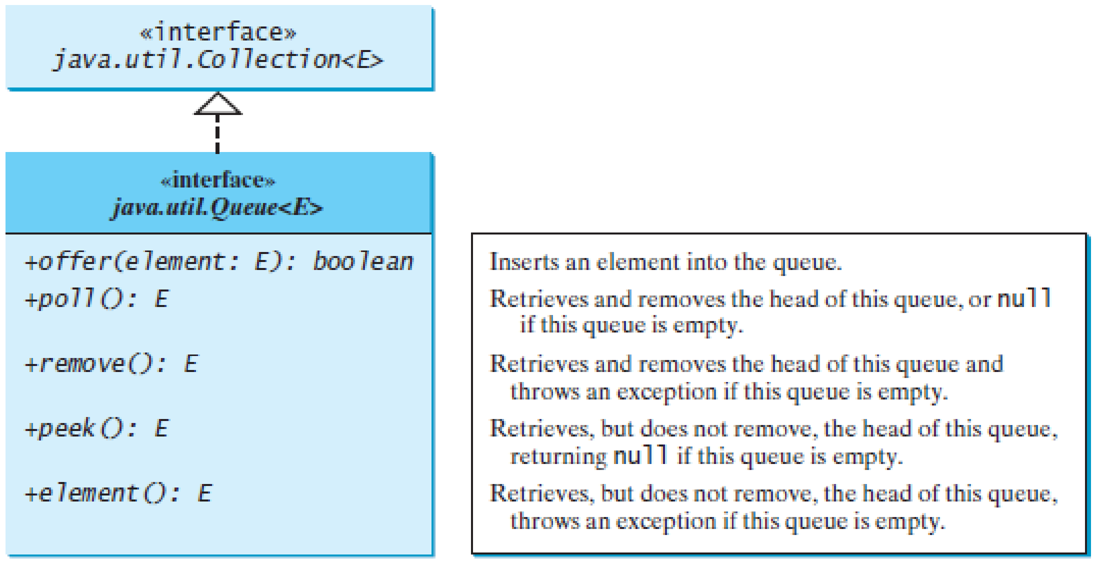
Description:

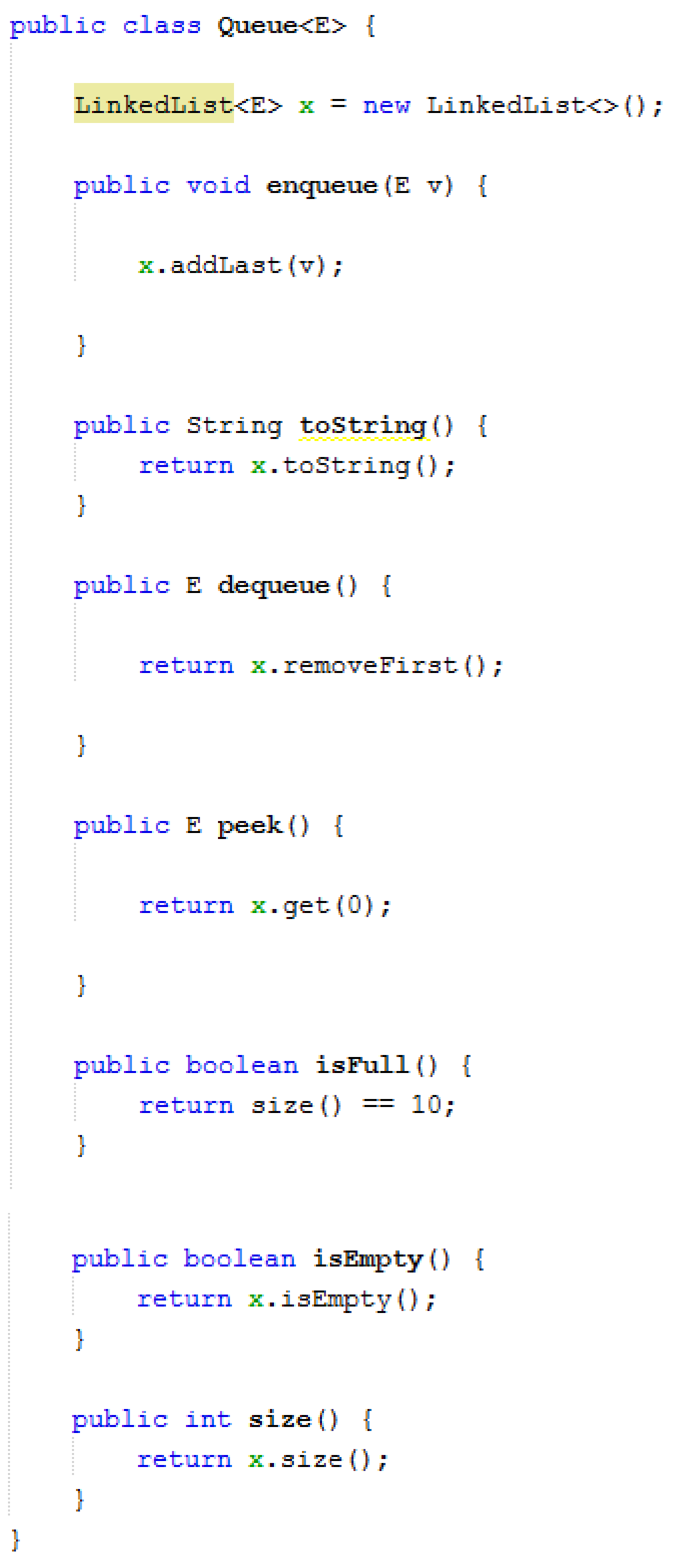
1. A queue represents a waiting list. It can be viewed as a special type of list whose elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head), as shown in the following figure.



2. A queue holds objects in a first-in, first-out fashion.

Implementation:


«interface»
java.util.Collection<E>
«interface»
java.util.Queue<E>
+offer(element: E): boolean
+poll(): E
+remove(): E
+peek(): E
+element(): E
Inserts an element into the queue.
Retrieves and removes the head of this queue, or null if this queue is empty.
Retrieves and removes the head of this queue and throws an exception if this queue is empty.
Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
Retrieves, but does not remove, the head of this queue, throws an exception if this queue is empty.



```java
public class Queue<E> {

    LinkedList<E> x = new LinkedList<>();

    public void enqueue(E v) {

        x.addLast(v);

    }

    public String toString() {
        return x.toString();
    }

    public E dequeue() {

        return x.removeFirst();

    }

    public E peek() {

        return x.get(0);

    }

    public boolean isFull() {
        return size() == 10;
    }


    public boolean isEmpty() {
        return x.isEmpty();
    }

    public int size() {
        return x.size();
    }
}
```

<u>Suggested Exercises:</u>

1. We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.

2. The problem is opposite of the previous one. We are given a Queue data structure that supports standard operations like enqueue () and dequeue (). We need to implement a Stack data structure using only instances of Queue and queue operations allowed on the instances.

3. Reversing a Queue : implement a method for reversing a queue Q

```
Input: [10, 22, 13, 40, 2]

Output: [2, 40, 13, 22, 10]
```

Experiment No. 7

Recursion

Description:

Recursion is a technique that leads to elegant solutions to problems that are difficult to program using simple loops. A method is recursive if it calls itself to accomplish its work. A recursive method is defined in terms of a smaller instance of itself. In the recursive method we have two cases a basic case which can be solved without recursion and a general case which solved by recursion, so there are two basic rules in recursion:

1. Base case: simplest case, always have at least one case that can be solved without using recursion.
2. Make progress: Any recursive call (general case) must progress toward a base case.

Examples:

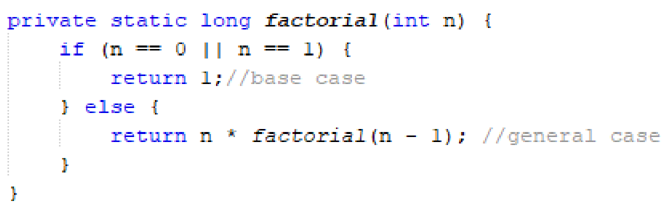
a) Factorial

The factorial of any positive integer is the product of all the numbers between 1 and the number.

$$n! = n \times n - 1 \times n - 2 \times \cdots \times 3 \times 2 \times 1$$

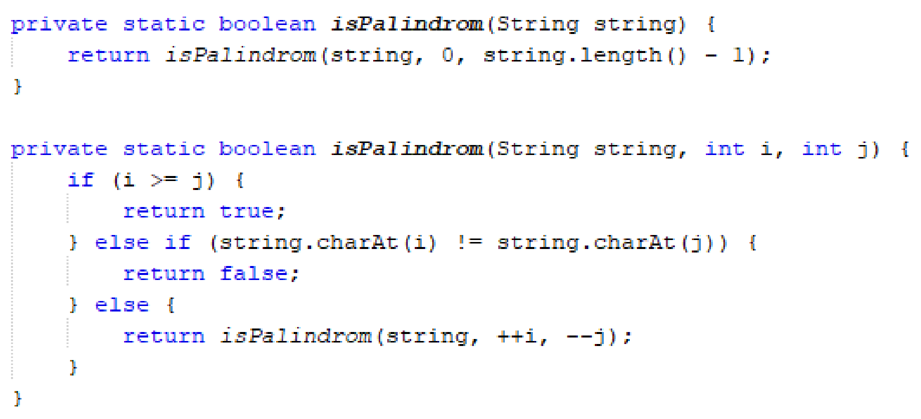
$$4! = 4 \times \underbrace{3 \times 2 \times 1}_{3!}$$
$$\downarrow$$
$$3! = 3 \times \underbrace{2 \times 1}_{2!}$$
$$\downarrow$$
$$2! = 2 \times \underbrace{1}_{1!}$$
$$\downarrow$$
$$1$$

As we seen $4! = 4 \times 3!$ and so on. In addition, the factorial of 1 is known and doesn't need to be calculated it equals 1. We conclude from this that the base case is 1 and the general case equals to the product of the number with the factorial of the number-1.



```
private static long factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;//base case
    } else {
        return n * factorial(n - 1); //general case
    }
}
```

b) Checking whether a string is a palindrome



```java
private static boolean isPalindrom(String string) {
    return isPalindrom(string, 0, string.length() - 1);
}

private static boolean isPalindrom(String string, int i, int j) {
    if (i >= j) {
        return true;
    } else if (string.charAt(i) != string.charAt(j)) {
        return false;
    } else {
        return isPalindrom(string, ++i, --j);
    }
}
```

In the following experiment we will see how the recursion facilitate some processes in the data structures and learn how to use it to solve the problems.

<u>Suggested Exercises:</u>

Write the following recursive methods:

1) A method to compute 2^n for a positive integer n.
2) A method to compute x^n for a positive integer n.
3) A method to reverse a string.
4) A method to reverse array.
5) A method to compute the GCD (greatest common divisor).
6) A recursive binary search method.
7) A method to compute the following series: $m(i) = 1 + \frac{1}{2} + \frac{1}{3} + \ ... \ + \frac{1}{i}$
8) A method to find the occurrence of a specific character in a string.

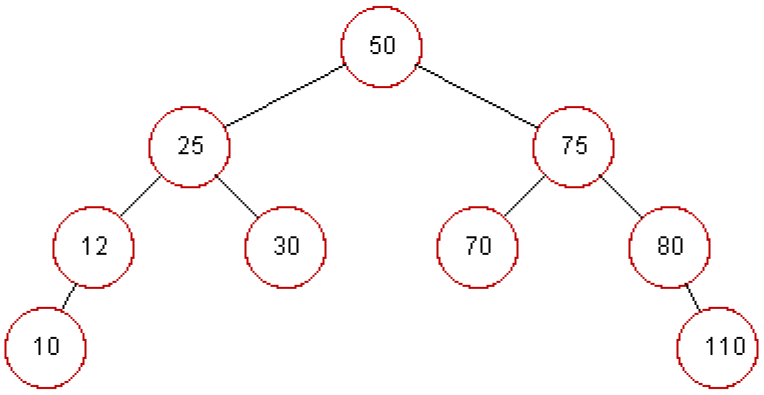
Experiment No. 8

Binary Search Tree (BST)

Description:

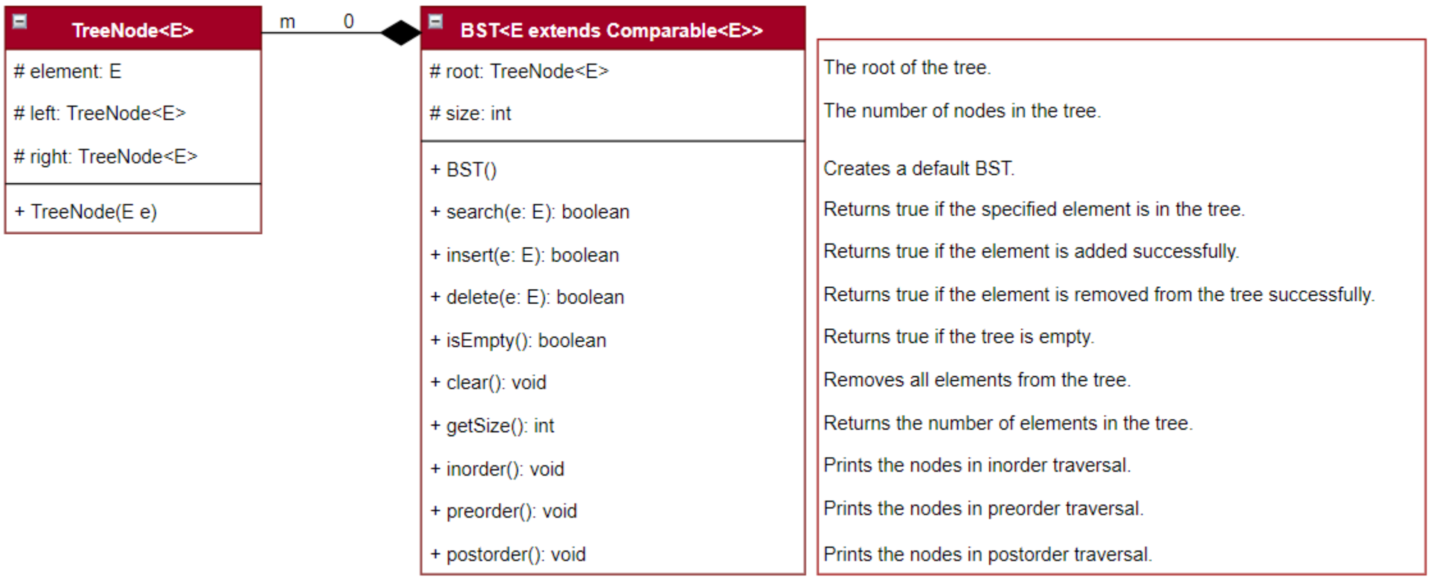
A tree provides a hierarchical organization in which data are stored in the nodes. A binary tree is a hierarchical structure. It either is empty or consists of an element, called the root, and two distinct binary trees, called the left subtree and right subtree, either or both of which may be empty, if both are empty the node is called leaf.

A BST is special type of binary tree has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node.



Implementation:

In this lab, BST class will be implemented in addition to different operation on BST. A binary search tree can be represented using a set of linked nodes. First, class TreeNode is implemented, where each node from this class holds an element in the tree and two links named left and right that reference the left child and right child, respectively.



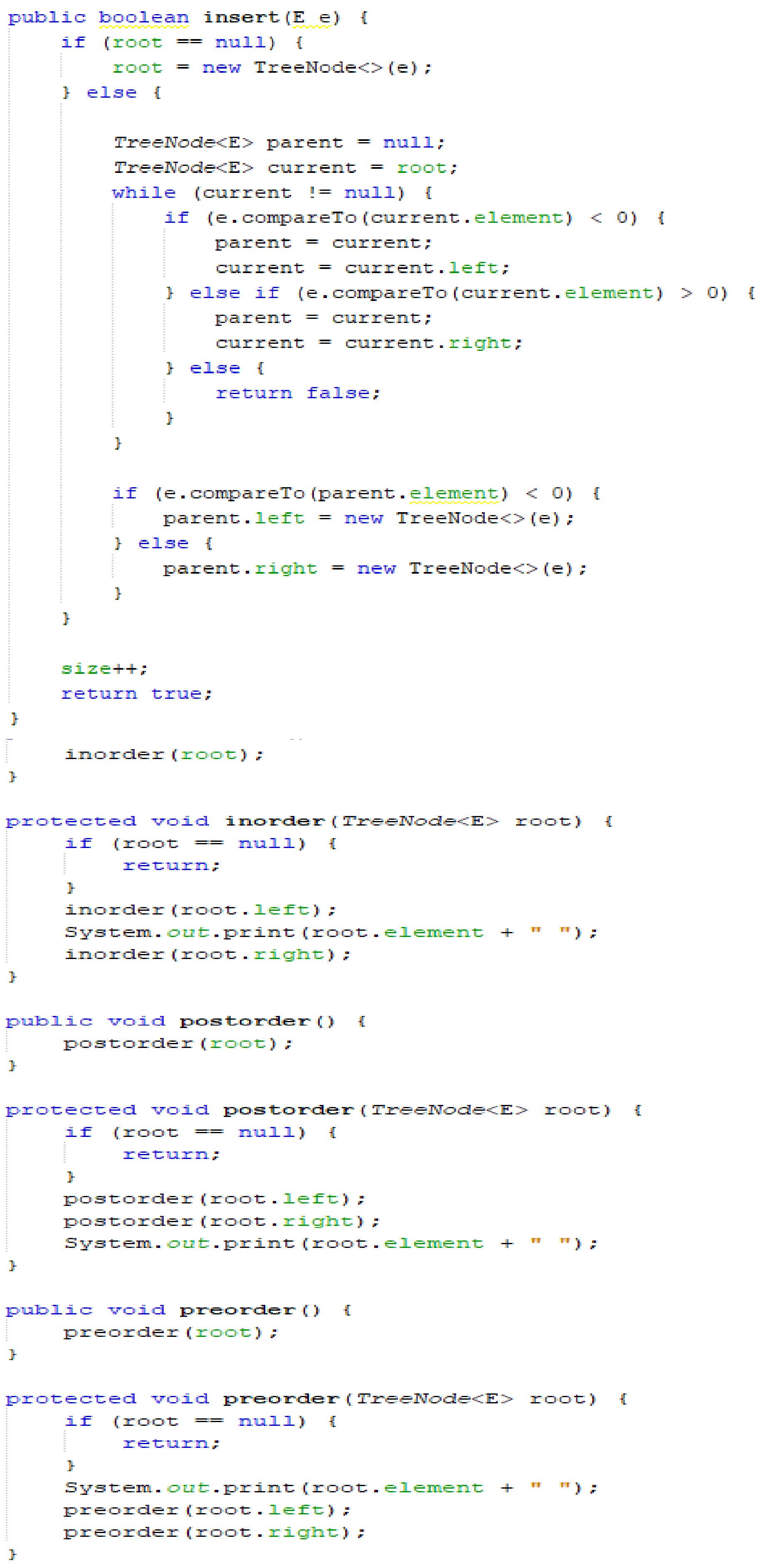
# root: TreeNode<E>	The root of the tree.
# size: int	The number of nodes in the tree.
+ BST()	Creates a default BST.
+ search(e: E): boolean	Returns true if the specified element is in the tree.
+ insert(e: E): boolean	Returns true if the element is added successfully.
+ delete(e: E): boolean	Returns true if the element is removed from the tree successfully.
+ isEmpty(): boolean	Returns true if the tree is empty.
+ clear(): void	Removes all elements from the tree.
+ getSize(): int	Returns the number of elements in the tree.
+ inorder(): void	Prints the nodes in inorder traversal.
+ preorder(): void	Prints the nodes in preorder traversal.
+ postorder(): void	Prints the nodes in postorder traversal.

a) TreeNode class implementation:

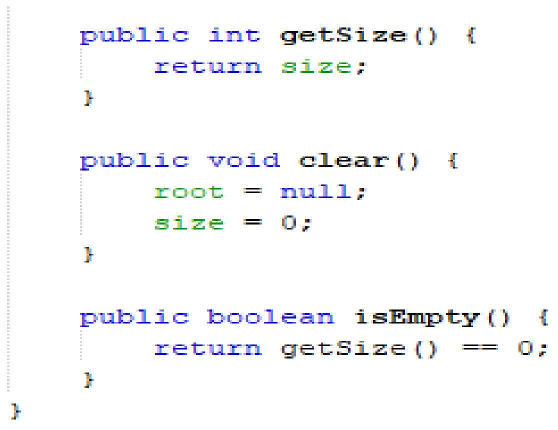
```
class TreeNode<E> {

    protected E element;
    protected TreeNode<E> left;
    protected TreeNode<E> right;

    public TreeNode(E e) {
        element = e;
    }
}
```

b) BST class implementation:

```
public class BST<E extends Comparable<E>> {

    protected TreeNode<E> root;
    protected int size = 0;

     /**
     * Returns true if the element is in the tree
     */
    public boolean search(E e) {
        TreeNode<E> current = root; // Start from the root

        while (current != null) {
            if (e.compareTo(current.element) < 0) {
                current = current.left;
            } else if (e.compareTo(current.element) > 0) {
                current = current.right;
            } else // element matches current.element
            {
                return true; // Element is found
            }
        }

        return false;
    }
```



```
public boolean insert(E e) {
    if (root == null) {
        root = new TreeNode<>(e);
    } else {

        TreeNode<E> parent = null;
        TreeNode<E> current = root;
        while (current != null) {
            if (e.compareTo(current.element) < 0) {
                parent = current;
                current = current.left;
            } else if (e.compareTo(current.element) > 0) {
                parent = current;
                current = current.right;
            } else {
                return false;
            }
        }

        if (e.compareTo(parent.element) < 0) {
            parent.left = new TreeNode<>(e);
        } else {
            parent.right = new TreeNode<>(e);
        }
    }

    size++;
    return true;
}
    inorder(root);
}

protected void inorder(TreeNode<E> root) {
    if (root == null) {
        return;
    }
    inorder(root.left);
    System.out.print(root.element + " ");
    inorder(root.right);
}

public void postorder() {
    postorder(root);
}

protected void postorder(TreeNode<E> root) {
    if (root == null) {
        return;
    }
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.element + " ");
}

public void preorder() {
    preorder(root);
}

protected void preorder(TreeNode<E> root) {
    if (root == null) {
        return;
    }
    System.out.print(root.element + " ");
    preorder(root.left);
    preorder(root.right);
}
```



```
    public int getSize() {
        return size;
    }

    public void clear() {
        root = null;
        size = 0;
    }

    public boolean isEmpty() {
        return getSize() == 0;
    }
}
```

Suggested Exercises:

1) Implement the method delete in BST class.
2) Implement methods to find the height, number of levels, number of leave nodes, number of internal nodes and total number of nodes in BST
3) Implement the method pathToNode.
4) Implement the method BFT (Breadth First Traversal).
5) Implement the methods isFull to check if BST is full.
6) Implement the method isLeaf.
7) Implement recursive methods to find the height, number of levels, number of leave nodes, number of internal nodes and total number of nodes in BST.
8) Write a GUI enables the user to create and deal with BST.

Experiment No. 9

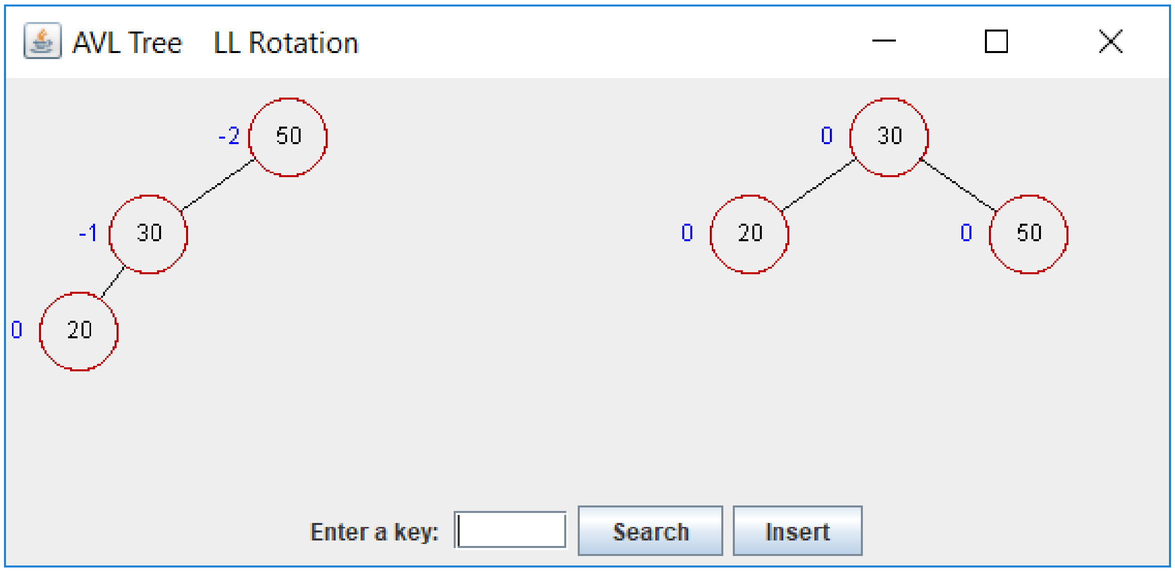
AVL Tree

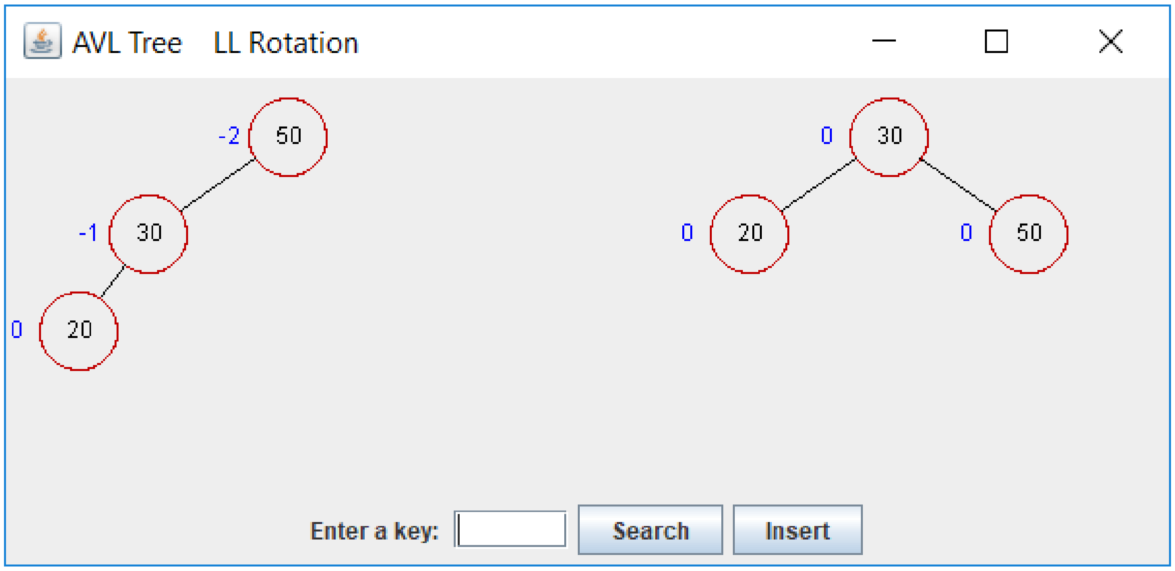
<u>Description:</u>

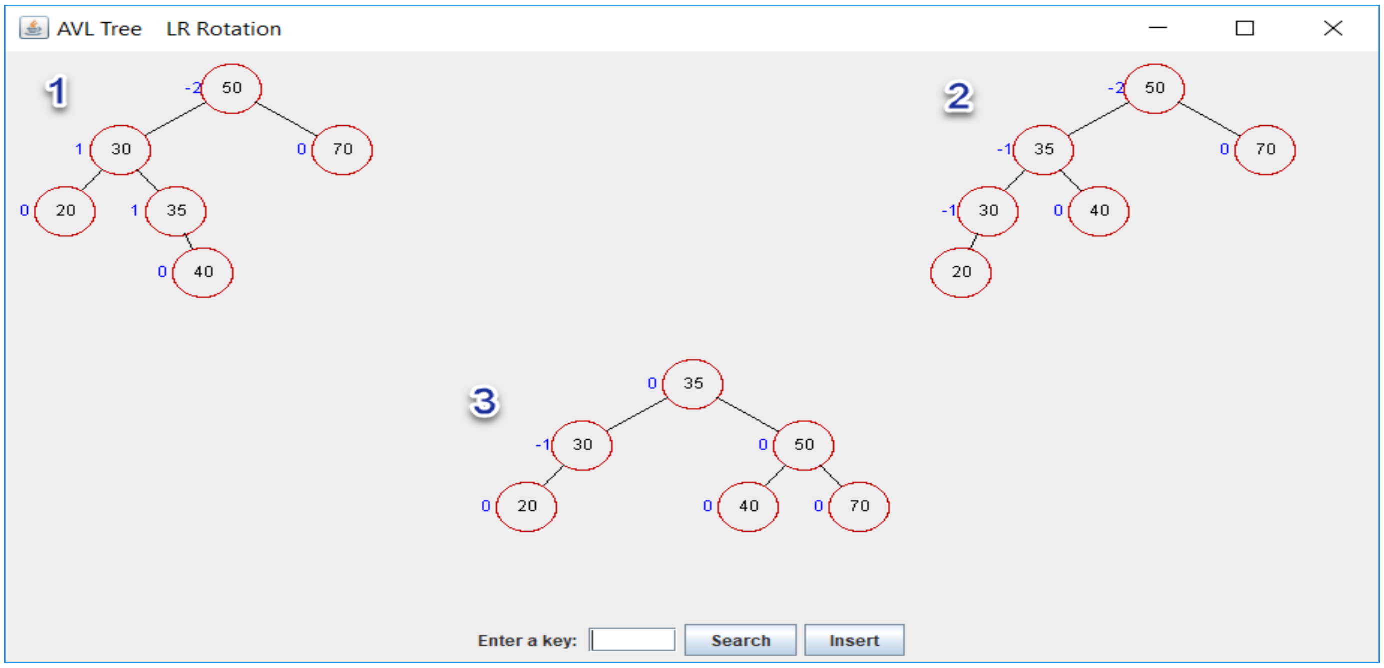
AVL Tree is a balanced binary search tree. the difference between the heights of every node’s two subtrees is 0 or 1. It can be shown that the maximum height of an AVL tree is O(log n).
The process for inserting or deleting an element in an AVL tree is the same as in a regular binary search tree, except that you may have to rebalance the tree after an insertion or deletion operation. The balance factor (BF) of a node is the height of its right subtree minus the height of its left subtree. A node is said to be balanced if its balance factor is -1, 0, or 1. A node is considered left-heavy if its balance factor is -1, and right-heavy if its balance factor is +1.
If a node is not balanced after an insertion or deletion operation, you need to rebalance it. The process of rebalancing a node is called rotation.
There are four possible rotations:

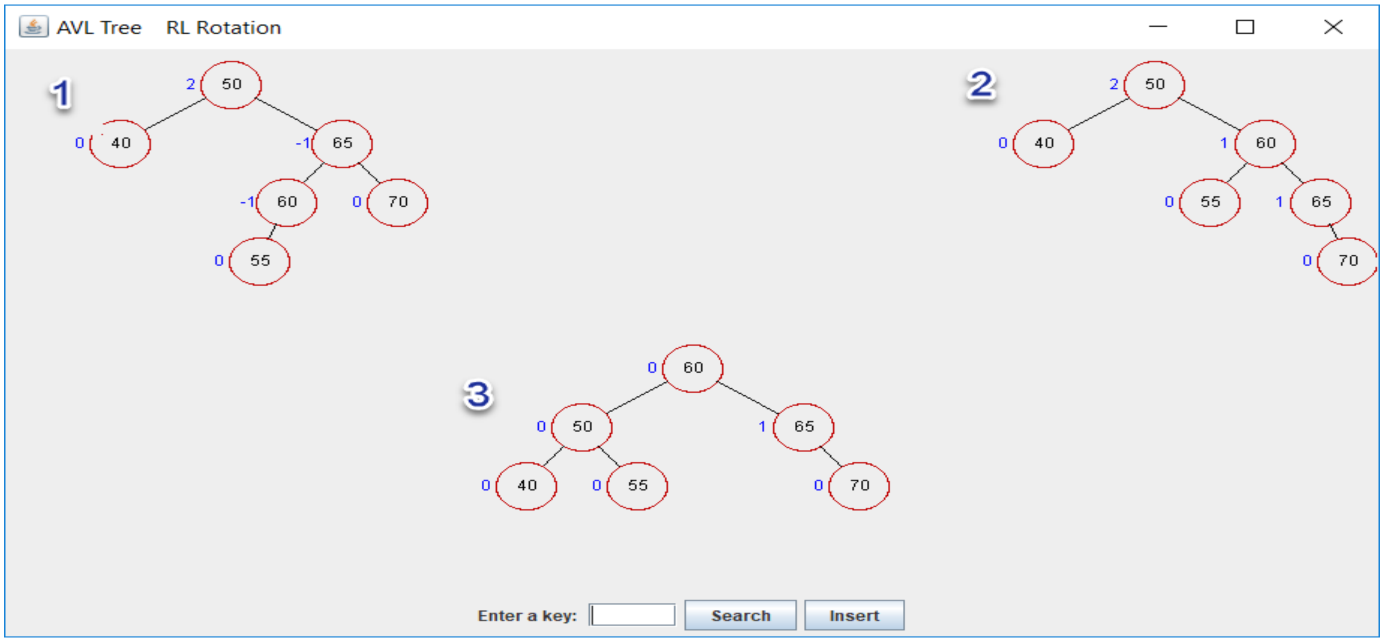
- LL: single right rotation around the imbalanced node (if BF of the imbalanced node = -2 and the BF of the left child of the imbalanced node=0 or -1).
- RR: single left rotation around the imbalanced node (if BF of the imbalanced node = +2 and the BF of the left child of the imbalanced node=0 or +1).
- LR: (double rotation) left rotation around the left child of the imbalanced node then right rotation around the imbalanced node (if BF of the imbalanced node = -2 and the BF of the left child of the imbalanced node =+1).
- RL: (double rotation) right rotation around the right child of the imbalanced node then left rotation around the imbalanced node (if BF of the imbalanced node = +2 and the BF of the left child of the imbalanced node= -1).

In the following figures example for each case is presented









Suggested Exercises:

1) Implement the AVL Tree class by extending the BST class implemented in the previous experiment.
2) Implement the methods balanceFactor, rotateRight, rotateLeft.
3) Override insert and delete methods.