ADSJ

wildcards in generics

```
#upperBounds
import java.util.*;
public class Main
{
   public static double sumOfList(List<? extends Number> list){
      double sum=0.0;
      for(Number n:list){
         sum+=n.doubleValue();
      }
      return sum;
   }
      public static void main(String[] args) {
            List<Integer> al=Arrays.asList(1,2,3,4);
            List<Double> bl=Arrays.asList(1.1,2.2,3.3,4.4);
            System.out.println("Sum of Integers: "+sumOfList(al));
            System.out.println("Sum of Doubles: "+sumOfList(bl));
      }
}

#lowerBounds
import java.util.*;
public class Main
{
   public static void addNumbers(List<? super Integer> list) {
      for (int i = 1; i <= 5; i++) {
         list.add(i);
      }
   }

   public static void main(String[] args) {
      List<Number> numberList = new ArrayList<>();
      addNumbers(numberList);
      System.out.println("Number list after adding integers: " + numberList);

      List<Object> objectList = new ArrayList<>();
      addNumbers(objectList);
      System.out.println("Object list after adding integers: " + objectList);
   }
}
```

LAMBDA

#1

```
interface Demo
{
   void say(String message);
}
public class Main{
   public static void main(String[] args)
   {
      Demo d=(name)->System.out.println("Hello "+name);

      d.say("Riki");
   }
}
```

#2

```
import java.util.*;
public class Main{
   public static void main(String[] args)
   {
      List<String> name=Arrays.asList("pikachu","Charizard","mew","mewtwo");

      name.forEach(names-> System.out.println(names));

      //method reference
      name.forEach(System.out::println);
   }
}
```

#3

```
import java.util.*;
public class Main{
   public static void main(String[] args)
   {
     List<String> pok=Arrays.asList("Pika","Charm","Blast","squi","mew");
     Collections.sort(pok,(a,b)->a.compareTo(b));
     System.out.println(pok);
   }
}
```

#4

```
import java.util.*;
import java.util.stream.*;
public class Main{
   public static void main(String[] args)
   {
     List<String>
pok=Arrays.asList("Chika","Pika","Charm","Blast","squi","mew");

      List<String> filtered = pok.stream().filter(name-
>name.startsWith("C")).collect(Collectors.toList());

      System.out.println(filtered);
   }
}
```

COLLECTIONS

Stack and queues

#1

```
import java.util.LinkedList;

public class Main<T>{
   private LinkedList<T> list=new LinkedList<>();
   public void push(T ele)
   {
      list.addFirst(ele);
   }
   public T pop()
   {
      return list.removeFirst();
   }
   public T peek(){
      return list.getFirst();
   }
   public boolean isEmpty(){
      return list.isEmpty();
   }
   public int size(){
      return list.size();
   }
   public static void main(String[] args)
   {
      Main<Integer> stack=new Main<>();
```

```
        stack.push(1);
        stack.push(3);
        stack.push(9);
        System.out.println("Stack size "+stack.size());
        System.out.println("Top Element "+stack.peek());
        System.out.println("Popped Element "+stack.pop());

    }
}
```

#2

```
import java.util.LinkedList;

public class Main<T>{
    private LinkedList<T> list=new LinkedList<>();
    public void enqueue(T ele)
    {
        list.addLast(ele);
    }
    public T dequeue()
    {
        return list.removeFirst();
    }
    public T peek(){
        return list.getFirst();
    }
    public boolean isEmpty(){
        return list.isEmpty();
    }
    public int size(){
        return list.size();
    }
    public static void main(String[] args)
    {
        Main<Integer> q=new Main<>();
        q.enqueue(1);
        q.enqueue(3);
        q.enqueue(9);
        System.out.println("Queue size "+q.size());
        System.out.println("Top Element "+q.peek());
        System.out.println("Popped Element "+q.dequeue());

    }
}
```

#3

```
import java.util.*;
public class Main<T>{
   private ArrayList<T> list=new ArrayList<>();
   public void push(T ele){
      list.add(ele);
   }
   public T pop(){
      return list.remove(list.size()-1);
   }
   public T peek(){
      return list.get(list.size()-1);
   }
   public boolean isEmpty(){
      return list.isEmpty();
   }

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

   public static void main(String[] args)
   {
      Main<String> stack=new Main<>();
      stack.push("5");
      stack.push("2");
      stack.push("9");
      System.out.println("Stack size: " + stack.size());
      System.out.println("Top element: " + stack.peek());
      System.out.println("Popped element: " + stack.pop());
      System.out.println("Stack size after pop: " + stack.size());
   }
}
```

#4

```
import java.util.*;
public class Main<T>{
   private ArrayList<T> list=new ArrayList<>();
   public void enqueue(T ele){
      list.add(ele);
   }
   public T dequeue(){
      return list.remove(0);
   }
```

```
    public T peek(){
        return list.get(0);
    }
    public boolean isEmpty(){
        return list.isEmpty();
    }

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

    public static void main(String[] args)
    {
        Main<String> q=new Main<>();
        q.enqueue("5");
        q.enqueue("2");
        q.enqueue("9");
        System.out.println("Queue size: " + q.size());
        System.out.println("Top element: " + q.peek());
        System.out.println("Popped element: " + q.dequeue());
        System.out.println("Stack size after pop: " + q.size());
    }
}
```

STREAM API

```
import java.util.stream.*;
import java.util.*;
public class Main{
    public static void main(String[] args)
    {
        List<String>
names=Arrays.asList("John","Pika","Char","raich","squi","chika","char");

        names.stream().filter(name-
>name.startsWith("J")).forEach(System.out::println);

        names.stream().map(String::toUpperCase).forEach(System.out::println);

        names.stream().distinct().forEach(System.out::println);

        names.stream().sorted().forEach(System.out::println);

        int sum = Stream.of(1,2,3,4,5).reduce(0,Integer::sum);
        System.out.println(sum);

        Integer[] num=Stream.of(1,2,3,4,5).toArray(Integer[]::new);
```

```
        System.out.println("Arrays: "+Arrays.toString(num));

        List<Integer> nums=Stream.of(1,2,3,4,5).collect(Collectors.toList());
        System.out.println("List: "+nums);

        Set<Integer> set=Stream.of(1,2,3,4,5,5).collect(Collectors.toSet());
        System.out.println("Set: "+set);

        Map<Integer,String> map=Stream.of(1,2,3).collect(Collectors.toMap(i->i,i->"Value"+i));
        System.out.println("Map: "+map);

        String result=Stream.of("Hello","World","Riki").collect(Collectors.joining(", "));
        System.out.println("String:"+result);

    }
}



//LP
import java.util.*;
class Pair<K,V>{
K key;
V value;
Pair(K key,V value){
this.key=key;
this.value=value;
}
public String toString(){
return this.key+" "+this.value;
}
}
class LP<K,V>{
Pair<K,V> htable[];
boolean bit[];
int sz;
LP(int n){
this.sz=n;
htable=new Pair[sz];
bit=new boolean[sz];
for(int i=0;i<sz;i++)
{
htable[i]=null;
bit[i]=true;
}
```

```
}
void insert(Pair<K,V> entry)
{ int home=hash(entry.key);
int i=probe(home);
if(i!=-1){
htable[i]=entry;
bit[i]=false;
}
else
System.out.println("Hash table is full, so insertion is not possible");
}
int hash(K key){
return key.hashCode()%sz;
}
int probe(int home){
int i=home;
do{
if(htable[i]==null)
return i;
i=(i+1)%sz;
}while(i!=home);
return -1;
}
void display(){
if(isEmpty()){
System.out.println("HashTable is empty");
}
else{
System.out.println("Hashtable entries are" );
for(int i=0;i<sz;i++){
if(htable[i]==null)
System.out.println(i+" NULL");
else
System.out.println(i+" "+htable[i]);
}
}
}
boolean isEmpty(){
boolean flag=true;
for(int i=0;i<sz;i++)
if(htable[i]!=null)
flag=false;
return flag;
}
void delete(K key){
int home=hash(key);
int i=home;
```

```
do{
if(htable[i]!=null && htable[i].key.equals(key)){ System.out.println("Entry going
to be deleted is "+htable[i]);
htable[i]=null;
return;
}
else{
i=(i+1)%sz;
}
}while(bit[i]!=true && i!=home);
System.out.println("Key not found");
}
void find(K key){
int home=hash(key);
int i=home;
do{
if(htable[i]!=null && htable[i].key.equals(key)){
System.out.println("Entry found at "+i+"th position");
System.out.println("and it is "+htable[i]);
return;
}
else{
i=(i+1)%sz;
}
}while(bit[i]!=true && i!=home);
System.out.println("Key not found");
}
}
public class LPDriver{
public static void main(String[] args){
LP<Integer,String> lp=new LP<>(5);
Pair<Integer,String> entry1=new Pair<>(529,"Renu");
lp.insert(entry1);
Pair<Integer,String> entry2=new Pair<>(575,"Riki");
lp.insert(entry2);
Pair<Integer,String> entry3=new Pair<>(597,"Zoro");
lp.insert(entry3);
lp.insert(new Pair<>(577,"Nami"));
lp.display();
lp.delete(597);
lp.display();
}
}
```

```
//QP
import java.util.*;
class Pair<K,V>
{
K key;
V value;
Pair(K key,V value){
this.key=key; this.value=value;
}
public String toString(){
return this.key+" "+this.value;
}
}
class QP<K,V>{
int sz;
Pair<K,V>[] htable;
boolean[] bit;
QP(int n)
{
this.sz=n;
htable=new Pair[sz];
bit=new boolean[sz];
for(int i=0;i<sz;i++)
{
htable[i]=null;
bit[i]=true;
}
}
void insert(Pair<K,V> entry)
{
int home=hash(entry.key);
int i=probe(home);
if(i!=-1){
htable[i]=entry;
bit[i]=false;
}
else{
System.out.println("Hash table is full,cannot insert elements");
}
}
int hash(K key){
return key.hashCode()%sz;
}
int probe(int home)
{
int i=0;
```

```
int pos; do{
pos=(home+i*i)%sz;
if(htable[pos]==null){
return pos;
}
i++;
}while(i<sz);
return -1;
}
void display(){
if(isEmpty())
{
System.out.println("Hash TAble is emptty");
}
else{
System.out.println("the entries in the hashtable are:\n");
for(int i=0;i<sz;i++){
if(htable[i]==null)
System.out.println(i+" NULL");
else
System.out.println(i+ " "+htable[i]);
}
}
}
boolean isEmpty(){
boolean flag=true;
for(int i=0;i<sz;i++)
if(htable[i]!=null)
flag=false;
return flag;
}
void delete(K key)
{
int home=hash(key);
int i=home;
do{
if(htable[i]!=null && htable[i].key.equals(key)){
System.out.println("Entry is goging to be deleted is "+htable[i]);
htable[i]=null;
return;
}
else{
i=(i+1)%sz;
}
}while(bit[i]!=true && i!=home);
System.out.println("Key not found");
}void find(K key){
```

```
int home=hash(key);
int i=home;
do{
if(htable[i]!=null && htable[i].key.equals(key)){
System.out.println("Entry found at "+i+"th position");
System.out.println("and it is "+htable[i]);
return;
}
else{
i=(i+1)%sz;
}
}while(bit[i]!=true && i!=home);
System.out.println("Key not found");
}
}
public class QPDriver{
public static void main(String[] args){
QP<Integer,String> lp=new QP<>(5);
Pair<Integer,String> entry1=new Pair<>(529,"Renu");
lp.insert(entry1);
Pair<Integer,String> entry2=new Pair<>(575,"Riki");
lp.insert(entry2);
Pair<Integer,String> entry3=new Pair<>(597,"Zoro");
lp.insert(entry3);
lp.insert(new Pair<>(579,"Nami"));
lp.display();
lp.delete(597);
lp.display();
}
}
```

//seperate chain

```
import java.util.*;
class Pair<K extends Comparable<K>,V>{
	K key;
	V value;
	Pair(K key , V value){
		this.key=key;
		this.value=value;
	}
	public String toString(){
		return this.key+" "+this.value;
	}
```

```
}
class PairNode<K extends Comparable<K>,V>{
	Pair<K,V> data;
	PairNode<K,V> next;
	PairNode(Pair<K,V> data, PairNode<K,V> next){
		this.data=data;
		this.next=next;
	}
}
class Seperatechain<K extends Comparable<K>,V>{
	PairNode<K, V> sctable[];
	PairNode<K,V> npnode, temp;
	int tsize;
	int noelements;
	Seperatechain(int n){
		tsize=n;
		noelements=0;
		sctable=new PairNode[tsize];
		for(int i=0;i<tsize;i++)
			sctable[i]=null;
	}
	int hash(K key){
		return key.hashCode()%tsize;
	}
	void insert(Pair<K,V> data){
		npnode=new PairNode<>(data, null);
		int home=hash(data.key);//0
		if(sctable[home]==null)
			sctable[home]=npnode;
		else{
			temp=sctable[home];
			while(temp.next!=null){
				temp=temp.next;
			}
			temp.next=npnode;
		}
		noelements++;
	}
	void display(){
		if(isEmpty()){
			System.out.println("HASH TABLE IS EMPTY");
			return;
		}
		else{
			System.out.println("elements of hash table are:");
			for(int i=0;i<tsize;i++)
			{
```

```
                if(sctable[i]==null)
                        System.out.println(i +" NULL");
                else{
                        temp=sctable[i];
                        System.out.print(i +" NOT NULL ");
                        while(temp!=null){
                                System.out.print(" -> "+ temp.data );
                                temp=temp.next;
                        }
                        System.out.println();
                }
        }
    }
}
void delete(K key){
    if(isEmpty()){
        System.out.println("Given data is not present in Hash Table");
        return ;
    }
    int home=hash(key);
    PairNode<K,V> cur=sctable[home];
    PairNode<K,V> prev=null;
    while(cur!=null && cur.data.key.compareTo(key)!=0){
        prev=cur;
        cur=cur.next;
    }
    if(cur!=null){
        if(cur==sctable[home])
            sctable[home]=cur.next;
        else
            prev.next=cur.next;
        System.out.println("Deleted data is "+ cur.data);
     }
    else
        System.out.println("Given data is not present in Hash Table");
}
void find(K key){
    if(isEmpty()){
        System.out.println("Given data is not present in Hash Table");
        return ;
    }
    int home=hash(key);
    PairNode<K,V> cur=sctable[home];
    while(cur!=null && cur.data.key.compareTo(key)!=0){
        cur=cur.next;
    }
    if(cur!=null){
```

```
                System.out.println("Entry found and it is "+ cur.data);
            }
            else
                System.out.println("Given data is not present in Hash Table");
        }
        boolean isEmpty(){
            boolean flag=true;
            for(int i=0;i<tsize;i++)
                if(sctable[i]!=null)
                    flag=false;
            return flag;
        }
}
class SeperatechainDriver{
        public static void main(String args[]){
            Scanner sc=new Scanner(System.in);
            Seperatechain<Integer,String> sch=new Seperatechain<>(8);
            Pair<Integer,String> entry;
            int ch;
            Integer key;
            String value;
            System.out.println("\n1.Insert\n2.Delete\n3.Find\n4.Display\n5.Exit");
            do{
                System.out.print("Enter your choice :");
                ch=sc.nextInt();
                switch(ch){
                    case 1: System.out.print("Enter key:");
                            key=sc.nextInt();
                            System.out.print("Enter value:");
                            sc.nextLine();
                            value=sc.nextLine();
                            entry=new Pair<>(key, value);
                            sch.insert(entry);
                            break;
                    case 2: System.out.print("Enter key:");
                            key=sc.nextInt();
                            sch.delete(key);
                            break;
                    case 3: System.out.print("Enter key:");
                            key=sc.nextInt();
                            sch.find(key);
                            break;
                    case 4: sch.display();
                            break;
                    case 5: System.exit(0);
                    default: System.out.println("Invalid choice");
                }
```

```
            }
            while(ch!=5);
        }
}
```

BST

```
import java.util.*;
public class Bst{
    class Node{
        int key;
        Node left,right;
        Node(int data)
        {
            key=data;
            left=null;
            right=null;
        }
    }
    private Node root;
    public Bst(){
        root=null;
    }
    public void insert(int data)
    {
        root=insertRec(root,data);
    }
    public Node insertRec(Node root,int data)
    {
        if(root==null)
        {
            root=new Node(data);
            return root;
        }
        else if(data<root.key)
        {
            root.left= insertRec(root.left,data);
        }
        else
            root.right=insertRec(root.right,data);
        return root;
    }
    public void inorder(){
        inorderRec(root);
```

```
}
public void inorderRec(Node root){
   if(root!=null)
   {
      inorderRec(root.left);
      System.out.print(root.key+" ");
      inorderRec(root.right);
   }
}
public void preorder(){
   preorderRec(root);
}
public void preorderRec(Node root)
{
   if(root!=null)
   {
      System.out.print(root.key+" ");
      preorderRec(root.left);
      preorderRec(root.right);
   }
}
public void postorder(){
   postorderRec(root);
}
public void postorderRec(Node root)
{
   if(root!=null){
   postorderRec(root.left);
   postorderRec(root.right);
   System.out.print(root.key+" ");
   }
}
public static void main(String[] args){
   Scanner sc=new Scanner(System.in);
   Bst bst=new Bst();
   String ch="";
   do{
      System.out.println("Enter the element to be inserted in the tree: ");
      int n=sc.nextInt();sc.nextLine();
      bst.insert(n);
      System.out.println("Do you wanna continue inserting elements:");
      ch=sc.nextLine();
   }while(ch.equals("yes"));
   System.out.println("Inorder Traversal ");
   bst.inorder();
   System.out.println("PreOrderTraversal ");
   bst.preorder();
```

```
        System.out.println("PostOrderTraversal ");
        bst.postorder();
    }
    }



//BST DELETE


import java.util.*;
public class Bst{
    class Node{
        int key;
        Node left,right;
        Node(int data){
            key=data;
            left=right=null;
        }
    }
    private Node root;
    Bst(){
        root=null;
    }
    public void insert(int data){
        root=insertRec(root,data);
    }
    public Node insertRec(Node root,int data){
        if(root==null)
        {
            root=new Node(data);
            return root;
        }
        else if(data<root.key)
        {
            root.left=insertRec(root.left,data);
        }
        else
            root.right=insertRec(root.right,data);
        return root;
    }
    public void inorder()
    {
        inorderRec(root);
    }
    public void inorderRec(Node root)
    {
```

```
    if(root!=null)
    {
        inorderRec(root.left);
        System.out.print(root.key+" ");
        inorderRec(root.right);
    }
}
public void delete(int key)
{
    root=deleteRec(root,key);
}
public Node deleteRec(Node root,int key)
{
    if(root==null)
    {
        return root;
    }
    if(key<root.key)
    {
        root.left=deleteRec(root.left,key);
    }
    else if(key>root.key)
    {
        root.right=deleteRec(root.right,key);
    }
    else{
        if(root.left==null)
            return root.right;
        else if(root.right==null)
            return root.left;

            root.key=minValue(root.right);
            root.right=deleteRec(root.right,root.key);
    }
    return root;
}
public int minValue(Node root)
{
    int minv=root.key;
    while(root.left!=null)
    {
        minv=root.left.key;
        root=root.left;
    }
    return minv;
}
```

```
public static void main(String[] args)
{
   Scanner sc=new Scanner(System.in);
   Bst bst=new Bst();
   String ch="";
   do{
      System.out.println("Enter the element to be inserted");
      int n=sc.nextInt();sc.nextLine();
      bst.insert(n);
      System.out.println("Do you want to keep inserting?");
      ch=sc.nextLine();
   }while(ch.equals("yes"));
   System.out.println();
   System.out.println("InorderTraversal");
   bst.inorder();
   System.out.println("Enter the element you want to delete:");
   int x=sc.nextInt();sc.nextLine();
   bst.delete(x);
   System.out.println("Inorder traversal after delting element:");
   bst.inorder();
   }
   }



//Priority queue

import java.util.*;

public class PriorityQueueDemo {
   public static void main(String[] args) {
      // Creating a PriorityQueue using various constructors
      PriorityQueue<Integer> pq1 = new PriorityQueue<>(); // Default
constructor
      PriorityQueue<Integer> pq2 = new PriorityQueue<>(List.of(10, 5, 15)); //
Constructor with initial elements
      Comparator<Integer> comparator = (a, b) -> b - a;
      PriorityQueue<Integer> pq3 = new PriorityQueue<>(comparator); //
Constructor with custom comparator

      // Adding elements using add() method
      pq1.add(20);
      pq1.add(30);
      pq1.add(10);

      // Adding elements using offer() method
      pq2.offer(25);
```

```
        pq2.offer(35);

        // Printing the elements in PriorityQueue
        System.out.println("PriorityQueue pq1: " + pq1);
        System.out.println("PriorityQueue pq2: " + pq2);

        // Peeking at the head of the queue using peek() method
        System.out.println("Peeked Element in pq1: " + pq1.peek());
        System.out.println("Peeked Element in pq2: " + pq2.peek());

        // Polling the head of the queue using poll() method
        System.out.println("Polled Element from pq1: " + pq1.poll());
        System.out.println("Polled Element from pq2: " + pq2.poll());

        // Removing a specific element using remove() method
        pq1.remove(30);

        // Checking the size of PriorityQueue using size() method
        System.out.println("Size of PriorityQueue pq1: " + pq1.size());
        System.out.println("Size of PriorityQueue pq2: " + pq2.size());

        // Checking if PriorityQueue is empty using isEmpty() method
        System.out.println("Is PriorityQueue pq1 empty? " + pq1.isEmpty());
        System.out.println("Is PriorityQueue pq2 empty? " + pq2.isEmpty());

        // Clearing the PriorityQueue using clear() method
        pq2.clear();
        System.out.println("PriorityQueue pq2 cleared.");

        // Printing the elements in PriorityQueue after clearing
        System.out.println("PriorityQueue pq2 after clearing: " + pq2);

        // Adding elements using addAll() method
        pq3.addAll(List.of(40, 45, 35));

        // Printing the elements in PriorityQueue pq3
        System.out.println("PriorityQueue pq3: " + pq3);
    }
}
```

AVL

```
import java.util.Scanner;

class Node {
```

```
    int data, height;
    Node left, right;

    Node(int d) {
        data = d;
        height = 1;
    }
}

class AVLTree {
    Node root;

    int height(Node N) {
        if (N == null)
            return 0;
        return N.height;
    }

    int max(int a, int b) {
        return (a > b) ? a : b;
    }

    Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

        x.right = y;
        y.left = T2;

        y.height = max(height(y.left), height(y.right)) + 1;
        x.height = max(height(x.left), height(x.right)) + 1;

        return x;
    }

    Node leftRotate(Node x) {
        Node y = x.right;
        Node T2 = y.left;

        y.left = x;
        x.right = T2;

        x.height = max(height(x.left), height(x.right)) + 1;
        y.height = max(height(y.left), height(y.right)) + 1;

        return y;
    }
```

```
int getBalance(Node N) {
    if (N == null)
        return 0;
    return height(N.left) - height(N.right);
}

Node insert(Node node, int data) {
    if (node == null)
        return new Node(data);

    if (data < node.data)
        node.left = insert(node.left, data);
    else if (data > node.data)
        node.right = insert(node.right, data);
    else
        return node;

    node.height = 1 + max(height(node.left), height(node.right));

    int balance = getBalance(node);

    if (balance > 1 && data < node.left.data)
        return rightRotate(node);

    if (balance < -1 && data > node.right.data)
        return leftRotate(node);

    if (balance > 1 && data > node.left.data) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }

    if (balance < -1 && data < node.right.data) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }

    return node;
}

Node delete(Node root, int data) {
    if (root == null)
        return root;

    if (data < root.data)
        root.left = delete(root.left, data);
```

```
    else if (data > root.data)
        root.right = delete(root.right, data);
    else {
        if ((root.left == null) || (root.right == null)) {
            Node temp = null;
            if (temp == root.left)
                temp = root.right;
            else
                temp = root.left;

            if (temp == null) {
                temp = root;
                root = null;
            } else
                root = temp;
        } else {
            Node temp = minValueNode(root.right);
            root.data = temp.data;
            root.right = delete(root.right, temp.data);
        }
    }

    if (root == null)
        return root;

    root.height = max(height(root.left), height(root.right)) + 1;

    int balance = getBalance(root);

    if (balance > 1 && getBalance(root.left) >= 0)
        return rightRotate(root);

    if (balance > 1 && getBalance(root.left) < 0) {
        root.left = leftRotate(root.left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root.right) <= 0)
        return leftRotate(root);

    if (balance < -1 && getBalance(root.right) > 0) {
        root.right = rightRotate(root.right);
        return leftRotate(root);
    }

    return root;
}
```

```
    Node minValueNode(Node node) {
        Node current = node;
        while (current.left != null)
            current = current.left;
        return current;
    }

    void preOrder(Node node) {
        if (node != null) {
            System.out.print(node.data + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        AVLTree tree = new AVLTree();
        Scanner scanner = new Scanner(System.in);
        int choice;

        while (true) {
            System.out.println("\nAVL Tree Operations Menu:");
            System.out.println("1. Insert an element");
            System.out.println("2. Delete an element");
            System.out.println("3. Display Preorder Traversal");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter element to insert: ");
                    int insertElement = scanner.nextInt();
                    tree.root = tree.insert(tree.root, insertElement);
                    break;
                case 2:
                    System.out.print("Enter element to delete: ");
                    int deleteElement = scanner.nextInt();
                    tree.root = tree.delete(tree.root, deleteElement);
                    break;
                case 3:
                    System.out.print("Preorder traversal of the AVL tree: ");
                    tree.preOrder(tree.root);
                    System.out.println();
```

```
                break;
            case 4:
                System.out.println("Exiting...");
                scanner.close();
                return;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    }
  }
}
```