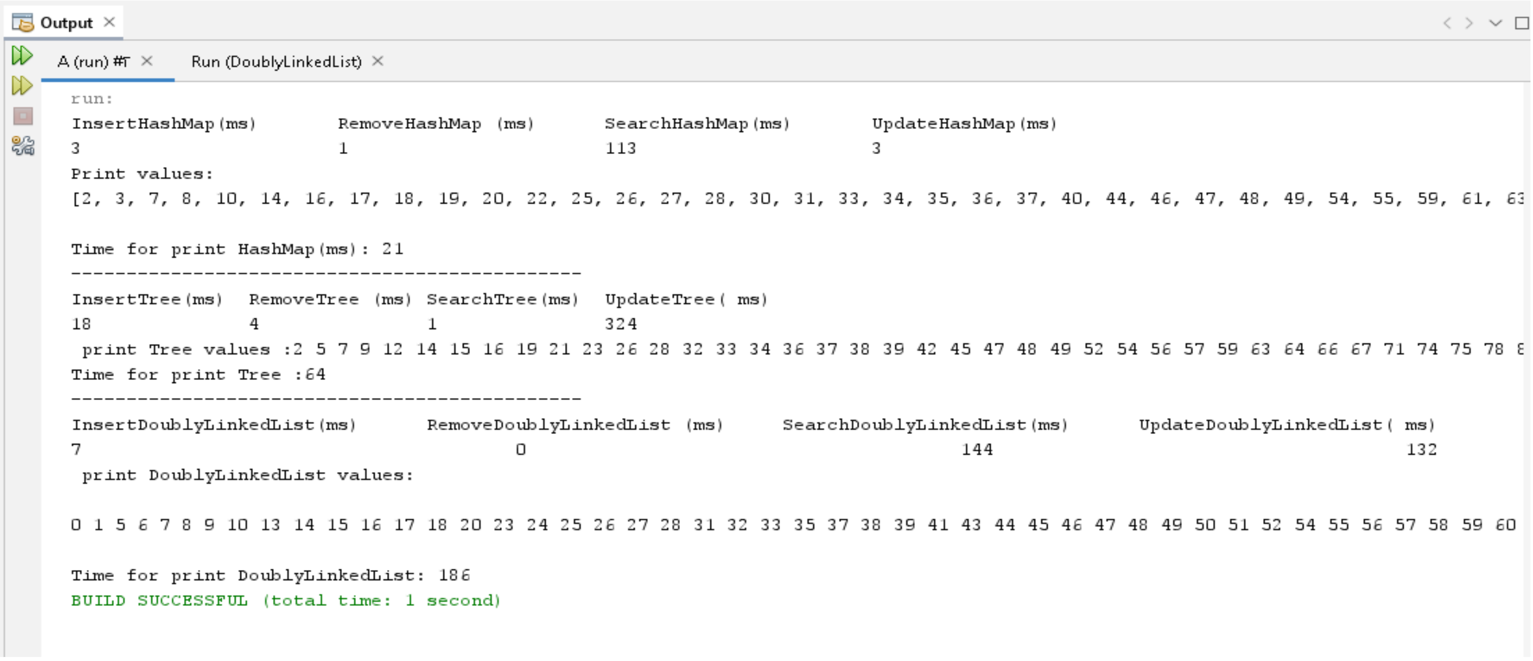
Assignment

Output:



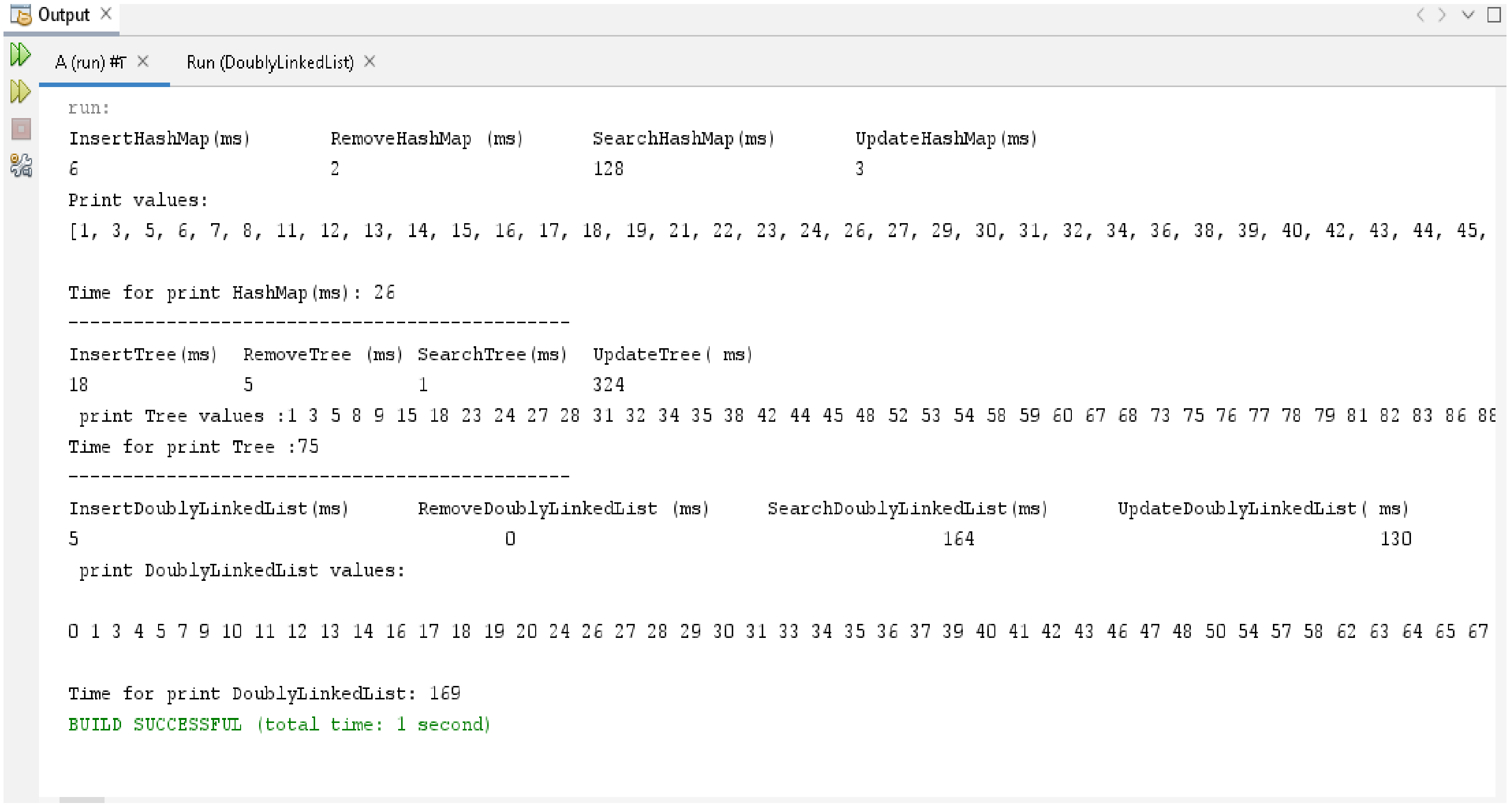
```
run:
InsertHashMap(ms)          RemoveHashMap  (ms)        SearchHashMap(ms)          UpdateHashMap(ms)
3                          1                          113                        3
Print values:
[2, 3, 7, 8, 10, 14, 16, 17, 18, 19, 20, 22, 25, 26, 27, 28, 30, 31, 33, 34, 35, 36, 37, 40, 44, 46, 47, 48, 49, 54, 55, 59, 61, 63

Time for print HashMap(ms): 21
----------------------------------------------
InsertTree(ms)  RemoveTree  (ms) SearchTree(ms)  UpdateTree( ms)
18              4                1               324
 print Tree values :2 5 7 9 12 14 15 16 19 21 23 26 28 32 33 34 36 37 38 39 42 45 47 48 49 52 54 56 57 59 63 64 66 67 71 74 75 78 8
Time for print Tree :64
----------------------------------------------
InsertDoublyLinkedList(ms)      RemoveDoublyLinkedList  (ms)      SearchDoublyLinkedList(ms)       UpdateDoublyLinkedList( ms)
7                                       0                                       144                                       132
 print DoublyLinkedList values:

0 1 5 6 7 8 9 10 13 14 15 16 17 18 20 23 24 25 26 27 28 31 32 33 35 37 38 39 41 43 44 45 46 47 48 49 50 51 52 54 55 56 57 58 59 60

Time for print DoublyLinkedList: 186
BUILD SUCCESSFUL (total time: 1 second)
```

otherRun(فترة تقريبا ساعة)



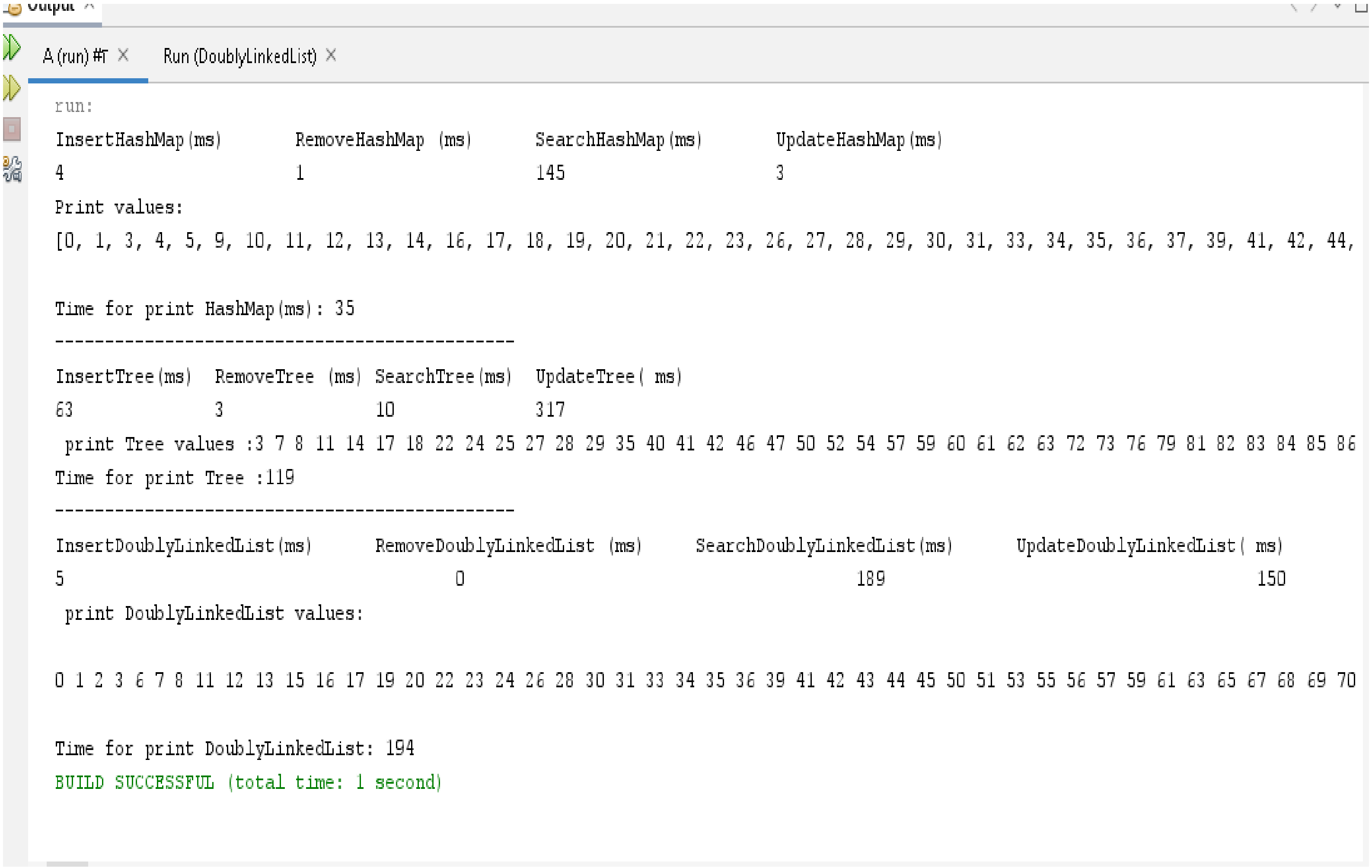
```
run:
InsertHashMap(ms)          RemoveHashMap  (ms)        SearchHashMap(ms)          UpdateHashMap(ms)
6                          2                          128                        3
Print values:
[1, 3, 5, 6, 7, 8, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 26, 27, 29, 30, 31, 32, 34, 36, 38, 39, 40, 42, 43, 44, 45,

Time for print HashMap(ms): 26
----------------------------------------------
InsertTree(ms)  RemoveTree  (ms) SearchTree(ms)  UpdateTree( ms)
18              5                1               324
 print Tree values :1 3 5 8 9 15 18 23 24 27 28 31 32 34 35 38 42 44 45 48 52 53 54 58 59 60 67 68 73 75 76 77 78 79 81 82 83 86 88
Time for print Tree :75
----------------------------------------------
InsertDoublyLinkedList(ms)      RemoveDoublyLinkedList  (ms)      SearchDoublyLinkedList(ms)       UpdateDoublyLinkedList( ms)
5                                       0                                       164                                       130
 print DoublyLinkedList values:

0 1 3 4 5 7 9 10 11 12 13 14 16 17 18 19 20 24 26 27 28 29 30 31 33 34 35 36 37 39 40 41 42 43 46 47 48 50 54 57 58 62 63 64 65 67

Time for print DoublyLinkedList: 169
BUILD SUCCESSFUL (total time: 1 second)
```

otherRun(فترة تقريبا ساعة)



```
run:
InsertHashMap(ms)       RemoveHashMap  (ms)      SearchHashMap(ms)        UpdateHashMap(ms)
4                       1                        145                      3
Print values:
[0, 1, 3, 4, 5, 9, 10, 11, 12, 13, 14, 16, 17, 18, 19, 20, 21, 22, 23, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 37, 39, 41, 42, 44,

Time for print HashMap(ms): 35
----------------------------------------------
InsertTree(ms)  RemoveTree  (ms) SearchTree(ms)  UpdateTree( ms)
63              3                10              317
 print Tree values :3 7 8 11 14 17 18 22 24 25 27 28 29 35 40 41 42 46 47 50 52 54 57 59 60 61 62 63 72 73 76 79 81 82 83 84 85 86
Time for print Tree :119
----------------------------------------------
InsertDoublyLinkedList(ms)      RemoveDoublyLinkedList  (ms)     SearchDoublyLinkedList(ms)      UpdateDoublyLinkedList( ms)
5                                       0                                       189                                     150
 print DoublyLinkedList values:

0 1 2 3 6 7 8 11 12 13 15 16 17 19 20 22 23 24 26 28 30 31 33 34 35 36 39 41 42 43 44 45 50 51 53 55 56 57 59 61 63 65 67 68 69 70

Time for print DoublyLinkedList: 194
BUILD SUCCESSFUL (total time: 1 second)
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.TreeMap;
import java.util.List;
import java.util.Map;
import java.util.Random;
import java.util.Scanner;
import java.util.Set;


public class Assignment{
   public static void insertTree ( BST t){

    Set<Integer> values = new HashSet<>();
     Random random = new Random();
      while (values.size() < 5000) {
         int val = random.nextInt(10000) + 1;
         if (!values.contains(val)) {
            values.add(val);
            t.insert(val);
         }
      }


   }
```

```
public static void DeleteTree ( BST tree){



    Random r= new Random();
    int[] values = new int[5000];
    for (int i = 0; i < 5000; i++) {
     values[i] = r.nextInt();
    }
    for (int i=0 ;i<5000;i++) {
         if ( tree.contains(values[i])){
            tree.delete(values[i]);
         }
    }


}
public static void printTree ( BST tree){
    System.out.println();
    System.out.print(" print Tree values :");
    tree.inOrder(tree.root);
 System.out.println();
 System.out.print("Time for print Tree :");
}








public static void searchTree ( BST tree){
```

```
         Random r=new Random(5000);
         int found=0;
         int Notfound=0;
        for (int i = 0; i < 5000; i++) {
            int val = r.nextInt();
            if (tree.search(val)) {
                found++;
//                System.out.println(val + " was found in the tree");
            } else {
                Notfound++;
//                System.out.println(val + " was not found in the tree");
            }
        }

    }
    public static void UpdateTree ( BST tree){
        Random r= new Random();
        int[] values = new int[5000];
        for (int i = 0; i < 5000; i++) {
         values[i] = r.nextInt();
        }
        for ( int j=0;j<5000;j++ ) {

          tree.updateElement(tree.root,tree.getvalues(tree.root),values[j] );
        }


    }
```

```
 public static void removeHashMap(HashMap<Integer,Integer>hp) {


   Random r= new Random();

   int[] keys = new int[5000];

   for (int i = 0; i < 5000; i++) {

     keys[i] = r.nextInt();

   }




   for (int key : keys) {

       if ( hp.containsKey(key))

       hp.remove(key);

   }

}


 public static void printHashMap(HashMap<Integer,Integer>hp) {

    System.out.println();

     System.out.println("Print values:");


   TreeMap<Integer, Integer> sortedValue = new TreeMap<>();

   for(int i=0;i<5000;i++){

      sortedValue.put(hp.get(i),i);


   }

   System.out.print(sortedValue.keySet());




   System.out.println("---------------------------");

   System.out.println();
```

```
    System.out.print("Time for print HashMap(ms): ");



 }



 public static void insertHashMap(HashMap<Integer, Integer> hp) {


  Set<Integer> values = new HashSet<>();

   Random random = new Random();

    for (int i=0;i<5000;i++) {

        int val = random.nextInt(10000);

        if (!values.contains(val)) {

            values.add(val);

            hp.put(i, val);

        }

    }


}


public static void searchHashMap(HashMap<Integer, Integer> hp) {

     Random r=new Random(5000);

     int found=0;

     int Notfound=0;

    for (int i=0;i < 5000;i++){

        if (hp.containsValue(r.nextInt())){

            found++;

        }

        else{
             Notfound++;

           }

       }

//      System.out.println( " Find "+ found +" value");

//      System.out.println( "  not Find "+ Notfound +" value");

   }


    public static void updateHashMap(HashMap<Integer, Integer> hp) {



       Random r= new Random();

       int[] values = new int[5000];

       for (int i = 0; i < 5000; i++) {

        values[i] = r.nextInt(5000);

       }


       for ( int j=0;j<5000;j++ ) {

           hp.put(j,values[j] );

       }



   }
     public static void insertDoublyLink(DoublyLinkedList1 DoubLinkList){
      Set<Integer> values = new HashSet<>();



       Random random = new Random();

       while (values.size() < 5000) {

           int val = random.nextInt(10000) + 1;
        if (!values.contains(val)) {
            values.add(val);
            DoubLinkList.insert(val);
        }
    }

}
   public static void removeDoublyLink(DoublyLinkedList1 DoubLinkList){


    Random r= new Random();
    int[] values = new int[5000];
    for (int i = 0; i < 5000; i++) {
     values[i] = r.nextInt();
    }
        for (int i = 0; i < 5000; i++) {
            if (DoubLinkList.contains(values[i]))

            DoubLinkList.remove(values[i]);

       }
 }



 public static void searchDoublyLink(DoublyLinkedList1 DoubLinkList) {
    Random ran=new Random ();
    int countFound=0;
    int countNotFound=0;
```

```
        for (int i = 0; i < 5000; i++) {
            int value = ran.nextInt();
            if (DoubLinkList.contains(value)) {
                countFound++;

            } else {
              countNotFound++;

            }

        }
//        System.out.println(countFound + " exists in the list");
//         System.out.println(countNotFound+ " does not exist in the list");
    }
    public static void UpdateDoublyLink(DoublyLinkedList1 DoubLinkList) {

         Random r= new Random();
        int[] values = new int[5000];
        for (int i = 0; i < 5000; i++) {
         values[i] = r.nextInt();
        }





        for (int j= 0; j < 5000; j++) {
            if (!DoubLinkList.contains(values[j])) {
              DoubLinkList.update( j,values[j]);
            }



        }

    }

    public static void printDoublyLink (DoublyLinkedList1 DoubLinkList) {


        System.out.println();

        System.out.println(" print DoublyLinkedList values:");

        System.out.println();


        DoubLinkList.sort();

        DoubLinkList.printNodes();

         System.out.println();

        System.out.println();

         System.out.print("Time for print DoublyLinkedList: ");

    }


    public static void main(String[] args) {

     Scanner in=new Scanner (System.in );

     HashMap <Integer, Integer> hp=new HashMap <>();

     BST tree =new BST();


        System.out.println("InsertHashMap(ms)" +"\t"+"RemoveHashMap (ms)"
+"\t"+"SearchHashMap(ms)" +"\t"+"UpdateHashMap(ms)");

     int start= (int)System.currentTimeMillis();

     insertHashMap (hp);

     int end=(int)System.currentTimeMillis();

     System.out.print((end- start));
```

```
 System.out.print("\t" +"\t"+"\t");


 start= (int)System.currentTimeMillis();

 removeHashMap(hp);

 end=(int)System.currentTimeMillis();

 System.out.print((end- start));


System.out.print("\t" +"\t"+"\t");


  start= (int)System.currentTimeMillis();

 searchHashMap(hp);

 end=(int)System.currentTimeMillis();

 System.out.print((end- start)+"");

 System.out.print("\t" +"\t"+"\t");




  start= (int)System.currentTimeMillis();

 updateHashMap (hp);

 end=(int)System.currentTimeMillis();

 System.out.print((end- start));

  System.out.print("\t" +"\t"+"\t");


  start= (int)System.currentTimeMillis();

 printHashMap (hp);

 end=(int)System.currentTimeMillis();

 System.out.print((end- start));

 System.out.println();

  System.out.println("---------------------------------------------");
```

```
     System.out.println("InsertTree(ms)" +"\t"+"RemoveTree (ms)" +"\t"+"SearchTree(ms)"
+"\t"+"UpdateTree( ms)" +"\t" );

     start= (int)System.currentTimeMillis();

    insertTree ( tree);

     end=(int)System.currentTimeMillis();

    System.out.print((end- start));

     System.out.print("\t" );


     System.out.print("\t" );

    start= (int)System.currentTimeMillis();

    DeleteTree(tree);

    end=(int)System.currentTimeMillis();

    System.out.print((end- start));



    System.out.print("\t" );

    System.out.print("\t" );

     start= (int)System.currentTimeMillis();

    searchTree(tree);

    end=(int)System.currentTimeMillis();

    System.out.print((end- start)+"");

    System.out.print("\t" );

    System.out.print("\t" );



     start= (int)System.currentTimeMillis();

    UpdateTree (tree);

    end=(int)System.currentTimeMillis();
```

```
      System.out.print((end- start));


       System.out.print("\t" );
      System.out.print("\t" );


       start= (int)System.currentTimeMillis();
      printTree ( tree);
      end=(int)System.currentTimeMillis();
      System.out.print((end- start));


       System.out.println( );
        System.out.println("---------------------------------------------");
       DoublyLinkedList1 DoubLinkList = new DoublyLinkedList1();
       System.out.println("InsertDoublyLinkedList(ms)" +"\t"+"RemoveDoublyLinkedList (ms)"
+"\t"+"SearchDoublyLinkedList(ms)" +"\t"+"UpdateDoublyLinkedList( ms)" +"\t" );
      start= (int)System.currentTimeMillis();
      insertDoublyLink (DoubLinkList);
       end=(int)System.currentTimeMillis();
      System.out.print((end- start));


      System.out.print("\t" +"\t"+"\t"+"\t"+"\t");


      start= (int)System.currentTimeMillis();
//      removeDoublyLink(DoubLinkList);
      end=(int)System.currentTimeMillis();
      System.out.print((end- start));


     System.out.print("\t" +"\t"+"\t"+"\t"+"\t");
```

```
       start= (int)System.currentTimeMillis();
      searchDoublyLink(DoubLinkList);
      end=(int)System.currentTimeMillis();
      System.out.print((end- start)+"");
      System.out.print("\t" +"\t"+"\t"+"\t"+"\t");


       start= (int)System.currentTimeMillis();
      UpdateDoublyLink (DoubLinkList);
      end=(int)System.currentTimeMillis();
      System.out.print((end- start));
       System.out.print("\t" +"\t"+"\t");

       start= (int)System.currentTimeMillis();
      printDoublyLink (DoubLinkList);
      end=(int)System.currentTimeMillis();
      System.out.print((end- start));
      System.out.println();

    }
}
class Node {
    int key;

    Node l,r,parent;

    public Node(int key) {
        this.key = key;
    }
```

```
}
class BST {

    Node root;



    public BST() {
        root = null;
    }

    void insert(int k) {
        root = insertRec(root, k);

    }
    Node root() {
        return root;
    }

    Node insertRec(Node root, int k) {
        if (root == null) {
            root = new Node(k);
            return root;
        }
        if (k < root.key) {
            root.l = insertRec(root.l, k);
        }
        if (k > root.key) {
            root.r = insertRec(root.r, k);
        }
        return root;
    }

public boolean contains(int value) {
        return contains(root, value);
    }

    private boolean contains(Node node, int value) {
        if (node == null) {
            return false;
        }
        if (node.key == value) {
            return true;
        }
        if (value < node.key) {
            return contains(node.l, value);
        } else {
            return contains(node.r, value);
        }
    }
    public int getvalues (Node root) {
         if (root == null) {
            return 0;
        }
        getvalues(root.l);
        getvalues(root.r);
        return root.key;
    }
    public void updateElement(Node root, int oldValue, int newValue) {
    if (root == null) {
        return;
    }

    if (root.key == oldValue) {
        root.key = newValue;
    } else if (root.key> oldValue) {
        updateElement(root.l, oldValue, newValue);
    } else {
        updateElement(root.r, oldValue, newValue);
    }
}



    void delete(int k) {
        root = deleteRec(root, k);
    }

    Node deleteRec(Node root, int k) {
        if (root == null) {
            return null;
        }
        if (k < root.key) {
            root.l = deleteRec(root.l, k);
        } else if (k > root.key) {
            root.r = deleteRec(root.r, k);
    } else {
        if (root.l == null) {
            return root.r;
        } else if (root.r == null) {
            return root.l;
        } else {
            root.key = minVal(root.r);
            root.r = deleteRec(root.r, root.key);
        }
    }
    return root;
}

int minVal(Node root) {
    int min = root.key;
    while (root.l != null) {
        min = root.l.key;
        root = root.l;
    }
    return min;
}

void inOrder(Node root) {
    if (root == null) {
        return;
    }
    inOrder(root.l);
  System.out.print (root.key+" ") ;
    inOrder(root.r);
    }
      boolean search(int val) {
        return searchRec(root, val);
    }

    boolean searchRec(Node current, int val) {
        if (current == null) {
            return false;
        }
        if (val == current.key) {
            return true;
        }
        return val < current.key ? searchRec(current.l, val): searchRec(current.r, val);
    }
}


 class DoublyLinkedList1 {
      class Node {
        int val;
        Node prev;
        Node next;

        public Node(int val) {
            this.val = val;
            this.prev = null;
            this.next = null;
        }
    }
```

```
Node head;

Node tail;

int size;

public DoublyLinkedList1() {

    this.head = null;

    this.tail = null;

}

 public boolean contains(int value) {

    Node current = head;

    while (current != null) {

        if (current.val == value) {

            return true;

        }

        current = current.next;

    }

    return false;

}



public void insert(int val) {

    Node newNode = new Node(val);

    if (tail != null) {

        tail.next = newNode;

        newNode.prev = tail;

        tail = newNode;

    } else {

        head = newNode;

        tail = newNode;

    }
    size++;
}
public void update( int index,int value) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }

    Node current = head;
   for (int i = 0; i < index; i++) {
        current = current.next;
    }
    current.val = value;
}
public void sort() {

    if (size <= 1) {
        return;
    }

    boolean swap = true;
    while (swap) {
        swap = false;
        Node current = head;
        while (current.next != null) {
            if (current.val> current.next.val) {

                int temp = current.val;
                current.val = current.next.val;
                current.next.val = temp;
```

```
          swap= true;
        }
        current = current.next;
      }
    }
  }


public void remove(int index) {
if (index < 0 || index >= size) {
 throw new IndexOutOfBoundsException();
}

Node current = head;
for (int i = 0; i < index; i++) {
 current = current.next;
}



if (current.prev != null) {
 current.prev.next = current.next;
} else {

 head = current.next;
}
if (current.next != null) {
 current.next.prev = current.prev;
} else {
  tail = current.prev;
 }
}
    public void printNodes() {

     Node  current = head;
     if(head == null) {
       System.out.println("Doubly linked list is empty");
       return;
     }

     while(current != null) {

       System.out.print(current.val + " ");
       current = current.next;
     }
   }
 }
```