# GIT Department of Computer Engineering
# CSE 222/505 - Spring 2022
# Homework 5 Report

**BURAK KOCAUSTA**
**1901042605**

## 1. SYSTEM REQUIREMENTS

There are 3 different tree implementations(except from interfaces and base class). Each one has their default constructors and they can be constructed just doing insertion or using left and right nodes in node link structures. Except for the BinaryTree all data's are required to be "Comparable".

```java
/**
 * No parameter constructor creates null root
 */
public BinaryTree ( ) {
```

```java
/**
 * Constructor using left and right trees.
 * @param data data
 * @param leftTree left subtree
 * @param rightTree right subtree
 */
public BinaryTree ( E data, BinaryTree<E> leftTree, BinaryTree<E> rightTree )
```

BinaryTree instances can be constructed with these 2 way.

```java
/**
 * No parameter constructor sets fields to null or 0.
 */
public LinkedBinaryHeap ( ) {
```

LinkedBinaryHeap can be constructed empty it has no requirements initially.

```java
/**
 * Constructor which takes root and comparator.
 * @param comp comparator
 * @param item root item
 */
public LinkedBinaryHeap ( E item, Comparator<E> comp ) {
```

LinkedBinaryHeap can be constructed with item and comparator. Item is required to be Comparable.

```java
/**
 * Constructor using left and right trees.
 * @param data data
 * @param leftTree left subtree
 * @param rightTree right subtree
 */
public LinkedBinaryHeap ( E data, LinkedBinaryHeap<E> leftTree, LinkedBinaryHeap<E> rightTree )
```

LinkedBinaryHeap can be constructed with merging 2 trees and one root data.

Tree updates itself according to the data.

```java
public interface SearchTree<E> {
    /**
     *
     * Inserts the given item properly to the tree, while insertion it does comparison.
     * @param item item
     * @return true if operation is successful, false otherwise.
     */
    boolean add ( E item );

    /**
     *
     * Removes target from tree.
     * @param target target
     * @return true if operation is successful, returns false if it is not.
     */
    boolean remove ( E target );

    /**
     *
     * Deletes target from tree.
     * @param target target
     * @return target if operation is successful, null if it is not.
     */
    E delete ( E target );

    /**
     *
     * Check operation for searching.
     * @param target target
     * @return true if it is found, false otherwise.
     */
    boolean contains ( E target );
  /**
   *
   * Finds the data and returns it reference.
   * @param target target
   * @return data if it is found, returns null otherwise.
   */
  E find ( E target );
```

SearchTree interface requires these abstract methods to be implemented.

```java
  /**
   * No parameter constructor creates an empty tree.
   */
  @SuppressWarnings("unchecked")
  public BinarySearchTree (  ) {
```

BinarySearchTree class has no parameter constructor, it has no requirements during construction.

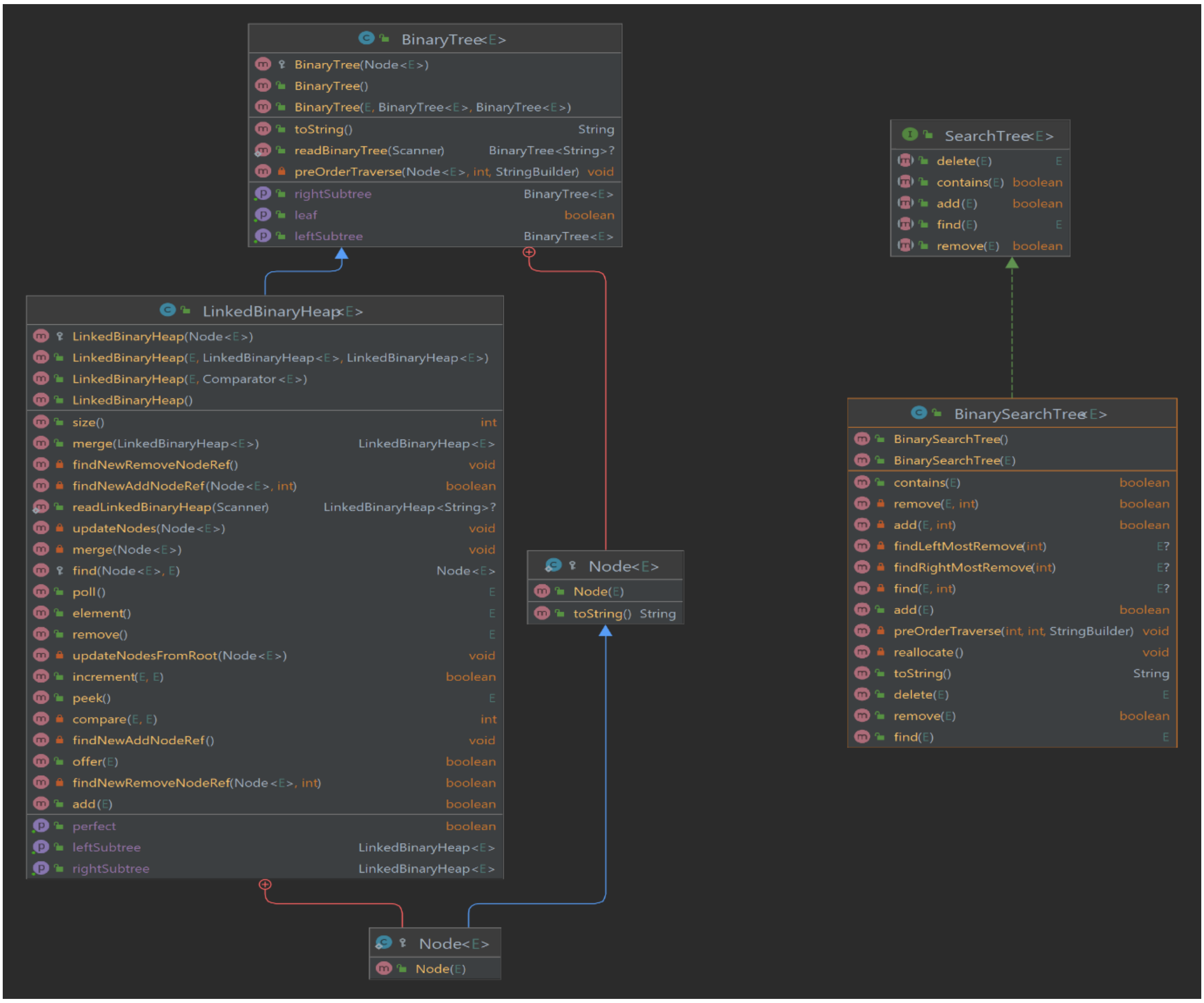
```java
/**
 * Constructor which takes root of the tree.
 * @param root root of the tree.
 */
public BinarySearchTree ( E root ) {
```

BinarySearchTree has one  parameter constructor. It requires a Comparable root data.

```java
import BinaryHeapGTU.*;
import BinarySearchTreeGTU.*;
```

All these implementations are inside these packages, therefore this import statement is required.

## 2. CLASS DIAGRAM



3. **PROBLEM SOLUTION APPROACH**

# Q1

First Question’s solution and explanation is added to Analysis and Solutions part of the report.

## Q2

Second Question’s solution and explanation is added to Analysis and Solutions part of the report.

## Q3

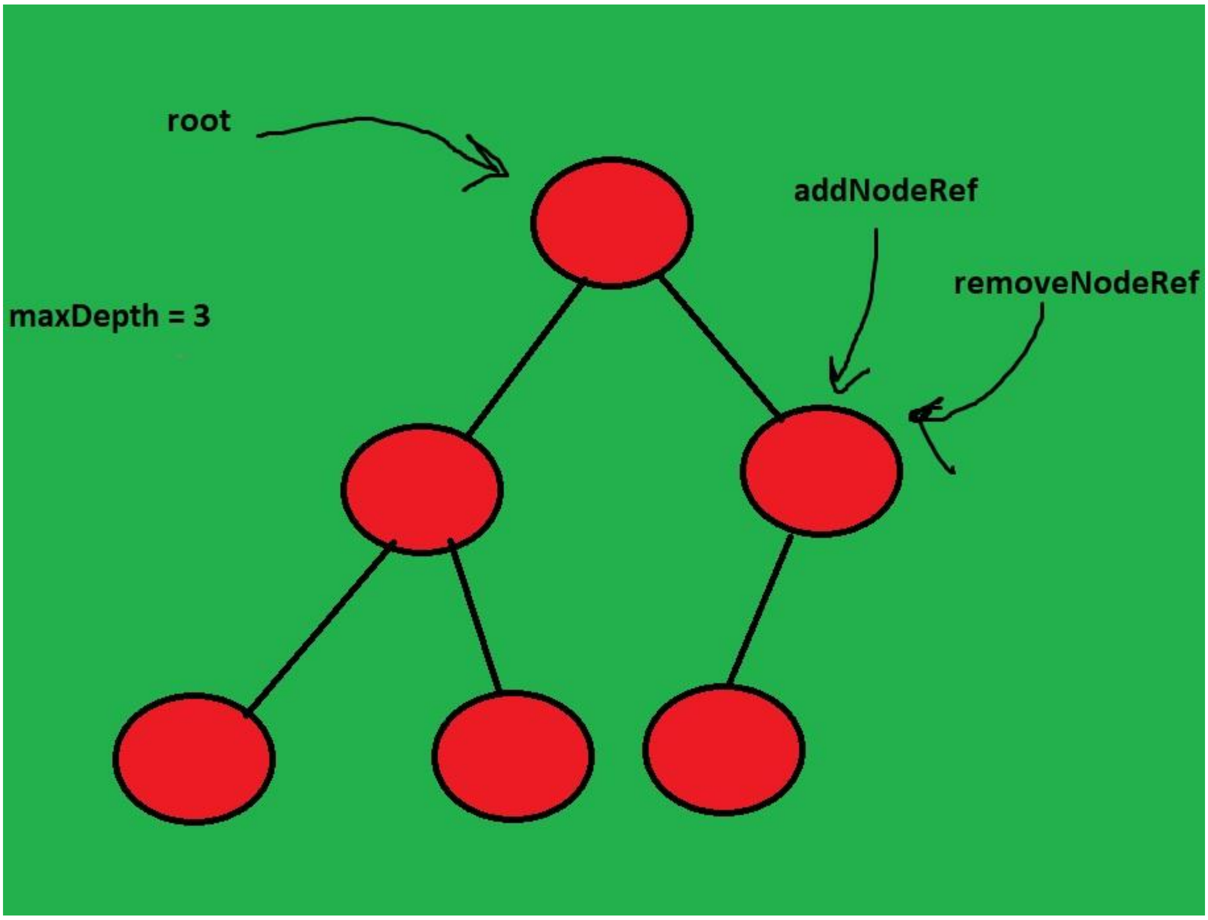
In Binary Heap problem, BinaryTree class is extended. BinaryTree class has Node class, and it is also extended with binary heap’s node class. I thought what could be added to binary heaps node class. In normal binary heap implementation, it is done with array structure, because adding last element can be done with amortized constant time. Initially I thought the possibility of can this to be done with node link structure.

```
protected static class Node<E> extends BinaryTree.Node<E> {

    /**
     * Node reference to the parent.
     */
    protected Node<E> parent = null;

    /**
     * Constructs a node with given data.
     * @param data data
     */
    public Node ( E data ) {
        super(data);
        parent = null;
    }
}
```

It is obvious that Node class needs a parent reference. Because there is no way to access parents like as array structure.

```java
/**
 * comparator for constructing different kind of heaps
 */
protected Comparator<E> comparator = null;

/**
 * reference to the insertion place.
 */
protected Node<E> addNodeRef = null;

/**
 * reference to the node which is parent of the node that will be put root before updating.
 */
protected Node<E> removeNodeRef = null;

/**
 * size of the tree.
 */
protected int size = 0;

/**
 * maximum depth of the nodes.
 */
protected int maxDepth = 0;
```

I added addNodeRef reference to the heaps private field to make insertion sometimes constant time. I thought this option creates a best case. After implementing insertion operation, I realized that it is impossible to insert binary heap in constant time. Similarly, I added removeNodeRef to make deletion to best case. removeNodeRef points data to be put root, and goes somewhere proper after deletion. In this class there is also maxDepth variable to hold greatest depth. This makes easier to make insertion or deletion while traversing. For the merging, I thought it as a member method of this class, which takes other tree as parameter and inserts it elements one by one. It has to be one by one because there is an hierarchy in heap, so It is not possible to just link left and right nodes. Incrementing part is ambiguous, therefore I thought it as, there is an increment method which takes key value and value that to be set. This value must be greater than key, according to the change heap will be update itself.




At the end, I decided that node link structure is not an appropriate solution for binary heap concept. I made complexity analysis of this class in complexity analysis and solutions part of this report.

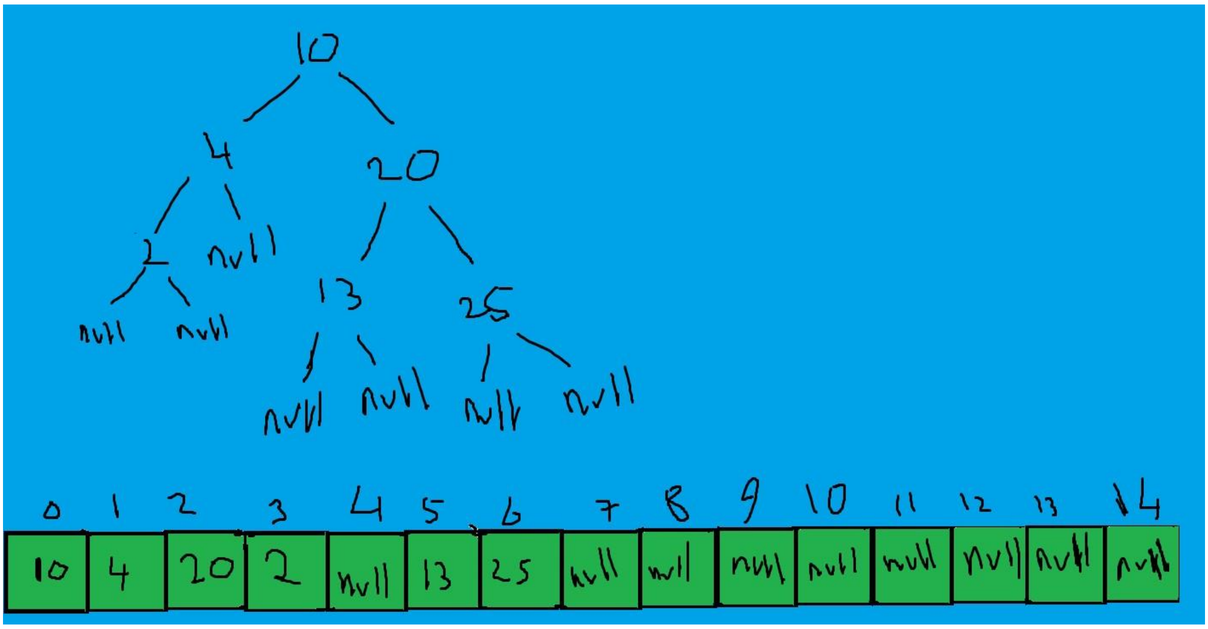
## Q4

For the BinarySearchTree class, Firstly I created SearchTree interface which has essential, generic methods for BinarySearchTree. In node link structure for binary search tree, It is very efficient to traverse because order is known, according to the comparisons going left or right solves the case. In array implementation traversing is similar because left child = 2*n + 1, right child = 2*n + 2. But this information is not useful as in the binary heap, because binary heap is complete tree. Binary search tree is not a complete tree. During insertion there can be too much reallocation. So, insertion’s time complexity is increased, It cannot be amortized constant time. When node is inserted, its child’s even if them are null, are having to be inserted to array.

```java
public class BinarySearchTree<E> implements SearchTree<E> {

    private E[] theData;
    private int capacity;

    private static final int INITIAL_CAPACITY = 31;
```

hold data as an array, and capacity information for checking the array is full or not. First element of the array is always root.

Insertion operation is not same with node link structure because of the reallocation time as I explained. Deletion operation is not same either. Because There isn’t any option to just link the nodes. Best case is a node that has no child, but in other cases there must be pretty much swap operations to rearrange the tree. Example binary Tree inside theData given below.

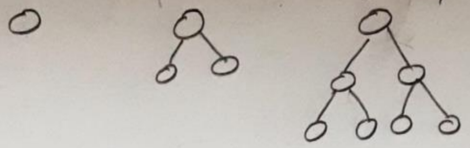



## 4. Complexity Analysis(Q3,Q4) and Solutions(Q1,Q2)

### Q1

Q1)

a) 1 2 3 } height



1 5 17 } total depths

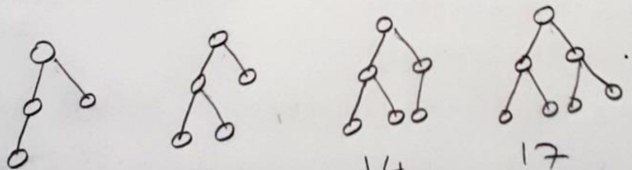
$2^0$ $2^1 \times 2 + 2^0$ $2^3 \times 3 + 2^1 \times 2 + 2^0$

In a perfect binary tree total depth can be calculated with height.

$$\boxed{\text{total depth} = \sum_{1}^{h} 2^{h-1} \times h}$$

For complete binary trees, It can be said that this value is maximum for this height. It is not possible the determine exact ~~total~~ total depth of complete binary tree.

For example for height = 3



8 11 14 17

For height 3 there are 4 different total depth possibilities. Formula founded above only works for the 17 because it is perfect.

So, $\boxed{\text{total depth} = \sum_{1}^{h} 2^{h-1} \times h}$ is definitely true for perfect binary trees.

Every perfect binary trees are also complete binary trees.

Q1)

b) For a perfect binary tree.

$2^h - 1 = n$

$2^h = n+1$

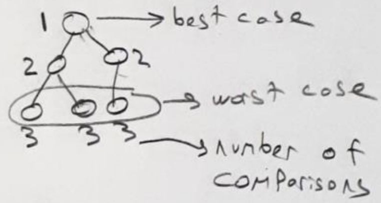
$\boxed{h = \log(n+1)}$

For a complete binary tree

$\boxed{h \cong \log(n+1)}$

→ In binary search if tree is not like line, in every step half of the tree is eliminated. Therefore, in a balanced tree, number of the comparisons cannot be number of element in the tree.

→ For the worst case in complete tree (it is assumed that key is in the tree) element is at one of the leaf node.




→ Best case is $\Theta(1)$ for binary search because element can be root of the tree.

→ For the worst case, if element is at leaf nodes total number of comparison is equal to its height. As calculated above $\boxed{\text{height} \cong \log(\text{number of node} + 1)}$ for complete binary trees. Therefore worst case is $\Theta(\log n)$.

∘ If this tree $\boxed{\text{weren't complete binary tree}}$, it cannot be said that worst case is logarithmic. In that case average case were logarithmic worst case were linear.

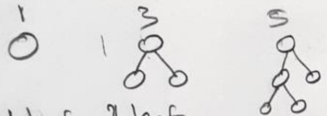
$$\boxed{\begin{aligned} T_{worst}(n) &= \Theta(\log n) \\ T_{best}(n) &= \Theta(1) \\ T_{av}(n) &= \Theta(\log n) \\ T(n) &= O(\log n) \end{aligned}}$$

→ for binary search in complete binary trees.

Q1)

c) For a binary tree to be full binary tree, number of nodes must be an $\boxed{\text{odd number}}$, so yes, there is a restriction on the number of nodes in a full binary tree. There is one root, to be full all nodes must have 2 or 0 children. So in a full binary tree total number of node $= \boxed{2k+1}$. It has to be odd.

→ In full binary tree, as indicated above number of the nodes are odd. Total number of the nodes in full binary trees are goes like 1, 3, 5, .... In every change 2 new node is added and they are leaf. Every two leaf creates one internal node.



1 leaf 0 internal | 2 leaf 1 internal | 3 leaf 2 internal

→ for every transition 1 leaf is gone, 2 new leaf is created. So, every consecutive full binary tree leaves incerements one by one. As similar when $\boxed{\text{2 leaf is created 1 leaf becomes internal node}}$. So number of leaves is number of internal node + 1.

| | | | | | |
|---|---|---|---|---|---|
| number of nodes | 1 | 3 | 5 | 7 .... | $2k+1$ |
| number of leaves | 1 | 2 | 3 | 4 ... | $k+1$ |
| number of internal nodes | 0 | 1 | 2 | 3 ... | $k$ |

→ Number of leaves has to be one more from number of internal node because of the reason indicated above. In n node full binary tree.

$$\boxed{\begin{array}{l} \frac{n+1}{2} = \text{number of leaves} \\ \frac{n-1}{2} = \text{number of internal node} \end{array}}$$

$\frac{n+1}{2} - 1 = \text{number of internal node}$

**Q3**

```java
public E peek ( ) {
    if ( root == null )
        return null;
    return root.data;
}
```

peek() method is constant time operation. **T(n) = Θ(1)**

```java
public E element ( ) throws NoSuchElementException {
    E item = peek();
    if( item == null )
        throw new NoSuchElementException("Tree is empty!");
    return item;
}
```

element() is constant time operation, it delegates peek(). T(n) = Θ(1)

```java
public boolean add( E item ) {
    if ( item == null )                              Θ(1)
        return false;

    if ( root == null ) {
        //create new root
        Node<E> temp = new Node<E>(item);                      Θ(1)
        addNodeRef = temp;
        removeNodeRef = temp;
        root = temp;
        temp.parent = null;
        size++;
        maxDepth++;
    }
    else {
        //check left
        if ( addNodeRef.left == null ) {
            Node<E> temp = new Node<E>(item);
            addNodeRef.left = temp;
            temp.parent = addNodeRef;
            if(isPerfect())                    Θ(1)
                maxDepth++;
            size++;
            updateNodes( temp );               O(log n)
        }
        // check right
        else if ( addNodeRef.right == null ) {
            Node<E> temp = new Node<E>(item);
            addNodeRef.right = temp;
            temp.parent = addNodeRef;
            size++;                                        O(n)
            updateNodes( temp );   Θ(log n)
            findNewAddNodeRef();   O(n)
        }
        findNewRemoveNodeRef();    O(n)
    }
    return true;
}
```

add() method is linear, if root is null it inserts in constant time, but after each insertion it has to find removeNode and addNode. These operations are performed in linear time. updateNodes() is not linear but finding removeNode makes operation definitely linear time.

**$T_b(n) = O(\log n) + O(n) = O(n)$, $T_w(n) = O(n) + O(n) = O(n)$**

**$T(n) = O(n)$** for add()

```
public boolean offer ( E item ) {
    return add(item);
}
```

offer() delegates add() so it is also, **$T(n) = O(n)$**

```
public E poll ( ) {
    if ( root == null )          O(1)
        return null;
    else {
        E temp = root.data;

        // check right
        if ( removeNodeRef != null && removeNodeRef.right != null ) {
            root.data = removeNodeRef.right.data;
            removeNodeRef.right = null;                                  O(1)
        }

        //check left
        else if ( removeNodeRef != null && removeNodeRef.left != null ) {
            root.data = removeNodeRef.left.data;
            removeNodeRef.left = null;
        }
        size--;
        if ( isPerfect() ) maxDepth--;
        if ( size == 0 ) {
            removeNodeRef = null;          O(1)
            addNodeRef = null;
            root = null;
            maxDepth = 0;
        }
        else {
            //update all values and heap.
            updateNodesFromRoot( (Node<E>) root );      O(logn)
            findNewRemoveNodeRef();
            findNewAddNodeRef();          O(n)
        }
        return temp;
    }
}
```

poll() method is linear, if root is null it instantly returns, makes null initialization to deleted node in constant time. But after the initialization, It updates the heap in **$2O(n) + \Theta(\log n)$** times inevitably.

So, **$T(n) = O(n)$** for poll() operation.

```java
public E remove( ) {
    return poll();
}
```

remove() delegates poll, so it is **T(n) = O(n)** also.

```java
public boolean isPerfect ( ) {
    return ( (size+1) & ( size )) == 0;
}
```

isPerfect() is constant time operation, **T(n) = Θ(1)**

```java
private void findNewAddNodeRef ( ) {
    if ( isPerfect() ) {
        Node<E> temp = ( Node<E> ) root;
        while ( temp.left != null ) {
            temp = ( Node<E> ) temp.left;
        }
        addNodeRef = temp;
    }
    else
        findNewAddNodeRef ( (Node<E>) root, 1 );    // search whole heap.
}
/**
 * Private helper method to arrange addNodeRef.
 * @param curDepth depth of current node
 * @param node node
 * @return true if it is arranged.
 */
private boolean findNewAddNodeRef ( Node<E> node, int curDepth ) {
    if( node == null || curDepth == maxDepth )
        return false;
    if ( curDepth + 1 == maxDepth ) {
        if ( (Node<E>) node.left == null ) {
            addNodeRef = node;
            return true;
        }
        else if ( (Node<E>) node.right == null ) {
            addNodeRef = node;
            return true;
        }
        return false;
    }
    else {
        boolean flag = findNewAddNodeRef( (Node<E>) node.left, curDepth + 1 );
        if( !flag )
            flag = findNewAddNodeRef( (Node<E>) node.right, curDepth + 1 );
        return flag;
    }
```

$\Theta(1)$

$\Theta(\log n)$

best

$O(n)$

$O(n)$

$\Theta(1)$

$O(n)$

$O(n)$

Finding new add node, for the best case **$T_b$(n) = Θ(logn)** because if it is perfect it goes only left, and tree is definitely complete. In the worst case it checks all elements one by one from start, so **$T_w$(n) = O(n).** At the end, **T (n) = O(n) + Θ(logn), so T(n) = O(n) and T(n) = Ω(logn)** for this private method.

```java
private void findNewRemoveNodeRef ( ) {
    if ( isPerfect() ) {                                  → Θ(1)
        Node<E> temp = ( Node<E> ) root;
        while ( temp.right != null ) {
            temp = ( Node<E> ) temp.right;    } → Θ(logn)   } best
        }
        removeNodeRef = temp.parent;
    }
    else
        findNewRemoveNodeRef( (Node<E>)root, 1 );   // search all heap.   → O(n)
}

/**
 * Helper private method to arrange new removeNodeRef recursively.
 * @param node node
 * @param curDepth depth of current node
 */
private boolean findNewRemoveNodeRef ( Node<E> node, int curDepth ) {
    if( node == null || curDepth == maxDepth )
        return false;
    if ( curDepth + 1 == maxDepth ) {
        if ( (Node<E>) node.right != null ) {
            removeNodeRef = node;
            return true;
        }                                                } Θ(1)
        else if ( (Node<E>) node.left != null ) {
            removeNodeRef = node;
            return true;
        }
        return false;
    }
    else {
        boolean flag = findNewRemoveNodeRef( (Node<E>) node.right, curDepth + 1 );  → O(n)
        if( !flag )
            flag = findNewRemoveNodeRef( (Node<E>) node.left, curDepth + 1 );  →
        return flag;
    }
```

Finding remove node is pretty similar to finding add node. They have same best and worst cases, so **T(n) = O(n)** for this private method.

```java
private void updateNodes ( Node<E> cur ) {
    if ( cur == null || cur.parent == null )
        return;
    int result = compare(cur.data, cur.parent.data);   } Θ(1)

    if ( result < 0 ) {
        E temp = cur.data;
        cur.data = cur.parent.data;          } Θ(1)
        cur.parent.data = temp;
    }
    updateNodes( cur.parent );  // go up  → Θ(logn)
}
```

Updating node from below is logarithmic because it depends on the height of the tree. It is definitely complete so algorithm is logarithmic.

**T(n) = Θ(logn)** for updateNodes()

```java
private void updateNodesFromRoot ( Node<E> cur ) {

    if ( cur == null || (cur.right == null && cur.left == null) )      } Θ(1)
        return;

    int result = 0;

    if ( cur.right == null && cur.left != null ) {
        result = 1;
    }

    else if ( cur.left == null && cur.right != null ) {                Θ(1)
        result = -1;
    }
    else
        result = compare(cur.right.data, cur.left.data);

    if ( result >= 0
            && ( compare(cur.data, cur.left.data) > 0 ) ) {

        E temp = cur.data;
        cur.data = cur.left.data;     } → Θ(1)
        cur.left.data = temp;
        updateNodesFromRoot( (Node<E>) cur.left );     → Θ(logn)
    }
    else if ( result <= 0
            && ( compare(cur.data, cur.right.data) > 0 ) ) {

        E temp = cur.data;
        cur.data = cur.right.data;    } Θ(1)
        cur.right.data = temp;
        updateNodesFromRoot( (Node<E>) cur.right );    → Θ(logn)
    }
}
```

Updating node from above is also logarithmic, because it depends on the height of the tree, and tree is definitely complete.

**T(n) = Θ(logn)** for updateNodesFromRoot()

```java
public String toString ( ) {
    StringBuilder sb = new StringBuilder();
    preOrderTraverse(root, 1, sb);
    return sb.toString();
}
```

toString() method makes preorder traverse, if Tree is regular binaryTree, complexity will be **T(n) = Θ(n²).** For the heap **T(n) = Θ(nlogn)**. Because heap is a complete tree, It is not possible to create odd trees with heap.

**T(n) = Θ(n²)** for regular binaryTree toString() method.

**T(n) = Θ(nlogn)** for binary heap's toString() method.

```java
private void preOrderTraverse ( Node<E> node, int depth, StringBuilder sb ) {
    for( int i = 1; i < depth; i++ )
        sb.append("  ");

    if(node == null)
        sb.append("null\n");
    else {
        sb.append(node.toString());
        sb.append("\n");
        preOrderTraverse(node.left, depth + 1, sb);
        preOrderTraverse(node.right, depth + 1, sb);
    }
}
```

Θ(nlogn) for heap

Θ(n)

Traversing preorderly, printing indentation decides the time complexity. For the heap it prints nlogn times. For the regular binaryTree it prints $n^2$ times in the worst case. Because heap is complete tree.

**T(n) = Θ($n^2$)** for regular binaryTree preOrderTraverse() method.

**T(n) = Θ(nlogn)** for binary heap's preOrderTraverse() method.

```java
public boolean increment ( E key, E val ) {

    if ( root == null || key == null || val == null)
        return false;

    Node<E> node = find( (Node<E>) root, key);
    if ( node == null )
        return false;

    int result = compare(val, key);
    if ( result < 0 )
        return false;

    node.data = val;
    updateNodesFromRoot(node);
    return true;
}
```

Θ(1)

O(n)

Θ(1)

Θ(logn)

Incrementing a key value, find methods takes linear time in the worst case, updating nodes takes logarithmic time but it does not affect the average case.
**T(n) = O(n) + Θ(logn) = O(n)** for increment()

```java
protected Node<E> find ( Node<E> node, E key ) {
    if ( node == null )
        return null;
    if ( node.data.equals(key) )
        return node;

    Node<E> retval = find( (Node<E>) node.left, key );
    if ( retval == null )
        retval = find( (Node<E>) node.right, key );

    return retval;
}
```

$\Theta(1)$

$\Theta(1) \rightarrow$ best

$\Theta(n)$ worst

Finding a key value takes linear time in the worst case, best case is constant because first element could be key, other best case is it could be the left side of the tree so search operation could be logarithmic. But at the end it traverses whole tree. **$T_b(n) = \Theta(1)$, $T_w(n) = \Theta(n)$.**

**$T(n) = O(n)$** for find()

```java
public LinkedBinaryHeap<E> merge ( LinkedBinaryHeap<E> other ) {
    if ( other != null )
        merge ( (Node<E>) other.root );
    return this;
}

/**
 * Helper private method to merge two trees.
 * @param node node
 */
private void merge ( Node<E> node ) {
    if ( node != null ) {
        add(node.data);
        merge((Node<E>) node.left);
        merge((Node<E>) node.right);
    }
}
```

$\rightarrow \Theta(nm+m^2)$

$nm + \frac{n*(m-1)}{2} \rightarrow \Theta(nm+m^2)$

Merging two tree, m indicates the other's size. In each recursive call add operation will take (n+k) times. In first call k is 0. Last call k is m-1. So k = 0 + m-1. k = m*(m-1)/2. So we can say Θ(n+m^2) for add. These method happens for every m element, therefore k*m is required. Exact complexity will become Θ(n*m + (n*m-1)/2 ).

**$T(n) = \Theta(n*m + m^2)$** for merge().

## Q4

```
public boolean add ( E item ) {
    if ( item == null )
        return false;                          } Θ(1)

    if ( theData[0] == null ) {
        theData[0] = item;
        return true;
    }
    return add(item, 0);   → O(n), Ω(logn)
}
private boolean add( E item, int cur ) {

    //reallocate till cur can be accessible
    while ( cur >= capacity )                  } Θ(n)        Θ(logn) → best
        reallocate();

    int left = 2*cur + 1;                      } Θ(1)        Θ(n) → worst
    int right = 2*cur + 2;

    // insert location is found
    if ( theData[cur] == null ) {
        theData[cur] = item;

        while ( left >= capacity )             } Θ(n)
            reallocate();
        while ( right >= capacity )
            reallocate();

        // insert childs
        theData[left] = null;                  } Θ(1)
        theData[right] = null;
        return true;
    }

    int result = ( (Comparable<E>) item).compareTo(theData[cur]);   } Θ(1)

    if ( result == 0 )
        return false;
    else if ( result < 0 )
        return add(item, left);     // go left    } O(n)
    else
        return add(item, right);    // go right
}
```

Adding an element to tree, in private helper method, best case is the case which has no reallocation. There is a possibility that no reallocation happens. In this case insertion takes logarithmic times. Worst case is the case when reallocation happens. Reallocation has linear time complexity, therefore time complexity of insertion becomes linear in the worst case. It is not amortised because, insertion in each node has another insertion like 2*n+1 and 2*n+2.

**T(n) = O(n) and T(n) = Ω(logn)** for add()

```
public E find( E target ) {
    if ( theData[0] == null )          → θ(1)
        return null;
    return find(target,0);             → O(n)
}

private E find( E target, int cur ) {
    if ( cur >= capacity || theData[cur] == null )     → O(1)
        return null;

    int left = 2*cur + 1;
    int right = 2*cur + 2;                                            } O(1)
    int result = ((Comparable<E>) target).compareTo ( theData[cur] );

    if ( result == 0 )
        return theData[cur];
    else if ( result < 0 )
        return find(target, left);      // go left    } θ(n)    h_av = θ(logn)
    else
        return find(target, right);     // go right
}
```

Searching in the tree, It is same with node link structured search tree because there is no reallocation. Worst case is linear because there is no guarantee that tree is complete or balanced. But average case is logarithmic. Because it goes left or right in each search for average cases. Best case is constant it could be the first element.

**$T_b$ (n)= ϴ(1), $T_{av}$ = ϴ(logn), $T_w$(n) = ϴ(n)**

**T(n) = O(n)** for find()

```java
public boolean remove ( E target ) {

    if ( target == null || theData[0] == null )
        return false;

    return remove(target, 0);        → O(n)
}
private boolean remove ( E target, int cur ) {
    if ( cur >= capacity || theData[cur] == null )        } θ(1)
        return false;

    int left = 2*cur + 1;
    int right = 2*cur + 2;
    int result = ((Comparable<E>) target).compareTo ( theData[cur] );        } θ(1)

    if ( result < 0 )
        return remove(target, left);        b → θ(1)

    else if ( result > 0 )
        return remove(target, right);        w → θ(n)

    else {  // if target is found
        // if it has no child
        if ( theData[left] == null && theData[right] == null )        } θ(1)   av → θ(logn)
            theData[cur] = null;

        // if it has only right child
        else if ( theData[left] == null )
            theData[cur] = findLeftMostRemove(right);        → O(n)

        // if it has only left child
        else if ( theData[right] == null )
            theData[cur] = findRightMostRemove(left);        → O(n)

        // if it has both child
        else
            theData[cur] = findRightMostRemove(left);        → O(n)
        return true;
    }
}
```

For removing from tree, In the best case it takes constant time, at worst case which is an odd tree, it takes linear time, because it traverses all elements. At average cases, it takes logarithmic time. Leftmost and rightmost removals are O(n), but these 3 methods are works together, in remove operation, It does not traverse whole array in leftmost and rightmost methods, continues from where it left. Sometimes these methods can call remove method also for special cases, but it does not affect the complexity from being linear.

The worst exceptional case in removal is, when rightmost child of left child has left child and no right child or leftmost child of right child has right child and no left child. This case is handled by these rightmost leftmost methods. Current elements are removed and returned from these methods, and they fill their places with calling remove method. Binary Search Tree's rule is protected with this operation.

**$T_b$ (n)= Θ(1), $T_{av}$ = Θ(logn), $T_w$(n) = Θ(n)**

**T(n) = O(n)** for remove()

```
private E findLeftMostRemove ( int cur ) {

    if ( cur >= capacity || theData[cur] == null )     } Θ(1)
        return null;

    int left = 2*cur + 1;      } Θ(1)
    int right = 2*cur + 2;

    if ( theData[left] == null ) {                                  w → Θ(n)
                                                                    av → Θ(logn)
        // if left and right is null remove data and return.
        if ( theData[right] == null ) {                             b → Θ(1)
            E target = theData[cur];      } Θ(1)
            theData[cur] = null;
            return target;
        }

        // if left is null right is not null, remove current data and return.
        else {
            E target = theData[cur];
            remove(theData[cur], cur);     → Θ(n)
            return target;
        }
    }
    // if left is not null go left
    else
        return findLeftMostRemove( left );
}
private E findRightMostRemove ( int cur ) {
    if ( cur >= capacity || theData[cur] == null )     } Θ(1)
        return null;

    int left = 2*cur + 1;      } Θ(1)
    int right = 2*cur + 2;

    if ( theData[right] == null ) {                                 w → Θ(n)

        // if left and right is null remove data and return.
        if ( theData[left] == null ) {                              av → Θ(logn)
            E target = theData[cur];      } Θ(1)
            theData[cur] = null;
            return target;
        }

        // if right is null, left is not null, remove current data, and return.    b → Θ(1)
        else {
            E target = theData[cur];
            remove(theData[cur], cur);     → Θ(n)
            return target;
        }
    }
    // if right is not null go right
    else
        return findLeftMostRemove( right );
}
```

Finding leftmost and rightmost elements, it takes constant time in the best case. These two methods are very similar, so they can be analyzed together. Both of them in the worst case finds their most(left or right) and remove in constant time, but sometimes their opposite direction are not null and their direction is null, in these case it returns the current target and removes it from tree, because it is the value that will put to removed value. This operation is averagely logarithmic, but there are cases that tree is not balanced. Because of that worst case is linear. But when it works with remove method, remove method takes linear time too, because current index is transmitted to these methods. **T(n) = O(n)** for findLeftMostRemove() and findRightMostRemove().

```java
public String toString ( ) {
    StringBuilder sb = new StringBuilder();
    preOrderTraverse(0, 1, sb);          → O(n^2)
    return sb.toString();
}

/**
 * Helper private method for toString.
 * @param cur current index
 * @param depth current depth
 * @param sb returned string.
 */
private void preOrderTraverse ( int cur, int depth, StringBuilder sb ) {
    for( int i = 1; i < depth; i++ )     } →  av → Θ(nlogn)
        sb.append("  ");                       w → Θ(n^2)

    if ( cur >= capacity )
        return;
                                         } Θ(1)
    int left = 2*cur+1;
    int right = 2*cur+2;

    if(theData[cur] == null)
        sb.append("null\n");
    else {                               } Θ(n)
        sb.append(theData[cur].toString());
        sb.append("\n");
        preOrderTraverse(left, depth + 1, sb);
        preOrderTraverse(right, depth + 1, sb);
    }
}
```

toString method prints tree in preorder traversal. It traverses the array without printing the depth in logn times or n times. But it does not matter because printing depth decides the complexity of this method. In the worst case tree is not balanced, so it takes quadratic times. In average cases it takes nlogn times. Because loop will work for every element, and in each time it iterates maximum of trees length, so it takes nlogn times in average cases.

$T(n) = O(n^2)$ , $T_{av}(n) = O(nlogn)$

## 5. TEST CASES

```java
public static void testLinkedBinaryHeap ( ) {

    System.out.println ( "___Testing LinkedBinaryHeap_________________________" );
    System.out.println( "create LinkedBinaryHeap\n" );
    LinkedBinaryHeap<Integer> lbh1 = new LinkedBinaryHeap<Integer>();
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");

    System.out.println ( "\nadd some Integer's to lbh1" );

    lbh1.add(Integer.valueOf(22));
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");

    System.out.println("lbh1.isLeaf() returns = " + lbh1.isLeaf());

    lbh1.add(Integer.valueOf(15));
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");

    System.out.println("lbh1.isLeaf() returns = " + lbh1.isLeaf());

    lbh1.add(Integer.valueOf(25));
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
    lbh1.offer(Integer.valueOf(47));
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
    lbh1.add(Integer.valueOf(11));
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
    lbh1.offer(Integer.valueOf(52));
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");

    System.out.println( "\nusing poll(remove) 3 times" );
    lbh1.poll();
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
    lbh1.poll();
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
    lbh1.poll();
    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
```

```java
    System.out.println( "\nadd much more values" );
    lbh1.add(Integer.valueOf(44));
    lbh1.add(Integer.valueOf(5));
    lbh1.add(Integer.valueOf(44));
    lbh1.add(Integer.valueOf(24));
    lbh1.add(Integer.valueOf(19));
    lbh1.add(Integer.valueOf(17));
    lbh1.add(Integer.valueOf(31));
    lbh1.add(Integer.valueOf(2));
    lbh1.add(Integer.valueOf(65));

    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");
    System.out.println( "\ncontinue adding" );

    lbh1.add(Integer.valueOf(16));
    lbh1.add(Integer.valueOf(42));
    lbh1.add(Integer.valueOf(123));
    lbh1.add(Integer.valueOf(6));
    lbh1.add(Integer.valueOf(22));

    lbh1.add(Integer.valueOf(13));
    lbh1.add(Integer.valueOf(21));
    lbh1.add(Integer.valueOf(56));
    lbh1.add(Integer.valueOf(35));
    lbh1.add(Integer.valueOf(1));

    System.out.println(lbh1 + "---------- size = " + lbh1.size() + "\n");


    System.out.println("Getting left sub tree");
    BinaryTree<Integer> lbh2 = lbh1.getLeftSubtree();
    System.out.println("\nlbh2(left sub tree of lbh1)\n" + lbh2 + "----------\n");
    System.out.println("right sub tree of lbh1\n" + lbh1.getRightSubtree() + "----------\n");
```

```java
System.out.println( "Remove all nodes from lbh1(after removing all try remove more for error handling)\n" );

int treeSize = lbh1.size();
for(int i = 0; i < treeSize; ++i)
    lbh1.poll();
lbh1.poll();
lbh1.poll();
System.out.println(lbh1 + "--------- size = " + lbh1.size() + "\n");

System.out.println( "Add new nodes to lbh1" );

lbh1.add(Integer.valueOf(25));
lbh1.add(Integer.valueOf(47));
lbh1.add(Integer.valueOf(11));
lbh1.add(Integer.valueOf(52));
lbh1.add(Integer.valueOf(44));
lbh1.add(Integer.valueOf(1));
lbh1.add(Integer.valueOf(18));
lbh1.add(Integer.valueOf(152));
lbh1.add(Integer.valueOf(4));
lbh1.add(Integer.valueOf(61));
lbh1.add(Integer.valueOf(23));

System.out.println(lbh1 + "--------- size = " + lbh1.size() + "\n");

System.out.println("\nCreate lbh3 and add some nodes");
LinkedBinaryHeap<Integer> lbh3 = new LinkedBinaryHeap<Integer>();
lbh3.add(Integer.valueOf(7));
lbh3.add(Integer.valueOf(9));
lbh3.add(Integer.valueOf(15));
lbh3.add(Integer.valueOf(2));
lbh3.add(Integer.valueOf(31));
lbh3.add(Integer.valueOf(8));

System.out.println("\nlbh3\n" + lbh3 + "--------- size = " + lbh3.size() + "\n");
```

```java
System.out.println( "Merge lbh1 and lbh3 using lbh1.merge(lbh3)\n" );
lbh1.merge(lbh3);
System.out.println("lbh1(after merging)\n" + lbh1 + "--------- size = " + lbh1.size() + "\n");

System.out.println("\nAdd some values to merged tree");
lbh1.add(Integer.valueOf(41));
lbh1.add(Integer.valueOf(68));
lbh1.add(Integer.valueOf(54));
lbh1.add(Integer.valueOf(33));
lbh1.add(Integer.valueOf(23));
lbh1.add(Integer.valueOf(5));

System.out.println("lbh1\n" + lbh1 + "--------- size = " + lbh1.size() + "\n");

System.out.println("\nlbh3\n" + lbh3 + "---------");
System.out.println("\nIncrement 7 with 14");
lbh3.increment(Integer.valueOf(7), Integer.valueOf(14));
System.out.println("lbh3\n" + lbh3 + "---------");

try {
    System.out.println("lbh3.peek() returns = " + lbh3.peek());
    System.out.println("lbh3.element() returns = " + lbh3.element() + "\n");
}
catch(Exception e) {
    System.out.println(e);
}
```

```java
System.out.println("Check LinkedBinaryHeap with Strings");
LinkedBinaryHeap<String> lbh4 = new LinkedBinaryHeap<String>();

System.out.println("add some strings to lbh4\n");
lbh4.add("car");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("aisle");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("chrome");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("burak");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("zone");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");

System.out.println("poll(remove) from lbh4");
lbh4.poll();
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
System.out.println("Add more values to lbh4");

lbh4.offer("run");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("anchor");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("time");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
lbh4.add("beat");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");

System.out.println("\nIncrement burak with joy");
lbh4.increment("burak", "joy");
System.out.println("lbh4\n" + lbh4 + "---------");

System.out.println("Check if tree is perfect = " + lbh4.isPerfect());
```

```java
lbh4.poll();
System.out.println("Poll from tree\n");
System.out.println(lbh4 + "---------- size = " + lbh4.size() + "\n");
System.out.println("Check if tree is perfect = " + lbh4.isPerfect());
```

```java
public static void testBinarySearchTree ( ) {

    System.out.println ( "___Testing BinarySearchTree_________________________" );

    System.out.println ( "\nCreate new Integer BinarySearchTree and assign it to SearchTree reference.\n" );
    SearchTree<Integer> bst1 = new BinarySearchTree<Integer>();

    System.out.println("bst1\n" + bst1 + "----------\n");

    System.out.println( "Adding values to the bst1\n" );
    bst1.add(Integer.valueOf(25));
    System.out.println(bst1 + "----------\n");
    bst1.add(Integer.valueOf(22));
    System.out.println(bst1 + "----------\n");
    bst1.add(Integer.valueOf(7));
    System.out.println(bst1 + "----------\n");
    bst1.add(Integer.valueOf(39));
    System.out.println(bst1 + "----------\n");
    bst1.add(Integer.valueOf(18));
    System.out.println(bst1 + "----------\n");

    System.out.println("Removing 18\n" + "after removing:\n");
    bst1.delete(Integer.valueOf(18));
    System.out.println(bst1 + "----------\n");
    System.out.println("Removing 22\n" + "after removing:\n");
    bst1.delete(Integer.valueOf(22));
    System.out.println(bst1 + "----------\n");
    System.out.println("Removing 25\n" + "after removing:\n");
    bst1.delete(Integer.valueOf(25));
    System.out.println(bst1 + "----------\n");
```

```java
    System.out.println("Removing 7 and 39 consecutively and remove 5 from empty tree(for testing error handling)\n");
    bst1.remove(Integer.valueOf(7));
    System.out.println(bst1 + "----------\n");

    bst1.remove(Integer.valueOf(39));
    bst1.remove(Integer.valueOf(5));
    System.out.println(bst1 + "----------\n");


    System.out.println( "Adding values again to the bst1\n" );
    bst1.add(Integer.valueOf(25));
    bst1.add(Integer.valueOf(22));
    bst1.add(Integer.valueOf(7));
    bst1.add(Integer.valueOf(39));
    bst1.add(Integer.valueOf(18));
    System.out.println(bst1 + "----------\n");

    System.out.println( "Trying to add same value to bst1 (18)\n" );
    bst1.add(Integer.valueOf(18));
    System.out.println(bst1 + "----------\n");

    System.out.println( "Trying to remove unexisted value from bst1 (3)\n" );
    bst1.remove(Integer.valueOf(3));
    System.out.println(bst1 + "----------\n");

    System.out.println( "bst1.find(25) returns = " + bst1.find(25) + "\n"
                      + "bst1.find(3) returns = " + bst1.find(3) + "\n" );
    System.out.println( "bst1.contains(7) returns = " + bst1.contains(7) + "\n"
                      + "bst1.contains(1) returns = " + bst1.contains(1) + "\n" );
```

```java
System.out.println( "Adding more data to bst1\n" );
bst1.add(Integer.valueOf(3));
bst1.add(Integer.valueOf(33));
bst1.add(Integer.valueOf(38));
bst1.add(Integer.valueOf(35));
bst1.add(Integer.valueOf(47));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 25\n" + "after removing:\n");
bst1.delete(Integer.valueOf(25));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 39\n" + "after removing:\n");
bst1.delete(Integer.valueOf(39));
System.out.println(bst1 + "----------\n");


System.out.println( "Adding more data to bst1\n" );

bst1.add(Integer.valueOf(67));
bst1.add(Integer.valueOf(40));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 22\n" + "after removing:\n");
bst1.delete(Integer.valueOf(22));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 35\n" + "after removing:\n");
bst1.delete(Integer.valueOf(35));
System.out.println(bst1 + "----------\n");
```

```java
System.out.println( "Adding more data to bst1\n" );

bst1.add(Integer.valueOf(12));
bst1.add(Integer.valueOf(5));
bst1.add(Integer.valueOf(17));
bst1.add(Integer.valueOf(52));
bst1.add(Integer.valueOf(25));
System.out.println(bst1 + "----------\n");


System.out.println("Removing 67\n" + "after removing:\n");
bst1.remove(Integer.valueOf(67));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 40\n" + "after removing:\n");
bst1.remove(Integer.valueOf(40));
System.out.println(bst1 + "----------\n");


System.out.println( "Adding more data to bst1\n" );
bst1.add(Integer.valueOf(1));
bst1.add(Integer.valueOf(36));
bst1.add(Integer.valueOf(75));
bst1.add(Integer.valueOf(48));
bst1.add(Integer.valueOf(8));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 33\n" + "after removing:\n");
bst1.remove(Integer.valueOf(33));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 7\n" + "after removing:\n");
bst1.remove(Integer.valueOf(7));
System.out.println(bst1 + "----------\n");
```

```java
System.out.println( "bst1.find(75) returns = " + bst1.find(75) + "\n"
                    + "bst1.find(1) returns = " + bst1.find(1) + "\n" );
System.out.println( "bst1.contains(47) returns = " + bst1.contains(47) + "\n"
                    + "bst1.contains(36) returns = " + bst1.contains(36) + "\n" );

System.out.println("Removing 18\n" + "after removing:\n");
bst1.remove(Integer.valueOf(18));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 25\n" + "after removing:\n");
bst1.remove(Integer.valueOf(25));
System.out.println(bst1 + "----------\n");

System.out.println("Removing 38\n" + "after removing:\n");
bst1.remove(Integer.valueOf(38));
System.out.println(bst1 + "----------\n");

System.out.println ( "\nCreate new String BinarySearchTree and assign it to SearchTree reference.\n" );
SearchTree<String> bst2 = new BinarySearchTree<String>("laughter");

System.out.println("bst2\n" + bst2 + "----------\n");
```

```java
System.out.println("Add some datas to bst2\n");

bst2.add("game");
bst2.add("power");
bst2.add("climax");
bst2.add("motivation");
bst2.add("joy");
bst2.add("inside");
bst2.add("day");
bst2.add("bark");
bst2.add("shore");
bst2.add("throw");
bst2.add("possession");
bst2.add("titane");
bst2.add("assure");
bst2.add("architecture");
bst2.add("assemble");
bst2.add("cocteau");
bst2.add("feasible");
bst2.add("inline");
bst2.add("danger");
bst2.add("soaked");
bst2.add("kind");
bst2.add("foggy");
bst2.add("law");
bst2.add("queen");
bst2.add("xymox");
System.out.println(bst2 + "----------\n");

System.out.println( "bst2.find(\"law\") returns = " + bst2.find("law") + "\n"
                    + "bst2.find(\"danger\") returns = " + bst2.find("danger") + "\n" );
System.out.println( "bst2.contains(\"shore\") returns = " + bst2.contains("shore") + "\n"
                    + "bst2.contains(\"emily\") returns = " + bst2.contains("emily") + "\n" );
```

```java
System.out.println("Removing titane\n" + "after removing:\n");
bst2.remove("titane");
System.out.println(bst2 + "----------\n");

System.out.println("Removing bark\n" + "after removing:\n");
bst2.remove("bark");
System.out.println(bst2 + "----------\n");

System.out.println("Removing laughter\n" + "after removing:\n");
bst2.remove("laughter");
System.out.println(bst2 + "----------\n");
```

## 6. RUNNING AND RESULTS

```
___Testing LinkedBinaryHeap_________________________
create LinkedBinaryHeap

null
---------- size = 0


add some Integer's to lbh1
22
  null
  null
---------- size = 1

lbh1.isLeaf() returns = true
15
  22
    null
    null
  null
---------- size = 2

lbh1.isLeaf() returns = false
15
  22
    null
    null
  25
    null
    null
---------- size = 3

15
  22
    47
      null
      null
    null
  25
    null
    null
---------- size = 4

11
  15
    47
      null
      null
    22
      null
      null
  25
    null
    null
---------- size = 5

11
  15
    47
      null
      null
    22
      null
      null
  25
    52
      null
      null
    null
---------- size = 6
```

```
using poll(remove) 3 times
15
  22
    47
      null
      null
    52
      null
      null
  25
    null
    null
---------- size = 5

22
  47
    52
      null
      null
    null
  25
    null
    null
---------- size = 4

25
  47
    null
    null
  52
    null
    null
---------- size = 3
```

```
continue adding
1
  2
    6
      19
        47
          null
          null
        22
          null
          null
      13
        25
          null
          null
        21
          null
          null
    5
      35
        56
          null
          null
        44
          null
          null
      17
        31
          null
          null
        null
  16
    24
      65
        null
        null
      52
        null
        null
    42
      44
        null
        null
      123
        null
        null
---------- size = 22
```

```
Getting left sub tree

lbh2(left sub tree of lbh1)
2
  6
    19
      47
        null
        null
      22
        null
        null
    13
      25
        null
        null
      21
        null
        null
  5
    35
      56
        null
        null
      44
        null
        null
    17
      31
        null
        null
      null
----------
```

```
right sub tree of lbh1
16
  24
    65
      null
      null
    52
      null
      null
  42
    44
      null
      null
    123
      null
      null
----------

Remove all nodes from lbh1(after removing all try remove more for error handling)

null
---------- size = 0
```

```
Add new nodes to lbh1
1
  4
    44
      152
        null
        null
      52
        null
        null
    23
      61
        null
        null
      47
        null
        null
  11
    25
      null
      null
    18
      null
      null
---------- size = 11


Create lbh3 and add some nodes

lbh3
2
  7
    9
      null
      null
    31
      null
      null
  8
    15
      null
      null
    null
--------- size = 6
```

```
Merge lbh1 and lbh3 using lbh1.merge(lbh3)

lbh1(after merging)
1
  4
    8
      15
        152
          null
          null
        44
          null
          null
      52
        null
        null
    23
      61
        null
        null
      47
        null
        null
  2
    7
      25
        null
        null
      11
        null
        null
    9
      18
        null
        null
      31
        null
        null
--------- size = 17
```

```
Add some values to merged tree
lbh1
1
  4
    8
      15
        152
          null
          null
        44
          null
          null
      41
        52
          null
          null
        68
          null
          null
    5
      33
        61
          null
          null
        54
          null
          null
      23
        47
          null
          null
        23
          null
          null
  2
    7
      25
        null
        null
      11
        null
        null
    9
      18
        null
        null
      31
        null
        null
--------- size = 23
```

```
lbh3
2
  7
    9
      null
      null
    31
      null
      null
  8
    15
      null
      null
    null
---------

Increment 7 with 14
lbh3
2
  9
    14
      null
      null
    31
      null
      null
  8
    15
      null
      null
    null
---------
lbh3.peek() returns = 2
lbh3.element() returns = 2
```

```
Check LinkedBinaryHeap with Strings
add some strings to lbh4

car
  null
  null
---------- size = 1

aisle
  car
    null
    null
  null
---------- size = 2

aisle
  car
    null
    null
  chrome
    null
    null
---------- size = 3

aisle
  burak
    car
      null
      null
    null
  chrome
    null
    null
---------- size = 4

aisle
  burak
    car
      null
      null
    zone
      null
      null
  chrome
    null
    null
---------- size = 5
```

```
poll(remove) from lbh4
burak
  car
    zone
      null
      null
    null
  chrome
    null
    null
---------- size = 4

Add more values to lbh4
burak
  car
    zone
      null
      null
    run
      null
      null
  chrome
    null
    null
---------- size = 5

anchor
  car
    zone
      null
      null
    run
      null
      null
  burak
    chrome
      null
      null
    null
---------- size = 6

anchor
  car
    zone
      null
      null
    run
      null
      null
  burak
    chrome
      null
      null
    time
      null
      null
---------- size = 7

anchor
  beat
    car
      zone
        null
        null
      null
    run
      null
      null
  burak
    chrome
      null
      null
    time
      null
      null
---------- size = 8
```

```
Increment burak with joy
lbh4
anchor
  beat
    car
      zone
        null
        null
      null
    run
      null
      null
  chrome
    joy
      null
      null
    time
      null
      null
---------
Check if tree is perfect = false
Poll from tree

beat
  car
    zone
      null
      null
    run
      null
      null
  chrome
    joy
      null
      null
    time
      null
      null
---------- size = 7

Check if tree is perfect = true
```

```
___Testing BinarySearchTree_________________________

Create new Integer BinarySearchTree and assign it to SearchTree reference.

bst1
null
----------

Adding values to the bst1

25
  null
  null
----------

25
  22
    null
    null
  null
----------

25
  22
    7
      null
      null
    null
  null
----------

25
  22
    7
      null
      null
    null
  39
    null
    null
----------
```

```
25
  22
    7
      null
      18
        null
        null
    null
  39
    null
    null
----------

Removing 18
after removing:

25
  22
    7
      null
      null
    null
  39
    null
    null
----------

Removing 22
after removing:

25
  7
    null
    null
  39
    null
    null
----------

Removing 25
after removing:

7
  null
  39
    null
    null
----------
```

```
Removing 7 and 39 consecutively and remove 5 from empty tree(for testing error handling)

39
  null
  null
----------

null
----------

Adding values again to the bst1

25
  22
    7
      null
      18
        null
        null
    null
  39
    null
    null
----------

Trying to add same value to bst1 (18)

25
  22
    7
      null
      18
        null
        null
    null
  39
    null
    null
----------
```

```
Trying to remove unexisted value from bst1 (3)

25
  22
    7
      null
      18
        null
        null
    null
  39
    null
    null
----------

bst1.find(25) returns = 25
bst1.find(3) returns = null

bst1.contains(7) returns = true
bst1.contains(1) returns = false
```

```
bst1.find(25) returns = 25
bst1.find(3) returns = null

bst1.contains(7) returns = true
bst1.contains(1) returns = false

Adding more data to bst1

25
  22
    7
      3
        null
        null
      18
        null
        null
    null
  39
    33
      null
      38
        35
          null
          null
        null
    47
      null
      null
----------
Removing 25
after removing:

22
  18
    7
      3
        null
        null
      null
    null
  39
    33
      null
      38
        35
          null
          null
        null
    47
      null
      null
----------
Removing 39
after removing:

22
  18
    7
      3
        null
        null
      null
    null
  35
    33
      null
      38
        null
        null
    47
      null
      null
----------
```

```
Adding more data to bst1

22
  18
    7
      3
        null
        null
      null
    null
  35
    33
      null
      38
        null
        null
    47
      40
        null
        null
      67
        null
        null
----------

Removing 22
after removing:

18
  7
    3
      null
      null
    null
  35
    33
      null
      38
        null
        null
    47
      40
        null
        null
      67
        null
        null
----------

Removing 35
after removing:

18
  7
    3
      null
      null
    null
  38
    33
      null
      null
    47
      40
        null
        null
      67
        null
        null
----------
```

```
Adding more data to bst1

18
  7
    3
      null
      5
        null
        null
    12
      null
      17
        null
        null
  38
    33
      25
        null
        null
      null
    47
      40
        null
        null
      67
        52
          null
          null
        null
----------
```

```
Removing 67
after removing:

18
  7
    3
      null
      5
        null
        null
    12
      null
      17
        null
        null
  38
    33
      25
        null
        null
      null
    47
      40
        null
        null
      52
        null
        null
----------
```

```
Removing 40
after removing:

18
  7
    3
      null
      5
        null
        null
    12
      null
      17
        null
        null
  38
    33
      25
        null
        null
      null
    47
      null
      52
        null
        null
----------
```

```
Adding more data to bst1

18
  7
    3
      1
        null
        null
      5
        null
        null
    12
      8
        null
        null
      17
        null
        null
  38
    33
      25
        null
        null
      36
        null
        null
    47
      null
      52
        48
          null
          null
        75
          null
          null
----------
```

```
Removing 33
after removing:

18
  7
    3
      1
        null
        null
      5
        null
        null
    12
      8
        null
        null
      17
        null
        null
  38
    25
      null
      36
        null
        null
    47
      null
      52
        48
          null
          null
        75
          null
          null
----------

Removing 7
after removing:

18
  5
    3
      1
        null
        null
      null
    12
      8
        null
        null
      17
        null
        null
  38
    25
      null
      36
        null
        null
    47
      null
      52
        48
          null
          null
        75
          null
          null
----------

bst1.find(75) returns = 75
bst1.find(1) returns = 1

bst1.contains(47) returns = true
bst1.contains(36) returns = true
```

```
Removing 18
after removing:

8
  5
    3
      1
        null
        null
      null
    12
      null
      17
        null
        null
  38
    25
      null
      36
        null
        null
    47
      null
      52
        48
          null
          null
        75
          null
          null
----------
```

```
Removing 25
after removing:

8
  5
    3
      1
        null
        null
      null
    12
      null
      17
        null
        null
  38
    36
      null
      null
    47
      null
      52
        48
          null
          null
        75
          null
          null
----------
```

```
Removing 38
after removing:

8
  5
    3
      1
        null
        null
      null
    12
      null
      17
        null
        null
  36
    null
    47
      null
      52
        48
          null
          null
        75
          null
          null
----------


Create new String BinarySearchTree and assign it to SearchTree reference.

bst2
laughter
  null
  null
----------
```

```
Add some datas to bst2

laughter
  game
    climax
      bark
        assure
          architecture
            null
            assemble
              null
              null
          null
        null
      day
        cocteau
          null
          danger
            null
            null
        feasible
          null
          foggy
            null
            null
    joy
      inside
        inline
          null
          null
        null
      kind
        null
        null
  power
    motivation
      law
        null
        null
      possession
        null
        null
    shore
      queen
        null
        null
      throw
        soaked
          null
```

```
  power
    motivation
      law
        null
        null
      possession
        null
        null
    shore
      queen
        null
        null
      throw
        soaked
          null
          null
        titane
          null
          xymox
            null
            null
----------

bst2.find("law") returns = law
bst2.find("danger") returns = danger

bst2.contains("shore") returns = true
bst2.contains("emily") returns = false
```

```
Removing titane
after removing:

laughter
  game
    climax
      bark
        assure
          architecture
            null
            assemble
              null
              null
          null
        null
      day
        cocteau
          null
          danger
            null
            null
        feasible
          null
          foggy
            null
            null
    joy
      inside
        inline
          null
          null
        null
      kind
        null
        null
  power
    motivation
      law
        null
        null
      possession
        null
        null
```

```
        null
      kind
        null
        null
  power
    motivation
      law
        null
        null
      possession
        null
        null
    shore
      queen
        null
        null
      throw
        soaked
          null
          null
        xymox
          null
          null
----------
```

```
Removing bark
after removing:

laughter
  game
    climax
      assure
        assemble
          architecture
            null
            null
          null
        null
      day
        cocteau
          null
          danger
            null
            null
        feasible
          null
          foggy
            null
            null
    joy
      inside
        inline
          null
          null
        null
      kind
        null
        null
  power
    motivation
      law
        null
        null
      possession
        null
        null
    shore
      queen
        null
        null
      throw
        soaked
          null
```

```
        null
  power
    motivation
      law
        null
        null
      possession
        null
        null
    shore
      queen
        null
        null
      throw
        soaked
          null
          null
        xymox
          null
          null
----------
```

```
Removing laughter
after removing:

inline
  game
    climax
      assure
        assemble
          architecture
            null
            null
          null
        null
      day
        cocteau
          null
          danger
            null
            null
        feasible
          null
          foggy
            null
            null
    joy
      inside
        null
        null
      kind
        null
        null
  power
    motivation
      law
        null
        null
      possession
        null
        null
    shore
      queen
        null
        null
      throw
        soaked
          null
          null
        xymox
          null
          null
```

“make” command compiles, and “make run” command runs the program.