

GIT Department of Computer Engineering

CSE 222/505 - Spring 2022

Homework 5 Report

Abdurrahman BULUT

1901042258

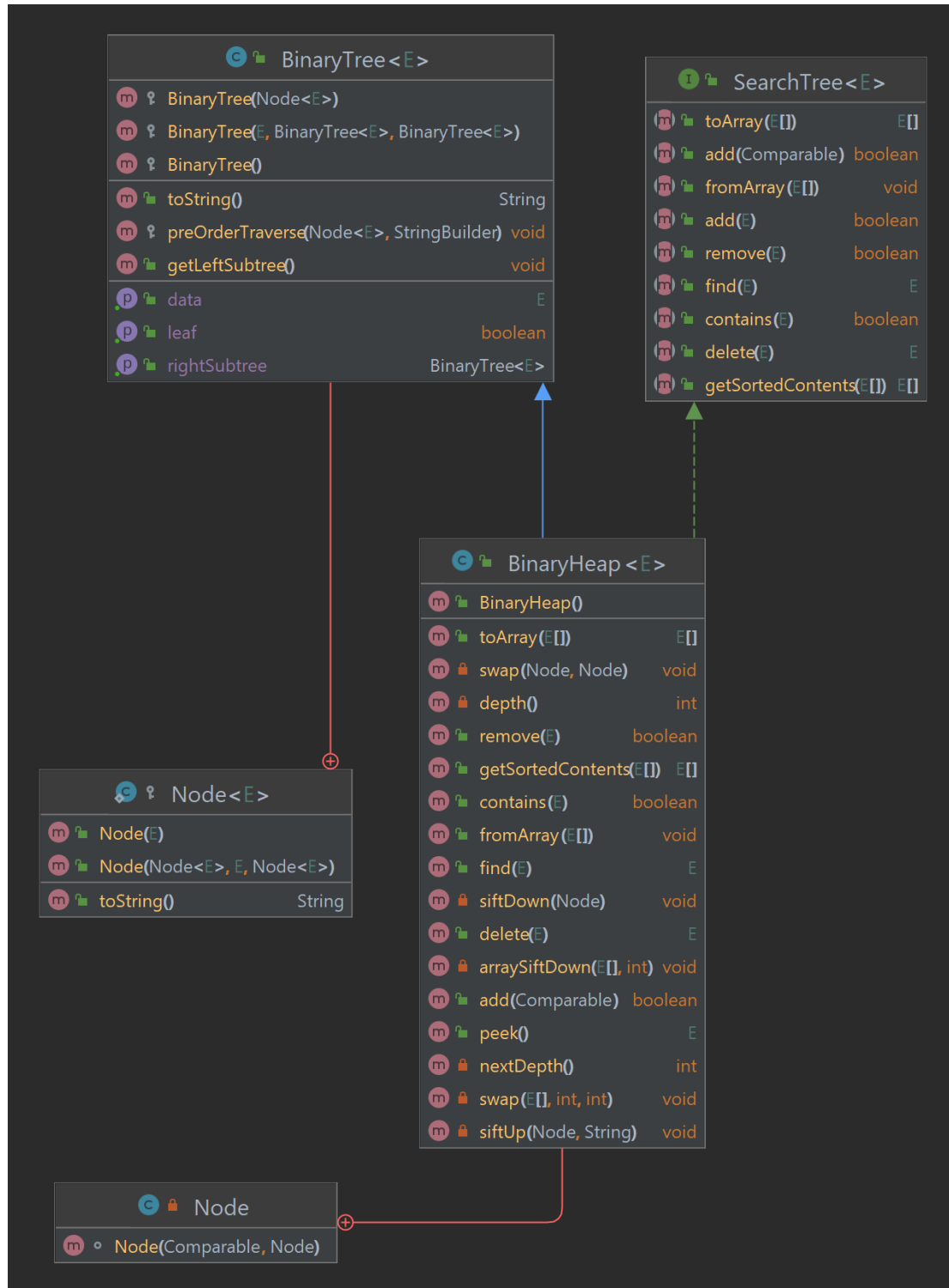
1. SYSTEM REQUIREMENTS

❖ Functional Requirements

➤ System

- openjdk version "11.0.14.1" 2022-02-08 LTS
- OpenJDK Runtime Environment Corretto-11.0.14.10.1 (build 11.0.14.1+10-LTS)
- OpenJDK 64-Bit Server VM Corretto-11.0.14.10.1 (build 11.0.14.1+10-LTS, mixed mode)

2. USE CASE AND CLASS DIAGRAMS



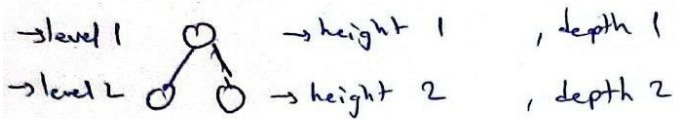
3. PROBLEM SOLUTION APPROACH

For question 2, I drew relations with arrows. I have put data values to table and inserted to binary tree. For question 3, I wrote a search Tree java file, a binary Tree file and binary Heap file. Binary heap class implements search Tree class. Most of the time Binary heap can be written with array. But we are asked to write using nodes. So, I created a node class. To go left and right, I used $2n+1$ and $2n+2$ formulas respectively to traverse. For question 4, I used an array to represent. Again, to move left side of the tree $2n+1$ is used. To move right side of the tree $2n+2$ is used.

4. Question 1

- a) Calculate the total depth of the nodes in a complete binary tree of height h . Note that total depth is 5 if height is 2 where the depth of the root is one and there are two nodes with depth 2.

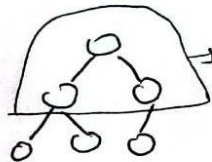
As example



So, Total depth is $1+2+2=5$ for this example.

A complete binary tree is a perfect binary tree except the last level.

for example :



this is a perfect BT.
and, $2^n - 1$ number of Nodes.

For Perfect BT Part;

$$\begin{array}{c} 1, 1 + 2, 2 + 3, 4 + 4, 8 \dots \\ \swarrow \quad \searrow \\ \text{height} \quad \text{numbers} \\ \quad \quad \text{of} \\ \quad \quad \text{nodes} \end{array}$$

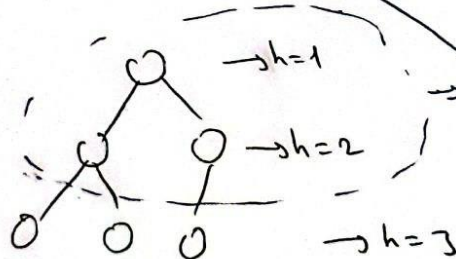
$$\Rightarrow \sum_{n=0}^h (n+1) \cdot 2^n, \quad n=h-1$$

\Rightarrow Depends on the last level of tree,
The total Depth will be;

P: Number of nodes on the last level

$$\underbrace{\left[\sum_{h=1}^h (h) \cdot 2^{h-1} \right]}_{\text{if it is perfect BT.}} - \underbrace{\left[(2^{h-1} - p), (h-1) \right]}_{\text{Number of nodes on the last level}}$$

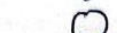
Example



\rightarrow It is always perfect BT

-

→ 1 comparison



```
graph TD; 0((0)) --- 1((1)); 0 --- 2((2)); 1 --- 3((3)); 1 --- 4((4)); 2 --- 5((5));
```

so; First we should calculate perfect BT Part ($h-1$)

p : Number of nodes on the last level

Total Node Number: $2^h - 1$, if it is Perfect AT
Total Node Number of our assumption: $2^h - 1 - p$

So, Average =
$$\frac{\left[\sum_{h=1}^n h^2 \cdot 2^{h-1} \right] - \left[(2^{h+1} - p) \cdot (h-1)^2 \right]}{2^h - p - 1 + 1 \rightarrow \text{Total}}$$

$$\begin{matrix} \searrow & \swarrow \\ \text{Not in the tree} & \text{last level nodes} \end{matrix}$$

- c) Is there a restriction on the number of nodes in a full binary tree? What is the number of internal nodes and number of leaves in an n node full binary tree?

The number of leaves in a non-empty full binary tree is one more than the number of internal nodes. (By Mathematical Induction)

Induction step: Given tree T with n internal nodes,

Pick internal node I with two leaf children.

Remove I 's children, call resulting tree T' ,

T has $I+1$ nodes (leaf)

- we start with 1 leaf and each branching step creates 2 new leaf nodes, and one leaf node turns into an internal node (for a net of +1 leaf in the tree). So the tree has $2b+1$ nodes, b internal nodes and $b+1$ leaves where b is the number of branching

$$n = 2b+1$$

$$b = (n-1)/2 \Rightarrow \text{internal nodes}$$

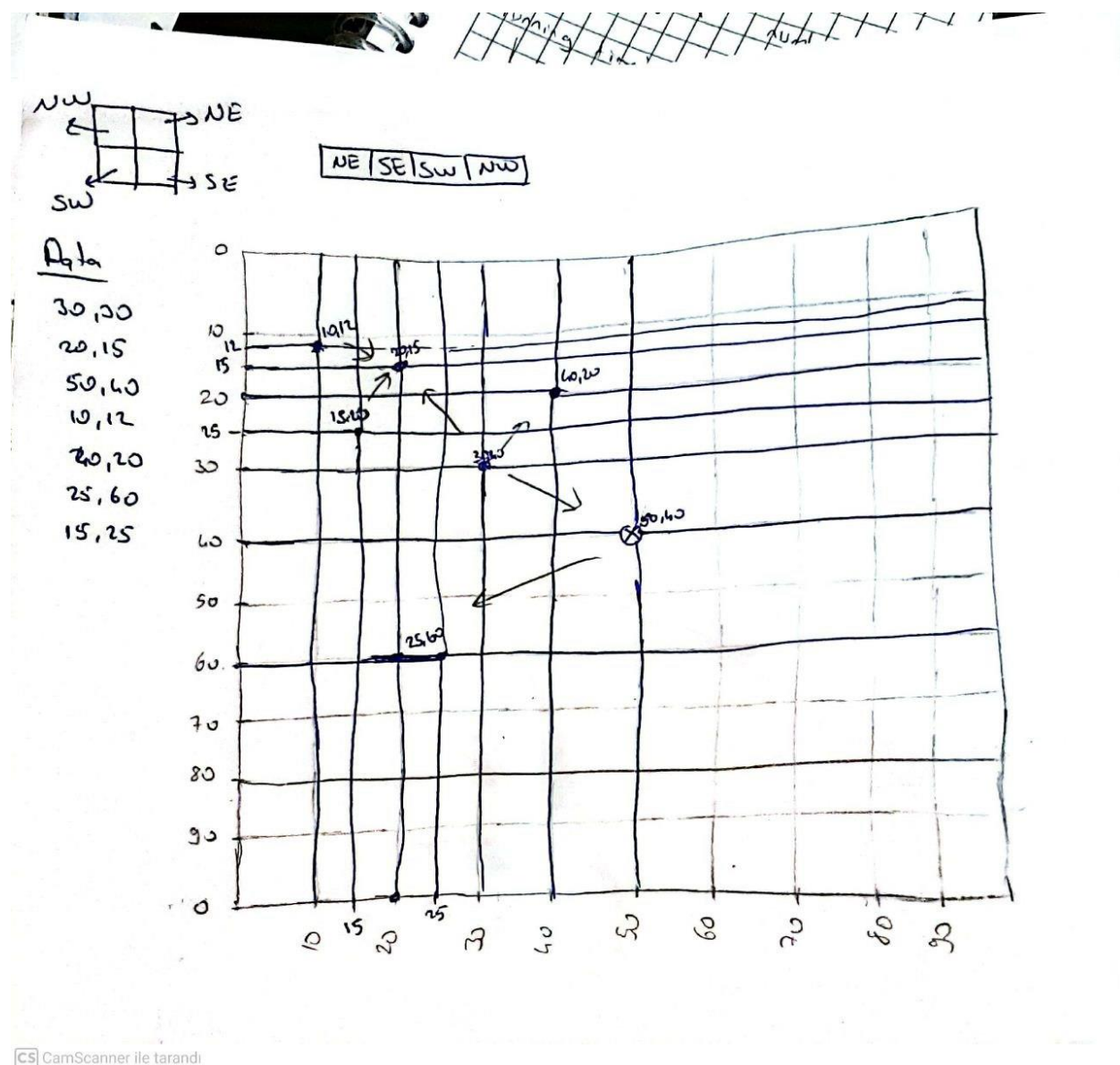
$$b+1 = \frac{n-1}{2} + 1 = \frac{n+1}{2} \Rightarrow \text{leaves}$$

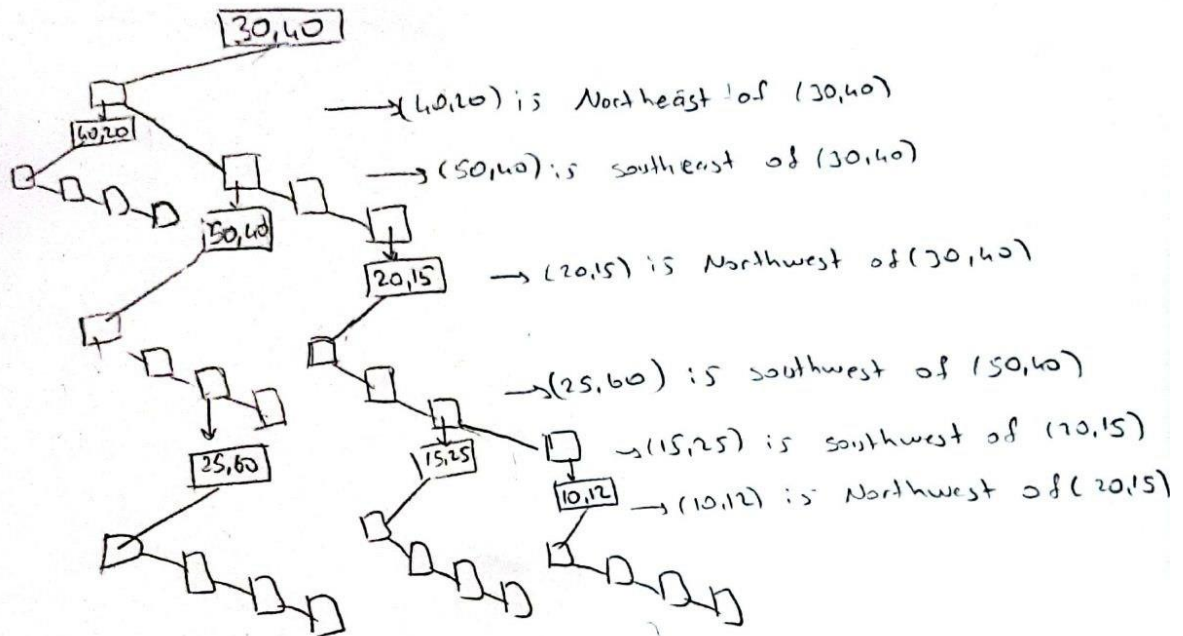
In binary tree each non-leaf node provides two edges. The full tree contains 2^n nodes. Each non-leaf node connected to an ancestor consumes one edge, which is true of all nodes except the root node of tree.

Question 2

Research about the quadtree structure for two-dimensional point data. Consider using the binary tree representation of general trees in our textbook to implement the quadtree structure. Insert the following elements one by one into an empty quadtree. Show the nodes traversed during each insertion and resulting tree after each insertion. Assume that the range is (0, 100) for both dimensions. Note that you are expected to draw the binary tree representation of the quadtree.

(30,30), (20,15), (50,40), (10,12), (40,20), (25,60), (15,25)





Analysis

```
package x q3\SearchTree.java x q3\BinaryTree.java x Driver.java x BinarySearchTree.java x q4\BinaryTree.java x q4\SearchTree.java x BinaryHeap.java x
29 of public boolean add(Comparable value) {
30     int tempSize = size + 1;
31     Node currentNode = root;
32     if (size == 0 || root == null) {
33         root = new Node(value, parent: null);
34         currentNode = root;
35         //System.out.println(currentNode.value + " is added at root");
36     } else {
37         boolean left = true;
38         for (int i = 0; i < nextDepth() - 1; i++) {
39
40             if (((Math.pow(2, nextDepth() + 1 - i) - 1)) - (tempSize)) >= Math
41                 .pow(2, nextDepth() - i - 1)) {
42                 currentNode = currentNode.leftChild;
43                 left = true;
44                 // System.out.println("moved left");
45             } else {
46                 currentNode = currentNode.rightChild;
47                 left = false;
48                 // System.out.println("moved right");
49             }
50
51             if (left) {
52                 tempSize = (int) (tempSize - Math.pow(2, nextDepth() - i
53                     - 1));
54             } else {
55                 tempSize = (int) (tempSize - Math.pow(2, nextDepth() - i));
56             }
57         }
58
59         if (currentNode.leftChild == null) {
60             currentNode.leftChild = new Node(value, currentNode);
61             //System.out.println(currentNode.leftChild.value
62             //    + " added to left. parent is " + currentNode.value);
63             siftUp(currentNode.leftChild, value: "add");
64
65         } else if (currentNode.rightChild == null) {
66             currentNode.rightChild = new Node(value, currentNode);
67             // System.out.println(currentNode.rightChild.value
68             //    + " added to right. parent is " + currentNode.value);
69             //siftUp(currentNode.rightChild, value: "add");
70     }
}
```

$O(n)$
 $O(1)$
 $O(n)$
 $O(1)$
 $O(1)$

```
package ... q3\SearchTree.java q3\BinaryTree.java Driver.java BinarySearchTree.java q4\BinaryTree.java q4\SearchTree.java BinaryHeap.java
79 @Override
80 public boolean remove(E target) {
81     E removedValue = root.value;
82
83     Node currentNode = root;
84     int tempSize = size;
85     boolean left = true;
86
87     if (size <= 1) {
88
89         root = null;
90         if (size > 0) {
91             size--;
92         }
93         return true;
94     } else if (size == 2) {
95         root.leftChild.parent = null;
96         root = root.leftChild;
97         size--;
98         return true;
99     }
100     for (int i = 0; i < depth(); i++) {
101
102         if (((Math.pow(2, depth() + 1 - i) - 1) - (tempSize)) >= Math
103             .pow(2, depth() - i - 1)) {
104
105             currentNode = currentNode.leftChild;
106
107             left = true;
108             // System.out.println("moved left");
109         } else {
110
111             currentNode = currentNode.rightChild;
112             left = false;
113             //System.out.println("moved right");
114         }
115
116         if (left) {
117             tempSize = (int) (tempSize - Math.pow(2, depth() - i - 1));
118         } else {
119             tempSize = (int) (tempSize - Math.pow(2, depth() - i));
120         }
121     }
122
123     root.value = currentNode.value;
124     if (left && size > 1) {
125         currentNode.parent.leftChild = null;
126     } else if (!left && size > 2) {
127         currentNode.parent.rightChild = null;
128     }
129     // System.out.println("root is now " + root.value)
130     siftDown(root);
131
132     if (size > 0) {
133         size--;
134     }
135
136     return false;
137 }
138
139
140
141 /*
142  * here lies the array methods of the heap.
143  */
144 public void fromArray(E[] array) {
145     E[] valueArray = array;
146     for (int i = 0; i < valueArray.length; i++) {
147         add(valueArray[i]);
148     }
149 }
150 }
```

$O(n)$

$O(1)$

$O(n)$

$O(1)$

Lightshot
Screenshot is saved to Screenshot_50.png. Click here to open in the folder.

```
package ... q3\SearchTree.java q3\BinaryTree.java Driver.java BinarySearchTree.java q4\BinaryTree.java q4\SearchTree.java BinaryHeap.java
118     } else {
119         tempSize = (int) (tempSize - Math.pow(2, depth() - i));
120     }
121 }
122
123 root.value = currentNode.value;
124 if (left && size > 1) {
125     currentNode.parent.leftChild = null;
126 } else if (!left && size > 2) {
127     currentNode.parent.rightChild = null;
128 }
129 // System.out.println("root is now " + root.value)
130 siftDown(root);
131
132 if (size > 0) {
133     size--;
134 }
135
136 return false;
137 }
138
139
140
141 /*
142  * here lies the array methods of the heap.
143  */
144 public void fromArray(E[] array) {
145     E[] valueArray = array;
146     for (int i = 0; i < valueArray.length; i++) {
147         add(valueArray[i]);
148     }
149 }
150 }
```

$O(1)$

$O(n)$

```
package x q3\SearchTree.java x q3\BinaryTree.java x Driver.java x BinarySearchTree.java x q4\BinaryTree.java x q4\SearchTree.java x BinaryHeap.java x
227
1 usage
228 @ public E[] toArray(E[] array) {
229     int counter = 0;
230     Node tempNode;
231     E[] valuesArray = (E[]) java.lang.reflect.Array.newInstance(array.getClass().getComponentType(), size); } O(n)
232     ArrayList<Node> queue = new ArrayList<>();
233     queue.add(root);
234
235     while (!queue.isEmpty()) {
236         tempNode = (Node) queue.remove(index: 0);
237         valuesArray[counter] = tempNode.value;
238         if (tempNode.leftChild != null) {
239             queue.add(tempNode.leftChild);
240         }
241         if (tempNode.rightChild != null) {
242             queue.add(tempNode.rightChild);
243         }
244         counter++;
245     }
246
247     return valuesArray;
248 }
249
250
251 /*
252  * These are methods used to shift nodes around.
253  */
254 2 usages
255 private void siftUp(Node node, String value) {
256     Node n = node;
257     if (n.parent == null) {
258         return;
259     }
260     for (int i = 0; i < nextDepth(); i++) {
261         if (n.value.compareTo(n.parent.value) > 0) {
262             swap(n, n.parent);
263             n = n.parent;
264         } else {
265             break;
266         }
267     }
268 }
```

Handwritten annotations in the image include:

- $O(n)$ next to line 231.
- $O(n)$ next to the while loop (lines 235-245).
- $O(n)$ next to the `siftUp` method signature (line 255).
- $O(n)$ next to the for loop in `siftUp` (lines 260-265).

Lightshot
Screenshot is saved to Screenshot_52.png. Click here to open in the folder.

```
package x q3\SearchTree.java x q3\BinaryTree.java x Driver.java x BinarySearchTree.java x q4\BinaryTree.java x q4\SearchTree.java x BinaryHeap.java x
30      */
31      1 usage
32      public Node( Node<E> left , E data , Node<E> right ) {
33          this.left = left;
34          this.data = data;
35          this.right = right;
36      }
37
38      /** Construct a node with given data and no children.
39      @param data The data to store in this node
40      */
41      1 usage
42      public Node( E data ) { this( left: null , data , right: null ); }
43
44      /** Return a string representation of the node.
45      @return A string representation of the data fields
46      */
47      @Override
48      public String toString() { return data.toString(); }
49
50      }
51
52      /** The root of the binary tree */
53      21 usages
54      protected Node<E> root;
55
56      /**
57      * No parameter constructor.
58      */
59      1 usage
60      protected BinaryTree() { root = null; }
61
62      /**
63      * Construct a node with given node as root.
64      * @param root node that will be root of BinaryTree.
65      */
66      2 usages
67      protected BinaryTree(Node<E> root ) { this.root = root; }
68
69      /**
70      * Constructs a new binary tree with data in its root leftTree
71
72
```

) O(1)

) O(1)

} O(1)

```
package x q3\SearchTree.java x q3\BinaryTree.java x Driver.java x BinarySearchTree.java x q4\BinaryTree.java x q4\SearchTree.java x BinaryHeap.java x
93      * Return the left subtree.
94      */
95      1 usage
96      public void getLeftSubtree() {
97          if( root != null && root.left != null ) {
98              new BinaryTree<>(root.left);
99          }
100         else {
101             return;
102         }
103     }
104 }
105
106 /** Return the right subtree.
107  * @return The right subtree or null if either the root or
108  * the right subtree is null
109  */
110 1 usage
111  public BinaryTree<E> getRightSubtree() {
112      if( root != null && root.right != null ) {
113          return new BinaryTree<>( root.right );
114      }
115      else {
116          return null;
117      }
118  }
119 }
120
121 /**
122  * Return the data of the root node.
123  * @return data which is hold in tree
124  */
125  public E getData() {
126      if( root != null ) {
127          return root.data;
128      }
129
130      System.out.println("Error : getData() root is null");
131      return null;
132  }
```

) O(1)

) O(1)

) O(1)

```

137 package q3;
138
139 if (root != null) {
140     return ( root.left == null && root.right == null );
141 }
142
143 System.out.println("Error : root null in isLeaf ! ");
144 return false;
145 }
146
147 /** Converts a sub-tree to a string.
148     Performs a preorder traversal.
149     @param node The local root
150     @param sb The StringBuilder to save the output
151     */
152     3 usages
153     protected void preOrderTraverse( Node<E> node , StringBuilder sb ) {
154
155         if( node != null ) {
156             sb.append( node.toString() ).append(" ");
157             preOrderTraverse( node.left , sb );
158             preOrderTraverse( node.right , sb );
159         }
160     }
161
162 /**
163     * Return prefix constructed string representation of the tree.
164     * @return String representation of current tree.
165     */
166     @Override
167     public String toString() {
168
169         StringBuilder sb = new StringBuilder();
170         preOrderTraverse( root , sb );
171         return sb.toString();
172     }
173 }
174 }

```

Handwritten notes in the first image:

- $O(1)$ (next to line 140)
- $O(n)$ (next to line 153)
- $O(n)$ (next to line 167)

Lightshot Screenshot is saved to Screenshot_55.png. Click here to

```

1 package q4;
2
3 public class BinarySearchTree<E extends Comparable<E>> extends BinaryTree<E> implements SearchTree<E>{
4
5     58 usages
6     E[] tree;
7     1 usage
8     public BinarySearchTree(int size) {
9         tree = (E[]) new Comparable[size];
10    }
11
12    @Override
13    public boolean remove(E target) { return false; }
14
15    @Override
16    public boolean add(Comparable d){
17        int i = 0; //Counter to traverse the array.
18        while (i < tree.length && tree[i] != null) { //End of Array or null location found.
19            if (d.compareTo(tree[i]) < 0) //Incoming data is smaller than data present in the array.
20                i = (2 * i) + 1; //Move to left side of the tree.
21            else
22                i = (2 * i) + 2; //Move to right side of the tree
23        }
24        tree[i] = (E) d;
25        return false;
26    }
27
28    @Override
29    public boolean contains(E target) { return false; }
30 }

```

Handwritten notes in the second image:

- $O(1)$ (next to line 8)
- $O(\log n)$ (next to line 16)
- $T_b \rightarrow O(1)$ (next to line 20)
- $T_w \rightarrow O(\log n)$ (next to line 21)
- $T_d = O(\log n)$ (next to line 22)


```
package x SearchTree.java x q3\BinaryTree.java x Driver.java x BinarySearchTree.java x q4\BinaryTree.java x

1 usage
33 01 public E find(E d) {
34     int i = 0; //Counter to traverse the array.
35     boolean found = false;
36     while (i < tree.length) { //End of Array or 'd' found.
37         if (d.compareTo(tree[i]) < 0) //Data to be searched is smaller than data present in the array.
38             i = (2 * i) + 1; //Move to left side of the tree.
39         else if (d.compareTo(tree[i]) > 0)
40             i = (2 * i) + 2; //Move to right side of the tree
41         else { // 'd' is same as tree[i] i.e. d found.
42             found = true;
43             break;
44         }
45     }
46     if (found) // 'd' found.
47         System.out.println(d + " found at " + i + ".");
48     else // 'd' not found.
49         System.out.println(d + " not found.");
50     return d;
51 }
52
2 usages
53 public int[] findDel(E d) {
54     int[] indices = new int[2];
55     boolean found = false;
56     int c = 0; //Child
57     int p = 0; //Parent
58     while (c < tree.length && tree[c] != null) { //End of Array or 'd' found.
59         if (d.compareTo(tree[c]) < 0) { //Data to be searched is smaller than data present in the array.
60             p = c; //Parent index
61             c = (2 * c) + 1; //Move to left side of the tree.
62         }
63         else if (d.compareTo(tree[c]) > 0) {
64             p = c; //Parent index
65             c = (2 * c) + 2; //Move to right side of the tree
66         }
67         else { // 'd' is same as tree[i] i.e. d found.
68             found = true;
69             break;
70         }
71     }
72     if (found) {
73         // ...
74     }
75 }
```

$O(n)$

$O(\log n)$

```
package ... SearchTree.java ... q3.BinaryTree.java ... Driver.java ... BinarySearchTree.java ... q4.BinaryTree.java ...
121 }
122
123 1 usage
124 public E delete(E d) {
125     if (findDel(d) == null) {
126         System.out.printf("\nKey not found, can't delete.\n\nThe list remains unchanged\n\n");
127         return null;
128     }
129     else {
130         int[] ref = findDel(d);
131         int c = ref[1];
132         int right = (2 * c) + 2;
133         int left = (2 * c) + 1;
134         System.out.printf("\ndeleting " + tree[c] + "\n");
135         if ((right > tree.length || left > tree.length) || (tree[left] == null && tree[right] == null)) { //No child case
136             delNoChild(c);
137         } else if ((tree[right] == null && tree[left] != null) || (tree[right] != null && tree[left] == null)) { //One child case
138             delOneChild(c, left, right);
139         } else { //Two Child Case
140             int temp = right; //Move to the right of the node to be deleted
141             int temp2 = 0;
142             while (tree[temp] != null) { //Until the right is empty
143                 temp2 = temp; //Because temp will be null at the end of the loop, a variable to store the previous value
144                 temp = (2 * temp) + 1; //Move to the left
145             }
146             tree[c] = tree[temp2]; //The value at temp2 copied at the node to be deleted
147             c = temp2; //Child updated to carry out deletion of no child/one child at temp2
148             //right and left updated to carry out deletion at temp2
149             right = (2 * c) + 2;
150             left = (2 * c) + 1;
151             if ((right > tree.length || left > tree.length) || (tree[left] == null && tree[right] == null)) { //No child case
152                 delNoChild(c);
153             } else if ((tree[right] == null && tree[left] != null) || (tree[right] != null && tree[left] == null)) { //One child case
154                 delOneChild(c, left, right);
155             }
156         }
157     }
158     return d;
159 }
160
```

package x SearchTree.java x q3\BinaryTree.java x Driver.java x BinarySearchTree.java x q4\BinaryTree.java x

```
71  * as its left subtree and rightTree as its right subtree.
72  * @param data data
73  * @param left left reference
74  * @param right right reference
75  */
76  @protected BinaryTree( E data , BinaryTree<E> left , BinaryTree<E> right ) {
77
78      root = new Node<>( data );
79
80      if( left.root != null ) {
81          root.left = left.root;
82      }
83
84      if( right.root != null ) {
85          root.right = right.root;
86      }
87
88  }
89
90  /** Return the left subtree.
91  @return The left subtree or null if either the root or
92  the left subtree is null
93  */
94  ! usage
95  public BinaryTree<E> getLeftSubtree() {
96
97      if( root != null && root.left != null ) {
98          return new BinaryTree<>( root.left );
99      }
100     else {
101         return null;
102     }
103 }
104
105 /** Return the right subtree.
106 @return The right subtree or null if either the root or
107 the right subtree is null
108 */
```

) Q(11)

) Q(11)

```
package × SearchTree.java × q3\BinaryTree.java × Driver.java × BinarySearchTree.java × q4\BinaryTree.java ×
106  @return The right subtree or null if either the root or
107  the right subtree is null
108  */
109  1 usage
110  public BinaryTree<E> getRightSubtree() {
111      if( root != null && root.right != null ) {
112          return new BinaryTree<>( root.right );
113      }
114      else {
115          return null;
116      }
117  }
118  }
119
120  /**
121   * Return the data of the root node.
122   * @return data which is hold in tree
123   */
124  public E getData() {
125      if( root != null ) {
126          return root.data;
127      }
128      System.out.println("Error : getData() root is null");
129      return null;
130  }
131  }
132
133  /** Determine whether this tree is a leaf.
134   * @return true if the root has no children
135   */
136  1 usage
137  public boolean isLeaf() {
138      if( root != null ) {
139          return ( root.left == null && root.right == null );
140      }
141      System.out.println("Error : root null in isLeaf ! ");
142      return false;
143  }
144  }
```

```
package × SearchTree.java × q3\BinaryTree.java × Driver.java × BinarySearchTree.java × q4\BinaryTree.java ×
138 if ( root != null ) {
139     return ( root.left == null && root.right == null );
140 }
141
142 System.out.println("Error : root null in isLeaf ! ");
143 return false;
144 }
145
146 /** Converts a sub-tree to a string.
147     Performs a preorder traversal.
148     @param node The local root
149     @param sb The StringBuilder to save the output
150 */
151 3 usages
152 protected void preOrderTraverse( Node<E> node , StringBuilder sb ) {
153
154     if( node != null ) {
155         sb.append( node.toString() ).append(" ");
156         preOrderTraverse( node.left , sb );
157         preOrderTraverse( node.right , sb );
158     }
159 }
160
161 /**
162     * Return prefix constructed string representation of the tree.
163     * @return String representation of current tree.
164     */
165 @Override
166 public String toString() {
167
168     StringBuilder sb = new StringBuilder();
169     preOrderTraverse( root , sb );
170     return sb.toString();
171 }
172 }
173 }
```

5. TEST CASES

Heap constructor worked

12 added to heap

44 added to heap

12 removed from heap

Binary Search Tree constructor worked with size 6

15 added to BST

13 added to BST

13 deleted

15 found at 0

Error: root null in is Leaf !

6. RUNNING AND RESULTS

