# Algorithm and Data Structures

Assignment 14: Report

Name: Maeda Taishin

Student ID: 1W21CF15

Date: 18/07/2023

# Binary Search Tree:

```java
import java.util.Stack;
public class BiTreeS {
    private Node root;
    public BiTreeS(){
        root = null;
    }
    public String search(int key){
        Node p = root;
        while(true){
            if(p == null){
                return null;
            }else{
                if(key == p.getKey()){
                    return p.getData();
                }else if(key < p.getKey()){
                    p = p.getLeft();
                }else if(key > p.getKey()){
                    p = p.getRight();
                }
            }
        }
    }
```

```java
    public boolean add(int key, String data){
        Node newNode = new Node(key, data);
        if(root == null){
            root = newNode;
            return true;
        }else{
            Node p = root;
            while(true){
                if(key == p.getKey()){
                    return false;
                }else if(key < p.getKey()){
                    if(p.getLeft() == null){
                        p.addLeft(newNode);
                        return true;
                    }else{
                        p = p.getLeft();
                    }
                }else if(key > p.getKey()){
                    if(p.getRight() == null){
                        p.addRight(newNode);
                        return true;
                    }else{
                        p = p.getRight();
                    }
                }
            }
        }
    }
```

```java
public boolean delete(int key){
    Node p = root;
    Node parent = null;
    while(true){
        if(p == null){
            return false;
        }else{
            if(key == p.getKey()){
                break;
            }else if(key < p.getKey()){
                parent = p;
                p = p.getLeft();
            }else if(key > p.getKey()){
                parent = p;
                p = p.getRight();
            }
        }
    }
    if(p.getLeft() == null && p.getRight() == null){
        if(p.getKey() < parent.getKey()){
            parent.deleteLeft();
        }else if(p.getKey() > parent.getKey()){
            parent.deleteRight();
        }
    }else if(p.getLeft() == null){
        if(p.getKey() < parent.getKey()){
            parent.addLeft(p.getRight());
        }else if(p.getKey() > parent.getKey()){
            parent.addRight(p.getRight());
        }
    }else if(p.getRight() == null){
        if(p.getKey() < parent.getKey()){
            parent.addLeft(p.getLeft());
        }else if(p.getKey() > parent.getKey()){
            parent.addRight(p.getLeft());
        }
    }else{
        Node tempParent = p;
        Node largestN = p.getLeft();
        while(largestN.getRight() != null){
            tempParent = largestN;
            largestN = largestN.getRight();
        }
        p.update(largestN.getKey(), largestN.getData());
        if(tempParent != p){
            tempParent.addRight(largestN.getLeft());
        }else{
            tempParent.addLeft(largestN.getLeft());
        }
    }
    return true;
}
```

```
103     public void printTree() {
104         if (root == null) {
105             System.out.println(x:"No nodes in this tree");
106         } else {
107             Stack<Node> stack = new Stack<Node>();
108             stack.push(root);
109             while (!stack.empty()) {
110                 Node p = stack.pop();
111                 System.out.print(p);
112                 if (p.getLeft() != null || p.getRight() != null)
113                     System.out.print(s:" has");
114                 if (p.getLeft() != null) {
115                     System.out.print(" " + p.getLeft() + " on left");
116                     stack.push(p.getLeft());
117                 }
118                 if (p.getRight() != null) {
119                     System.out.print(" " + p.getRight() + " on right");
120                     stack.push(p.getRight());
121                 }
122                 System.out.println(x:"");
123             }
124         }
125     }
```

```
127     public static void main(String[] args) {
128         //Test case 1
129         BiTreeS tree = new BiTreeS();
130         tree.add(key:9, data:"n1");
131         tree.add(key:5, data:"n2");
132         tree.add(key:10, data:"n3");
133         tree.add(key:2, data:"n4");
134         tree.add(key:7, data:"n5");
135         tree.add(key:11, data:"n6");
136         tree.add(key:1, data:"n7");
137         tree.add(key:4, data:"n8");
138         tree.add(key:3, data:"n9");
139         tree.add(key:6, data:"n10");
140         tree.add(key:8, data:"n11");
141         tree.add(key:12, data:"n12");
142         tree.printTree();
143 
144         //Test case 2
145         System.out.println("Search " + 1);
146         System.out.println("Result " + tree.search(key:1));
147 
148         System.out.println("Search " + 11);
149         System.out.println("Result " + tree.search(key:11));
150 
151         System.out.println("Search " + 20);
152         System.out.println("Result " + tree.search(key:20));
153 
154 
155         //Test case 3
156         System.out.println("Delete " + 6);
157         tree.delete(key:6);
158         tree.printTree();
159 
160         System.out.println("Delete " + 10);
161         tree.delete(key:10);
162         tree.printTree();
163 
164         System.out.println("Delete " + 5);
165         tree.delete(key:5);
166         tree.printTree();
167     }
168 
```

```java
    private class Node {
        private int key;
        private String data;
        private Node right;
        private Node left;

        public Node(int key, String data) {
            this.key = key;
            this.data = data;
            this.right = null;
            this.left = null;
        }

        public int getKey() {
            return key;
        }

        public String getData() {
            return data;
        }

        public void addLeft(Node n) {
            left = n;
        }

        public void addRight(Node n) {
            right = n;
        }

        public void deleteRight() {
            right = null;
        }

        public void deleteLeft() {
            left = null;
        }

        public Node getLeft() {
            return left;
        }

        public Node getRight() {
            return right;
        }

        public void update(int key, String data) {
            this.key = key;
            this.data = data;
        }

        public String toString() {
            return "<" + key + ", " + data.toString() + ">";
        }
```

# Code Explanation

Lines 1-2:

Import java package/(java.util.Stack)

Line 3:

Node “root” is defined in the BiTreeS class.

Lines 4-6:

A BiTreeS function is created to initialize a binary search tree with a null root.

**Lines 7-22: (Function “search” is created to search a node with “key” and return the string data of the node**

Line 8:

Node “p” representing a pointer is set as “root”.

Lines 9-21:

A while-loop is defined to keep checks until the “key” is found or nothing is to be searched anymore.

Lines 10-19:

An if-else statement is defined. If “p” is null, the program will return null, representing that nothing is found. Otherwise, another if-else statement is created to check if the “key” matches, being smaller or greater than the key of “p”. If the “key” matches the key of “p”, then the program will return the string data stored in “p”. Else-if the “key” is smaller than the key of “p”, then the program will reassign the pointer “p” to be the left child node of “p”. Else-if the “key” is greater than the key of “p”, then the program will reassign the pointer “p” to be the right child node of “p”.

**Lines 23-50: (A function that is defined to add a node with “Key” and “data”)**

Line 24:

Node “newNode” is set having “key” and “data”.

Lines 25-49:

An if-else statement is defined to check whether “root” is equal to null.

Lines 26-27:

If “root” is equal to null, “root” will be assigned to “newNode” and return true. Otherwise, the pointer “p” will be assigned as “root”.

Lines 30-48:

A while loop is defined to continuously find an available space for the node.

Lines 31-47:

An if-else statement is created inside the while loop. If the “key” is equal to the key of “p”, then the program will return false.

Else if the “key” is smaller than the key of “p”, another if-else statement is defined to check whether the left child node of “p” is equal to null or not (available or not). If so, the program will add the new node at this position and return true. If not, then the program will reassign the pointer “p” to be the left child node of “p”.
Else if the “key” is greater than the key of “p”, another if-else statement is defined to check whether the right child node of “p” is equal to null or not (available or not). If so, the program will add the new node at this position and return true. If not, then the program will reassign the pointer “p” to be the right child node of “p”.

**Lines 51-102: (A function that is defined to delete a node with “Key”)**

Lines 52-53:

Node “p” and “parent” are initialized as “root” and null respectively.

Lines 54-68:

A while loop is defined to continuously find an available space for the node.

Lines 55-67:

An if-else statement is created inside the while loop. If “p” is equal to null, then the program will return false meaning that it is not found. Otherwise, another if-else statement is defined. If the “key” matches the key of “p”, then the program will break the loop.
Else-if the “key” is smaller than the key of “p”, then “parent” representing the pointer of the parent node will be assigned as “p”, and then the program will reassign the pointer “p” to be the left child node of “p”.
Else-if the “key” is greater than the key of “p”, then “parent” representing the pointer of parent node will be assigned as “p”, and then the program will reassign the pointer “p” to be the right child node of “p”.

Lines 69-100:

An if-else statement is defined considering three different cases. If the matched node “p” does not have any child node on both sides, then the program will remove “p” by deleting a child node of “parent” containing “p”. Inside the statement, another if-else statement is defined by checking whether the key of “p” is smaller than that of “parent”. If so, the program will delete the left child of the “parent”. Else-if the key of “p” is larger, then the program will delete the right child of the “parent”.
Else-if the key of the left child is equal to null meaning that “p” has only the right child, then the program will remove “p” by reassigning a child node of “parent” containing “p”. Inside, another if-else statement is defined by checking whether the “key” of “p” is smaller than that of “parent”. If so, then the program will reassign the left child of the “parent” with the right child of “p”. Else-if the key of “p” is greater than that of “parent”, then the program will reassign the right child of “parent” with the right child of “p”.

Else-if the key of the right child is equal to null meaning that “p” has only the left child, then the program will remove “p” by reassigning a child node of “parent” containing “p”. Inside, another if-else statement is defined by checking whether the “key” of “p” is smaller than that of “parent”. If so, then the program will reassign the left child of the “parent” with the left child of “p”. Else-if the key of “p” is greater than that of “parent”, then the program will reassign the right child of “parent” with the left child of “p”.

Otherwise, if “p” has child nodes in both sides, Node “tempParent” and “largestN” are initialized as temporary parent nodes and set as the largest node in the sub-tree as “p” and left child of p respectively.

Lines 90-93:

A while loop is defined. As long as there is “largestN” has the right child, the program will reassign “tempParent” as “largestN” and “largestN” as the right child of “largestN”.

Line 94:

The program is set to update the content of “largestN”.

Lines 95-99:

An if-else statement is defined to check if “tempParent” is not equal to “p”, then the program will reassign the right child of “tempParent” with the left child of largestN”. Otherwise, the program will reassign the left child of “tempParent” with the left child of “largestN”.

Line 101:

The program will return true.

**Lines 103-125: (A function that is created to print the number of nodes in the tree)**

Line 104-124:

An if-else statement is defined. If “root” is equal to null, then the program will print out “No nodes in this tree” representing the emptiness of the tree. Otherwise, the program will create an empty “Stack<Node>” named “stack” in which it will be used to perform the traversal of the binary tree in the iterations.

Lines 109-123:

A while loop is defined and continues to run until the stack becomes empty. Inside the loop, the program will pop a node “p” from the top of the stack using “stack.pop()”. Then, the program will print the current node “p” using the “toString()” defined in the “Node” class which will be discussed later.

Lines 112-121:

Three if-else statements are defined to check if the current node “p” has any children on the left or right. If so, then the program will print “has”.

If the node “p” has a left child, then the program will print the information of the left child, print “on left” after, and push the left child to the stack for the next process.

If the node “p” has a right child, then the program will print the information of the right child, print “on right” after, and push the right child to the stack for the next process.

Line 122:

A line break will be printed to move to th next line.

**Lines 127-167 (A main function that is created to test three different cases)**

“BiTreeS” class is created to indicate the binary search tree and includes functions/methods for adding, searching, and deleting nodes.

Test case 1: Creates a binary search tree and prints its nodes.

Test case 2: Searches for nodes with the specific keys and print out the result.

Test case 3: Delete nodes with the specific keys from the tree and print out the updated tree.

**Lines 169-224 (A class Node)**

Line 169:

A private inner class called “Node” is defined that can be only accessed within the enclosing class.

Lines 170-173:

An integer field “key” and “String data” string field is initialized to represent the key or value and additional data associated with the node respectively. On top of that, Node “right” and “left” is defined to indicate the right and left child node of the current node respectively.

Lines 175-180:

A node class is created to take parameters including “key” and “data” in order to initialize a new node with the given key and data as well as initially setting the “right” and “left” reference to null.

Lines 182-212:

Get methods: “getKey()” returns the “key” value of the node. “getData()” returns the “data” associated with the node.

Add methods: “addLeft(Node n)” set the left child of the current node to the provided “Node” “n”. “addRight(Node n)” set the right child of the current node to the provided “Node” “n”.

Delete methods: “deleteRight()” removes the element to the right child node which deletes it from the tree. “deleteLeft()” removes the element to the Left child node which deletes it from the tree.

Get methods for children: “getLeft()” returns the left child node of the current node. “getRight()” returns the right child node of the current node.

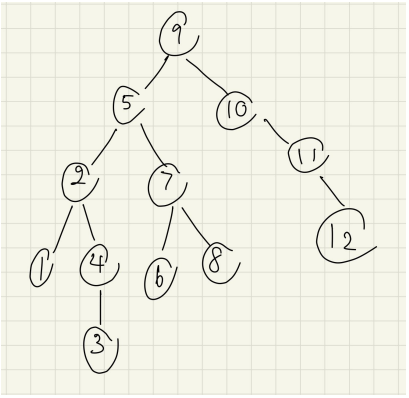
Update method: This method updates the “key” and “data” values of the node.

toString Method: This method returns a string representation of the node by printing out as “<key, data>” format.

# Execution Results:

**Test case 1**

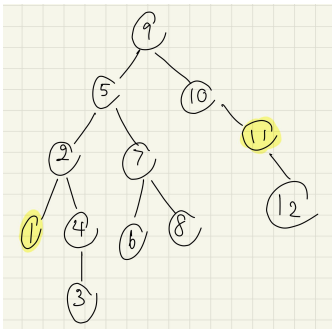
```
chawinwaimaleongora-ek@Chawins-MacBook-Air Code %
BiTreeS
<9, n1> has <5, n2> on left <10, n3> on right
<10, n3> has <11, n6> on right
<11, n6> has <12, n12> on right
<12, n12>
<5, n2> has <2, n4> on left <7, n5> on right
<7, n5> has <6, n10> on left <8, n11> on right
<8, n11>
<6, n10>
<2, n4> has <1, n7> on left <4, n8> on right
<4, n8> has <3, n9> on left
<3, n9>
<1, n7>
```




As can be seen, “9” is the root of this tree in which “5” and “10” are children of node/parent “9” in this case. In addition, 1,2,3,4,5,6,7,8 are considered as the left subtree and 10,11,12 are considered as the right subtree of “9”. Looking deeper, 1,2,3,4 are considered as the left subtree and 6,7,8 are considered as the right subtree of “5”. “10” only has its right subtree including 11 and 12. And the concept continues until every node has no more children or subtrees.

**Test case 2**

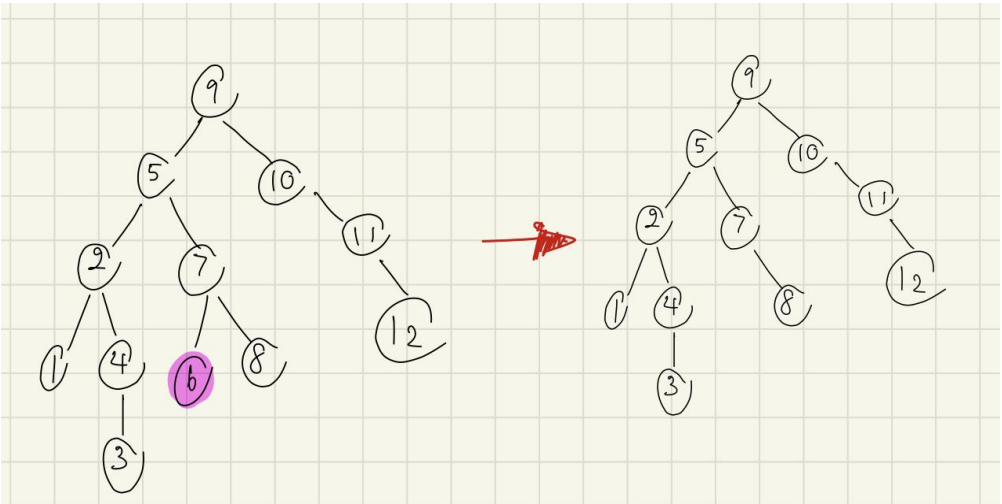
```
Search 1
Result n7
Search 11
Result n6
Search 20
Result null
```

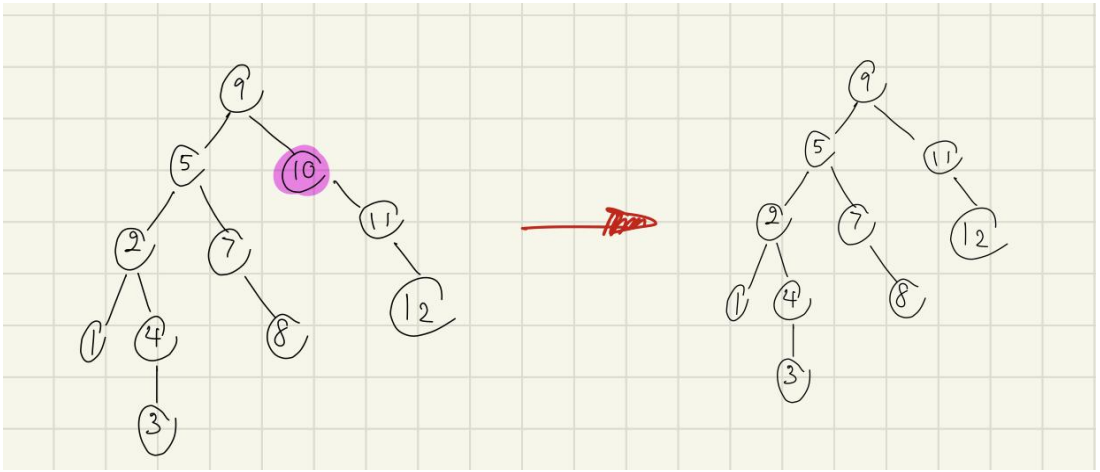


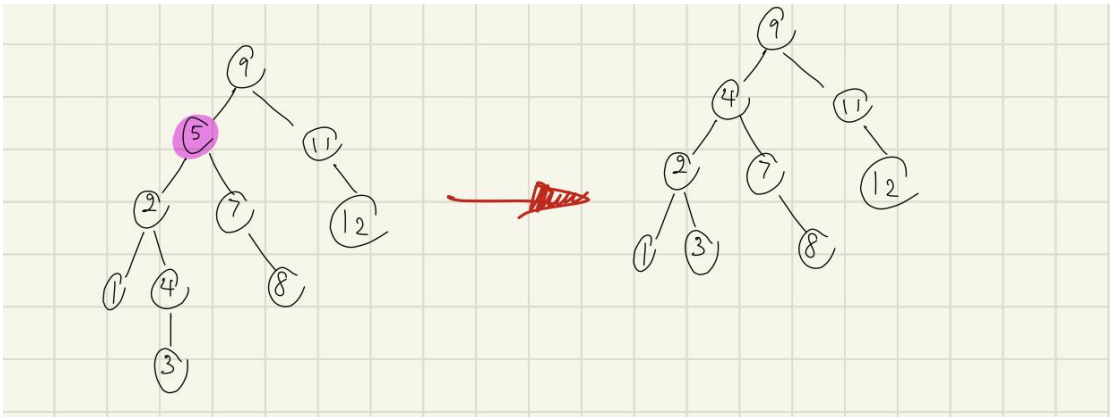

Continuing from Test case 1, the program asked to search for the node number of the integer “1”. As a result, “1” is the node number “n7” of the tree. The program also asked to search for the node number of the integer “11”. As a result, “11” is node number “n6” of the tree. However, when the program asked to search for the node number “20”, the program printed null meaning that “20” can not be found in this tree.

**Test case 3**

```
Delete 6
<9, n1> has <5, n2> on left <10, n3> on right
<10, n3> has <11, n6> on right
<11, n6> has <12, n12> on right
<12, n12>
<5, n2> has <2, n4> on left <7, n5> on right
<7, n5> has <8, n11> on right
<8, n11>
<2, n4> has <1, n7> on left <4, n8> on right
<4, n8> has <3, n9> on left
<3, n9>
<1, n7>
Delete 10
<9, n1> has <5, n2> on left <11, n6> on right
<11, n6> has <12, n12> on right
<12, n12>
<5, n2> has <2, n4> on left <7, n5> on right
<7, n5> has <8, n11> on right
<8, n11>
<2, n4> has <1, n7> on left <4, n8> on right
<4, n8> has <3, n9> on left
<3, n9>
<1, n7>
Delete 5
<9, n1> has <4, n8> on left <11, n6> on right
<11, n6> has <12, n12> on right
<12, n12>
<4, n8> has <2, n4> on left <7, n5> on right
<7, n5> has <8, n11> on right
<8, n11>
<2, n4> has <1, n7> on left <3, n9> on right
<3, n9>
<1, n7>
chawinwaimaleongora-ek@Chawins-MacBook-Air Code %
```





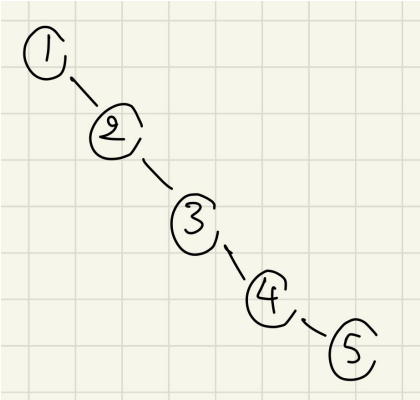




When deleting integer “6” from the tree, “6” is removed without affecting any other node’s positions since it does not have any children or subtrees. However, when integer “10” is deleted from the tree, “10” will be removed and the position of “11” and “12” are changed shifting from the position of “11” to “10 and “12” to “11” meaning that “11” and “12” moved to a higher level of the tree. On top of that, when deleting “5” from the tree, the entire subtrees of “5” are moved up to a higher level of the tree. Note that, “4” became the parent of the rest of the integers, and the initial position of “4” will be replaced by “3” for the appropriate tree structures.

# Time complexity:

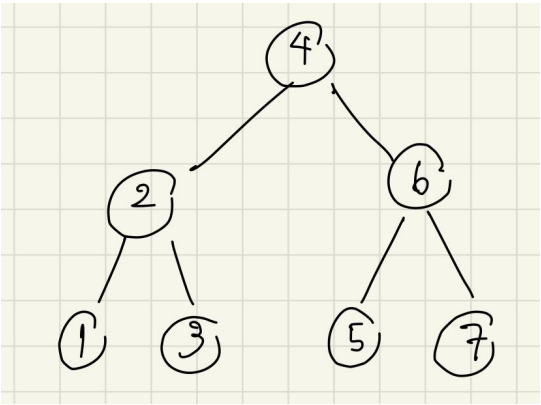
The time complexity of the binary search tree can vary depending on the structure of the tree and how balanced is it.

Worst-case time complexity:




This will be the case when the binary search tree is a skewed binary tree. In other words, the height of the binary search tree is n (shown in the picture above). Hence, the time complexity will be $O(n)$.

Best-case time complexity:




This will be the case when the structure of the binary search tree is balanced as shown in the picture above. This means that the height of the binary search tree becomes $log(n)$. Hence, the time complexity will be $O(logn)$.

Adding and deleting:

The time complexity of adding (inserting) and deleting a node from a balanced binary search tree will be $O(logn)$. For adding in the balanced tree, each insertion reduces the search by half. For deleting in the balanced tree, it is required to traverse down the tree through the left or right child pointers, so each step in the traversal reduces the search by half.