Assignment #14

Rikiya Takehi
1W21CS04-6
Algorithms and Data Structures
12/07/2022

Problem Statement

Write an algorithm for binary search tree using Java language. Boolean add, string search and Boolean delete should be included in the program. Test cases 1, 2, and 3 are given, each corresponding to add, search and delete.

Program (four pages)

```java
import java.util.Stack;

public class BinarySearchTree {
    private Node root;

    public BinarySearchTree() {
        root = null;
    }

    public String search(int key) {
        Node p=root;
        while(true){
            if(p==null){
                return null;
            }else if(key==p.key){
                return p.data;
            }else if(key<p.key){
                p=p.left;
                continue;
            }else if(key>p.key){
                p=p.right;
                continue;
            }
        }
    }

    public boolean add(int key, String data) {
        if(root==null){
            root=new Node(key, data);
        }

        Node p=root;
        while(true){
            if(key<p.key){
                if(p.left==null){
                    p.left=new Node(key, data);
                    return true;
                }else{
                    p=p.left;
                    continue;
                }
            }else if(key>p.key){
                if(p.right==null){
                    p.right=new Node(key, data);
                    return true;
                }else{
                    p=p.right;
                    continue;
                }
            }else{
                return false;
            }
        }
    }

    public boolean delete(int key) {
        Node p = root;
        Node c = root;
        boolean LC = false;
        while(c.key != key){
            p = c;
            if(c.key>key){
                LC = true;
                c = c.left;
```

```java
            }else{
                LC = false;
                c = c.right;
            }
            if(c ==null){
                return false;
            }
        }

        if(c.left==null && c.right==null){
            if(c==root){
                root = null;
            }
            if(LC ==true){
                p.left = null;
            }else{
                p.right = null;
            }
        }

        else if(c.right==null){
            if(c==root){
                root = c.left;
            }else if(LC){
                p.left = c.left;
            }else{
                p.right = c.left;
            }
        }
        else if(c.left==null){
            if(c==root){
                root = c.right;
            }else if(LC){
                p.left = c.right;
            }else{
                p.right = c.right;
            }
        }else if(c.left!=null && c.right!=null){

            Node s   = Successor(c);
            if(c==root){
                root = s;
            }else if(LC){
                p.left = s;
            }else{
                p.right = s;
            }
            s.right = c.right;
        }
        return true;
    }

    public Node Successor(Node dNode){
        Node s =null;
        Node sP =null;
        Node c = dNode.left;
        while(c!=null){
            sP = s;
            s = c;
            c = c.right;
        }

        if(s!=dNode.left){
            sP.right = s.left;
```

```java
            s.left = dNode.left;
        }
        return s;
    }

    public void printTree() {
        if (root == null) {
            System.out.println("No nodes in this tree");
        } else {
            Stack<Node> stack = new Stack<Node>();
            stack.push(root);
            while (!stack.empty()) {
                Node p = stack.pop();
                System.out.print(p);
                if (p.getLeft() != null || p.getRight() != null)
                    System.out.print(" has");
                if (p.getLeft() != null) {
                    System.out.print(" " + p.getLeft() + " on left");
                    stack.push(p.getLeft());
                }
                if (p.getRight() != null) {
                    System.out.print(" " + p.getRight() + " on right");
                    stack.push(p.getRight());
                }
                System.out.println("");
            }
        }
    }

    public static void main(String[] args) {
        // Test case 1
        BinarySearchTree tree = new BinarySearchTree();
        tree.add(9, "n1");
        tree.add(5, "n2");
        tree.add(10, "n3");
        tree.add(2, "n4");
        tree.add(7, "n5");
        tree.add(11, "n6");
        tree.add(1, "n7");
        tree.add(4, "n8");
        tree.add(3, "n9");
        tree.add(6, "n10");
        tree.add(8, "n11");
        tree.add(12, "n12");
        tree.printTree();

        // Test case 2
        System.out.println("Search " + 1);
        System.out.println("Result " + tree.search(1));

        System.out.println("Search " + 11);
        System.out.println("Result " + tree.search(11));

        System.out.println("Search " + 20);
        System.out.println("Result " + tree.search(20));


        // Test case 3
        System.out.println("Delete " + 6);
        tree.delete(6);
        tree.printTree();

        System.out.println("Delete " + 10);
        tree.delete(10);
```

```java
        tree.printTree();

        System.out.println("Delete " + 5);
        tree.delete(5);
        tree.printTree();
    }

    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() + ">";
        }
    }

}
```

Program explanation

Binary search tree is a name for the placement of values in a sorted order. Looking up, addition and deletion could be done.

Lines 10-25 are the action of searching a value in a binary search tree. In line 12-13, when the node is null, it fails. In lines 14-15, if the node is the key itself, it returns the number as a data. In lines 17-22, if the key is smaller than key of the node, it replaces the node with the child on the left, and if the key is bigger than the key of the node, it replaces the node with the child on the right side. Then, repeats the same phase until the key is found

```java
public String search(int key) {
    Node p=root;
    while(true){
        if(p==null){
            return null;
        }else if(key==p.key){
            return p.data;
        }else if(key<p.key){
            p=p.left;
            continue;
        }else if(key>p.key){
            p=p.right;
            continue;
        }
    }
}
```

For the adding method, if the root is null, replace the root with a new node (lines 28-30). Then, if the key is smaller than the value of the node, the node is replaced with the left child unless the left child is null (lines 34-41). The same goes for the right child when the key is bigger than the key of the node (lines 42-48). If the key is neither smaller nor bigger than the key of the node, it returns false (lines 50-52). If the child becomes null, it replaces/adds the child with a new node.

```java
27  public boolean add(int key, String data) {
28      if(root==null){
29          root=new Node(key, data);
30      }
31
32      Node p=root;
33      while(true){
34          if(key<p.key){
35              if(p.left==null){
36                  p.left=new Node(key, data);
37                  return true;
38              }else{
39                  p=p.left;
40                  continue;
41              }
42          }else if(key>p.key){
43              if(p.right==null){
44                  p.right=new Node(key, data);
45                  return true;
46              }else{
47                  p=p.right;
48                  continue;
49              }
50          }else{
51              return false;
52          }
53      }
54  }
```

For the deleting method, it is basically divided into two parts. The first part includes finding the node and dealing with the sub-tree according to the numbers of children. In lines 60 to 72, if the current key is bigger than the key, the left child becomes the current node and same goes for the right child. This is a similar process to the search method. This is done until the current key becomes the key.

```
60        while(c.key != key){
61            p = c;
62            if(c.key>key){
63                LC = true;
64                c = c.left;
65            }else{
66                LC = false;
67                c = c.right;
68            }
69            if(c ==null){
70                return false;
71            }
72        }
```

Then, as we have found the node that is deleted in lines 60-72, in lines 74-83, the situation for when there are no children is written. In this case, the root and the children are all null.

```
74        if(c.left==null && c.right==null){
75            if(c==root){
76                root = null;
77            }
78            if(LC ==true){
79                p.left = null;
80            }else{
81                p.right = null;
82            }
83        }
```

When there is the left child but not right child, the left child becomes the root when the current node is the root (lines 86-87).

```
85        else if(c.right==null){
86            if(c==root){
87                root = c.left;
88            }else if(LC){
89                p.left = c.left;
90            }else{
91                p.right = c.left;
92            }
93        }
```

When the right child is there but not the left, the same procedure is done but with the right child.

```
        else if(c.left==null){
            if(c==root){
                root = c.right;
            }else if(LC){
                p.left = c.right;
            }else{
                p.right = c.right;
            }
```

When there are children on both left side and right side, it is necessary to use another algorithm that finds the maximum element in the left subtree. By using this, you can find the node that goes inside the deleted node.

```
        }else if(c.left!=null && c.right!=null){

            Node s   = Successor(c);
            if(c==root){
                root = s;
            }else if(LC){
                p.left = s;
            }else{
                p.right = s;
            }
            s.right = c.right;
        }
```

Here is the program to know the maximum in the left subtree. The successor should be replaced multiple times according to the length of the tree.

```
    public Node Successor(Node dNode){
        Node s =null;
        Node sP =null;
        Node c = dNode.left;
        while(c!=null){
            sP = s;
            s = c;
            c = c.right;
        }

        if(s!=dNode.left){
            sP.right = s.left;
            s.left = dNode.left;
        }
        return s;
    }
```

Experimental settings

Input data for test case 1 (add):

N1~12 in random orders.

Input data for test case 2 (search):

Search numbers: “1”, “11”, “21”.

Input data for test case 3 (delete):

Delete numbers: “6”, “10”, “5”

Results

Add:

```
(base) rikiyatakehi@RikiyanoMacBook-puro assignment 14 % java BinarySearchTree
<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>
```

Search:

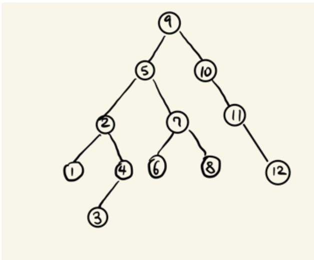
```
Search 1
Result n7
Search 11
Result n6
Search 20
Result null
```

Delete:

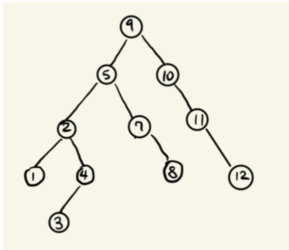
```
Delete 6
<9, n1> has <5, n2> on left <10, n3> on right
<10, n3> has <11, n6> on right
<11, n6> has <12, n12> on right
<12, n12>
<5, n2> has <2, n4> on left <7, n5> on right
<7, n5> has <8, n11> on right
<8, n11>
<2, n4> has <1, n7> on left <4, n8> on right
<4, n8> has <3, n9> on left
<3, n9>
<1, 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>
```

Discussion of the Results

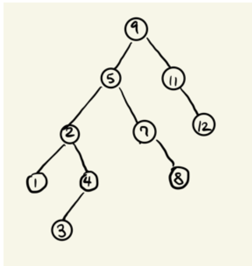
The results are shown on the compiler, but writing them as a graph showed the following figures.



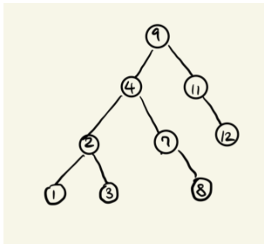
After Add Sequence



Delete 6



Delete 10



Delete 5

As shown, the binary search tree is very easy to see the overall picture of the values and elements. However, I have wondered the reason why binary search is being used. Although I could not have an image of efficiently using binary search tree to do any sort of work, after I have made the program and learned the functions, I have some ideas of suitable implementations.

Binary search tree can be used effectively when dealing with information that could be sorted out in order. For example, alphabet and numbers could be sorted out. A group of people could be sorted out using a binary tree. If the group of people is less than 100 people or so, it could be easily sorted out just by an array. However, if the number is over 100,000 and needs to be sorted out by adding, deleting and searching each person, it would be easier to sort using a binary tree. Adding and deleting would be faster than many of the other options. If the target value is near the end of the tree, it would be very fast to add and delete the numbers because it only requires a few steps. However, it would take a lot of time if the target numbers are placed near the start of the tree; the procedure would affect all the following tree branches.