15 Monday August 34th Wk • 227-138

2022 AUGUST

M	T	W	T	F	S	S	M	T	W	T	F	S	S
1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22	23	24	25	26	27	28
29	30	31											

Binary Search Tree

Normally, in previous data structure, we used $O(n)$ as the time complexity, where 'n' signifies the no. of nodes. Here, in BST $O(H)$ shall be used, where H refers to the height of the tree. In average case, $H = \log n$, so time complexity reduces significantly. That's why in fast look up @ scenarios, BST is used.

Properties :- (Binary Search Tree)

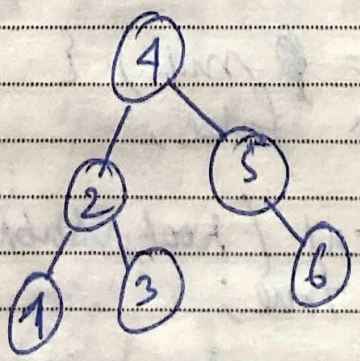
(1) All properties of Binary Tree are included in Binary Search Tree. (For e.g. the root will always have 2 children and not more than that, etc.

(2) Left Subtree Nodes < Root [∵ here values are considered]

(3) Right Subtree Nodes > Root

(4) Left and Right Subtree are also BST with no duplicates.

E.g →



Special Property :-

Inorder Traversal of BST gives a Sorted Sequence.

SEPTEMBER 2022

August
34th Wk • 228-137

16 Tuesday

BST Search

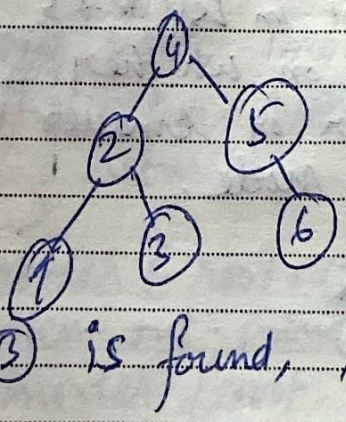
Just to note that BST makes search efficient.

Key | compare with Root

Left subtree (if key < root)

Right subtree (if key > root)

For e.g,



Key = 3

(a) compare Key = 3 with (4).

(b) As Key < root, go to LS. compare Key with (2).

(c) As Key > root, go to RS.

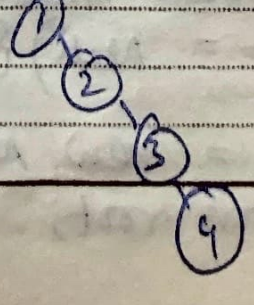
(d) once (3) is found, return true.

Also, time complexity = $O(H)$, where H = height of tree.
In average case, $H = \log n$, n = nodes

Best case :- when the tree is perfectly balanced, meaning there are equal nodes on the left and right sub-tree of the root. Then time complexity = $O(H)$, where $H = \log n$ and represents the height of the tree.

Worst case :- If the given tree is a skewed tree, then time complexity = $O(n)$, n = nodes.

e.g →



← this is a skewed tree and all the nodes are in the right sub-tree.

17 August
34th Wk • 229-136
Wednesday

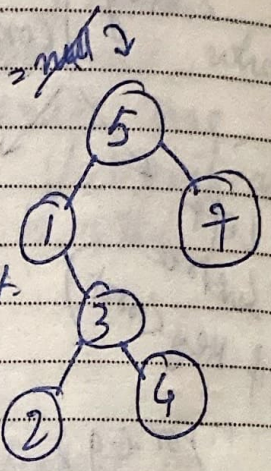
2022 AUGUST

M	T	W	T	F	S	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

* Strategy :- Most problems will be solved using recursion, i.e by dividing into sub-problems and making recursive calls on sub-trees.

Build a BST

Values[] = {5, 1, 3, 4, 2, 7}



Same structure to be followed for BST

```
Node {
    int data;
    Node left;
    Node right;
}
```

- Place the first element as root
- Then compare every element & Place it in LS or RS, depending on whether it's greater or smaller than the compared node.

Insertion in a BST

```
Public class BST
{
    Static class Node {
        int data;
        Node left;
        Node right;

        Node (int data) {
            this.data = data;
        }
    }

    Public Static Node insert (Node root, int val) {
        if (root == Null) {
            root = new Node(val);
            return root;
        }
```

```
if (root == null) {
    // left subtree
    root.left = insert (root.left, val);
}
else {
    // right subtree
    root.right = insert (root.right, val);
}
return root;
}

Public static void inorder (Node root) {
    if (root == null) {
        return;
    }
    inorder (root.left);
    System.out.print( root.data + " ");
    inorder (root.right);
}

Public static void main (String args[]) {
int values[] = {5, 1, 3, 4, 2, 7};
Node root = null;

for (int i=0 ; i< values.length; i++) {
    root = insert (root, values[i]);
}
inorder (root);
System.out.println();
}
}
```

19 August 2022
34th Wk • 231-134
Friday

AUGUST
1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31
M T W T F S S M T W T F S S

Search a BST

8 If root > Key → left Subtree

9 If root = Key → "True"

10 If root < Key → Right Subtree.

11 In worst case scenarios, we are travelling to the leaf, so O(H) is the time complexity, H = Height of the tree.
12

→
```
Public static boolean search (node root, int key) {
    if (root == null) {
1       return false;
    }
2   if (root.data > Key) {      // Search left Subtree
        return search (root.left, key);
3   }

4   else if (root.data == Key) {   // Search root
        return true;
5   }
    else {   // Search rightSubtree
6       return search (root.right, Key);
    }
}
Public static void main (String args[]) {
    int values[] = {8, 5, 3, 1, 4, 6, 10, 11, 14};
    Node root = null;
    if (search (root, 1)) {
        System.out.println ("found");
    } else {
        System.out.println ("not found");
```

33

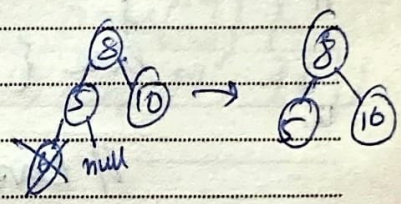
✱✱ Deletion of a node ✱✱

Deletion of a node is a special case. whenever, deletion comes into picture, we need to consider 3 cases -

Case 1 → No child (Leaf node)
Case 2 → one child
Case 3 → Two children

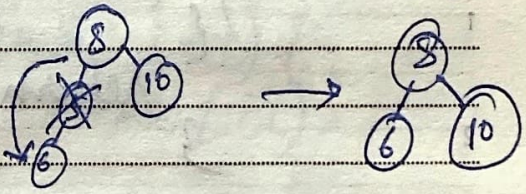
Case 1: No child

✓ Delete Node
✓ Return null to the parent.



Case 2: one child

✓ Delete Node
✓ Replace with child Node

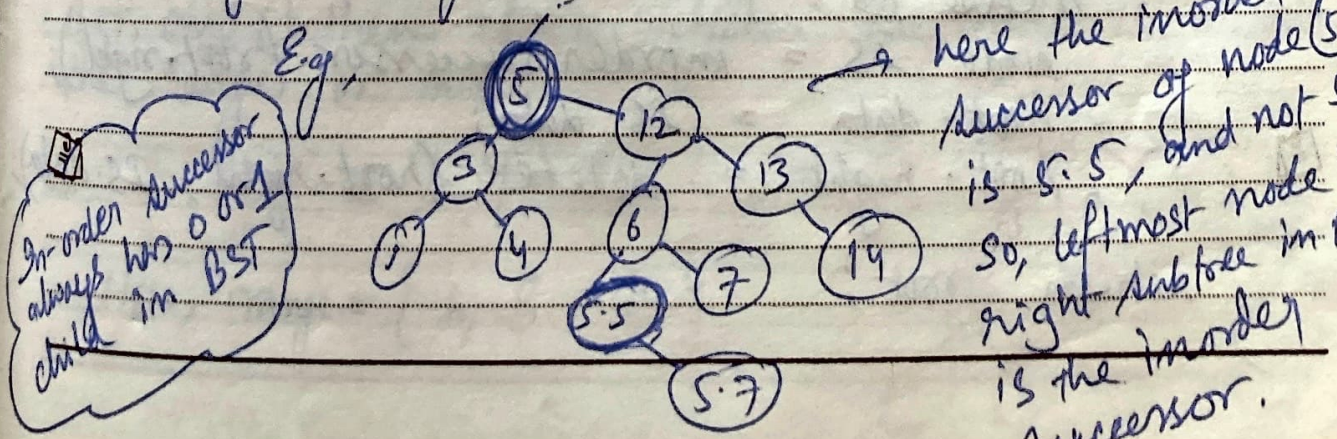


Case 3: Two children

✓ Replace value with inorder Successor
✓ delete the node for inorder Successor.

This is a bit complicated. Here we need to search the inorder Successor first and then we need to replace it. Also, in BST, ~~left~~ in-order = leftmost in right subtree.

Eg.,



→ here the inorder successor of node 5 is 5.5, and not 5.7. So, leftmost node in right subtree in BST is the inorder successor.

21 Sunday
August
34th Wk • 233-132

2022 AUGUST

M	T	W	T	F	S	S	M	T	W	T	F	S	S
1	2	3	4	5	6	7	8	9	10	11	12	13	14
15	16	17	18	19	20	21	22	23	24	25	26	27	28
29	30	31											

```
→ Public Static Node delete (Node root, int val)
  {
8   if (root.data > val)
    {
9     root.left = delete (root.left, val);
    }
10  else if (root.data < val) {
      root.right = delete (root.right, val);
11  }
    else {   //root.data == val
12
      // case 1
      if (root.left == null && root.right == null)
1     {
        return null;
2     }

3     // case 2
      if (root.left == null){
4       return root.right;
      }
5     else if (root.right == null) {
        return root.left;
6     }

      // case 3
      Node IS = inordersuccessor (root.right);
      root.data = IS.data;
      root.right = delete (root.right, IS.data);
    }
    return root;
  }
```

```
Public static Node inordersuccessor (Node root) {
    while (root.left != null) {
        root = root.left;
    }
    return root;
}
Public static void main (String args[]) {
    int values[] = {8, 5, 3, 1, 4, 6, 10, 11, 14};
    Node root = null;
    for (int i=0; i< values.length; i++) {
        root = insert (root, values[i]);
    }
    delete (root, 5);
    inorder (root);
}
```

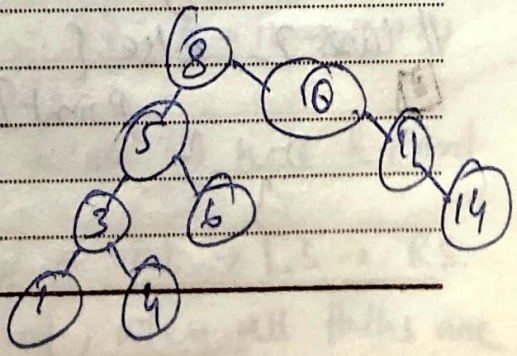
Print in Range

X>6 & Y>10 : range is inclusive of end limits.

So, basically we need to check all the nodes of the BST and then check whether the node is within the range or not.

X>6 & Y>10

Here, range = {6, 8, 10}.



23 August
35th Wk• 235-130
Tuesday

2022 AUGUST
1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31
M T W T F S S M T W T F S S

Here also we shall consider 3 cases —

case 1: $x <= root <= y$ → left, → right (if x=y=root, check case 1.)

case 2: $x > root$ → right

case 3 :- $y < root$ → left

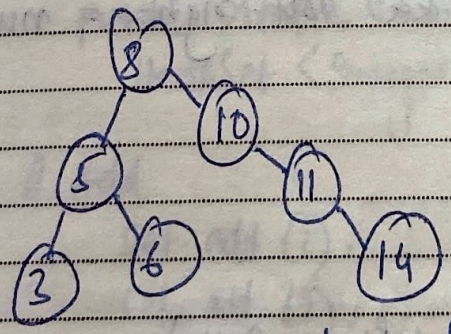
```
Public static void printInRange (Node root,
                                 int x, int y)
{
    if (root == null) {
        return;
    }
// case 1   if (root.data >= x && root.data <= y) {
        printInRange (root.left, x, y);
        System.out.print (root.data + " ");
        printInRange (root.right, x, y);
    }
// case 3   else if (root.data >= y) {
        printInRange (root.left, x, y);
    }
// case 2   else {
        printInRange (root.right, x, y);
    }
}
```

```
Public Static void main (String args[]) {
    int values[] = {8, 5, 3, 1, 4, 6, 10, 11, 14};
    node root = null;
    for (int i=0; i< values.length; i++) {
        root = insert (root, values[i]); // calls 'insert'
    }
    printInRange (root, 6, 10);
}
```

** Root to Leaf Paths **

In this question, we need to search all the paths from root to leafs in the BST.

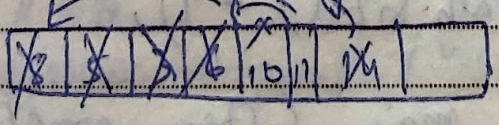
E.g.



P1: 8 → 5 → 3
P2: 8 → 5 → 6
P3: 8 → 10 → 11 → 14

To solve this, we need a data structure, where we can keep and remove elements as we traverse the BST. So, we shall use ArrayList.

Now,



P1: 8 → 5 → 3
P2: 8 → 5 → 6
P3: 8 → 10 → 11 → 14

- ✓ As we go to a node, we add the node.
- ✓ As we backtrack from a node, we delete the node & Print the path obtained.
- ✓ We shall follow pre-order traversal → root → LS → RS.
- ✓ At the end, we shall delete the root, when all paths are traversed.

25 August 2022
Thursday
35th Wk • 237-128

AUGUST
1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31
M T W T F S S M T W T F S S

→ Public static void printPath (ArrayList<Integer> path) {

```
    for (int i=0; i< path.size(); i++) {
        System.out.print(path.get(i) + "->");
    }
    System.out.println();
}

Public static void printRoot2Leaf (Node root, ArrayList
                                   <Integer> path) {
    if (root ==null) {
        return;
    }
    path.add(root.data);

    // leaf node
    if (root.left == null && root.right == null) {
        printPath (path);
    }

    // non-leaf nodes
    else {
        printRoot2Leaf (root.left, path);
        printRoot2Leaf (root.right, path);
    }
    path.remove (path.size() - 1);
}
Public static void main (String args[]) {
    int values[] = {8, 5, 3, 1, 4, 6, 10, 11, 14};
    Node root = null;
    for (int i=0; i< values.length; i++) {
        root = insert (root, values[i]);
    }
    printRoot2Leaf (root, new ArrayList<>());
}
```