

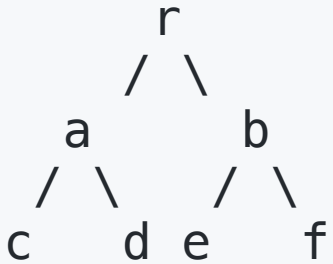
# Binary Tree

Definitions + `flatten`, `add` methods

# Definition

Binary tree is a tree data structure in which each node has at most two children, which are referred to as the *left* child and the *right* child.

# Diagram



What is the *height* of this tree?

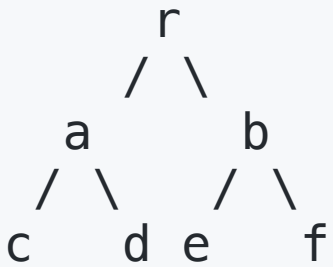
## Definitions (Cont.)

- *rooted binary tree* has a root node and every node has at most two children.
- *full binary tree* is a tree in which every node has either 0 or 2 children.
- *balanced binary tree* is a binary tree structure in which the left and right sub-trees of every node differ in height by no more than 1.

## Diagram (Cont.)

Assuming *left* node is *less* than parent node and *right* node is greater than parent node.

- Let's try to flatten it to a list
  - [c, a, d, r, e, b, f]
- Can we write a recursive code to do that?



## Flatten BST (naive)

```
void flatten(Node node, List<Node> rslt) {  
    // TODO: write a method that flattens a tree in a recursive fashion  
  
}  
  
Node root = ...;  
List<Node> list = new ArrayList<Node>();  
  
flatten(root, list);
```

# Flatten BST (naïve)

```
void flatten(Node node, List<Node> rslt) {  
    if (node == null) {  
        return;  
    } else {  
        flatten(node.left, rslt);  
        rslt.add(node);  
        flatten(node.right, rslt);  
    }  
}
```

```
Node root = ...;  
List<Node> list = new ArrayList<Node>();  
  
flatten(root, list);
```

How can we avoid recursion?

# Flatten BST (without recursion)

```
List<Node> flatten(Node root) {  
    List<Node> rslt = new ArrayList<Node>();  
    Stack<Node> stack = new Stack<Node>();  
  
    while (true) {  
        // Go to the left extreme insert all the elements to stack  
        while (root != null) {  
            stack.push(root);  
            root = root.left;  
        }  
  
        // check if Stack is empty, if yes, exit from everywhere  
        if (stack.isEmpty()) {  
            return;  
        }  
  
        // pop the element from the stack, yield it and add the nodes at the right to the Stack  
        root = stack.pop();  
        rslt.add(root);  
        root = root.right;  
    }  
  
    return rslt;  
}  
  
Node root = ...;  
List<Node> list = flatten(root, list);
```



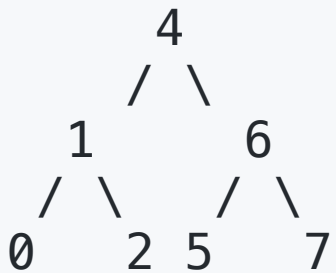
# Binary Search Property

Binary Search Property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree.

## add method

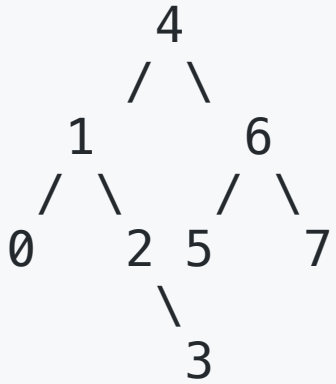
- How to implement `add` method which satisfies "Binary Search Property"
  - let's `add(3)`

## Diagram



## add method (Cont.)

### Diagram



# Exercise

## add method (naïve)

```
public void insert(Node node, T data) {  
    if (data < node.data) {  
        // TODO  
  
    } else if (data > node.data) {  
        // TODO  
  
    }  
}
```

## add method (naïve)

```
public void insert(Node node, T data) {  
    if (data < node.data) {  
        if (node.left != null) {  
            insert(node.left, data);  
        } else {  
            node.left = new Node(data);  
        }  
    } else if (data > node.data) {  
        if (node.right != null) {  
            insert(node.right, data);  
        } else {  
            node.right = new Node(data);  
        }  
    }  
}
```

How can we modify this method to not be void ?

## add method (without side-effect)

```
Node add(Node root, T data) {  
    if (root == null) {  
        Node temp = new Node(data, root, null, null);  
        root = temp;  
    } else if (data < root.data) {  
        root.left = add(root.left, data);  
    } else {  
        root.right = add(root.right, data);  
    }  
  
    return root;  
}
```

How can we avoid recursion?

# add method (iterative)

```
Node add(Node root, T data) {
    Node curr = root;

    // pointer to store parent node of current node
    Node parent = null;

    if (root == null) {
        return new Node(data, null, null);
    }

    // traverse the tree and find parent node of data
    while (curr != null)
    {
        parent = curr;

        if (data < curr.data) {
            curr = curr.left;
        } else {
            curr = curr.right;
        }
    }

    // construct a new node and assign to appropriate parent pointer
    if (data < parent.data) {
        parent.left = new Node(data, null, null);
    } else {
        parent.right = new Node(data, null, null);
    }

    return root;
}
```

# Exercise

## Height of tree

```
int height(Node node) {  
    if (node == null || (node.left == null && node.right == null)) {  
        // TODO: base case ...  
    } else {  
        return 1 + /* TODO */;  
    }  
}
```



# Height of tree

```
int height(Node node, int depth) {  
    if (node == null || (node.left == null && node.right == null)) {  
        return 0;  
    } else {  
        return 1 + Math.max(height(node.left), height(node.right));  
    }  
}
```

# Homework!

```
void range(T[] rslt, int index, Node n, T lo, T hi, Comparator<T> comp) {  
    // TODO: write a method that flattens the BST into an array  
    // where the result are in range lo and hi using the comparator  
    // Also, code should avoid traversing the subtree if it's out of range  
  
}  
  
T[] rslt = (T[]) new Object[manyItems];    // worst case!  
T lo = ...;  
T hi = ...;  
Comparator<T> comp = ...;  
  
range(rslt, 0, root, lo, hi, comp);
```