Non-Linear Structures - Trees

Learning Objectives

1. Understand and define various tree structures
2. Understand and implement various algorithms with the tree structures

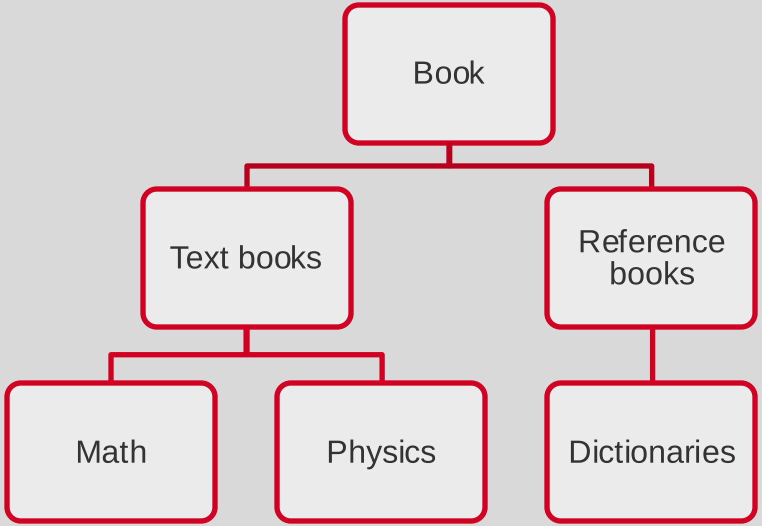
Agenda

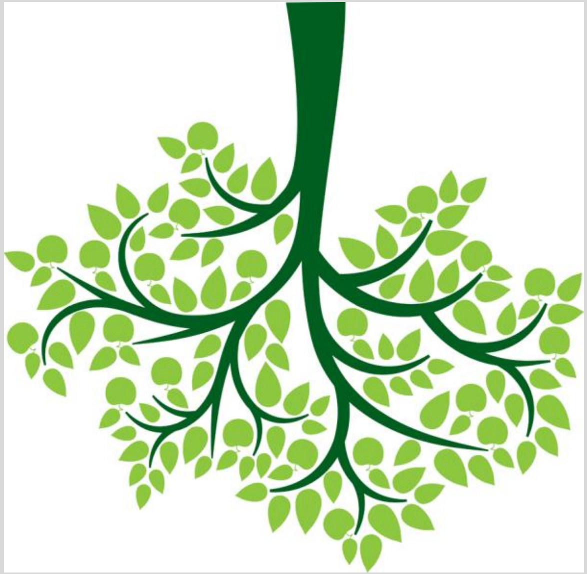
- Tree
  - Tree
  - Binary Tree
  - Tree Traversal
  - Binary Search Tree
  - Balanced Tree

Tree

What is a Tree?

- Suppose we need a data structure to keep track of book types in a library. Can a list or an array implement this structure? Is it efficient?





What is a Tree?

- An efficient structure to implement hierarchical data
- A hierarchy of nodes → node: data + references (sounds familiar?)
- Starts from root → expands to branches → ends at leaves

Terms for Describing a Tree

- Top node: **root**
- The sequence of connections (arcs) between root and a node: **path**
  - number of arcs in path: **path length**
  - number of nodes in path to node x (= path length + 1): **level of x**
- Node at the lower end of each path: **leaf node**
- Total number of nodes: **size**

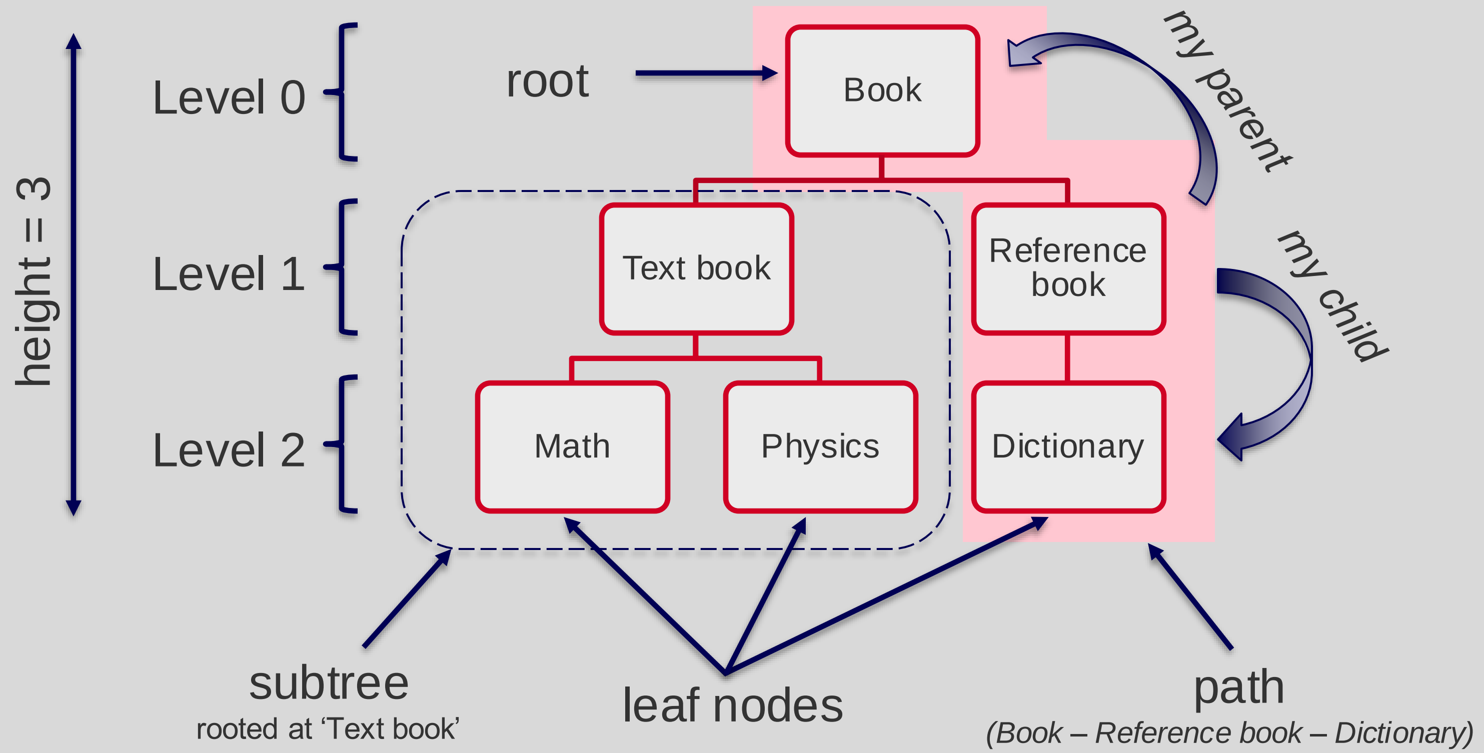
Terms for describing a Tree

- A node connected directly above another node: **parent**
  - parent, parent of parent, …: **predecessors/ancestors**
- A node connected directly below another node: **child**
  - children, children of children, …: **successors**
- Sometime, order of children matters: **ordered tree**
- Number of nodes in the longest path from root to a leaf node: **height/depth of tree**
- Set of all nodes connected under a certain node: **subtree**

Terms for describing a Tree

- Parent may have multiple children
    - But child has only one parent
- Root node is level 0 (*sometimes 1*), has no parent
    - Level refers to the path from the root to the node
- Leaf node has no children

Tree Example



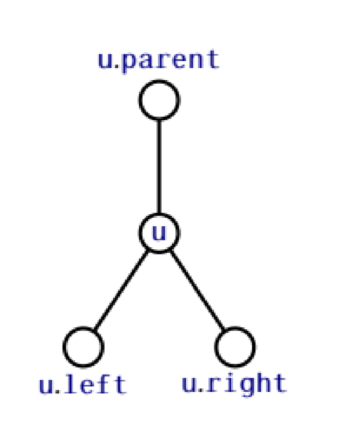
Binary Tree

Binary Tree

- **Binary Tree:** each node has at most 2 children.
- Normally, binary trees are **ordered**: distinguished **left-child** and **right-child**

→ How many references a node should have?

<table>
<tr><td colspan="2">parent</td></tr>
<tr><td colspan="2">data</td></tr>
<tr><td>left</td><td>right</td></tr>
</table>



Binary Tree Traversal

Traversal is process to visit nodes in tree:

- **Depth First:** proceed as far as possible to the left/right *(recursive)*
    - pre-order
    - in-order
    - post-order
- **Breadth First**: proceed layer by layer, left to right *(iterative)*

Binary Tree Traversal

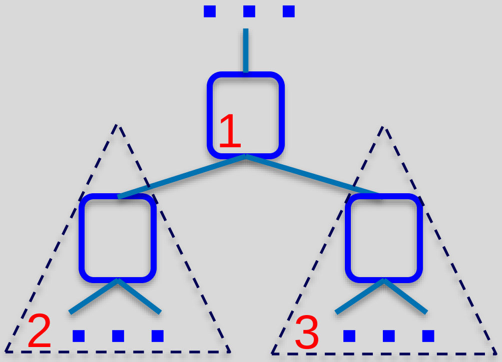
- **Visit**: temporary stop at the node, do something with its data
- Node can be referred to many times, but should be visited only once.
- The initial reference is always **root**.

→ Traversal could be used to re-compute the size of the tree

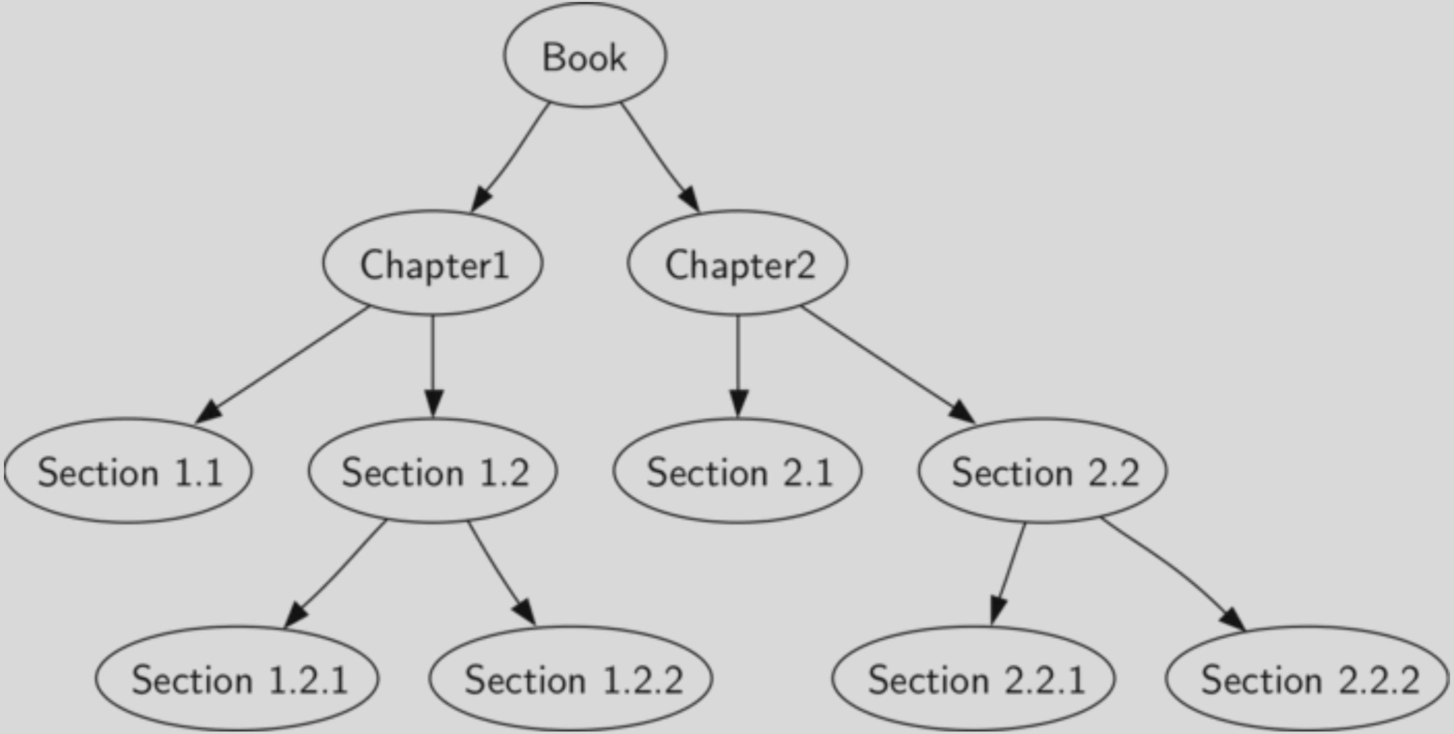
Pre-order Traversal

- At each node, visit: node → left subtree → right subtree

```
private void traversePreRecursive() {
    print("\nPre-order traversal recursive: ");
    preRecursive(root);
}
private void preRecursive(BTNode node) {
    if (node != null)  {
        print(" " + node.data); //node visit
        preRecursive(node.left);
        preRecursive(node.right);
    }
}
```



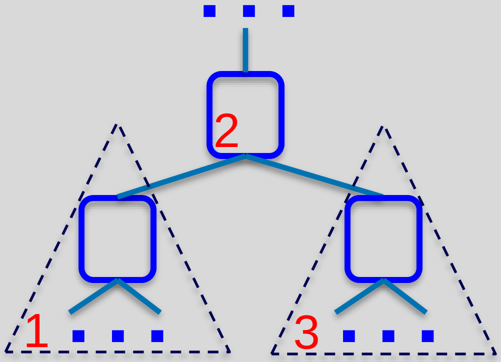
Pre-order Traversal



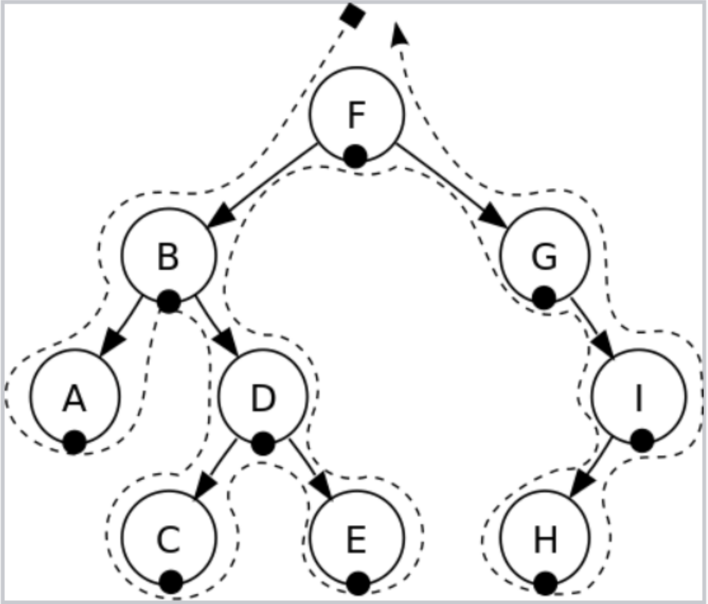
In-order Traversal

- At each node, visit: left subtree → node → right subtree

```
private void traverseInRecursive() {
    print("\nIn-order traversal recursive: ");
    inRecursive(root);
}
private void inRecursive(BTNode node) {
    if (node != null)  {
        inRecursive(node.left);
        print(" " + node.data); //node visit
        inRecursive(node.right);
    }
}
```



In-order Traversal

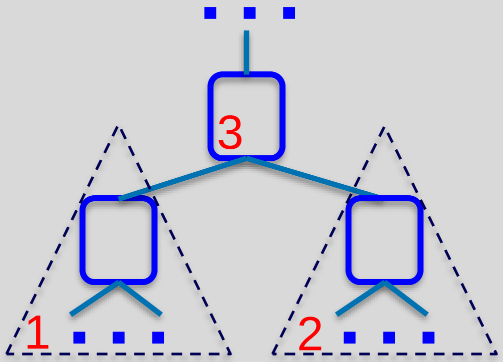


In-order: A, B, C, D, E, F, G, H, I.

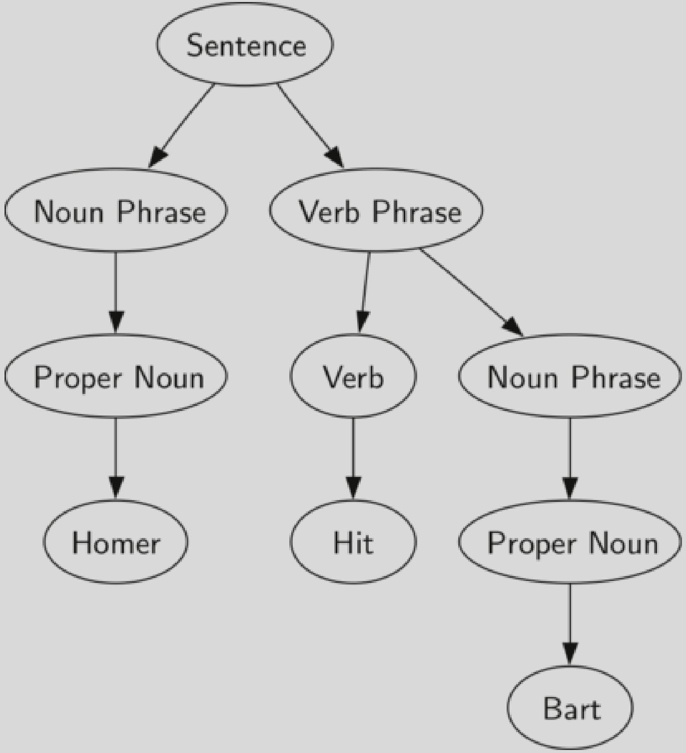
Post-order Traversal

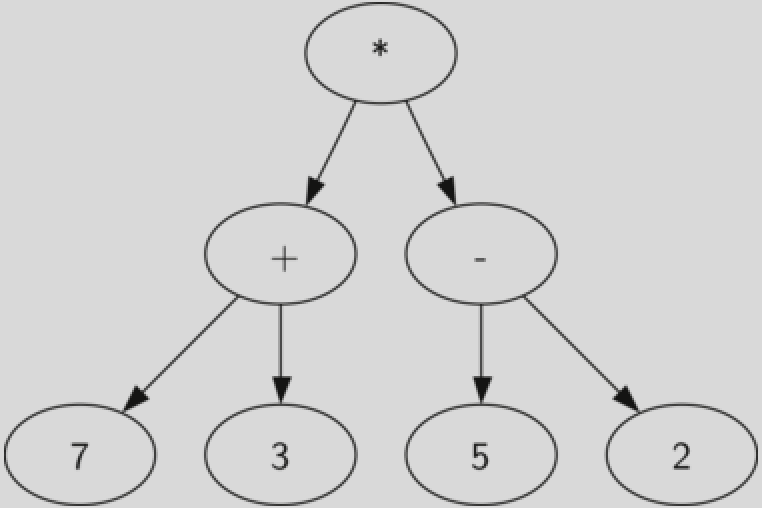
- At each node, visit: left subtree → right subtree → node

```
private void traversePostRecursive() {
    print("\nPost-order traversal recursive: ");
    postRecursive(root);
}
private void postRecursive(BTNode node) {
    if (node != null)  {
        postRecursive(node.left);
        postRecursive(node.right);
        print(" " + node.data); //visit node
    }
}
```



Post-order Traversal

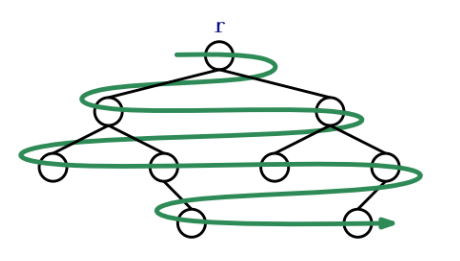




Breadth First Traversal

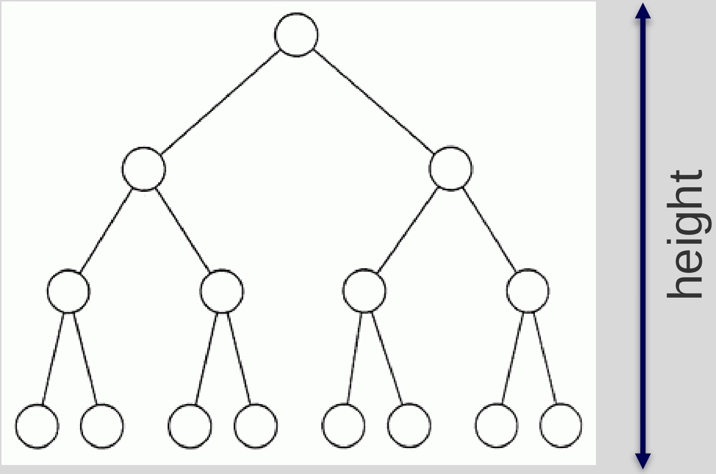
- Start from root, visit nodes layer by layer, from left to right

```
private void bfTraverse() {
    print("\nBreadth-first traversal iterative: ");
    Queue q = new Queue();

    if (root != null) q.enQueue(root);

    while (!q.isEmpty()) {
        Node node = q.peekFront();
        q.deQueue();
        print(" " + node.data); // visit node
        if (node.left != null) q.add(node.left);
        if (node.right != null) q.add(node.right);
    }
}
```



Why Binary Tree?

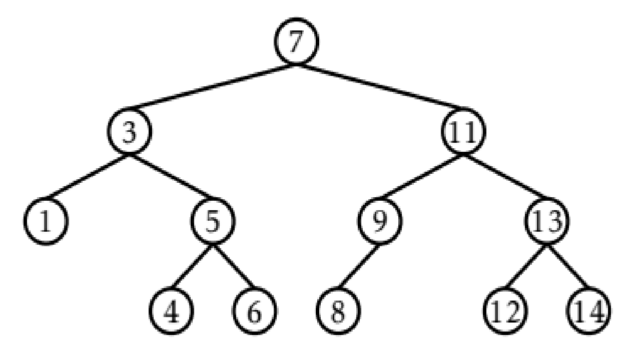
- Start from root, it takes maximum ***h*** steps to reach an arbitrary node (***h*** is height of the tree)
- Binary tree can be configured to have ***h*** = log(**tree size**).



Binary Search Tree (BST)

- A special kind of binary tree
- **Binary search tree property:** at each node, **key data** of that node is greater than all **key data** in the left subtree, and is smaller than all **key data** in the right subtree

→ Why **key data** rather than just **data**?



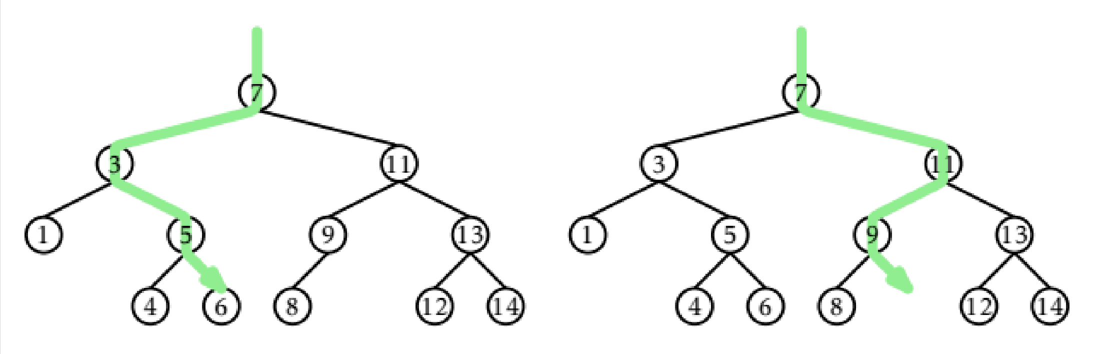
Value of key data increases

BST – Search

- The BST property is extremely useful to quickly locate a value **x** in the tree.
- Start from the root **r**, at each node **u**, there are three cases:
  1. If **x** < **u**.*data* → search **u**.left;
  2. If **x** > **u**.*data* → search **u**.right;
  3. If **x** == **u**.*data* → found the node **u** containing **x**.
- The search terminates when Case 3 occurs, or when **u == null**
- If **u** == **null**, **x** is not in the tree

BST – Search

```
BTNode find(int x) {        //search for node with key x
    BTNode node = root;
    while (node != null) {
        if (x < node.data) node = node.left;
        else if (x > node.data) node = node.right;
        else return node;
    }
    return null;
}
```

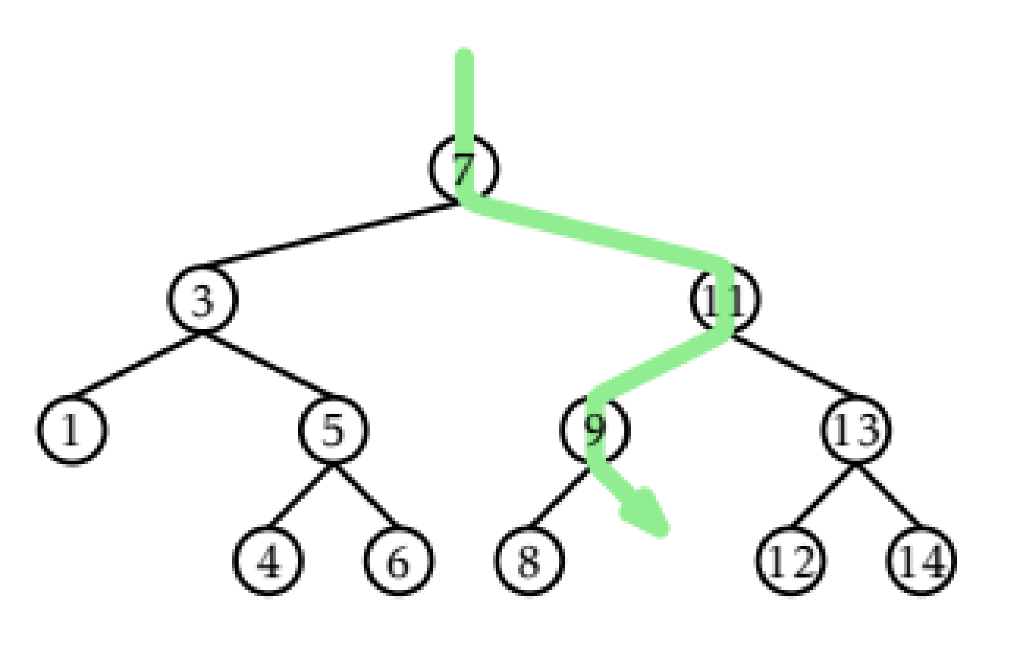


BST – Insert

- To maintain the BST property, each key value has to be unique
- Search for the key value to be added:
  1. Already exist → does not need to (cannot) be added
  2. Does not exist → can be added as a child of an appropriate existing node → which node?
  3. If key does not exist, the last visited node in the tree should become parent for the new node

BST – Insert

Add 10



BST – Insert

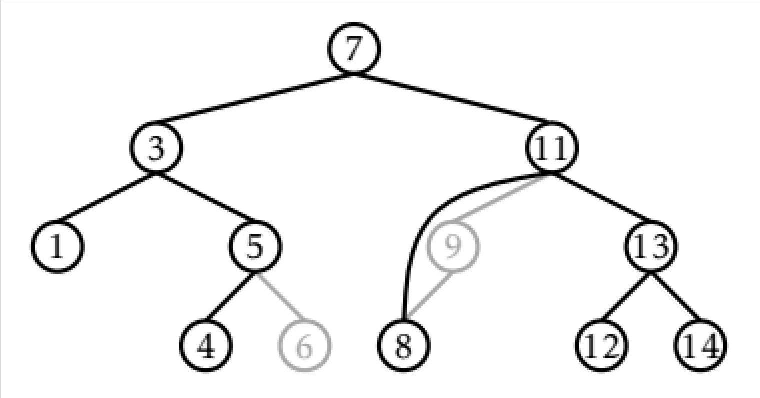
```java
// add a new value to this BST, no duplication is allowed
// return the new added node or null (if the value exists)
public BinaryTreeNode<T> add(T value) {
    if (root == null) {
        root = new BinaryTreeNode<T>(parent:null, value);
        size++;
        return root;
    }
    BinaryTreeNode<T> node = root;
    while (node != null) {
        // left or right?
        if (value.compareTo(node.data) < 0) {
            if (node.left == null) {
                BinaryTreeNode<T> newNode = new BinaryTreeNode<T>(node, value);
                node.left = newNode;
                size++;
                return newNode;
            }
            node = node.left;
        } else if (value.compareTo(node.data) > 0) {
```

```java
        } else if (value.compareTo(node.data) > 0) {
            if (node.right == null) {
                BinaryTreeNode<T> newNode = new BinaryTreeNode<T>(node, value);
                node.right = newNode;
                size++;
                return newNode;
            }
            node = node.right;
        } else {
            // duplication
            return null;
        }
    }
    // this return statement will never run
    // but the code won't compile without it
    return null;
}
```

BST – Remove

First, search the key value and return the **node** reference

- **Case 1:** If **node** is a leaf: detach **node** from its parent (update references!)
- **Case 2:** If **node** has only one child: let that child replaces **node** (splice)



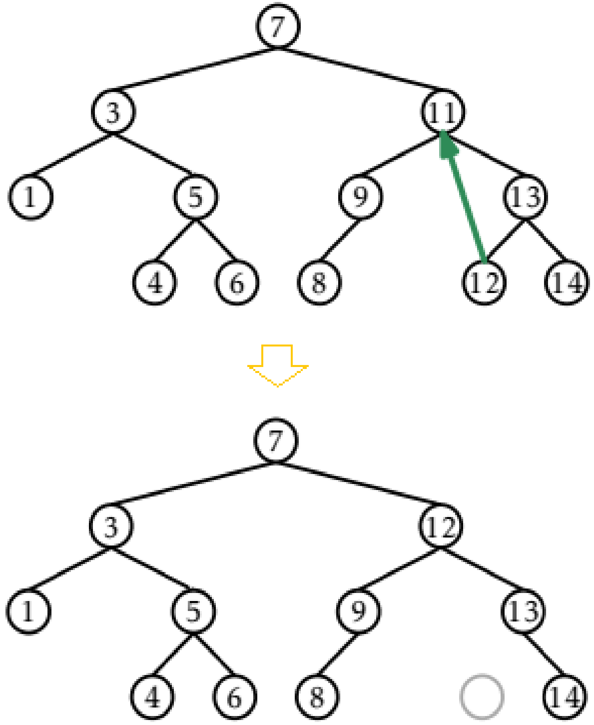
BST – Remove

- **Case 3:** If **node** has two children, find a nearby node **w** which has less than 2 children

  →node with smallest data in the right subtree

  →or node with largest data in the left subtree

- Let that node **w** replaces **u**



BST – Remove

```java
// remove a value from the tree
// return the parent node of the removed node
// OR null, if the node cannot be found or it is the root
public BinaryTreeNode<T> remove(T value) {
  // Empty tree
  if (size == 0) {
    return null;
  }
  // Step 1: find the node containing value
  BinaryTreeNode<T> node = root;
  while (node != null) {
    if (value.compareTo(node.data) == 0) {
      break;
    }
    if (value.compareTo(node.data) > 0) {
      node = node.right;
    } else {
      node = node.left;
    }
  }
  if (node == null) {
    // no node found
    return null;
  }
```

BST – Remove

```java
// Step 2A: node to be removed has no children
if (node.left == null && node.right == null) {
  // the node to be removed is root?
  if (node == root) {
    root = null;
    size = 0;
    return null;
  }
  // update the parent left or right
  if (node.parent.left == node) {
    node.parent.left = null;
  } else {
    node.parent.right = null;
  }
  size--;
  return node.parent;
}
```

```java
// Step 2B: node to be removed has one left child OR one right child
if ((node.left != null && node.right == null) ||
    (node.left == null && node.right != null)) {
  // find the correct child to replace node
  BinaryTreeNode<T> correctChild;
  if (node.left != null) {
    correctChild = node.left;
  } else {
    correctChild = node.right;
  }
  // the node to be removed is root?
  if (node == root) {
    root = correctChild;
    correctChild.parent = null;
    size--;
    return null;
  }
  // update node's parent to point to correctChild
  if (node.parent.left == node) {
    node.parent.left = correctChild;
    correctChild.parent = node.parent;
  } else {
    node.parent.right = correctChild;
    correctChild.parent = node.parent;
  }
  size--;
  return node.parent;
}
```

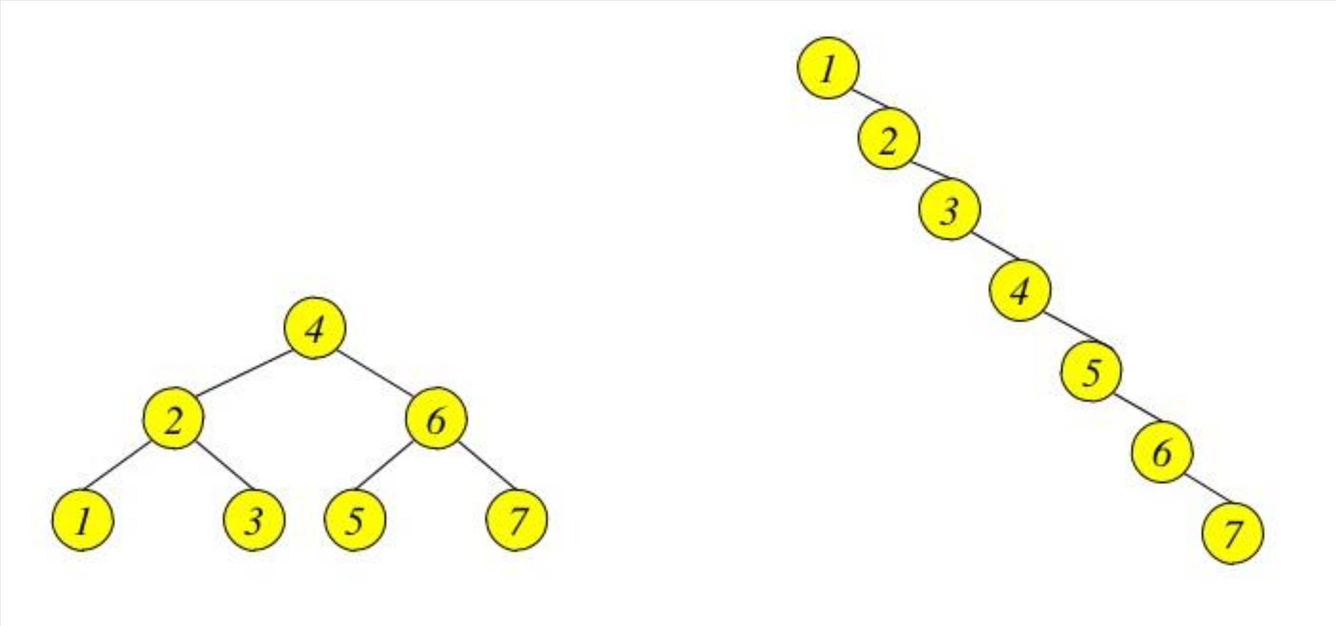
BST – Remove

```
// Step 2C: node to be removed has two children
// get the left-most node on the right subtree
// OR the right-most node on the left subtree
BinaryTreeNode<T> replaceNode = node.right;
while (replaceNode.left != null) {
  replaceNode = replaceNode.left;
}
// exchange value
T tmp = replaceNode.data;
replaceNode.data = node.data;
node.data = tmp;
// now, remove replaceNode, which is similar to step 2A and step 2B
// it is better if you create seperate methods for those operations
// for simplicity, I just put everything in the same place
```

Balanced Tree

Balanced vs. Unbalanced

- How fast to search for '**7**' in the following trees?



Balanced vs. Unbalanced

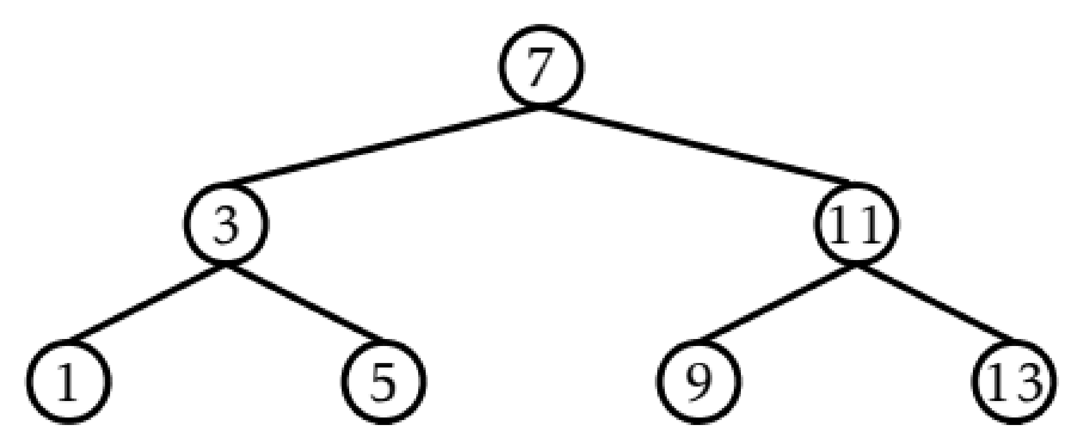
- **Balanced tree:** for every node in the tree, the height of its left subtree and height of its right subtree differ no more than 1.
- **Perfectly balanced tree:** balanced, and all leaves located in two last levels
- Much of complexity of operations in trees belong to the search for the node.

→ Processing the balanced trees (do not have to be perfect) are faster

Complete Binary Tree

- **Complete binary tree:** tree that is completely filled (all parents have 2 children), all leaf nodes are in last level

→ i^{th} level has exactly 2^i nodes



Complete Binary Tree

- Height of complete binary tree

= height of a perfectly balanced tree

= log(***size***)

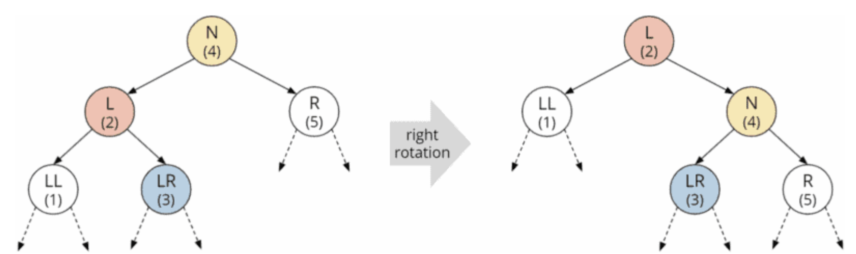
→ Maximum log(***size***) steps to reach arbitrary node!

AVL Tree

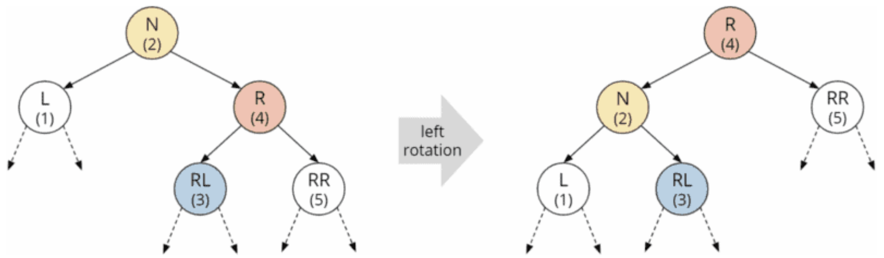
- Invented by Adelson-Velskii and Landis in 1962.
- AVL tree is balanced
- For every node in an AVL tree:
  - The difference between the left child's height and right child's height is at most 1.
  - This difference is called the balance factor.
- All sub-trees of an AVL tree are also AVL trees.

Tree Rotation

- Tree rotation is an operation to keep a tree balance.
- Tree can rotate left or rotate right.
- Rotation changes sub-trees' heights but keep the BST property.



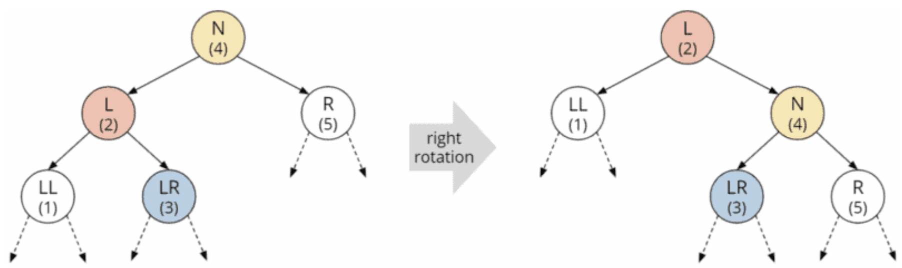
Right rotation in the AVL tree



Left rotation in an AVL tree

- Image source: https://www.happycoders.eu/algorithms/avl-tree-java/

Tree Rotation



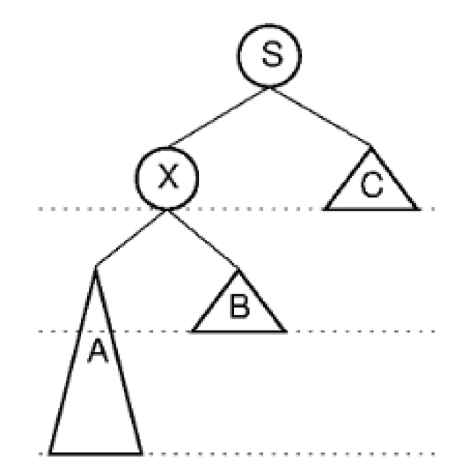
Right rotation in the AVL tree

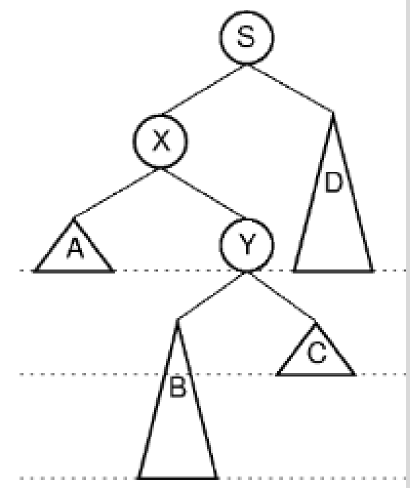
```java
// rotate right around the sub-stree rooted at node
// and return the new root
public BinaryTreeNode<T> rotateRight(BinaryTreeNode<T> node) {
    BinaryTreeNode<T> parent = node.parent;
    BinaryTreeNode<T> leftChild = node.left;
    BinaryTreeNode<T> rightOfLeftChild = leftChild.right;

    leftChild.right = node;
    node.parent = leftChild;

    node.left = rightOfLeftChild;
    if (rightOfLeftChild != null) {
        rightOfLeftChild.parent = node;
    }

    if (parent != null) {
        if (node == parent.left) {
            parent.left = leftChild;
        } else {
            parent.right = leftChild;
        }
        leftChild.parent = parent;
    } else {
        leftChild.parent = null;
        root = leftChild;
    }
    node.updateHeight();
    leftChild.updateHeight();
    return leftChild;
}
```

Balancing AVL Tree

- Add and Remove operations can make an AVL tree become unbalanced.
- Add operation
  - After an Add operation, assume the AVL tree becomes left-heavy.
    - The newly added node is on the left child.
    - But it can be on the left sub-tree of the left child OR the right sub-tree of the left child.

Left-Heavy AVL Tree





The newly added node is on the left sub-tree of the left child (case 1)

The newly added node is on the right sub-tree of the left child (case 2)

Image source: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/AVL.html

Balancing AVL Tree

- Case 1: A single right rotation around root (S) is needed

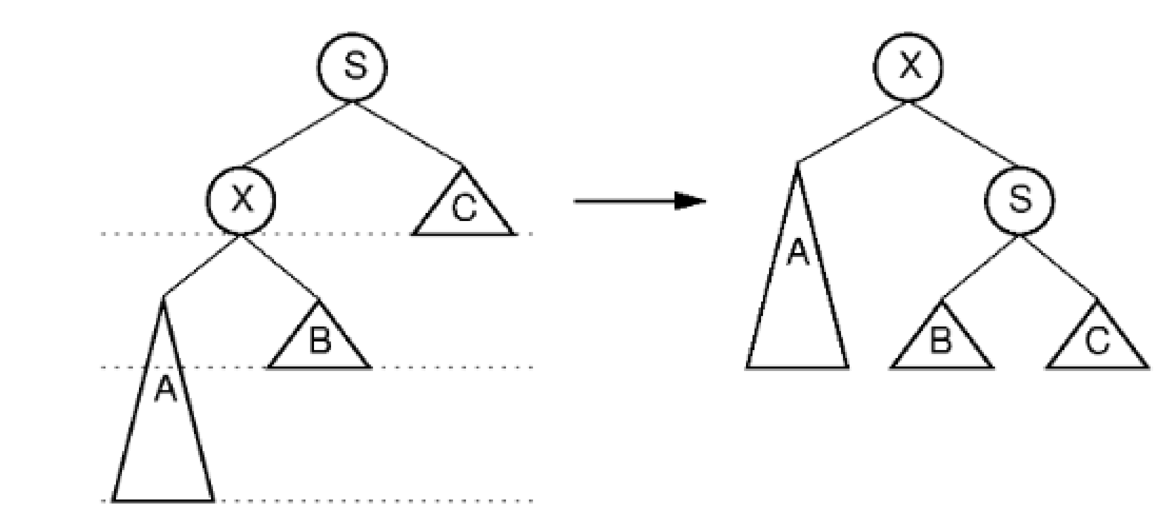


Image source: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/AVL.html

Balancing AVL Tree

- Case 2: Two rotations are needed:
  - A left rotation around root's left child (X).
  - A right rotation around root (S).

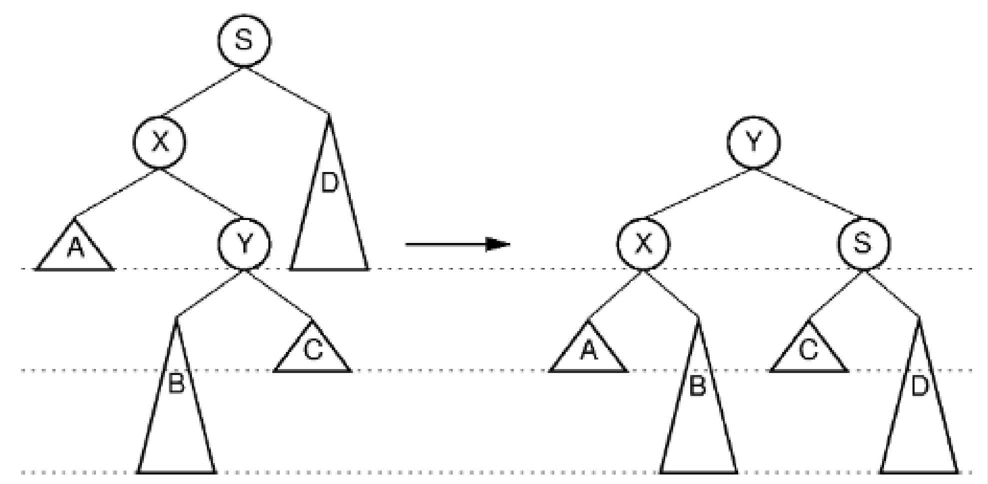
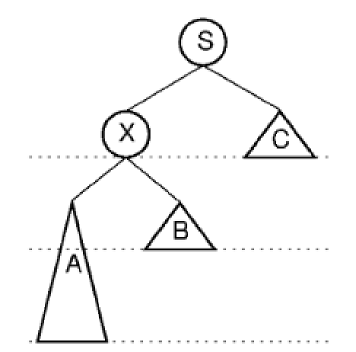


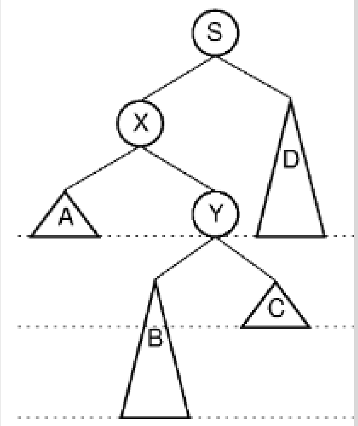
Image source: https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/AVL.html

Balancing AVL Tree

- For right-heavy AVL tree, balancing can be done similarly.
- This process works similarly for the Remove operation.
- What is the complexity of the rebalancing process?
    - Rotation works in a constant time.
    - After a node is rebalanced, the process continue with its parent.
    - The maximum number of nodes needs rebalancing is the height of the AVL tree = lg(N).
    - The complexity of Add/Remove = O(lg(N)).

Balancing AVL Tree





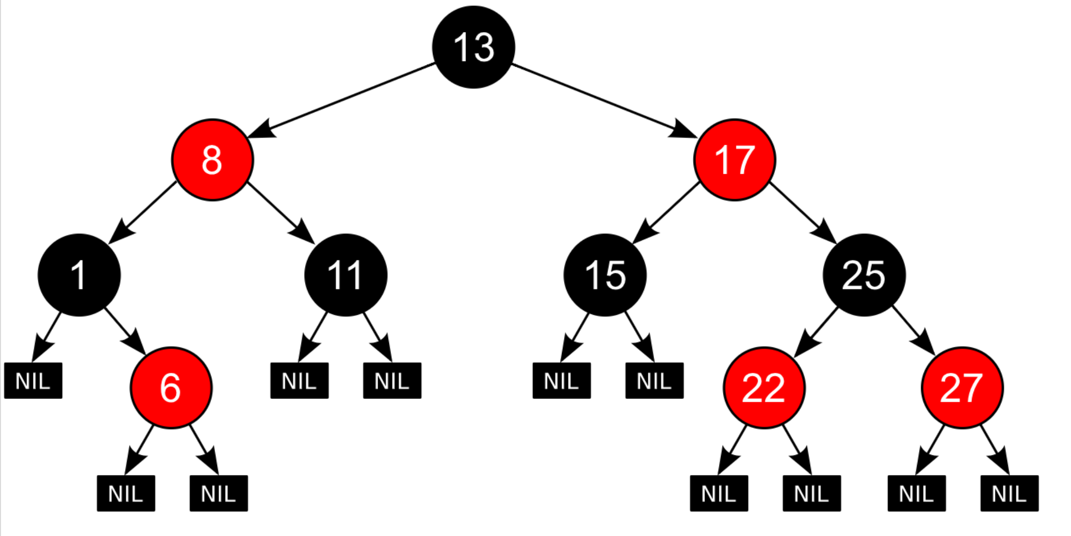

```java
// balance around a given node
// and return the new node at that location
private BinaryTreeNode<T> balanceNode(BinaryTreeNode<T> node) {
    int bf = node.getBalanceFactor();
    if (bf < -1) {
        BinaryTreeNode<T> leftChild = node.left;
        int bf2 = leftChild.getBalanceFactor();
        if (bf2 < 0) {
            return rotateRight(node);
        } else {
            rotateLeft(leftChild);
            return rotateRight(node);
        }
    } else if (bf > 1) {
        BinaryTreeNode<T> rightChild = node.right;
        int bf2 = rightChild.getBalanceFactor();
        if (bf2 > 0) {
            return rotateLeft(node);
        } else {
            rotateRight(rightChild);
            return rotateLeft(node);
        }
    }
    return node;
}
```

Red-Black Tree

- Invented by Leonidas J. Guibas and Robert Sedgewick in 1978.
- Red-Black tree is approximately balanced.
- The leaf nodes are always null (not contain data).
- For every node in a Red-Black tree:
    - Is either red or black.
    - All leaves are black.
    - A red node does not have red child.
    - Every path from a given node to any of its leaf nodes must go through the same number of black nodes.

Red-Black Tree

- Why Red-Black trees are balanced?
  - The longest path from the root to a leaf (not counting the root) is at most twice as long as the shortest path from the root to a leaf.



- Image source: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Balancing Red-Black Tree

- Tree rotation is also used to balance Red-Black trees.
- After an Add/Remove operation, the Red-Black properties are reviewed. If those properties do not hold, rotation and recoloring are executed.

Red-Black Tree vs AVL Tree

- AVL tree is more "balanced" (the balance factor is -1, 0, or 1), so searching on AVL tree is faster.
- However, as Red-Black tree requires less rebalancing, Adding and Removing data on Red-Black tree is faster.
- Depending on the operations that execute most of the time, an appropriate tree should be used.
- Java TreeMap implementation is based on Red-Black tree.