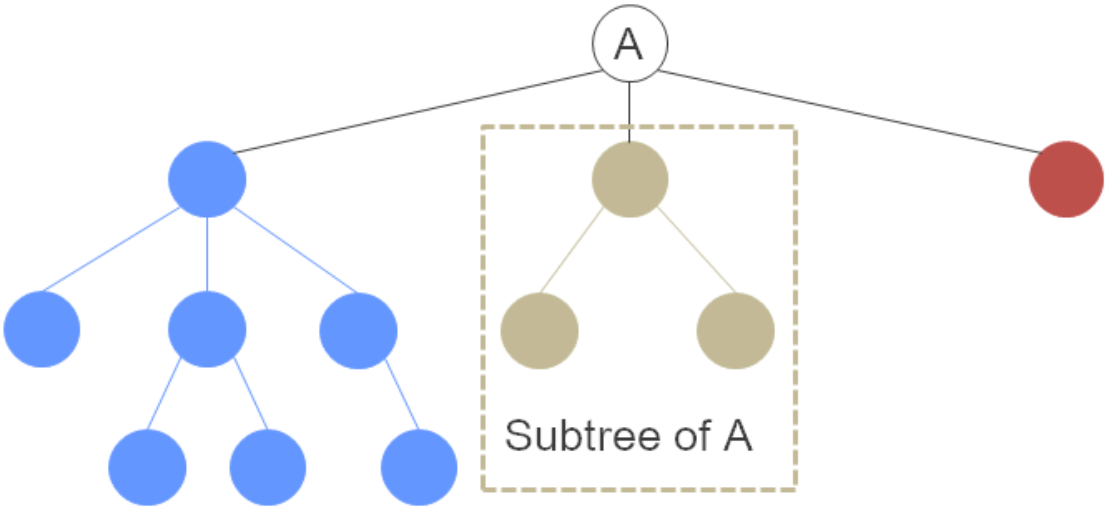
二叉搜索树 Binary Search Trees

当日常生活中接收到一堆文件的时候，我们可能会本能地想到要根据特定的方案来构建一个有效的结构，使得我们每次在寻找这些文件的时候都能快速有效地寻找到目标，而在编程任务中我们同样也需要处理类似的问题。而二分搜索树就是一个可以运用在电脑中的结构，它用简单的模型构建出了一个有效的搜索方案。

树

结构

树是节点的集合，而集合可以是空的，如果不是空的，树则由一个区分节点 r（根 root）和零个或多个非空子树 T_1、T_2、....、T_k 组成，每个子树的根都由一条来自 r 的有向边连接着



树

一个树有以下属性：

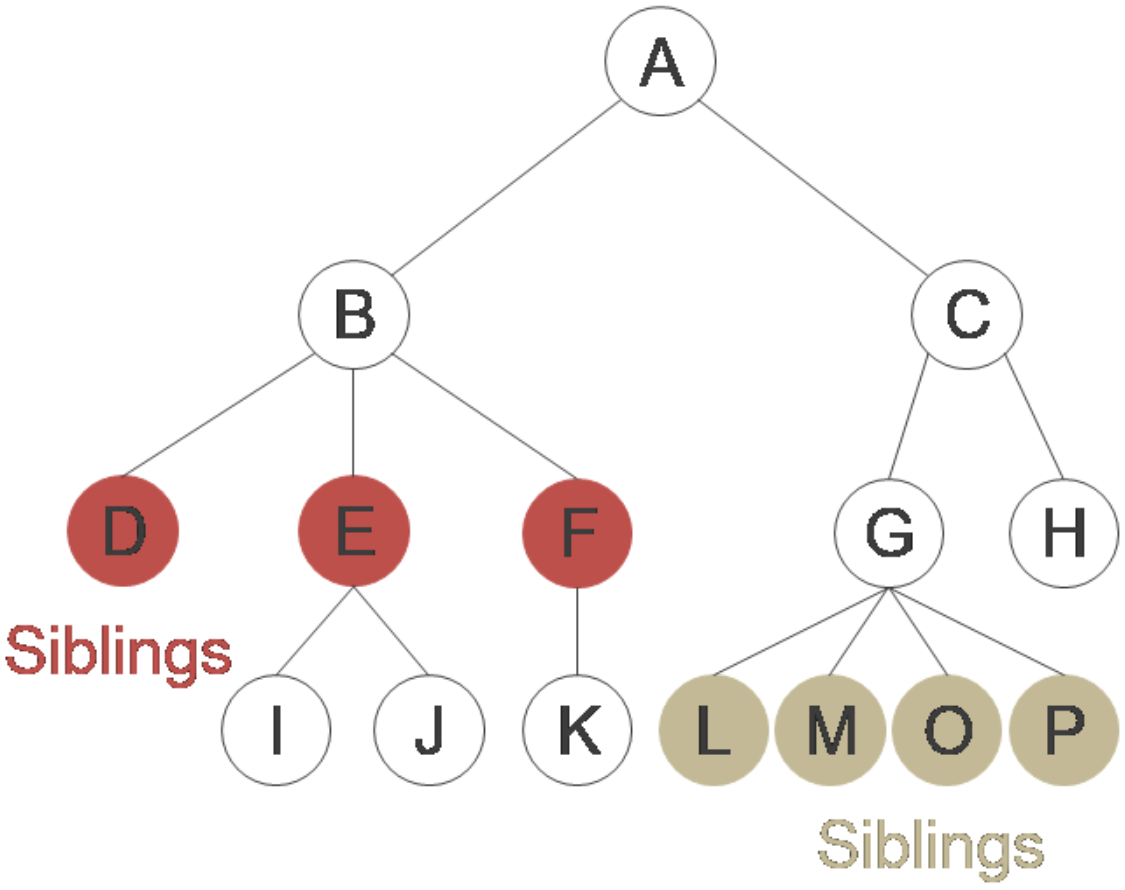
- 根 (root) 与叶 (leaf)
- 子节点 (Child) 和父节点 (Parent)

  除根节点外，每个节点都有一个父节点

  一个节点可以有零个或多个子节点

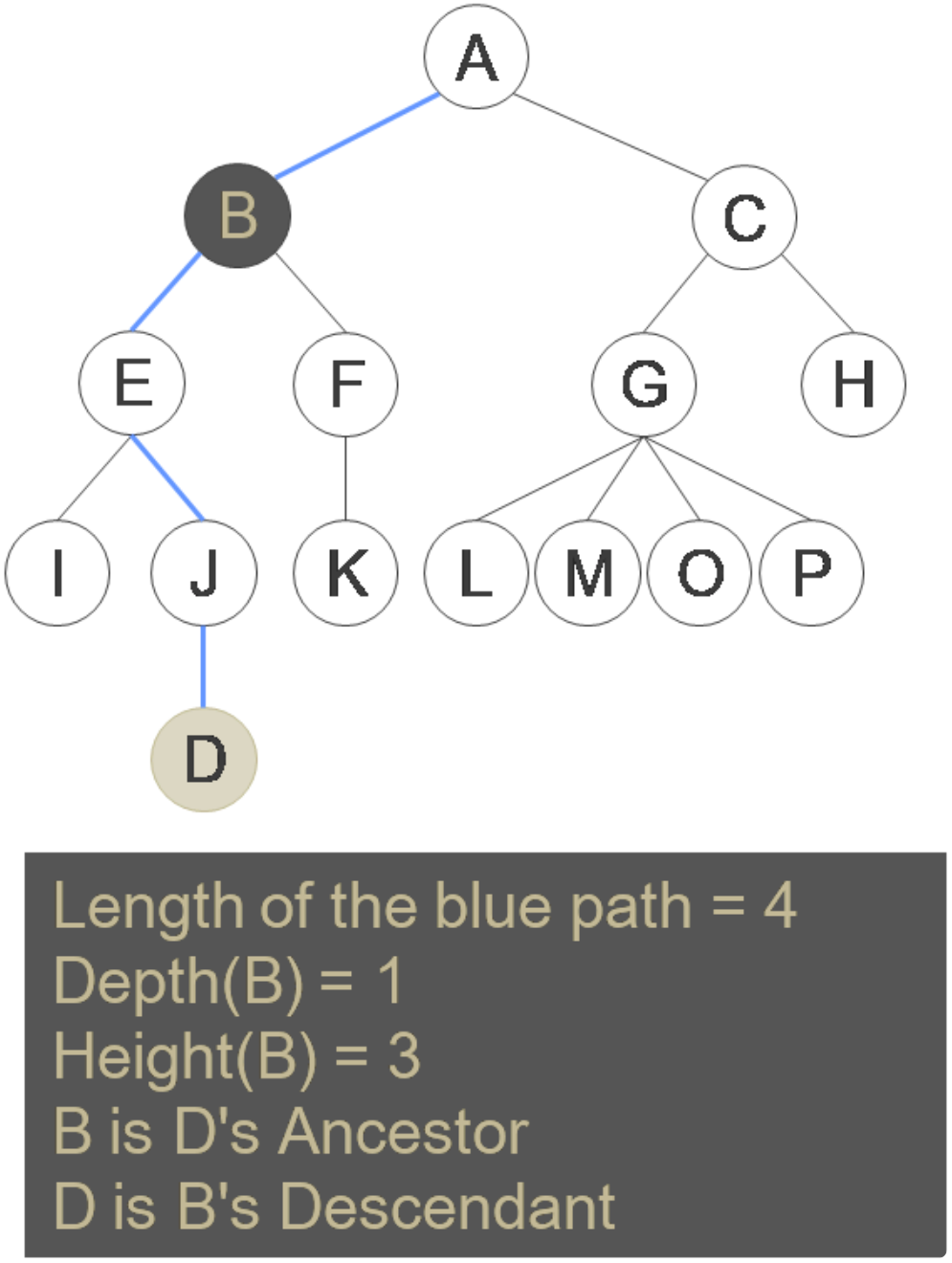
  叶节点没有子节点
- 同父节点 (Sibling)

  具有相同父节点的节点



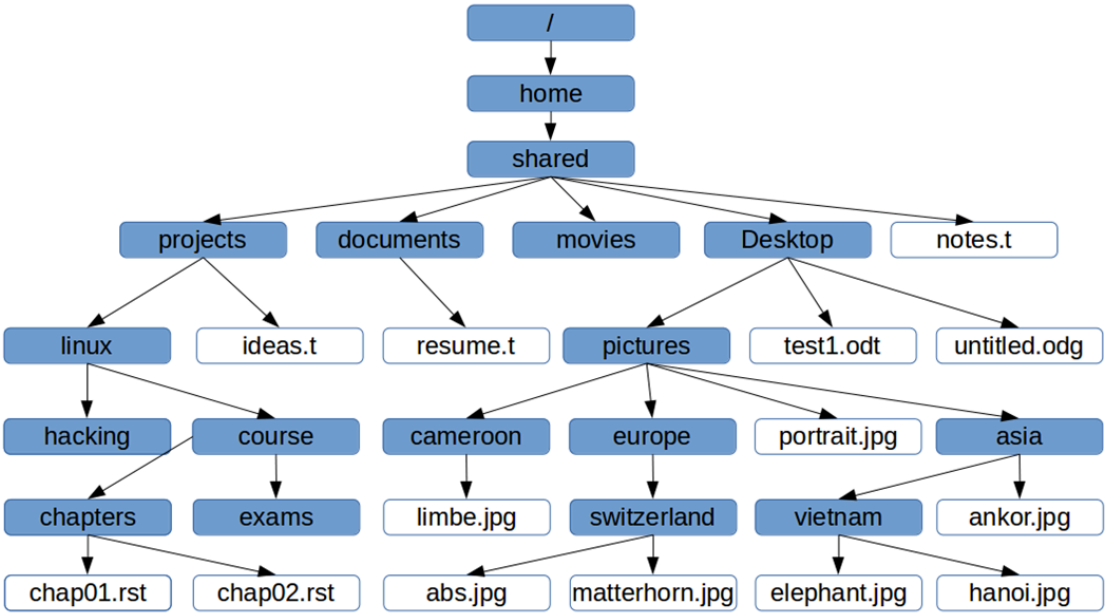
树的结构

- 路径 (Path)
  一连串的边
- 路径长度 (Length of a path)
- 节点深度 (Depth of a node)
  通向根节点的唯一路径的长度
- 节点高度 (Height of a node)
  到叶的最长路径长度
- 树高 (Tree height)
  根的高度
  最深树叶的深度
- 祖先和后代 (Ancestor and descendant)
  如果有一条从 n1 到 n2 的路径
  n1 是 n2 的祖先，n2 是 n1 的后代



树的属性

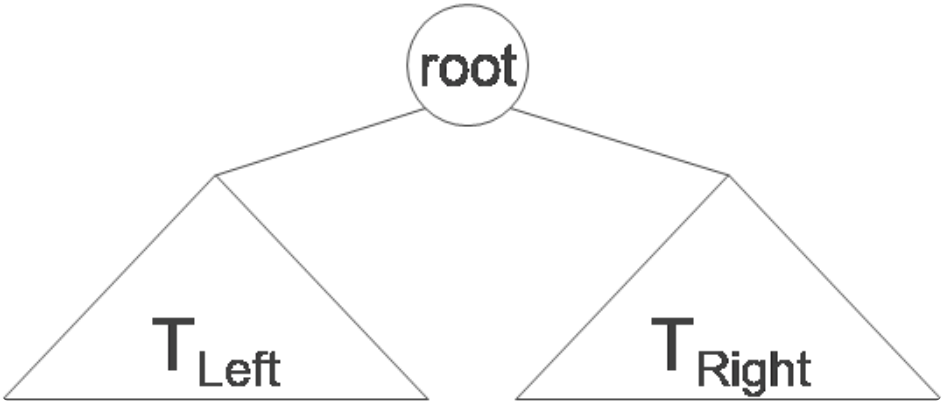
在 UNIX 系统中，文件的目录结构也是一个树，如下图所示。



UNIX目录结构

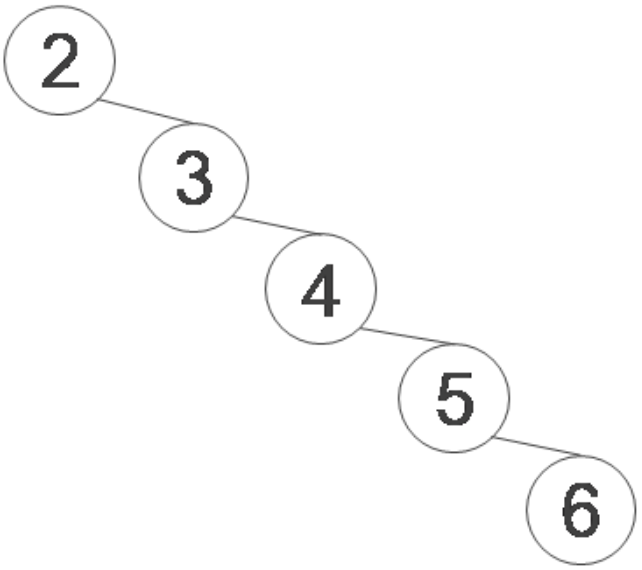
二叉树 Binary Trees

在二叉树中，所有的父节点至多只能拥有两个子节点。



二叉树

二叉树的平均深度远远小于 N，尽管在最糟糕的情况下，深度可能高达 N - 1，如下图所示。



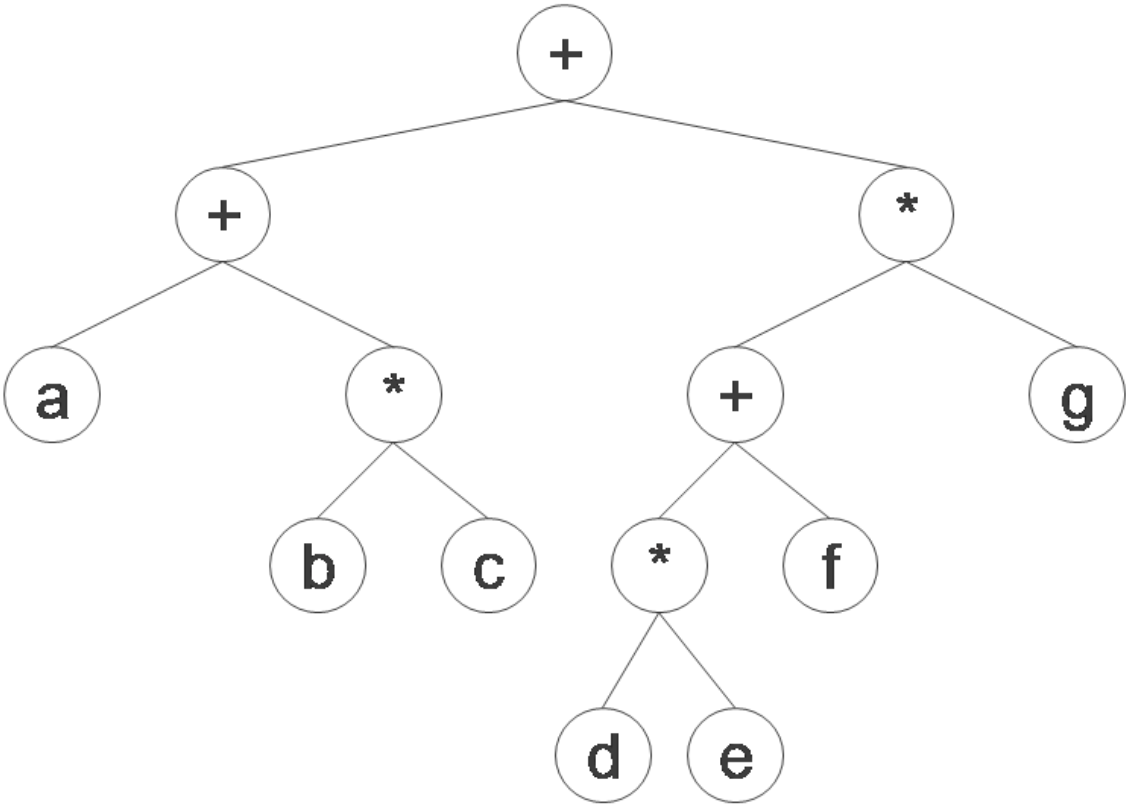
最差情况二叉树

遍历

二叉树的遍历分为前序遍历 (Pre-order)，中序遍历 (In-order) 和后序遍历 (Post-order)。

前序遍历 Pre-order

顺序：父节点，左节点，右节点



Expression tree for: (a + b*c) + (d*e + f) * g

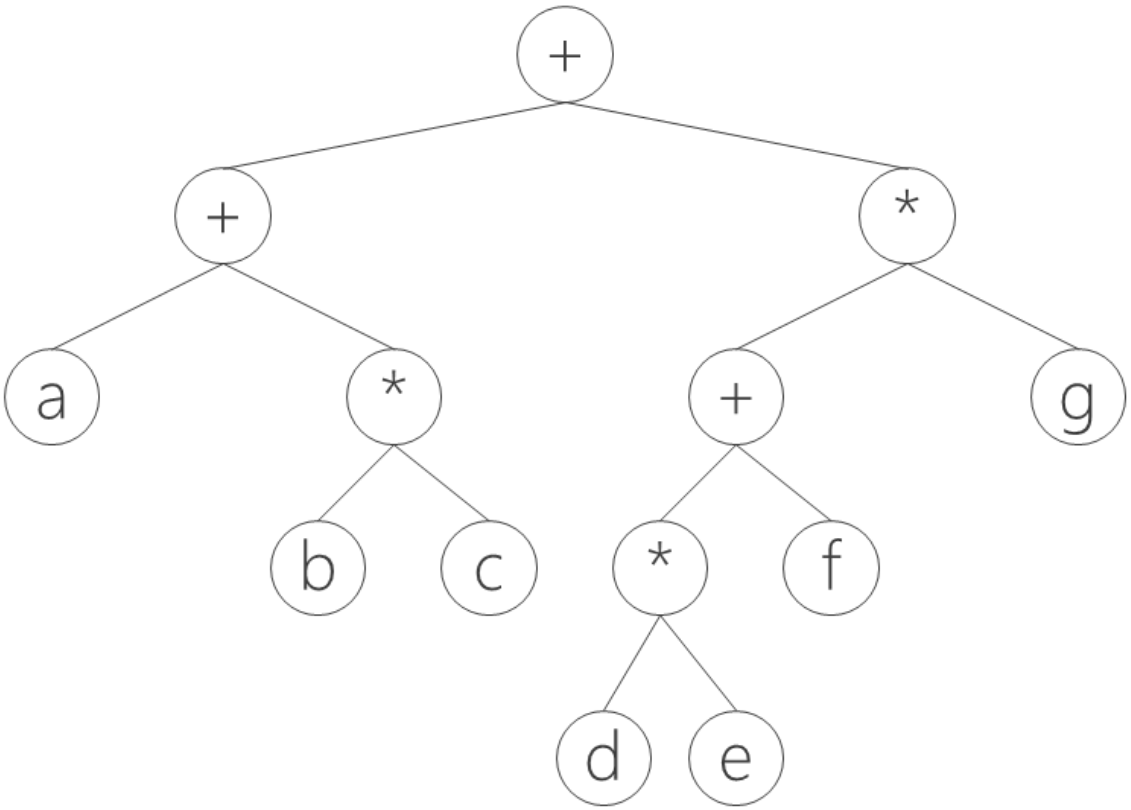
前序遍历

代码实现

```
PREORDER(root)
   IF root = Null
      return
   PRINT(root)
   PREORDER(LEFT(root))
   PREORDER(RIGHT(root))
```

中序遍历 In-order

顺序：左节点，父节点，右节点



Expression tree for: (a + b*c) + (d*e + f) * g

中序遍历

代码实现

```
INORDER(root)
   IF root = Null
      return
   INORDER(LEFT(root))
   PRINT(root)
   INORDER(RIGHT(root))
```

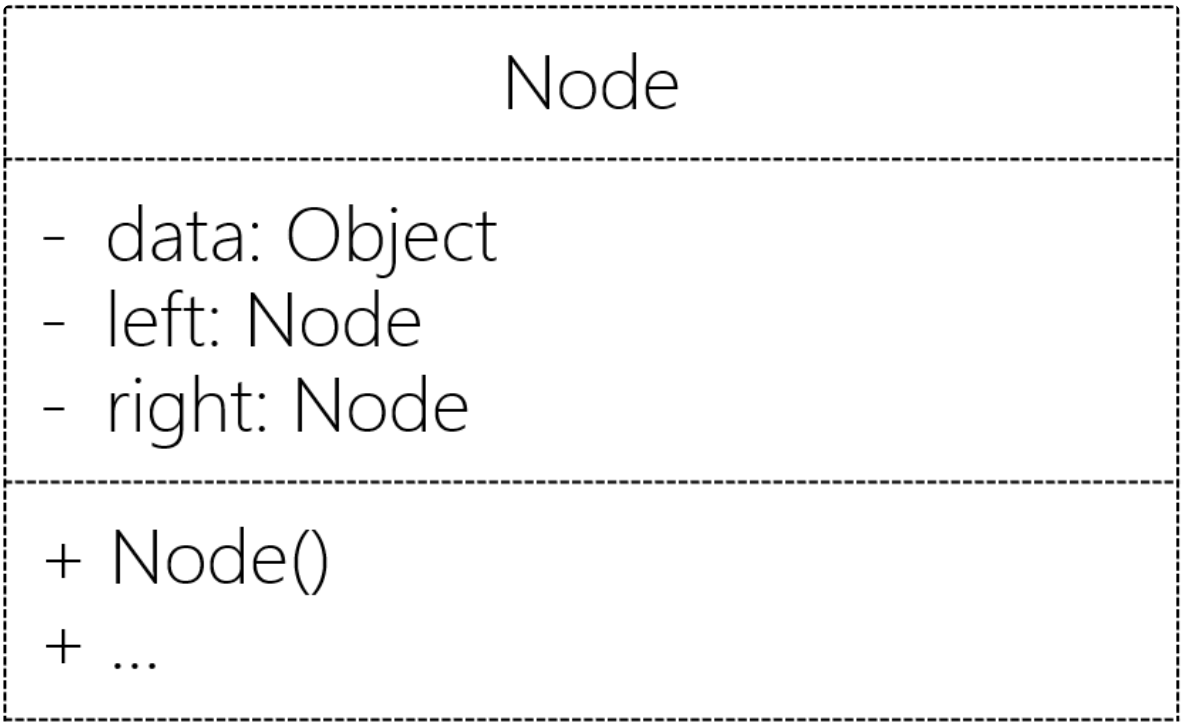
后序遍历 Post-order

顺序：左节点，右节点，父节点

代码实现

```
POSTORDER(root)
   IF root = Null
      return
   POSTORDER(LEFT(root))
   POSTORDER(RIGHT(root))
   PRINT(root)

```

节点结构



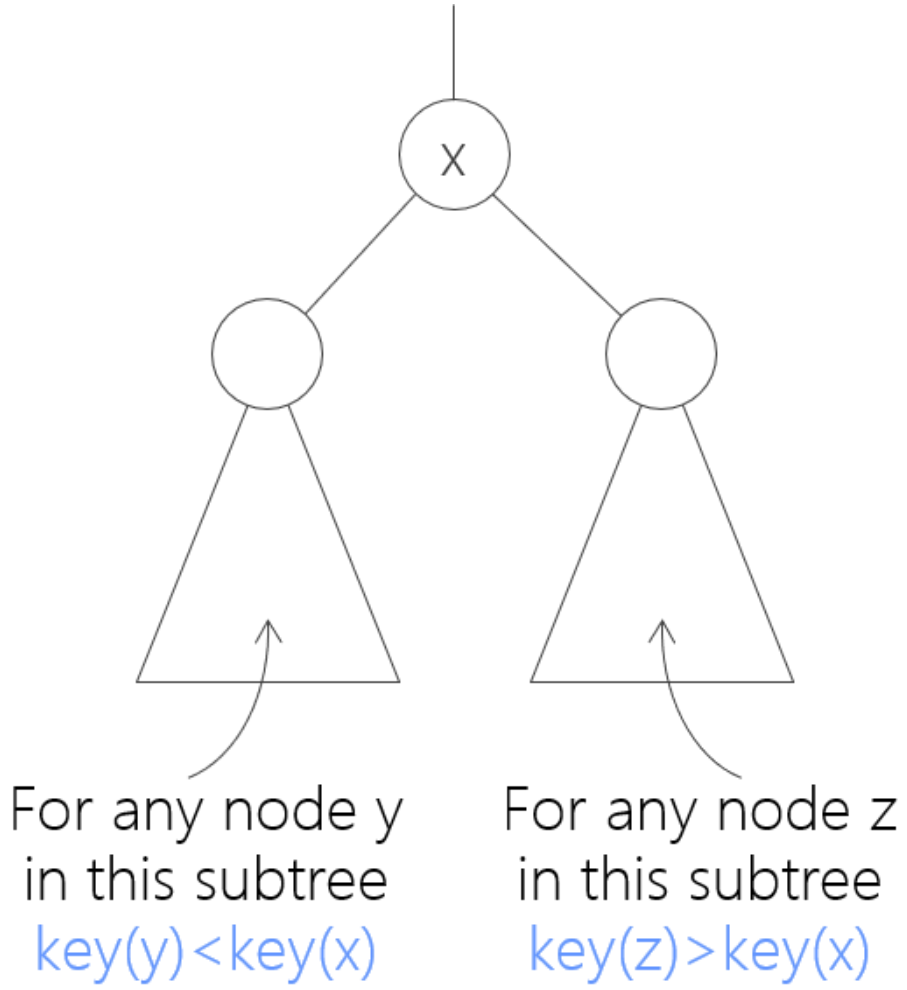
节点结构

代码实现

```java
public class Node{
    private int key;
    private Node left, right;
    public Node(int key) {
        this.key = key;
        this.left = null;
        this.right = null;
    }


    public int getKey() {return key;}
    public Node getLeft() {return left;}
    public Node getRight() {return right;}

    public void setkey(int key) {this.key= key;}
    public void setLeft(Node node) {left = node;}
    public void setRight(Node node) {right= node;}

    /** Get text to be printed */
    public String getText() {return String.valueOf(key);}
}
```

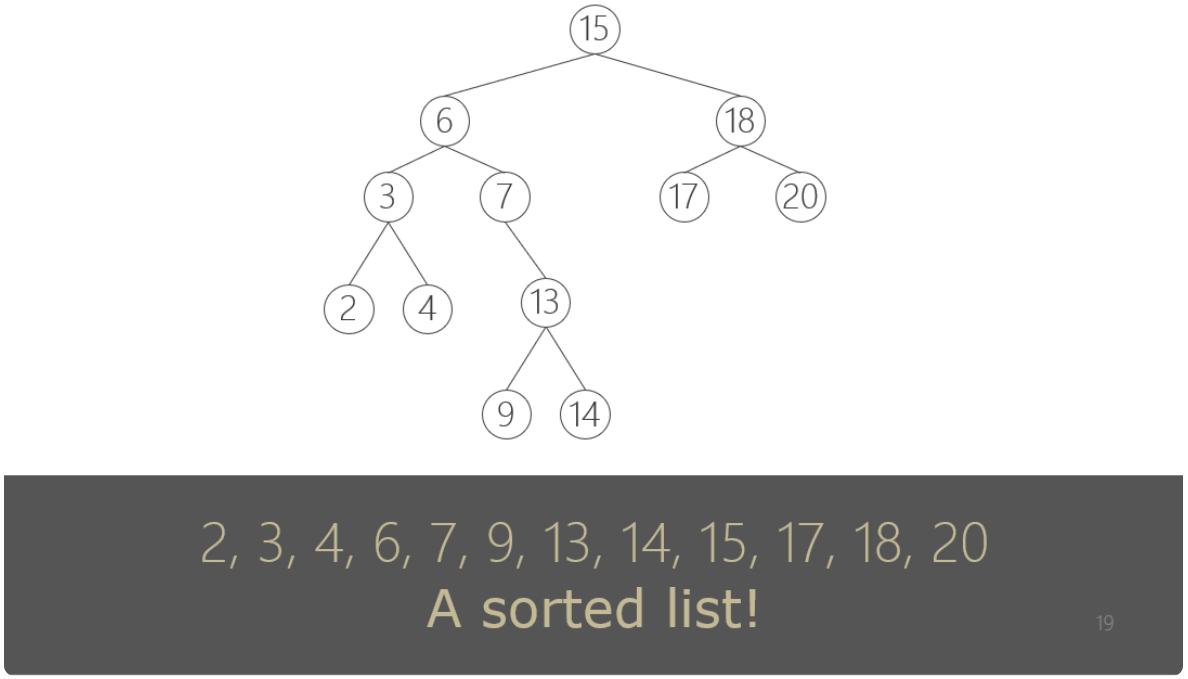
二叉搜索树 Binary Search Trees

在二叉搜索树中，对于每个节点 X，其左侧子树上的所有键值都小于 X 中的键值，其右侧子树上的所有键都大于 X 中的键值。



二叉搜索树

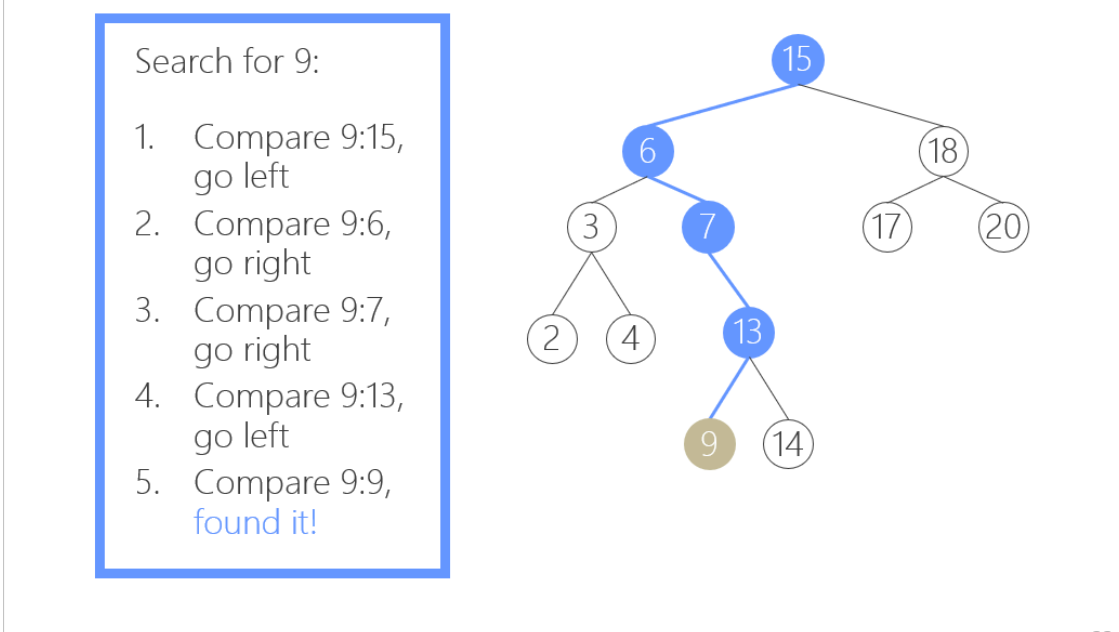
值得注意的是，当使用**中序遍历**导出节点的话，一般可以将树中的元素以排列好的顺序展现出来。



二叉搜索树与排列

检索过程

在二叉搜索树中搜索一个元素较为简单，只需要从父节点开始与节点元素比较大小，并导向正确的方向即可，即大于此节点元素时向右走，小于此节点元素时向左走，当检索结束却未找到目标元素时，返回null即可。



检索过程

代码实现

检索元素

时间复杂度约为 $O(\text{树高})$

```
FIND(root, x)
   IF root=Null
      return Null
   IF root.key=x
      return root
   IF root.key>x
      return FIND(root.left, x)
   return FIND(root.right, x)
```

```java
public Node findNode(int x) {
    return findNode(root, x);
}

protected Node findNode(Node root, int x) {
    /**
    * findNode returns a node whose key is "x" in a subtree
whose root is "root"
    * input: root - the root of the subtree x - the key to be
found output: a node
    * whose key is "x", or null if "x" is not found
    */
    if (root == null)
        return null;
    if (x < root.getKey())
        return findNode(root.getLeft(), x);
    if (x > root.getKey())
        return findNode(root.getRight(), x);
    return root;
}
```

寻找最大/最小值

时间复杂度约为 $O(\text{树高})$

```
FIND-MIN(root)
   IF root=Null
      return Null
   IF root.left=Null
      return root
   return FIND-MIN(root.left)
```

```
public int minKey(Node root) {
    // returns the minimum key in a subtree whose root is
"root"
    while (root.getLeft() != null) {
        root = root.getLeft();
    }
    return root.getKey();
}
```

```
FIND-MAX(root)
   IF root=Null
      return Null
   IF root.right=Null
      return root
   return FIND-MAX(root.right)
```

插入

插入的函数为 $insert(root, x)$，插入的过程就像查找一样沿着树向下查找。如果找到 x，则不做任何操作，这样做是为了剔除重复数据；否则，在遍历路径的最后一个位置插入 x。这个操作的时间复杂度约为 $O(\text{树高})$。

代码实现

```
INSERT(root, x)
   // returns the (updated) root of the subtree
   IF root=Null
      return CREATE-NODE(x)
   IF x<root.key
      root.left = INSERT(root.left, x)
   ELSE IF x>root.key
      root.right = INSERT(root.right, x)
   return root
```

```
public void insertNode(int x) {
    root = insertNode(root, x);
}

protected Node insertNode(Node root, int x) {
    /**
    * insertNode inserts a key "x" into a subtree whose root is
"root" input: root
    * - the root of the subtree x - the key to be inserted
output: the (updated)
    * root of the subtree
    */
    // Note that the root here is a parameter, not the class
member "root"

    if (root == null) { // insert the new node right here
```

```
            return new Node(x);
        }
        if (x < root.getKey()) // try to insert in the left subtree
            root.setLeft(insertNode(root.getLeft(), x));
        else if (x > root.getKey())// try to insert in the right
subtree
            root.setRight(insertNode(root.getRight(), x));
        // note that if x==root.getKey(), nothing is done
        return root;
    }
```

删除

当删除一个节点时，需要考虑五种情况来维持二叉搜索树的结构稳定。

1. 删除的节点没有子节点

   心安理得地删除

2. 删除的节点只有左节点

   以左节点代替

3. 删除的节点只有右节点

   以右节点代替

4. 删除的节点有两个子节点

   用右子树的最小节点来替代，即右子树最左边的节点

5. 没有搜索到目标

   返回null

代码实现

这个操作的时间复杂度约为 $O(\text{树高})$。

```java
public void deleteNode(int x) {
    root = deleteNode(root, x);
}

protected Node deleteNode(Node root, int x) {
    /**
    * deleteNode deletes the node whose key is "x" from a
subtree whose root is
    * "root" input: root - the root of the subtree x - the key
of the node to be
    * deleted output: the (updated) root of the subtree
    */
    // Return if the tree is empty
    if (root == null)
        return root;
    if (x < root.getKey()) // try to delete in the left subtree
        root.setLeft(deleteNode(root.getLeft(), x));
    else if (x > root.getKey()) // try to delete in the right
subtree
        root.setRight(deleteNode(root.getRight(), x));
    else {
        // delete the root
        if (root.getLeft() == null) // root has no left child,
pass the right child to grandparent
            return root.getRight();
        if (root.getRight() == null) // root has no right
child, pass the left child to grandparent
            return root.getLeft();
        // root has two children, replace root with the minimum
node in its left subtree
        root.setkey(minKey(root.getRight()));
        // delete the minimum node in its left subtree
        root.setRight(deleteNode(root.getRight(),
root.getKey()));
    }
    return root;
}
```

```java
public int minKey(Node root) {
    // returns the minimum key in a subtree whose root is
"root"
    while (root.getLeft() != null) {
        root = root.getLeft();
    }
    return root.getKey();
}
```