数据结构与算法7-二分搜索树

笔记本: 我的笔记
创建时间: 2020/10/4 8:10 **更新时间:** 2020/10/4 8:24
作者: liuhouer
标签: 算法

为什么要有树结构?

- 将数据使用树结构存储后，出奇的高效

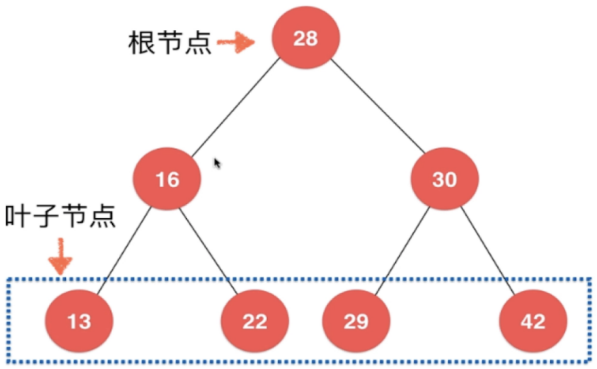
二分搜索树（Binary Search Tree）

平衡二叉树：AVL；红黑树

堆；并查集

线段树；Trie (字典树，前缀树)

二叉树

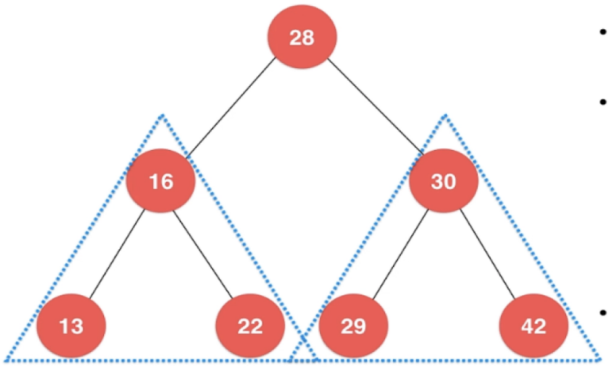


- 二叉树具有具有唯一根节点

```
class Node {
    E e;
    Node left;  ← 左孩子
    Node right; ← 右孩子
}
```

- 二叉树每个节点最多有两个孩子

二分搜索树 Binary Search Tree



- 二分搜索树是二叉树
- 二分搜索树的每个节点的值：
  - 大于其左子树的所有节点的值
  - 小于其右子树的所有节点的值
- 每一棵子树也是二分搜索树

1.自己实现一个二分搜索树

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;


public class BST<E extends Comparable<E>> {


    private class Node{
        public E e;
        public Node left, right;


        public Node(E e){
            this.e = e;
            left = null;
            right = null;
        }
    }


    private Node root;
    private int size;


    public BST(){
        root = null;
        size = 0;
    }


    public int size(){
        return size;
    }


    public boolean isEmpty(){
        return size == 0;
    }


    // 向二分搜索树中添加新的元素e
    public void add(E e){
        root = add(root, e);
    }


    // 向以node为根的二分搜索树中插入元素e，递归算法
    // 返回插入新节点后二分搜索树的根
    private Node add(Node node, E e){
```

```
        if(node == null){
            size ++;
            return new Node(e);
        }


        if(e.compareTo(node.e) < 0)
            node.left = add(node.left, e);
        else if(e.compareTo(node.e) > 0)
            node.right = add(node.right, e);


        return node;
    }


    // 看二分搜索树中是否包含元素e
    public boolean contains(E e){
        return contains(root, e);
    }


    // 看以node为根的二分搜索树中是否包含元素e, 递归算法
    private boolean contains(Node node, E e){


        if(node == null)
            return false;


        if(e.compareTo(node.e) == 0)
            return true;
        else if(e.compareTo(node.e) < 0)
            return contains(node.left, e);
        else // e.compareTo(node.e) > 0
            return contains(node.right, e);
    }


    // 二分搜索树的前序遍历
    public void preOrder(){
        preOrder(root);
    }


    // 前序遍历以node为根的二分搜索树, 递归算法
    private void preOrder(Node node){


        if(node == null)
            return;


        System.out.println(node.e);
        preOrder(node.left);
        preOrder(node.right);
    }


    // 二分搜索树的非递归前序遍历
    public void preOrderNR(){


        Stack<Node> stack = new Stack<>();
        stack.push(root);
        while(!stack.isEmpty()){
            Node cur = stack.pop();
            System.out.println(cur.e);


            if(cur.right != null)
                stack.push(cur.right);
            if(cur.left != null)
```

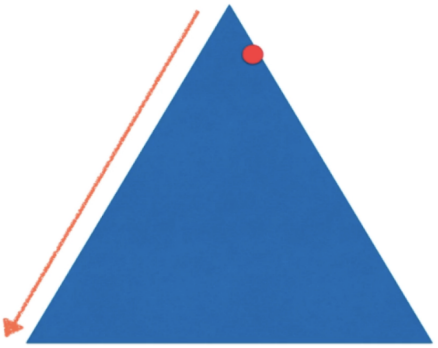
```
                stack.push(cur.left);
        }
    }


    // 二分搜索树的中序遍历
    public void inOrder(){
        inOrder(root);
    }


    // 中序遍历以node为根的二分搜索树, 递归算法
    private void inOrder(Node node){


        if(node == null)
            return;


        inOrder(node.left);
        System.out.println(node.e);
        inOrder(node.right);
    }


    // 二分搜索树的后序遍历
    public void postOrder(){
        postOrder(root);
    }


    // 后序遍历以node为根的二分搜索树, 递归算法
    private void postOrder(Node node){


        if(node == null)
            return;


        postOrder(node.left);
        postOrder(node.right);
        System.out.println(node.e);
    }


    // 二分搜索树的层序遍历
    public void levelOrder(){


        Queue<Node> q = new LinkedList<>();
        q.add(root);
        while(!q.isEmpty()){
            Node cur = q.remove();
            System.out.println(cur.e);


            if(cur.left != null)
                q.add(cur.left);
            if(cur.right != null)
                q.add(cur.right);
        }
    }


    // 寻找二分搜索树的最小元素
    public E minimum(){
        if(size == 0)
            throw new IllegalArgumentException("BST is empty!");


        return minimum(root).e;
    }


    // 返回以node为根的二分搜索树的最小值所在的节点
```

```
    private Node minimum(Node node){
        if(node.left == null)
            return node;
        return minimum(node.left);
    }


    // 寻找二分搜索树的最大元素
    public E maximum(){
        if(size == 0)
            throw new IllegalArgumentException("BST is empty");


        return maximum(root).e;
    }


    // 返回以node为根的二分搜索树的最大值所在的节点
    private Node maximum(Node node){
        if(node.right == null)
            return node;


        return maximum(node.right);
    }


    // 从二分搜索树中删除最小值所在节点, 返回最小值
    public E removeMin(){
        E ret = minimum();
        root = removeMin(root);
        return ret;
    }


    // 删除掉以node为根的二分搜索树中的最小节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMin(Node node){


        if(node.left == null){
            Node rightNode = node.right;
            node.right = null;
            size --;
            return rightNode;
        }


        node.left = removeMin(node.left);
        return node;
    }


    // 从二分搜索树中删除最大值所在节点
    public E removeMax(){
        E ret = maximum();
        root = removeMax(root);
        return ret;
    }


    // 删除掉以node为根的二分搜索树中的最大节点
    // 返回删除节点后新的二分搜索树的根
    private Node removeMax(Node node){


        if(node.right == null){
            Node leftNode = node.left;
            node.left = null;
            size --;
            return leftNode;
        }


        node.right = removeMax(node.right);
```

```
        return node;
    }


    // 从二分搜索树中删除元素为e的节点
    public void remove(E e){
        root = remove(root, e);
    }


    // 删除掉以node为根的二分搜索树中值为e的节点, 递归算法
    // 返回删除节点后新的二分搜索树的根
    Node remove(Node node, E e){


        if( node == null )
            return null;


        if( e.compareTo(node.e) < 0 ){
            node.left = remove(node.left , e);
            return node;
        }
        else if(e.compareTo(node.e) > 0 ){
            node.right = remove(node.right, e);
            return node;
        }
        else{   // e.compareTo(node.e) == 0


            // 待删除节点左子树为空的情况
            if(node.left == null){
                Node rightNode = node.right;
                node.right = null;
                size --;
                return rightNode;
            }


            // 待删除节点右子树为空的情况
            if(node.right == null){
                Node leftNode = node.left;
                node.left = null;
                size --;
                return leftNode;
            }


            // 待删除节点左右子树均不为空的情况


            // 找到比待删除节点大的最小节点, 即待删除节点右子树的最小节点
            // 用这个节点顶替待删除节点的位置
            Node successor = new Node(minimum(node.right).e);
            size ++;


            successor.right = removeMin(node.right);
            successor.left = node.left;


            node.left = node.right = null;
            size --;


            return successor;
        }
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();
        generateBSTString(root, 0, res);
        return res.toString();
```

```
    }

    // 生成以node为根节点，深度为depth的描述二叉树的字符串
    private void generateBSTString(Node node, int depth, StringBuilder res){

        if(node == null){
            res.append(generateDepthString(depth) + "null\n");
            return;
        }

        res.append(generateDepthString(depth) + node.e +"\n");
        generateBSTString(node.left, depth + 1, res);
        generateBSTString(node.right, depth + 1, res);
    }

    private String generateDepthString(int depth){
        StringBuilder res = new StringBuilder();
        for(int i = 0 ; i < depth ; i ++)
            res.append("--");
        return res.toString();
    }
}
```

广度优先遍历的意义



- 更快的找到问题的解
- 常用于算法设计中 - 最短路径

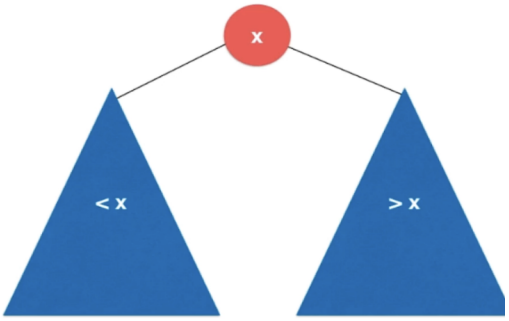
前序遍历

```
function traverse(node):
    if(node == null)
        return;

    访问该节点
    traverse(node.left)
    traverse(node.right)
```

- 最自然的遍历方式
- 最常用的遍历方式

中序遍历

```
function traverse(node):
    if(node == null)
        return;

    traverse(node.left)
    访问该节点
    traverse(node.right)
```



- 二分搜索树的中序遍历结果是顺序的

后序遍历

```
function traverse(node):
    if(node == null)
        return;

    traverse(node.left)
    traverse(node.right)
    访问该节点
```

- 后序遍历的一个应用：
- 为二分搜索树释放内存

2.二分搜索树的时间复杂度

二分搜索树 Binary Search Tree

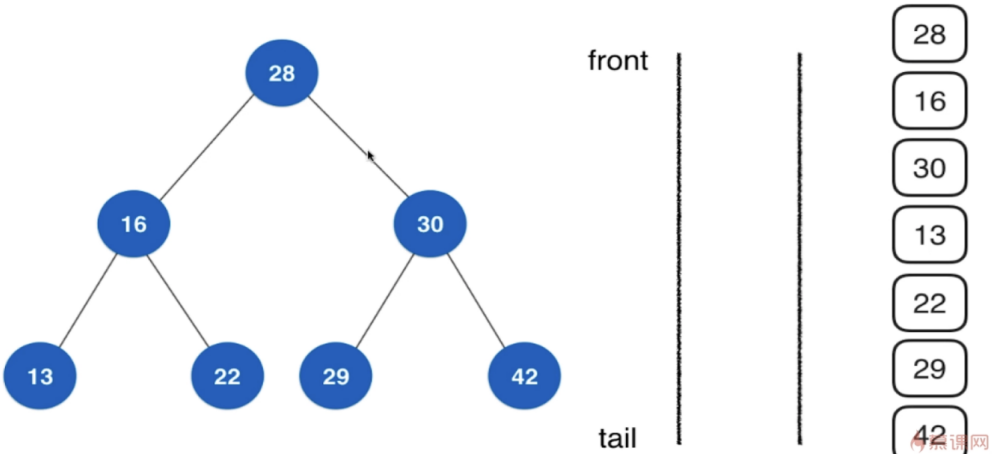
h层，一共多少个节点?

$$2^0+2^1+2^2+2^3+2^4+\ldots+2^{h-1}$$

$$=\frac{1\times(1-2^h)}{1-2}=2^h-1\ =n \qquad h=\log_2(n+1)$$

$$=O(\log_2 n)\ =O(\log n)$$

二分搜索树的层序遍历



logn和n的差距

	logn	n	
n = 16	4	16	相差4倍
n = 1024	10	1024	相差100倍
n = 100万	20	100万	相差5万倍