树 Tree

1 基本信息

1.1 引入

为了维护一个有序的数据，包括

- 查找一个元素
- 插入一个元素
- 删除一个元素

用我们已经学过的现有数据结构有

数组

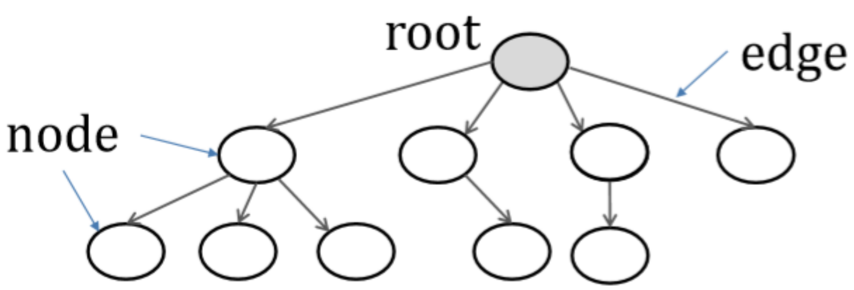
- 查找O(logn)
- 插入O(n)
- 删除O(n)

链表

- 查找O(n)
- 插入O(n)
- 删除O(n)

在接下来的学习中，我们将讨论数据结构(树、哈希表)，这些数据结构可用于更高效的数据结构存储

1.2 树的介绍



树的定义

树包括了

- 一组node
- 一组edge，每一个edge连接了一对node

每个node可能有**一个或多个数据**

- Key: 搜索元素时使用的查找项目
- 具有相同Key的多个数据项称为重复项

在树的顶端的node被称为树的root

node的关系

考虑一棵树T，设u和v是T中的两个node，如果以下条件之一成立，我们说u是v的ancestor

- u = v
- u 是 v 的parent
- u 是 v 的ancestor的parent

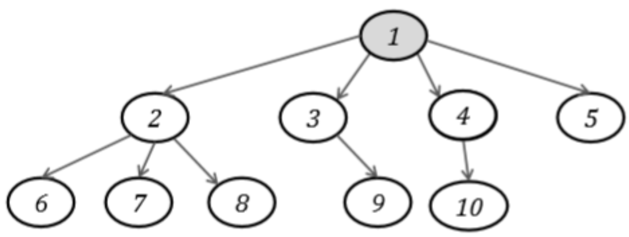
也就是说v的ancestor有包括v本身，v的parent，v的parent上面一直查找满足是上面的parent

考虑一棵树T，设u和v是T中的两个node，如果以下条件之一成立，我们说u是v的descendant

- u = v
- u 是 v 的child
- u 是 v 的descendant的child

如果u != v, u是v的proper ancestor，同样，v也是u的proper descendant

node的类型



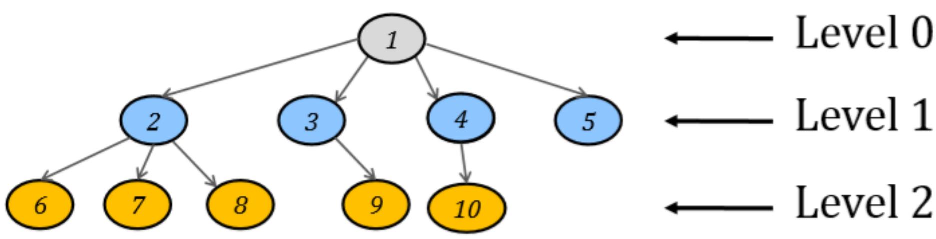
- leaf node：没有child的node

  如：5，6，7，8，9，10

- internal node：具有一个或多个child node的node

  如：1，2，3，4

Path，Depth，Level 和 Height



Path

- node的属性
- 只有一条path(一组edge序列)将每个node连接到root

Depth

- node的属性
- 一个node的depth = 从root到这个node所经过的edge数量
- root的depth = 0

Level

- 树的属性
- 拥有相同depth的node构成了树的level

- root所在的树的level = 0

Height

- 树的属性
- 树的height是树的node中最大的depth，也是最大的level

1.3 树的性质

性质1

一个有n个nodes的树有n-1条edge

证明

$对于每个非root的node，它有且只有一条edge指向自己$
$root没有edge指向自己$
$因为一棵树一共有n-1个非root的node，所以树有n-1条edge$

性质2

设树的每个internal node都至少有2个child node

如果m为leaf node的数量，则internal node的数量最多为m-1

证明

$假设internal\ \ node\ \ v有x_v个child\ \ node，假设这棵树有m个leaf\ \ node$
$平均来说，internal\ \ node的child的数量是x$
$Level\ \ H: 最多有\frac{m}{x}个parent\ \ node与这些leaf\ \ node直接相连$
$Level\ \ H-1: 对于\frac{m}{x}个intrnal\ \ node，最多有\frac{m}{x^2}个parent\ \ node与这些leaf\ \ node直接相连$
$...$
$所以internal\ \ node的总和为sum=\frac{m}{x}+\frac{m}{x^2}+...+1$
$当x=2时，sum=m-1为最大值$

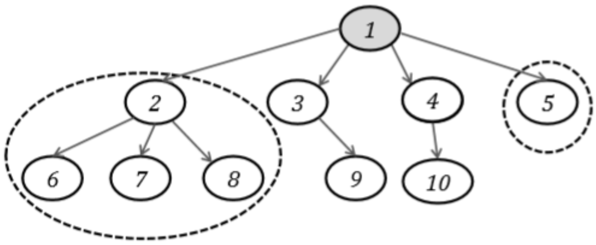
性质3

一个有n个node(n ≥ 2)的完全二叉树的Height为$log_2(n+1)$

证明

$假设树的高度为h$
$Level\ 0：2^0$
$Level\ 1：2^1$
$Level\ 2：2^2$
$Level\ 3：2^3$
$...$
$Level\ h-1：2^{h-1}$
$故n=2^0+2^1+...+2^{h-1}$
$解得h=log_2(n+1)$

1.4 树的递归性



树的每一个node都是一个更小的树的root

- 将这些树称为子树，以区别于整个树
  如 node 2 是一个子树的root，node 5 是一个只有一个node子树

1.5 k叉树和二叉树

k叉树

k叉树是一个有根树，其中每个internal node最多有k个child node

其中，k=2的时候称为二叉树

二叉树

在二叉树中，internal node最多有两个child node

二叉树的定义

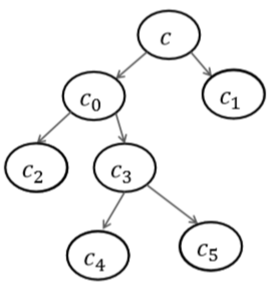
基本定义

二叉树可以是

- 空的
- 一个node（二叉树的root），包括
  一个左子树，该左子树也是一个二叉树
  一个右子树，该右子树也是一个二叉树

Full Level的定义

一个二叉树中一个level L，如果它有2^l个node，那么它是full level



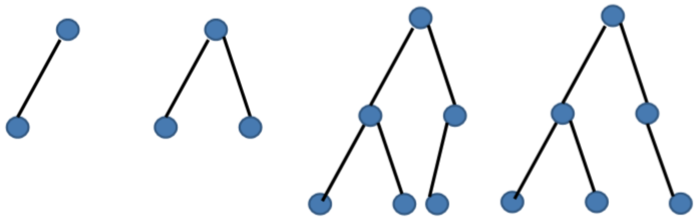
level 0和level 1是full level，level 2 和level 3不是

Complete Binary Tree的定义

满足下列条件的，一个Height为h的二叉树是完全二叉树

Level 0,1,...,h-1是full level

在Level h，所有的leaf node都尽可能的靠左



1，2，3是完全二叉树，4不是完全二叉树

2 树的遍历

2.1 介绍

树的遍历涉及到访问树里所有的node

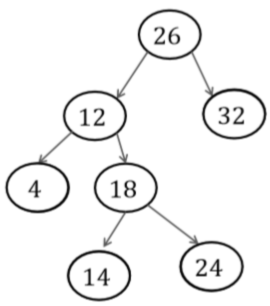
要理解遍历，记住二叉树的**递归定义**是有帮助的，其中每个node都是子树的root

2.2 前序遍历 Preorder Traversal

思路

根左右

- 访问root N
- 递归访问N的左子树
- 递归访问N的右子树



26，12，4，18，14，24，32

伪代码——递归形

```
Algorithm:Preorder(Node root)

print(root)

if(root.left != null)
    Preorder(root.left)

if(root.right != null)
    Preorder(root.right)
```

伪代码——迭代形

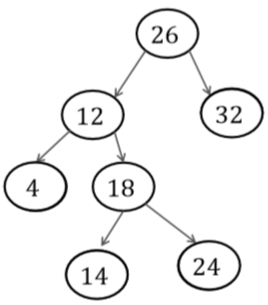
```
Algorithm:Preorder(Node root)

Creat stack s
s.push(root)
while(s!=empty)
    node = s.top()
    print(node)
    s.pop()
    if(node.right!=null)
        s.push(node.right)
    if(node.left!=null)
        s.push(node.left)
```

2.3 中序遍历 Inorder Traversal

思路

左根右

- 递归访问N的左子树
- 访问root N
- 递归访问N的右子树



4，12，14，18，24，26，32

伪代码——递归形

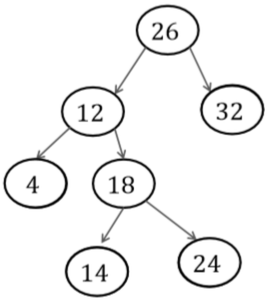
```
Algorithm:Preorder(Node root)

if(root.left != null)
    Preorder(root.left)

print(root)

if(root.right != null)
    Preorder(root.right)
```

2.4 后序遍历 Postorder Traversal

思路

左右根

- 递归访问N的左子树
- 递归访问N的右子树
- 访问root N



4，14，24，18，12，32，26

伪代码——递归形

```
Algorithm:Preorder(Node root)

if(root.left != null)
    Preorder(root.left)


if(root.right != null)
    Preorder(root.right)

print(root)
```

2.5 层序遍历 Level Traversal

思路

从顶到底，从左到右

用一个队列存储

从root开始，当出队一个node时，入队它的所有child node

伪代码——递归形

```
Algorithm:LevelTraversal(Node root)

Create queue q
q.enqueue(root)

while(q!=null)
    node = q.dequeue()
    enqueue(node.allsubNode)

```

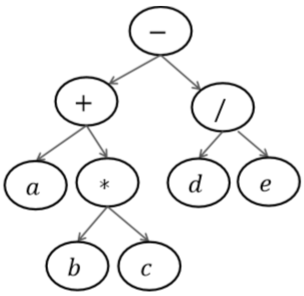
3 二叉树的应用

3.1 算术表达式的计算

思路

((a+(b*c))-(d/e))

目前我们只考虑有操作符+,-,*,/的情况



所有的leaf node都是变量或者常数

所有的internal node都是操作符

前序遍历可以计算该表达式

从root开始

sub((add(a,mul(b,c)), div(d,e)))

伪代码

```
Algorithm:Calculate(Node root)

if(root is operand)
    return operand_root(Calculate(root.left),Calculate(root.right))

else if(root is operator)
    return root
```

中序遍历可以把它转化成代数式

从root开始

- 访问左子树之前，打印【(】
- 访问左子树

- 打印root
- 访问右子树
- 访问右子树之后，打印【)】

3.2 可变长度字符编码——哈夫曼编码 Huffman Encoding

霍夫曼编码是最优的前缀编码，即空间代价最小

不可变长度编码

字符编码将每个字符映射为一个数字

计算机通常使用固定长度的字符编码

- ASCII对每个字符使用8位
- Unicode对每个字符使用16位

问题：固定长度编码浪费空间

思路

变长编码

- 使用不同长度的编码不同的字符
- 经常出现的字符分配更短的编码

要求：任何字符的编码不能是其他字符编码的前缀(例如，不能有00和001)

方法

通读文本以确定频率

创建一个node列表，其中包含文本中出现的每个字符的(字符、频率)对

删除并“合并”两个最低频率的node，形成一个新节点，即它们的parent node

将parent node添加到节点列表中

重复步骤3)和4)，直到列表中只有一个node，这个节点将是霍夫曼树的root

Huffman编码使用二叉树

例

例如，有一串字符，已知它们的对应值和出现频率

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

	①	F	C	U	D	L	E
	9	24	32	37	42	42	120

		C	②	U	D	L	E
		32	33	37	42	42	120

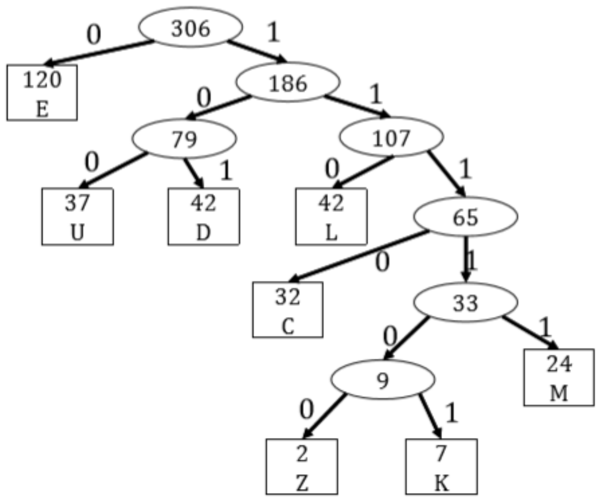
			U	D	L	③	E
			37	42	42	65	120

				L	③	④	E
				42	65	79	120

					④	⑤	E
					79	107	120

						E	⑥
						120	186

							⑦
							306



高级二叉树 Advanced Binary Tree

1 优先队列 Priority Queue

1.1 介绍

与普通队列(FIFO)不同，优先队列保证元素始终按升序离开(或按Delete-max降序离开)，而不管它们被插入的顺序是什么

我们将使用名为"二叉堆 binary heap"的数据结构实现一个优先队列，以实现以下保证

- 空间复杂度：O(n)
- 插入：O(log n)
- delete-min：O(log n)

二进制堆数据结构是一个数组对象，我们可以将其视为一个完整的二叉树

- Level 0 到 h-1是满的

- Level h的leaf node越左越好

1.2 二叉堆 BinaryHeap （小顶堆）

定义

设集合S有n个不重复整数，S上的二叉堆是一个二叉树T，满足

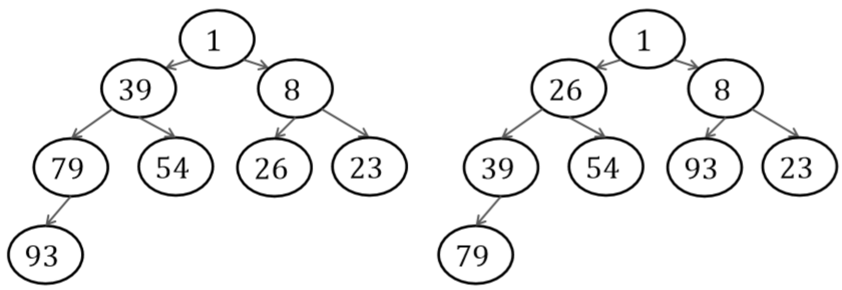
- T是完全二叉树
- T中的每个node u对应S中的一个不同的整数，这个整数称为u的key (存储在u)
- 如果u是internal node，则u的key小于其child node点的key

注意

- T有n个node
- u的key是以u的子树中最小的值（root一定是最小的）

例

S = {93, 39, 1, 26, 8, 23, 79, 54}



集合S的二叉堆**不是唯一**的

S的最小整数必须是root的key值

伪代码——二叉堆插入O(log n)

```
//小顶堆
//node:待插入的新node
//root:Binary Heap的root
Algorithm:BinaryHeapInsert(Node node, Node root):
Node rootNode ← root
Node leafNode ← node
//先把它插入在最靠右的位置
InsertRightMost(node,root)
//调换位置
while(true)
    Node u ← leafNode
    if(u is root) then
        return
    else if (u.key > u.parent.key)
        return
    else
        Node p ← u.parent
        swap(p,u)
```

伪代码——二叉堆删除O(log n)

```
Algorithm:Delete-min(Node root)

returnValue ← rote.key

//找到，删除最靠右的Node并返回
z ← Delete_The_Right_Most_Node(root)
root.key ← z.key
u ← root
while(true)
    if(u.isLeaf) then
        return returnValue
    else if(u.key < u.child)
        return returnValue
    else
        //找到u.child中最小key的那一个
        Node child = u.child
        swap(u,child)
        u ← child

```

伪代码——找到最右边的位置O(log n)

```
Algorithm:InsertRightMost(Node node,Node root)
We have BinaryHeap h
//把二叉堆现有Node数量+1转换成二进制形式
size ← (++h.size).convert_It_Into_Binary_Format
Node u ← root
//跳过最前面一位的1
for i ← 1 to size.lenth - 2
    if(size.charAt(i) == 0) then
        u ← u.leftChild
    else if(size.charAt(i) == 1) then
        u ← u.rightChild

if(size.charAt(siez.length - 1) == 0) then
        node ← u.leftchild
else if (size.charAt(siez.length - 1) == 1) then
        node ← u.rightchild

```

1.3 应用

人工智能(A *算法)

操作系统(负载平衡)

图搜索(最短路径算法)

2 动态数组 Dynamic Arrays

2.1 介绍

之前的讨论是基于指针的(用于将Parent Node与其Child Node连接起来)

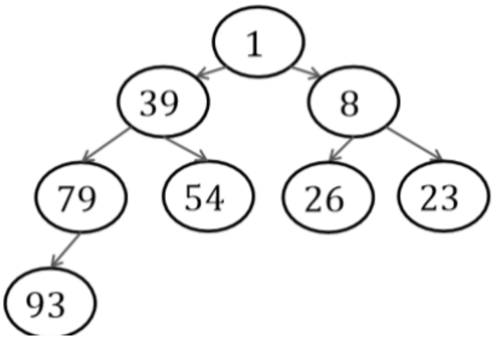
现在我们用一种**“无指针”**的方法来实现二进制堆，它在实践中实现了更低的空间消耗

2.2 例

设T为任意有n个节点的完全二叉树，我们将节点线性化如下

- 将较低Level的Node放在较高Level的Node之前
- 在同一层中，从左到右排列Node

我们将线性化的节点序列存储在一个长度为n的数组A中



1	39	8	79	54	26	23	93

2.3 动态数组的性质

注意这里第一个节点存在A[1]处

性质1

假设二叉堆T的一个Node u 存储在A[i]处。然后，u的左子结点存储在A[2i]处，右子结点存储在A[2i+1]处

性质2

假设二叉堆T的一个Node u 存储在A[i]处，那么u的Parent Node存储在A[i/2]（向下取整）

性质3

最底层最右边的leaf Node存储在A[n]

2.4 动态数组的效果

空间复杂度：O(n)

插入：O(log n)

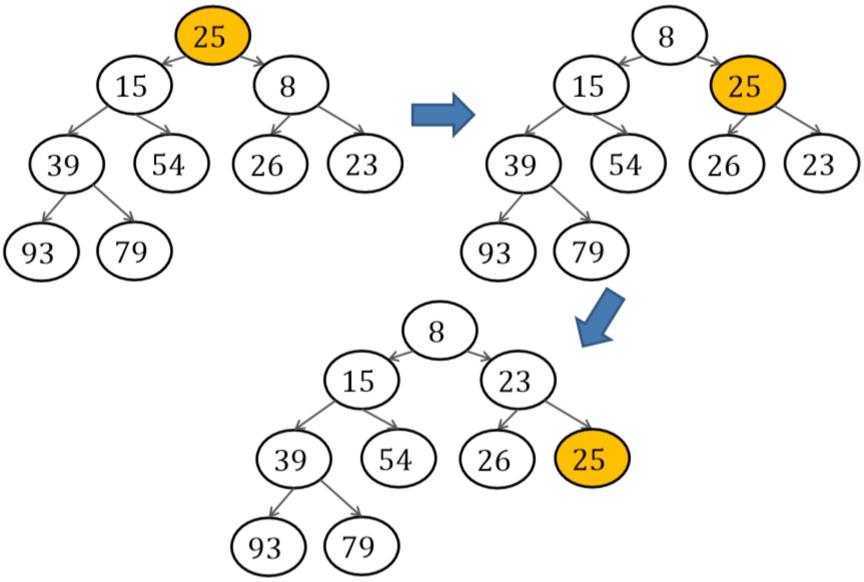
Delete-min：O(log n)

2.5 Root-fix 操作

我们有一个完全二叉树T有root r，它保证

- root的左子树是一个binary heap
- root的右子树是一个binary heap
- 然而root本身构成的二叉树不一定是binary heap，这也就意味着root的key可以大于左子节点或者右子节点

Root-fix可以在O(log n)时间内完成，方式与delete-min算法相同



2.6 用动态数组构建Binary Heap

思路

从最后一个元素开始，对每一个元素使用Root-fix操作，fix到最后是即满足条件

伪代码O(n)

```
//A是一个只是把Node放进去的数组，没有按顺序排好
//A的起始位置是0
Algorithm:CreateBinaryHeap(A[])
n ← A.length
for i ← n to 1
    RootFix(A[],i)
```

54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	93	8	1	23	39
54	26	15	39	8	1	23	93
54	26	1	39	8	15	23	93
54	8	1	39	26	15	23	93

1	8	15	39	26	54	23	93

时间复杂度分析O(n)

在数组A转换成二叉堆的时间复杂度为O(n)

证明

把A看作是一棵完全二叉树T，T的高度为h

为了不失去一般性，假设所有的$Level$都是$Full\ \ Level$

因此T中的$node$个数为$2^{h+1}-1$

那么构建一个$Binary\ \ heap$所需要的时间复杂度为

$T=\sum_{i=0}^{h}O(i*2^{h-i})=O(2^{h+1}-1)=O(n)$

3 二叉查找树 Binary Search Tree(BST)

3.1 介绍

二叉搜索树(特别是平衡二叉搜索树)是本课程中最强大的数据结构

这无疑是计算机科学中最重要的数据结构之一

在极端情况下，BST相当于一个链表，因此，我们通过研究AVL-tree来保证BST的运算性能

BST保证了

- 空间复杂度：O(n)
- 前续查找：O(h)
- 插入：O(h)
- 删除：O(h)

其中n是BST含有的node数量，h是BST的高度

3.2 二叉搜索树的定义

定义

一个有n个整数的集合，如果它是一个BST，它要满足以下条件

- T有n个node
- T中的每一个node u都是S里面一个唯一不重复的值，也叫做u的key
- 对于每一个internal node u，保证了

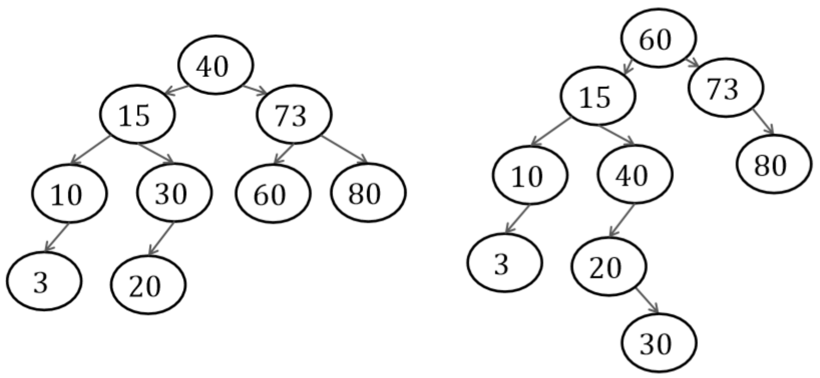
  u的key比所有它的左子树的node上的key都要大

  u的key比所有它的右子树的node上的key都要小

> 对于二叉搜索树，中序遍历得到的是一个递增的有序序列

例

S = {3，10，15，20，30，40，60，73，90}



3.3 前续查找O(h)

定义

设S是一整数，我们希望将S存储在一个数据结构中，以支持以下操作

- 前向查找：给定一个整数q，查找它在S中的前向，它是S中最大的不超过q的一个整数
- 插入：向S添加一个新的整数
- 删除：从S中删除一个整数

例

假设S ={3、10、15、20、30、40、60、73、80}

- 23的前向是20
- 15的前向是15
- 2的前向不存在

伪代码——前续查找

```
//T：二叉搜索树
//q：前序查找中的整数q
Alogrithm:PredecessorQuery(BST T, int q)
p ← -∞//最终p的值为返回值
u ← T.root
```

```
while(true)
    if(u == null)
        return p
    else if(u == q)
        p ← q
        return p
    else if(u > q)
        u ← u.leftchild
    else if(u < q)
        p ← u.key
        u ← u.rightchild

```

java代码——前续查找（递归）

```java
    public long predecessorQuery(Node root, long q) {
        //节点为空
        if (root == null) {
            return -1;
            //二叉搜索树中存在key=q的节点
        } else if (root.element == q) {
            return root.element;
            //当前节点的key>q的时候，将左子节点(更小)作为查询root节点
        } else if (root.element > q) {
            return predecessorQuery(root.leftChild, q);
            //当前节点的key<q的时候，保存当前的节点的key，将右子节点(更大)作为查询root节点，并且返回的key与当前节点的key做对比，如果是为-1证明无法找到
        } else {
            long temp = predecessorQuery(root.rightChild, q);
            if (temp == -1) {
                return root.element;
            } else {
                return temp;
            }
        }
    }
```

时间复杂度分析O(h)

显然，我们在访问的每个节点上花费O(1)时间

由于BST的高度为h，所以总查询时间为O(h)

3.3 后续查找O(h)

前续查找的对立是后续查找

整数q在S中的后续数是S中不小于q的最小整数

假设S ={3、10、15、20、30、40、60、73、80}

- 23的后续数是30
- 15的后续数是15
- 81的后续数不存在

给定一个整数q，后续查询返回S中q的后续数

根据对称性，我们从前面(关于前任查询)的讨论中知道，可以在O(h)时间内使用BST来查询后续数

java代码——后续查找（递归）

```java
    public long successorQuery(Node root, long q) {

        if (root == null) {
            return -1;
        } else if (root.element == q) {
            return root.element;
        } else if (root.element > q) {
            long temp = successorQuery(root.leftChild, q);
            if (temp == -1) {
                return root.element;
            } else {
                return temp;
            }
        } else {
            return successorQuery(root.rightChild, q);
        }

    }
```

3.4 BST的插入操作O(h)

假设我们需要插入一个新的整数e

首先创建一个新的leaf node z来存储key e

这可以通过从根到叶的路径降序来完成

伪代码——BST 插入（递归）

```
Algorithm:BST(key e, Node root)

node z ← new leafNode(e)
u ← root

if(u == null)
    u ← z

if (e < u.key)
    if(u.leftChild != null)
        BSTInsertion(key e, u.leftChild)
    else
        u.leftchild ← z
        return
else if(e ≥ u.key)
    if(u.rightChild != null)
        BSTInsertion(key e, u.rightChild)
    else
        u.rightchild ← z
        return

```

3.5 BST的删除操作O(h)

伪代码——BST 删除（递归）

```
Algorithm:BSTDelete(key e, Node root)

u ← root

if(u == null)
    return

if (e < u.key)
    if(u.leftChild != null)
        BSTDelete(key e, u.leftChild)
    else
        return
else if(e > u.key)
    if(u.rightChild != null)
        BSTInsertion(key e, u.rightChild)
    else
        return
else if(e = u.key)
    if(u.isLeafNode)
        u.father.remove(u)
    else if(u.rightChild != null)
        Node v = successorQuery(root, e)
        u.key = v.key
        if(v.isLeafNode)
            v.father.remove(v)
        else
            v = v.rightChild
    else if(u.rightChild == null)
        u = u.lefeChild



```

4 平衡二叉搜索树 Balancd Binary Search Tree

4.1 BST的高度

给定一个集合S有n个整数，它的BST的最大可能高度是多少

- h = n 最坏情况
- 且此时查找、插入、删除的开销都是O(n)

如何让每个操作的时间达到O(log n)

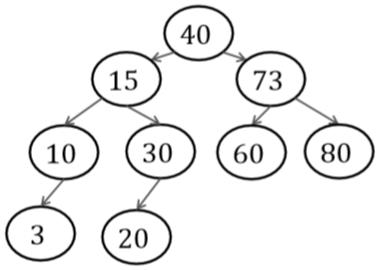
平衡二叉搜索树

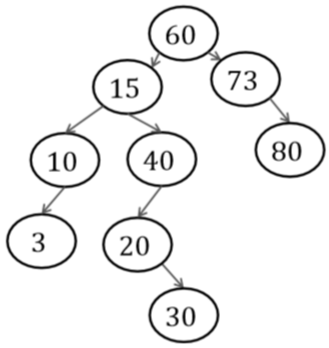
4.2 平衡二叉树的定义

如果在每个internal node u 在二叉树T上保持以下条件，则二叉树T是平衡的

- u左子树的高度与u右子树的高度最多相差1

例





前一个是平衡的，后一个是不平衡的

4.2 平衡二叉树的高度

有n个Node的平衡二叉树高度h = O(log n)

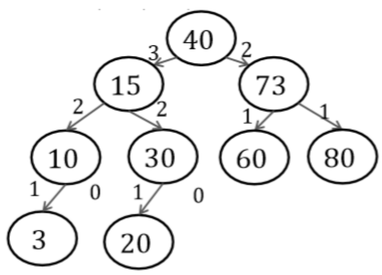
当平衡二叉树的高度为O(log n)时，可以得出一个平衡二叉搜索树的查询操作代价为O(log n)

4.3 平衡二叉搜索树的定义

一个包含n个整数的集合S上的AVL树是一个平衡二叉搜索树T，其中以下内容在每个internal node u上都成立

- u存储其左右子树的Height

例



通过存储子树高度，internal node可以知道它是否变得不平衡