L2. 二叉搜索树

搜索： Search (rt, x)

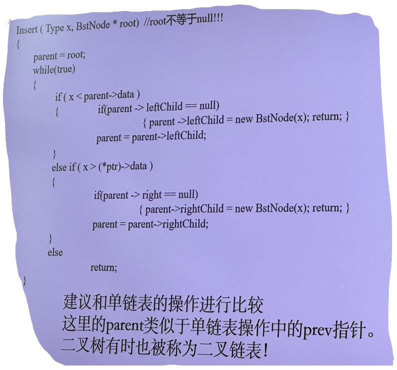
```
{ if (rt == null)
    return null;
  if (rt→data == x)
    return rt;
  else if (x < rt→data)
    rt→left = Search (rt→left, x)   // 左子树
  else
    rt→right = Search (rt→right, x)
```

插入：递归 (先查找)

```
Node*  Insert (rt, x)   // 返回插入后的树
{ if (rt = null) return new Node(x);
  if (rt→key == x) return rt;
  else if (x < rt→key)
     rt→left = Insert (rt→left, x)   // 返回后前的根节点不一定是 rt→left.
  else
     rt→right = ...
  return rt. }
```

迭代

```
void Insert ( Node ** result, rt, x )   // 将 x 插入 BST rt 且将新的 BST 的根传到 result 指向位置
{ if ( rt = null) { *result = new Node ; return ; }
  else if (rt→key == x) { *result = rt; return ;}
  else if (x < rt→key)
     Insert ( & rt→left, rt→right, x)   // 参数1是参数2的地址.
  else
     Insert (& rt→right, rt→right, x) }
```



```
Insert ( Type x, BstNode * root)  //root不等于null!!!
{
    parent = root;
    while(true)
    {
        if ( x < parent->data )
        {       if(parent -> leftChild == null)
                        { parent ->leftChild = new BstNode(x); return; }
                parent = parent->leftChild;
        }
        else if ( x > (*ptr)->data )
        {
                if(parent -> right == null)
                        { parent->rightChild = new BstNode(x); return; }
                parent = parent->rightChild;
        }
        else
                return;
}
```

建议和单链表的操作进行比较
这里的parent类似于单链表操作中的prev指针。
二叉树有时也被称为二叉链表!

删除：(先查找)

① 叶子节点 / 单子树的节点　　直接删除.

parent → right = ptr → left　// 对前一个节点进行操作.

(释放?)

53

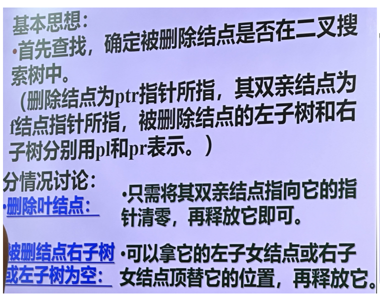
② 二叉子树的节点　　// 是左/右子树的分界线.

用key拷贝 / 直接移动

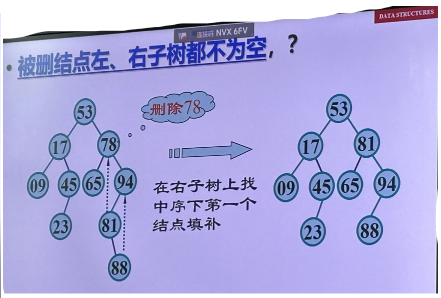
a. 查找右子树的最小key (中序)
b. 查找左子树的最大key

} ⇒ 移到删除节点, 保证分界线成立, 同时满足情况①.

(是叶子节点/单子树节点)



基本思想:
·首先查找，确定被删除结点是否在二叉搜索树中。
（删除结点为ptr指针所指，其双亲结点为pr结点指针所指，被删除结点的左子树和右子树分别用pl和pr表示。）
分情况讨论:
·删除叶结点: ·只需将其双亲结点指向它的指针清零，再释放它即可。
被删除结点右子树或左子树为空: ·可以拿它的左子女结点或右子女结点顶替它的位置，再释放它。



递归:

```
Remove (const Type&x , BstNode* & ptr) {
    BstNode * temp;
    if ( ptr == null )  return;
    if ( x < ptr -> data )   //左子树删除         → leftChild 改为新的左子树.
        Remove ( x, ptr->leftChild );
    else if ( x > ptr->data ) //右 ...
        Remove ( x, ptr -> rightChild );
    else if (ptr->leftChild != null && ptr->right != null ) {
        temp = Min ( ptr -> rightChild );
        //找 ptr 右子树中序第一结点
        ptr -> data = temp -> data.   //填补
        Remove (ptr->data, ptr ->rightChild);   // ptr右子树删除temp. }
    else { //ptr结点只有一个/无子女
        temp=ptr;
        if (ptr-> leftChild == null)
            ptr = ptr -> rightChild ;
        else if (ptr-> rightChild == null)
            ptr = ptr -> leftChild;
        delete temp; }
}
```

迭代

"替罪羊"树

得失: 
搜索最快 O(log n)
但插入、删除麻烦

L3. AVL树

AVL树: ① 空树

② 是二叉搜索树，且左、右子树均为AVL树，两高度差不超过1。

AVL树的高度.　高度为h　最多节点 $2^{h+1}-1$

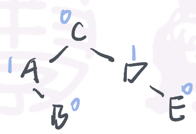
最小节点　递归　MinNode(h)

$\geq 1+$ MinNode(h-1) + MinNode(h-2)

Febonachii 数列

节点的平衡因子(balance factor): 每个结点附加一个数字，为右子树高度 − 左子树高度.

$\forall x \in AVL,\ bf(x) \in \{-1, 0, 1\}$



AVL树的插入.　在 $bf(x)=\pm 1$ 时插入后可能不再平衡.

平衡化旋转 { 单旋转: 左/右旋；双旋转: 左/右平衡

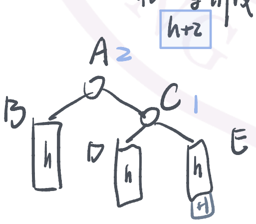
插入新结点后要从插入位置沿根回溯，检查各节点的平衡因子.

回溯到某节点高度不平衡，停止，取下面三层结点.

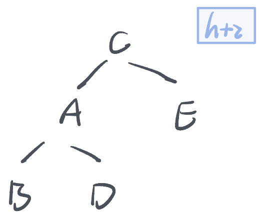
若三个节点在一条直线，则单旋转平衡化.

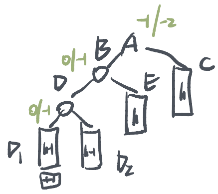
在一条折线，则双旋转平衡化.

① 选新的根
② 重置
③ 计算 balance factor

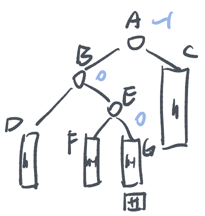


Ⓑ A Ⓓ C Ⓔ

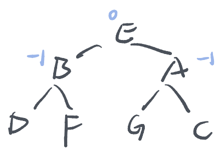




Ⓓ₁ D Ⓓ₂ B Ⓔ A Ⓒ



Ⓓ B Ⓕ E Ⓖ A Ⓒ



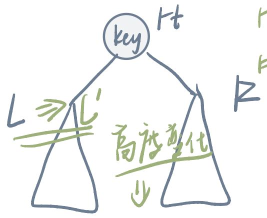
//AVL树的插入算法

① 新树 ② 是否变高

```
Insert (AVLNode* & tree, int x, int & taller) {      引用传递
int  success;
if(tree == NULL){
  tree = new  AVLNode(x);
  success = (tree != NULL) ? 1 : 0;
  if(success) taller = 1;
}
else if (x < tree→data) {
  success = Insert (tree→left , x, taller);
  if(taller){
    switch (tree→balance){
      case -1:  LeftBalance (tree, taller) ; break;   //调整旋转
      case  0:  tree→balance = -1; taller=1 break;    //原来为0，变为-1
      case  1:  tree →balance = 0 ; taller = 0;       //原来为1，变为0
      }
  }
  else if (x > tree→ data){
    success = Insert (tree→ right , x, taller);
    if(taller) {
     switch (tree→ balance) {
       case -1: tree→balance = 0; taller = 0; break;
       case  0: tree→ balance = 1; break;
       case  1: RightBalance (tree, taller);
       }
   }
   return success;
}
```

-1 一定左边插入 故 LeftBalance

1 一定右边插入

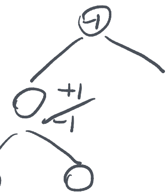


⇓
H→BF的变化
⇓
|H→BF| > 1 时，进行调整
返回新BF，及高度变化信息

H→BF ±1时，一定不变高
H→BF = 0 时，会变。

```
Left Balance (tree, taller) {
  switch (tree→left →BF) {
    case 1:                 //折线，双旋

    case -1:                //直线，单旋
    default :  assert(0);
```



递归比较

① 检查是否BST
中序遍历

② 检查是否平衡 及 BF是否正确
求BST高度

AVL树的实现？

AVL 树的删除、 1° 被删结点 x 至多有一个子女.

·删去该点 x.

x 双亲指向 x 的指针 改为指向 x 的子女 / NULL.

以 x 为根的子树高度减一.

2° 有两个子女.

搜寻 左最大 或 右最小. y.

y 内容 → x   删去 y.(至多一个子女)

高度减少:

x → 根. 回溯对各个节点影响

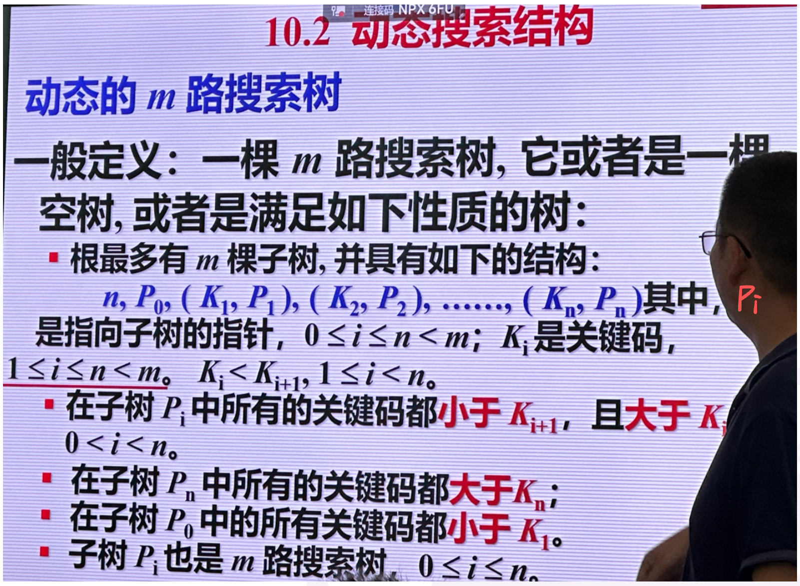
bool shorter

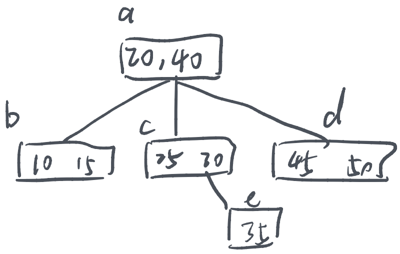
case 1:

case 2:

case 3:

L4: B⁻树





Node* p = P_n
for (i=1, 2, ...)
if (x = key_i)
else if (x < key_i) {
p = P_{i-1} break }
递归 Search (P, X)

Search.
① 确定子树
② 在子树递归.(读取子树的结点)

B-树.

m路搜索树,且为空树或有:
① 根节点两个子节点
② 除根节点,内部节点至少有⌈m/2⌉个子节点
③ 所有外部节点(失败结点)都位于同一层. (叶子节点高于一级).

$\lceil \frac{m}{2} \rceil \sim m$

B树的插入: 满足则分裂.

NANJING UNIVERSITY

(RBTree)

L5：红-黑树：催化 B-树

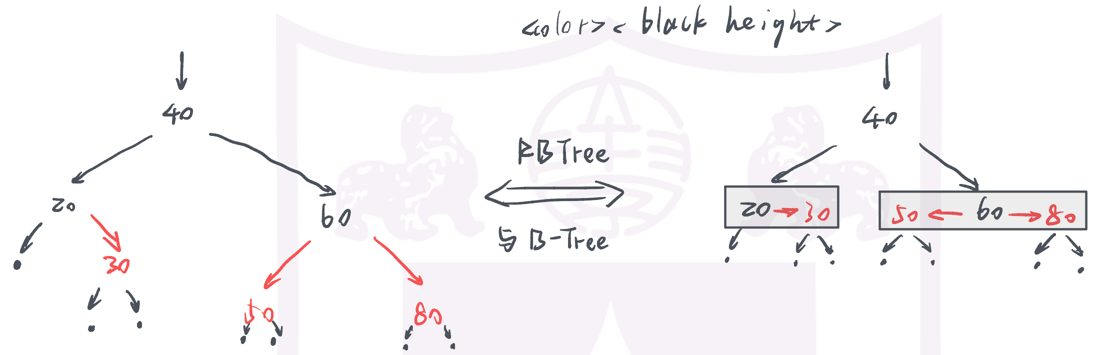
二叉搜索树：　　根节点、外部节点为黑

<color>　红节点不能有红子节点

<black height> 所有根至外部节点的路径上，黑节点数目相同.

ARBTree. Almost-RBTree. 根节点为红.

<color><black height>

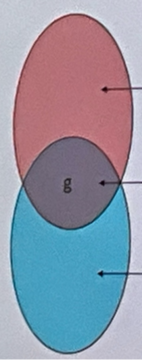


L6：时间复杂性分析

算法效率的比较方式

- 算法的效率通过它执行的关键操作的数量来度量，但是
  - 对于不同的具体输入，算法所需要的关键操作数量有所不同
  - 同样的操作在不同的计算机上需要的时间是不同的
- 算法复杂性表示为输入规模n的函数f(n)：平均情况，最坏情况
- 比较复杂性时主要看f(n)的渐进增长率（Asymptotic Growth Rate）
  - 如果算法A和算法B所需要的关键操作数量分别是f1(n)和f2(n)，A优于B是指当n足够大时，f1(n)必然小于f2(n)。

Relative Growth Rate



Ω(g):functions that grow at least as fast as g 至少与g一样快

Θ(g):functions that grow at the same rate as g 与g一样快

O(g):functions that grow no faster as g 不会快于g

"Big Oh"

- Basic idea
  - $f(n) \in O(g(n))$ if for sufficiently large input n, $g(n) \ge f(n)$

$\exists c, n_0$, s.t. $n > n_0$, we have $f(n) \le c g(n)$

- Definition – "$\varepsilon - N$"
  - Giving $g: N \to R^+$, then $O(g)$ is the set of $f: N \to R^+$, such that for some $c \in R^+$ and some $n_0 \in N$, $f(n) \le cg(n)$ for all $n \ge n_0$

Example

- Let $f(n)=n^2$, $g(n)=n\log n$, then:

L'Hospital's rule

  - $f \notin O(g)$, since

$$\lim_{n\to\infty}\frac{n^2}{n\log n}=\lim_{n\to\infty}\frac{n}{\log n}=\lim_{n\to\infty}\frac{1}{\frac{1}{n\ln 2}}=+\infty$$

  - $g \in O(f)$, since

$$\lim_{n\to\infty}\frac{n\log n}{n^2}=\lim_{n\to\infty}\frac{\log n}{n}=\lim_{n\to\infty}\frac{1}{n\ln 2}=0$$

南京大學
NANJING UNIVERSITY

Asymptotic Order

- Logarithm $log\ n$

  $logn \in O(n^{\alpha})$ for any $\alpha>0$ 对数慢于幂

- Power n^k

  $n^k \in O(c^n)$ for any $c>1$ 幂慢于指数

- Factorial $n!$

"Big Ω"

① $g(n) \in O(g)$
$\in \Omega(g)$

- Basic idea
  - $f(n) \in \Omega(g(n))$ if for sufficiently large input n, $f(n) \geq g(n)$
  - Dual of "O"

② $f_1 \in O(f_2)$ $f_2 \in O(f_3)$
$f_1 \in O(f_3)$?
传递

- Definition – "$\boldsymbol{\varepsilon} - \boldsymbol{N}$"
  - Giving g: $N \to R^+$, then $\Omega(g)$ is the set of f:$N \to R^+$, such that for some $c \in R^+$ and some $n_0 \in N$, $f(n) \geq cg(n)$ for all $n \geq n_0$

自反
传递
不是偏序
但不仅对称

The Set Θ

等价关系

- Basic idea of $f(n) \in \Theta(g(n))$
  - Roughly the same
  - $\Theta(g) = O(g) \cap \Omega(g)$
- Definition – "$\boldsymbol{\varepsilon} - \boldsymbol{N}$"
  - Giving g:$N \to R^+$, then $\Theta(g)$ is the set of f:$N \to R^+$, such that for some $c_1, c_2 \in R^+$ and some $n_0 \in N$,

    $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, for all $n \geq n_0$

Properties of O, Ω and Θ

- Transitive property:
  - If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
- Symmetric properties ≤ ≥
  - $f \in O(g)$ if and only if $g \in \Omega(f)$
  - $f \in \Theta(g)$ if and only if $g \in \Theta(f)$

  $g \in \Theta(f)$

  $f,)$
- Order of sum function
  - $O(f+g) = O(max(f,g))$

Recursion in Algorithm Design

- Counting the Number of Bits
  - Input: a positive decimal integer n
  - Output: the number of binary digits in n's binary representation

Int BitCounting (int n)

```
If(n==1) return 1;
Else
  return BitCounting(n div 2)+1;
```

$$T(n)=\begin{cases}0 & n=1\\ T(\lfloor n/2\rfloor)+1 & n>1\end{cases}$$

南京大学
NANJING UNIVERSITY

1、二叉搜索树：定义（递归） 最坏情况全为 $\Theta(n)$

$O(h)$ 或 $O(\log n)$ {
- 查找
- 插入
- 删除：3种情况 {
  - 叶子节点，直接删
  - 只有一棵子树 {
    - 无父结点：子节点→根节点
    - 有父结点：父结点指向子节点
  - 两棵子树 左子树最大/右子树最小→代替原节点

结论：排序算法时间复杂性 $O(n\log n)$.

⟺ 初始化并创建一棵二叉搜索树，其时间复杂性为 $O(n\log n)$

⇒ 合并两二叉搜索树：$O(n+m)$ n、m为元素个数

2、平衡搜索树：AVL树、红黑树、B树

- AVL树：定义（递归）：$|h_R - h_L| \le 1$.

$O(\log n)$ {
- 查找
- 插入：最多旋转1次
- 删除：最多旋转 $O(\log n)$次.

平衡因子 $bf(x) = h_L - h_R$.

- 红黑树：定义（递归）：BST、根、叶子为黑，没有连续红，黑高度相同.

$O(\log n)$ {
- 查找
- 插入：最多旋转1次：新节点为红色、先BST插入，再换颜色，再旋转
- 删除：最多旋转1次：先BST删除，再换颜色，再旋转

- B-树：m叉搜索树：每个内部节点最多m个孩子，以及 $1\sim m-1$个元素.

  含p个元素的节点有 $p+1$个孩子.

  ↳ $k_1,\dots,k_p$，孩子 $c_0,\dots,c_p$，有（$k_i < c_i < k_{i+1}$，$1\le i<p$）

  B树定义（递归）：mST、根节点至少2个孩子，内部节点至少有 $\lceil m/2 \rceil$个孩子. 外部节点在同一层.

  高度：T是高度为h的m阶B-树. $d=\lceil m/2 \rceil$. n是T的元素个数，则

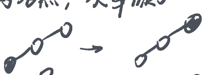
  $\underbrace{2d^{h-1}-1}_{\text{元素下限}} \le n \le \underbrace{m^h-1}_{\text{元素上限}}$; $\log_m(n+1) \le h \le \log_d\left(\frac{n+1}{2}\right)-1$.

$O(\log n)$ {
- 查找
- 插入：可能分裂
- 删除：可能合并

3、伸展树（分裂树） （二叉搜索树）每次将被访问的节点设为根节点、

- 旋转操作：将结点向上搬运一层. 左旋/右旋.（单旋）
- 伸展操作：将结点旋转为某结点.（一般是根结点）.（双旋）

PPT：简单旋转（两结点，一次单旋）

一字形旋转 

之字形旋转 

摊销分析：一次操作 $\le 1+3\log n$.

4、替罪羊树. BST中指定 $\alpha>1$. 只要高度不超 $\alpha\log n$ 就不平衡了.

每个节点记录子节点数量 size(v) 和子树高度 height(v).

一旦高度超过 $\alpha\log n$. 找插入路径上最不平衡节点重新平衡

检查 height(v) 是否大于 α size(v).

伸展树find的另一个递归算法

```
Node *find(root, X) //返回新的根结点
{
        if(         root == NULL                    //这种情况只可能在空树的时候发生
        ||          root->data == X          //X在根结点，直接返回
        ||          X < root->data && root->left == NULL  //没有找到X，把最接近X的root当作要旋转的结点
        ||          X > root->data && root->right == NULL)
                    return root;
        else if(X < root->data)      //左子树
        {           Node * left = root->left;
                    if(X < left->data)          //一字型旋转
                    {           left->left = find(left->left, X);          //递归
                                return splayRotate_slash_Left(root);       //一字型旋转
                    }
                    else if(X > left->data)
                    {
                                left->right = find(left->right, X);
                                return splayRotate_Zig_left(root);
                    }
                    else
                                return splayRotate_single_left(root);
        }
        else //X > root->data
        {
                    对称处理
        }
}
```

5、并查集：用树表示分离的集合。

每个集合是一棵树，除根节点外每个节点都有父指针。

查找（代表元素的根节点） $O(h)$

合并（两个集合） $\Theta(1)$

假设一系列操作 u 次合并和 f 次查找（$f>u$），则时间复杂度为 $O(fu)$。

利用重量规则或高度规则，可降为 $O(f+u)=O(f)$。

[重量规则] 若根为 i 的树节点数小于根为 j 的树节点数，则将 j 作为 i 的父节点。

[高度规则] 若根为 i 的树高度小于根为 j 的树高度，则将 j 作为 i 的父节点。

路径压缩改进find()性能

路径紧缩：待查节点到根节点路径上每一个节点都改为指向根节点。

路径分割：待查节点到根节点路径上每一个节点（除根节点及其子节点）都指向其祖父节点。

路径对折：待查节点到根节点路径上每一个节点（除根及其子节点）每隔一个都指向其祖父节点。

```
struct unionFindNode{
  int parent;  //父节点数量
  bool root; //记录是否为根

  unionFindNode()
    { parent = 1; root = True; }  //初始化
};
```

```
while (currNode != Root){
    int ParNode = node[currNode].parent;
    node[currNode].parent = Root;
    currNode = ParNode;
}
```

6、斐波那契堆

堆：一个大根堆（小根堆）是这样一棵树，其中每个节点都大于等于（小于等于）其子节点的值。

可合并堆：支持以下五个操作：Make-Heap()、Insert(H, x)、Minimum(H)、Extract-Min(H)、Union(H1, H2)

附加操作：DecreaseKey(H, x, k)、Delete(H, x)。

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

斐波那契堆：

- 有根树的森林
  - 一个节点可以有多个子结点
  - 所有树的子结点在一个双向链表上
  - H.min指向最小的根。
- 满足“最小堆序”
  - 一个结点的Key总是大于等于它的父节点的Key
- 具体表示：
  - 结点包含父节点指针x.p（常量时间内找到父节点）
  - 子节点使用循环双向链表保存，x.left,x.right分别是左右子节点。（常量时间内完成结点删除、链表合并）
  - x.child指向x的子节点的链表
  - x.degree:子节点的个数
  - x.mark：标记
  - H.n：堆的结点个数

7、左式堆：每个结点的key小于其子结点key；不平衡。

合并操作可在 $O(\log N)$ 中完成、插入、删除可转化为堆的合并。

- 零路径长(Null Path Length，NPL)
  - 对于结点X，Npl(X)定义为从X到一个只有零个或一个子结点的最短路径的长度。
- Npl(null)等于-1
- Npl(x) = MIN(Npl(x->left), Npl(x->right)) + 1
- 对于只有0个或1个子结点的结点，其NPL为0
- 左式堆的定义：对于堆中每个结点X，其左子结点的Npl大于等于其右子结点的Npl，那么它称为左式堆
- 对于左式堆中的每个结点X，Npl(X) = Npl(X->right) + 1

N个结点的左式堆的右路径最多含有 $\log(N+1)$ 个结点。

合并操作：

- //当H1和H2都不是空堆时

```
…
if(h1.data <= h2.data)
{
        Node * hr1 = Merge(h1->right, h2);
        if(NPL(h1->left) < NPL(hr1->npl))
        {/*对换*/        h1->right = h1->left; h1->right = hr1;        }
        else
                h1->right = hr1;

        h1->npl = NPL(h1->right) + 1;
        return h1;
}
```

NANJING UNIVERSITY

8、图遍历：DFS及其应用

dfs(G,v)
  Mark v as "discovered".
  For each vertex w that edge vw is in G:
    If w is undiscovered:
      dfs(G,w)
    Otherwise:
      "Check" vw without visiting w.
  Mark v as "finished".

A vertex must in one of the status:
- undiscovered，discovered，Finished
- 可达的顶点将顺序经历上面三个状态

That is: exploring vw, visiting w, exploring from there as much as possible, and backtrack from w to v.

找图中的连通分图.

基本思想：遍历每个顶点v

1. 对每一个尚未确定所在连通子图的顶点v，通过DFS寻找到和v连通的所有顶点，对于找到的每个顶点w，设置w的连通子图编号为v。
2. 如果顶点v在之前的DFA搜索中已经出现在某个连通子图中就不需要在遍历。

DFS框架的性质（回顾）

- 图的DFS过程中有两次处理顶点的机会：
  - 顶点状态由undiscovered（white）变成discovered（black）时进行处理（Preorder）
  - 顶点状态由discovered（gray）变成finished（black）时进行处理，（Postorder）
    - 此时顶点的后继顶点的状态要么是finished（black）状态，要么是discovered（gray）状态
    - 如果邻接顶点是discovered（gray）状态，那么此邻接顶点和当前顶点形成一个回路

使用DFS框架处理问题

- 如果一个问题能够转化为如下的形式，那么可以考虑在DFS框架下进行处理
  - 问题的目标是求解各个顶点v对应的值F(v)
  - 如果v的邻接顶点是$u_1,u_2,\ldots,u_k$，那么F(v)可以根据$F(u_1)$, $F(u_2),\ldots,F(u_k)$得到
- 处理的方法大致如下：
  - 按照PostOrder进行处理（也就是顶点由discovered变成finished时进行处理）
  - 函数增加参数来记录各个顶点对应的F值，同时可能需要一些新参数来记录相关信息
  - 在PostOrder进行处理，计算F(v)
- 如果PostOrder处理的时间复杂性是O(1)或者O(k)的(注意所有结点的k值加起来就是边的数量)，那么算法的复杂性就是O(m+n)的，其中m是顶点数，n是边数。