Chapter 1. 插入、归并排序

1.1. 插入排序(稳定，原址)

```
INSERTION-SORT(A)
for j=2 to A.length
    key=A[j]
    //Insert A[j] into the sorted sequence A[1…j-1]
    i=j-1
    while i>0 and A[i]>key
        A[i+1]=A[i]
        i=i-1
    A[i+1]=key
```

1. 2. 归并排序

```
MERGE(A,p,q,r)
n1=q-p+1
n2=r-q
let L[1…n1+1] and R[1…n2+1] be new arrays
for i=1 to n1
    L[i]=A[p+i-1]
for j=1 to n2
    R[j]=A[q+j]
L[n1+1]=∞
R[n2+1]=∞
i=1
j=1
for k=p to r
    if L[i]≤R[j]
        A[k]=L[i]
        i=i+1
    else A[k]=R[j]
        j=j+1
```

```
MERGE-SORT(A,p,r)(稳定)
if p<r //当只有一个元素(p=r)或者没有元素(p>r)时递归终止
    q=⌊(p+r)/2⌋
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

Chapter 2. 最大和子数组

```
FIND-MAX-CROSING-SUBARRAY(A,low,mid,high)//求包含 mid 的最大字数组
left-sum=-∞//mid 左侧最大值，包括 mid
sum=0
for i=mid downto low//这里从 mid 算起，因此 max-left 最大为 mid
    sum=sum+A[i]
    if sum>left-sum
        left-sum=sum
        max-left=i
right-sum=-∞//mid  右侧最大值
sum=0
for j=mid+1 to high //这里从 mid+1 算起，因此 max-right 最小为 mid+1
    sum=sum+A[j]
    if sum>right-sum
        right-sum=sum
        max-right=j
return (max-left,max-right,left-sum+right-sum)
```

```
FIND-MAXIMUM-SUBARRAY(A,low,high)
if high==low//递归终止
    return(low,high,A[low])
else mid=⌊(low + high)/2⌋
    (left-low,left-high,left-sum)=
        FIND-MAXIMUM-SUBARRAY(A,low,mid)//这里包含只含有单个 mid 的情况
    (right-low,right-high,right-sum)=
        FIND-MAXMUM-SUBARRAY(A,mid+1,high)//这里不包含只含有单个 mid 的情况
    (cross-low,cross-high,cross-sum)=
        FIND-MAX-CROSSING-SUBARRAY(A,low,mid,high) //这里不包含只含有单个 mid 的情况
    if left-sum≥right-sum and left-sum≥cross-sum
        return (left-low,left-high,left-sum)
    elseif right-sum≥left-sum and right-sum≥cross-sum
        return(right-low,right-high,right-sum)
    else return(cross-low,cross-high,cross-sum)
```

注意：cross 必然包含两个元素，至少为[mid,mid+1]

Chapter 3. 堆排序

3.1. 堆性质维护

维护最大堆的性质(单独对某一个节点调用该函数，并不能保证以该节点为根节点的子堆满足最大堆的性质，即不发生递归调用的时候(该节点的子节点比该节点小)，可能该节点子节点的子节点比该节点大)

```
MAX-HEAPIFY(A,i)
l=LEFT(i)
r=RIGHT(i)
if l≤A.heap-size and A[l]>A[i]
    largest=l
else largest=i
if r≤A.heap-size and A[r]>A[largest]
    largest=r
if largest≠i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A,largest)
```

3. 2. 构造最大堆

```
BUILD-MAX
A.heap-size=A.length//这句什么用?
for i=⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A,i)
```

3. 3. 堆排序（非原址，非稳定）

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i=A.length downto 2
    exchange A[1] with A[i]
    A.heap-size=A.heap-size-1
    MAX-HEAPIFY(A,1)
```

若堆索引从 **1** 开始算，那么 **L=2*i   R=2*i+1**

若堆索引从 **0** 开始算，那么 **L=2*i+1   R=2*i+2**

3. 4. 基于最小最大二叉堆的优先队列

```
HEAP-MAXIMUM(A)
return A[1]

HEAP-EXTRACT-MAX(A)
if A.heap-size<1
        error"heap underflow"
max=A[1]
A[1]=A[A.heap-size]//将最后一个数放置到第一个
A.heap-size=A.heap-size-1//减少堆的维度
MAX-HEAPIFY(A,1)//维护堆的性质
return max

HEAP-INCREASE-KEY(A,i,key)
if key<A[i]
        error"new key is smaller than current key"
A[i]=key
while i>1 and A[PARENT(i)]<A[i]
        exchange A[i] with A[PARENT(i)]
        i=PARENT(i)

MAX-HEAP-INSERT(A,key)
A.heap-size=A.heap-size+1
A[A.heap-size]=-∞
HEAP-INCREASE-KEY(A,A.heap-size,key)
```

Chapter 4. 快速排序

4.1. 快速排序（原址，非稳定）

```
QUICKSORT(A,p,r)
if p<r
    q=PARTITION(A,p,r)
    QUICKSORT(A,p,q-1)
    QUICKSORT(A,q+1,r)
```

```
PARTITION(A,p,r)//其实 p=r 的情况下也能运行
x=A[r]
i=p-1
for j=p to r-1//循环到 r-1 的原因：等于 x 的值已经放在最右侧了，对该值不需要循环
    if A[j]≤x
        i=i+1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```

PARTITION 的随机化版本

```
RANDOMIZED-PARTITION(A,p,r)
i=RANDOM(p,r)
exchang A[r] with A[i]//必须将该值置于最后，才能调用 PARTITION
return PARTITION(A,p,r)
```

4. 2. 优化版本 1

对于重复元素较多的情况下，采用这种方式效率较高

```
PARTITION_REPEAT1(A,p,r)
x=A[r]
i=p-1            //小于 x 的最大索引
boundary=r-1         //以最后一个元素为主元，非主元的最大索引
for j=p to boundary
    if A[j]<x
        i=i+1
        EXCHANGE A[i] with A[j]
    elseif A[j]==x
        EXCHANGE A[j] with A[boundary]     //将于 x 相同的值先放到最后
        j=j-1//将索引为 cnt 的数放到 j 位置，但这个数尚未进行判断，因此要将 j-1(抵消自增量)
        boundary = boundary -1//由于 boundary 位置上已经是与 x 相同的数，因此循环边界递减
n=r- boundary
for j=0 to n-1
    EXCHANGE A[i+1+j] with A[r-j]
return i+1 and i+n
```

i+1 是与 x 值相同的区间内的开始，i+rn 是与 x 值相同的区间的结束
[p,i]区间内的元素小于 x
[i+1,i+rn]区间内的元素等于 x，[i+rn+1,r]的元素大于 x

```
MODIFIED_PARTITION(A,p,r,M)
i=p-1
cnt=r          //非主元M的最大索引
for j=p to cnt     //与上一个版本有差异，因为最后一个元素并不是M，M的位置是未知的
    if A[j]<M
        i=i+1
        EXCHANGE A[i] with A[j]
    elseif A[j]==M   //关键：将于M相同的值暂时放到A的最后边
        EXCHANGE A[j] with A[cnt] //将于x相同的值先放到最后
        j=j-1//将第cnt个数放到j位置上，但这个数尚未进行判断，因此要将j-1(抵消自增量)
        cnt=cnt-1     //由于cnt位置的值已经与M相等，因此递减循环边界cnt
rn=r-cnt
for j=0 to rn-1     //将等于M的区间挪到中间
    EXCHANGE A[i+1+j] with A[r-j]
return i+1 and i+rn
```

i+1 是与 x 值相同的区间内的开始，i+rn 是与 x 值相同的区间的结束
[p,i]区间内的元素小于 x
[i+1,i+rn]区间内的元素等于 x
[i+rn+1,r]的元素大于 x

4. 3. 优化版本 2

对于重复元素较多的情况下，采用这种方式效率较高

```
PARTITION_REPEAT2(A,p,r)
x=A[r]
i1=p-1        //小于x的最大索引
i2=p-1        //等于x的最大索引(算最后一个)
for j=p to r-1
    if A[j]<x
        i1=i1+1
        EXCHANGE A[i1] with A[j]
        i2=i2+1
        if i1≠i2
            EXCHANGE A[i2] with A[j]
    elseif A[j]==x
        i2=i2+1
        EXCHANGE A[i2] with A[k]
i2=i2+1
EXCHANGE A[i2] with A[r]
return i1+1 and i2
```

Chapter 5. 线性时间排序

5.1. 计数排序(稳定，非原址)

```
COUNTING-SORT(A,B,k)
let C[0…k] be a new array
for i=0 to k
        C[i]=0
for j=1 to A.length
        C[A[j]]=C[A[j]]+1
//C[i] now contains the number of elements equal to i
for i=1 to k
        C[i]=C[i]+C[i-1]
//C[i] now contains the number of elements less than or equal to i
  //存的是值为 i 的元素的最大索引
for j=A.length down to 1
        B[C[A[j]]]=A[j]
        C[A[j]]=C[A[j]]-1
```

5.2. 基数排序

```
RADIX-SORT(A,d)
for i=1 to d
        use a stable sort to sort array A on digit i
```

5. 3. 桶排序

```
BUCKET-SORT(A)
n=A.length
let B[0…n-1] be a new array
for i=0 to n-1
        make B[i] an empty list
for i=1 to n
        insert A[i] into list B[⌊nA[i]⌋]
for i=0 to n-1
        sort list B[i] with insertion sort
concatenate the lists B[0],B[1],…,B[n-1] together in order
```

5.4. 遗忘比较交换算法

```
COMPARE-EXCHANGE(A,i,j)
if A[i]>A[j]
    exchange A[i] with A[j]

INSERTION-SORT(A)
for j=2 to A.length//循环不变式：A[1…j-1]是已排序的序列
    for i=j-1 down to 1
        COMPARE-EXCHANGE(A,i,i+1)
```

Chapter 6. 中位数和顺序统计量

```
RANDOMIZED_SELECT(A,p,r,i)//这里的 pr 是绝对下标，i 是相对大小
if p==r//只有一个元素时，退出
        return A[p]
q=RANDOMIZED_PARTITION(A,p,r);
k=q-p+1
if k==i//若q就是要找的下标
        return A[q]    //别写成了A[i]
elseif i<k
        return RANDOMIZED_SELECT(A,p,q-1,i)
else return RANDOMIZED_SELECT(A,q+1,r,i-k)
```

Chapter 7. 基本数据结构

7.1. 二叉树前序遍历非递归算法

外循环体：对于当前指针 cur：

首先：内循环体：对于当前指针 cur

1) cur 不为空：访问该节点，并将该节点压入栈，并使 cur 指向该节点的左孩子(无论左孩子是否存在)
2) cur 为空：栈顶元素为最左端的节点，内循环结束

然后：对于栈

1) 栈为空：树已遍历，外循环结束
2) 栈不为空：弹出栈顶节点(该节点已被访问过)，将指针指向该节点的右孩子(无论右孩子是否存在)

```
PRE-ORDER-STACK(T)
let S be a STACK sized T.size
cur=T.root
while( S.empty==False or cur≠NULL)  //这个条件怎么理解：栈为空且指针为空才表明树已经完全输出
    while(cur≠NULL)  //循环终止时，栈顶元素(节点指针)指向没有左孩子的节点，cur指向空
        visit(cur)
        S.PUSH(cur)
        cur=cur.left
    if S.empty==Flase
        cur=S.POP
        cur=cur.right
```

7. 2. 二叉树中序遍历非递归算法

外循环体：对于当前指针**cur**：

首先：内循环体：对于当前指针**cur**

1) **cur**不为空：将**cur**指向的节点压入栈，并使**cur**指向该节点的左孩子(无论左孩子是否存在)
2) **cur**为空：栈顶元素为最左端的节点，内循环结束

然后：对于栈

1) 栈为空：树已遍历，外循环结束
2) 栈不为空，则弹出栈顶节点，并访问该节点，并使**cur**指向该节点的右孩子(无论右孩子是否存在

```
IN-ORDER-STACK(T)
let S be a STACK sized T.size
cur=T.root
while( S.empty==False or cur≠NULL)  //这个条件怎么理解：栈为空且指针为空才表明树已经完全输出
    while(cur≠NULL)  //循环终止时，栈顶元素(节点指针)指向没有左孩子的节点，cur指向空
        S.PUSH(cur)
        cur=cur.left
    if S.empty==Flase
        cur=S.POP
        visit(cur)
        cur=cur.right
```

7. 3. 二叉树后序遍历非递归算法 1

外循环体：对于当前指针cur

首先：内循环体：对于当前指针cur

1) **cur不为空：将入栈计数增加1(该节点的入栈计数变成了1)，然后将该节点压入栈，并使cur指向该节点的左孩子(无论左孩子是否存在)**
2) **cur为空：栈顶元素为最左端的节点，内循环结束**

然后：对于栈：

1) **栈为空：树已遍历，外循环结束**
2) **栈不为空：弹出栈顶元素记为N1**

- N1 的入栈计数为 2，访问该元素
- **N1 的入栈计数为 1，入栈计数增加 1(入栈计数变成了 2)，重新将该节点压入栈，并使 cur 指向该节点的右孩子(无论右孩子是否存在)**

```
POST-ORDER-STACK(T)
let S be a STACK sized T.size
let every Node's cnt be zero
cur=T.root
while( S.empty==False or cur≠NULL) //这个条件怎么理解：栈为空且指针为空才表明树已经完全输出
    while(cur≠NULL) //循环终止时，栈顶元素(节点指针)指向没有左孩子的节点，cur指向空
        cur.cnt=cur.cnt+1
        S.PUSH(cur)
        cur=cur.left
    if S.empty==Flase
        cur=S.POP
        if cur.cnt==2
            visit(cur)
            cur=NULL //保证下一次循环直接跳过内层的 while
        else cur.cnt=cur.cnt+1
            S.PUSH(cur)
            cur=cur.right
```

7.4. 二叉树后序遍历非递归算法 2

初始化：首先将根节点入栈，**cur**置空，**pre**置空**(cur**指向栈顶元素，**pre**指向上一次访问的元素**)**

循环体：对于栈

1) **栈不为空：cur指向栈顶元素，记为N1**
    - **若N1的左右孩子均不存在，或pre指针指向的节点是N1的孩子：弹出栈顶元素N1，并访问，并将pre指向该已被访问过的节点N1**
    - **若N1存在孩子，且pre指向的节点不是N1的孩子：若N1的右孩子存在，则将右孩子入栈，若N1的左孩子存在，再将左孩子入栈**
2) 栈为空：树已遍历，循环结束

关键点：节点压入栈的顺序为后序遍历的反序，即先当前，再有孩子，再左孩子

```
POST-ORDER-STACK
let S be a STACK sized T.size
S.PUSH(T.root)
pre=cur=NULL
while S.empty==False//栈不为空时进入循环
    cur=S.TOP//获取栈顶元素(非弹出)
    if cur.left==NULL and cur.right==NULL or pre≠NULL and pre.p=cur
        visit(cur)
        S.POP//弹出该元素
        pre=cur
    else_if cur.right≠NULL
            S.PUSH(cur.right)
        if cur.left≠NULL
            S.PUSH(cur.left)
```

7.5. 二叉树的前序遍历的非递归非栈算法

```
POST-ORDER-ELSE
pre=NULL//前一节点初始化为空
cur=T.root//当前节点初始化为根节点
while(cur≠NULL)
    if pre==cur.p //当前节点是上一节点的子节点
        visit(cur)      //访问当前节点
        pre=cur
        if cur.left≠NULL
            cur=cur.left
        elseif cur.right≠NULL
            cur=cur.right
        else
            cur=cur.p
    elseif pre==cur.left//上一节点是当前节点的左孩子
        pre=cur
        if cur.right≠NULL
            cur=cur.right
        else
            cur=cur.p
    else//上一节点是当前节点的右孩子
        pre=cur
        cur=cur.p
```

访问出现在左孩子判断前

7. 6. 二叉树的中序遍历的非递归非栈算法

```
IN-ORDER-ELSE
pre=NULL//前一节点初始化为空
cur=T.root//当前节点初始化为根节点
while(cur≠NULL)
    if pre==cur.p //当前节点是上一节点的子节点
        pre=cur
        if cur.left≠NULL
            cur=cur.left
        elseif cur.right≠NULL
            visit(cur)      //访问当前节点
            cur=cur.right
        else
            visit(cur)      //访问当前节点
            cur=cur.p
    elseif pre==cur.left//上一节点是当前节点的左孩子
        pre=cur
        visit(cur)      //访问当前节点
        if cur.right≠NULL
            cur=cur.right
        else
            cur=cur.p
    else//上一节点是当前节点的右孩子
        pre=cur
        cur=cur.p
```

访问出现在右孩子判断前

7.7. 二叉树的后序遍历的非递归非栈算法

```
POST-ORDER-ELSE
pre=NULL//前一节点初始化为空
cur=T.root//当前节点初始化为根节点
while(cur≠NULL)
    if pre==cur.p //当前节点是上一节点的子节点
        pre=cur
        if cur.left≠NULL
            cur=cur.left
        elseif cur.right≠NULL
            cur=cur.right
        else
            visit(cur)      //访问当前节点
            cur=cur.p
    elseif pre==cur.left//上一节点是当前节点的左孩子
        pre=cur
        if cur.right≠NULL
            cur=cur.right
        else
            visit(cur)      //访问当前节点
            cur=cur.p
    else//上一节点是当前节点的右孩子
        visit(cur)      //访问当前节点
        pre=cur
        cur=cur.p
```

访问出现在返回父节点之前

7. 8. 二叉树树的析构：

7. 8. 1. 通过后续遍历的栈算法 2 的变形来实现

```
~TREE(T)
let S be a STACK sized T.size
pre=cur=NULL
S.PUSH(T.root)
while S.empty==False//栈不为空时进入循环
    cur=S.TOP//获取栈顶元素(非弹出)
    if cur.left==NULL and cur.right==NULL    //与后续遍历的不同之处
        S.POP//弹出该元素
        pre=cur
        cur=cur.p
        if cur≠NULL
            if pre==cur.left
                cur.left==NULL
            else cur.right=NULL
        delete pre//释放被弹出的栈顶元素的内存
    else_if cur.right≠NULL
            S.PUSH(cur.right)
        if cur.left≠NULL
            S.PUSH(cur.left)
```

7. 8. 2. 通过指针路径算法的变形来实现

```
~TREE(T)
pre=NULL//前一节点初始化为空
cur=T.root//当前节点初始化为根节点
while(true)
    if cur==NULL
        break //当前节点为空时，退出循环
    if pre==cur.p //当前节点是上一节点的子节点
        if cur.left≠NULL
            pre=cur
            cur=cur.left
            continue
        if cur.right≠NULL
            pre=cur
            cur=cur.right
            continue
        pre=cur
        cur=cur.p
        if cur≠NULL and pre==cur.left
            i=1
        elseif cur≠NULL and pre==cur.right
            i=2
        delete pre      continue
    elseif pre==cur.left//上一节点是当前节点的左孩子
        if cur.right≠Null
            pre=cur
            cur=cur.right
            continue
        pre=cur
        cur=cur.p
        if cur≠NULL and pre==cur.left
            i=1
        elseif cur≠NULL and pre==cur.right
            i=2
        delete pre      continue
    elseif pre==cur.right//上一节点是当前节点的右孩子
        pre=cur
        cur=cur.p
        if cur≠NULL and pre==cur.left
            i=1
        elseif cur≠NULL and pre==cur.right
            i=2
        delete pre      continue
    else switch(i)
```

```
case 1: if cur.right≠Null
            pre=cur
            cur=cur.right
            continue
        pre=cur
        cur=cur.p
        if cur≠NULL and pre==cur.left
            i=1
        elseif cur≠NULL and pre==cur.right
            i=2
        delete pre     continue
case 2: pre=cur
        cur=cur.p
        if cur≠NULL and pre==cur.left
            i=1
        elseif cur≠NULL and pre==cur.right
            i=2
        delete pre     continue
```

7.9. 二叉树的遍历总结

对于后续遍历的栈算法 2 与非栈非递归算法的比较：

两者均有 pre 与 cur 指针,但不同的是

Stack2 算法中：cur 指向的是栈顶元素,pre 指向的是上一次访问的元素

Fix 算法中：cur 指向的是指针路径的顶端元素，pre 指向的是指针路径的顶端第二个元素，cur 与 pre 必定为父子或子父关系。注意：pre 并非指向访问的元素，只是指针路径中的顶端第二个元素

对于如下的一棵树，**Fix** 算法中的指针路径(不存在跳跃，必须连续前进)为

1-2-4-8-4-9-4-2-5-10-5-2-1-3-6-11-6-3-7-3-1-Null

```
                              1
                 2                                  3
        4                 5                 6                 7
   8         9                 10        11
```

Chapter 8. 二叉搜索树

8.1. 插入二叉搜索树

```
TREE_INSERT(T,z)
y=T.nil
x=T.root
while x≠T.nil//循环结束时 x 指向空，y 指向上一个 x
    y=x
    if z.key<x.key
        x=x.left
    else x=x.right
z.p=y//将这个叶节点作为 z 的父节点
if y==T.nil
    T.root=z
elseif z.key<y.key
    y.left=z
else y.right=z
```

```
TREE-SEARCH(x,k) x 指向根节点
while x≠T.nil and k≠x.key//当找到该元素或者达到搜索路径的顶端(叶节点的孩子节点)循环结束
    if k<x.key
        x=x.left
    else x=x.right
return x
```

以 x 为根节点的子树的最大值

```
TREE-MAXIMUM(x)
while x.right≠T.nil//沿着右孩子路径一直搜索到没有右孩子的节点
    x=x.right
return x
```

以 x 为根节点的子树的最小值

```
TREE-MINIMUM(x)
while x.left≠T.nil//沿着左孩子路径一直搜索到没有左孩子的节点
    x=x.left
return x
```

8. 2. 后继元素

1、若该元素含有右孩子，那么后驱元素必定在以右孩子为根节点的子树中

2、**若该元素没有右孩子，那么搜索子树的根节点 y 第一次以左孩子的身份作作为其父节点的孩子，那么该父节点就是后驱元素(第一次父比子大)**

```
TREE-SUCCESSOR(x)
if x.right≠T.nil
    return TREE-MINIMUM(x.right)//找到以右孩子为根节点的最大值
y=x.p
while y≠T.nil and x≠y.left//循环结束时 x 为 y 的左孩子
    x=y
    y=y.p
return y//y 若为空，则代表无后继元素
```

8. 3. 前驱元素

1、若该元素含有左孩子，那么前驱元素必定在以左孩子为根节点的子树中

2、**若该元素没有左孩子，那么搜索子树的根节点 y 第一次以右孩子的身份作作为其父节点的孩子，那么该父节点就是后驱元素(第一次父比子小)**

```
TREE-PREDECESSOR(x)
if x.left≠T.nil
    return TREE-MAXIMUM(x.left)//找到以左孩子为根节点的最大值
y=x.p
while y≠T.nil and x≠y.right//循环结束时x为y的右孩子
    x=y
    y=y.p
return y//y 若为空，则代表无前驱元素
```

查找关键字为k的元素

8. 4. 删除

删除节点的辅助函数：用另一棵树替换一棵树并成为其双亲的孩子节点

需要更改的指针：v 的父节点，以及 u 的父节点的相应的孩子节点

```
TRANSPLANT(T,u,v)
if u.p==T.nil
    T.root=v
elseif u==u.p.left
    u.p.left=v
else u.p.right=v
if v≠T.nil
    v.p=u.p
```

//u.p=u.left=u.right=NIL 这句不能有，需要完整保留以 u 为根节点的子树(u 的双亲未必是 NIL)

删除元素**版本 1**：(假定删除 z 节点)

①若 z 节点没有孩子，那么直接删除 z 即可

②若 z 节点只有一个孩子，那么将这个孩子作为根节点的子树替换以 z 为根节点的子树，并成为 z 的双亲的孩子

③若 z 节点有两个孩子，那么找到 z 的后继 y(一定在右子树中)，并让 y 占据 z 的位置。z 的原来右子树部分成为 y 的新的右子树，并且 z 的左子树成为 y 的新左子树

删除指定关键字的节点

```
TREE-DELETE1(T,z)
1 if z.left==T.nil
2       TRANSPLANT(T,z,z.right)
3 elseif z.right==T.nil
4       TRANSPLANT(T,z,z.left)
5 else y=TREE-MINIMUM(z.right) //找到z的后继，由于z存在左右孩子，故后继为右子树中的最小值
6       if y≠z.right//如果y是z的右孩子，那么y的右子树会保留，只需要更新y的左子树即可
7             TRANSPLANT(T,y,y.right)
8             y.right=z.right
9             y.right.p=y
10        TRANSPLANT(T,z,y)
11        y.left=z.left
12        y.left.p=y
```

6-12 行可改为以下形式：无论何种情况都会更新 y 的左右子树

```
6        TRANSPLANT(T,y,y.right)
7        y.right=z.right
8        y.right.p=y
9        TRANSPLANT(T,z,y)
10       y.left=z.left
11       y.left.p=y
```

删除元素**版本 2**：(假定删除 z 节点)

①若 z 节点没有孩子，那么直接删除 z 即可

②若 z 节点只有一个孩子，那么将这个孩子作为根节点的子树替换以 z 为根节点的子树，并成为 z 的双亲的孩子

③若 z 节点有两个孩子，那么找到 z 的前驱 y(一定在左子树中)，并让 y 占据 z 的位置。z 的原来左子树部分成为 y 的新的左子树，并且 z 的右子树成为 y 的新右子树

删除指定关键字的节点

```
TREE-DELETE2(T,z)
1 if z.left==T.nil
2       TRANSPLANT(T,z,z.right)
3 elseif z.right==T.nil
4       TRANSPLANT(T,z,z.left)
5 else y=MAXIMUM(T,z.left)//找到z的前驱，由于z存在左右孩子，故前驱为左子树中的最大值
6       if y≠z.left//如果y是z的左孩子，那么y的左子树会保留，只需要更新y的右子树即可
7               TRANSPLANT(T,y,y.left)
8               y.left=z.left
9               y.left.p=y
10      TRANSPLANT(T,z,y)
11      y.right=z.right
12      y.right.p=y
```

6-12 行可改为以下形式：无论何种情况都会更新 **y** 的左右子树

```
6       TRANSPLANT(T,y,y.left)
7       y.left=z.left
8       y.left.p=y
9       TRANSPLANT(T,z,y)
10      y.right=z.right
11      y.right.p=y
```

Chapter 9. 红黑树

9.1. 定义

9.1.1. 节点

1、节点的属性

1) val：关键字
2) left：左孩子节点
3) right：右孩子节点
4) parent：父节点
5) color：颜色

2、节点的性质

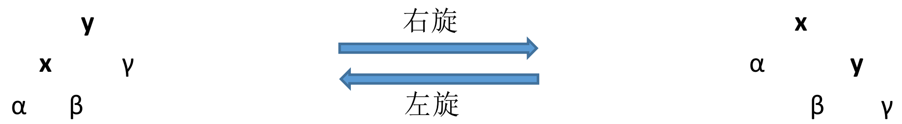
1) 每个节点或是红色的，或是黑色的
2) 根节点是黑色的
3) 每个叶节点(nil)是黑色的
4) 如果一个节点是红色的，则它的两个子节点都是黑色的
5) 对每个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点

9.1.2. 树

1、属性

1) nil：哨兵节点
2) root：根节点

9. 2. 旋转



左右旋的变换中，需要改变的就是节点 β，节点 α 和 γ 不需要改变

9. 2. 1. 左旋

```
LEFT-ROTATE(T,x)
1 y=x.right
2 x.right=y.left
3 if y.left≠T.nil
4       y.left.p=x     //1-4行首先令节点b成为x的右孩子(改动两个指向：x.right 以及 b.p)
5 y.p=x.p
6 if x.p==T.nil
7       T.root=y
8 elseif x==x.p.left
9       x.p.left=y
10 else x.p.right=y // 5-10行再令节点y代替x(改动两个指向： y.p 以及 x.p.left or x.p.right or root)
11 y.left=x
12 x.p=y           //11-12最后令x成为y的左孩子(改动两个指向：y.left 以及 x.p)
```

9. 2. 2. 右旋

```
RIGHT-ROTATE(T,y)
1 x=y.left
2 y.left=x.right
3 if x.right≠T.nil
4       x.right.p=y     //1-4行首先令节点b成为y的左孩子(改动两个指向：y.l 以及 b.p)
5 x.p=y.p
6 if y.p==T.nil
7       root=x
8 elseif y==y.p.left
9       y.p.left=x
10 else y.p.right=x // 5-10行再令节点x代替y(改动两个指向： x.p 以及 y.p.left or y.p.right or root)
11 x.right=y
12 y.p=x           //11-12最后令y成为x的右孩子(改动两个指向：x.right 以及 y.p)
```

(颜色改动+旋转变换)后性质 5 是否成立：看变换后 x、y 节点的父节点黑高是否发生变化即可

9. 3. 插入

```
RB-INSERT(T,z)
1 y=T.nil
2 x=T.root
3 while x≠T.nil
4       y=x
5       if z.key<x.key
6            x=x.left
7       else x=x.right
8 z.p=y
9 if y==T.nil
10      T.root=z
11 elseif z.key<y.key   //这里与567行最好保持一致
12      y.left=z
13 else y.right=z
14 z.left=T.nil
15 z.right=T.nil
16 z.colcor=RED
17 RB-INSERT-FIXUP(T,z)
```

插入的节点被设定为红色：

那么可能会违背性质2或4，但只能是其中之一

①当插入的节点是第一个时，此时根节点是红色，违背了性质2，但其子节点与父节点均为T.nil 是黑色，没有违反性质4

②当插入的节点不是根节点，并且其父节点也为红色时，违背了性质4

纠正思路：

对于错误①的修正，只需要将根节点设为黑色即可

对于错误②的修正，由于z与其父节点均为红色，那么祖父节点必为黑色，根据z的叔节点的颜色状况以及z作为z.p的左右孩子，分三种情况讨论：

9.3.1. 插入辅助

①当 z 的父节点是祖父节点的左孩子时：(叔节点为祖父节点的右孩子)

情况1：z节点的父节点以及z节点的叔节点都是红色：将z节点的父节点以及叔节点置为黑色，z节点的祖父节点置为红色，继续循环z的祖父节点(z=z.p.p)

z.p.p　　　　　　　　　　z.p.p
z.p　　y　　→　　z.p　　y
z　　　　　　　　　　　　z

z可为z.p的左或右孩子

情况2：z节点的父节点为红色，叔节点为黑色，z为父节点的右孩子，对z的父节点做一次左旋，转为情况3:(旋转前后z所表示的关键字发生改变，但是z的祖父节点没有变)

z.p.p　　　　　　　　　　z.p.p
z.p　　y　　→　　z　　y
z　　　　　　　　　　　　z.p--------->z’

情况3：z节点的父节点为红色，叔节点为黑色，z为父节点的左孩子，首先将父节点设为黑色，祖父节点设为红色，然后对祖父节点做一次右旋

z.p.p　　　　　　　　　　z.p
z.p　　y　　→　　z　　z.p.p
z　　　　　　　　　　　　y

②当 z 的父节点是祖父节点的右孩子时，(叔节点为祖父节点的左孩子)

情况1：z节点的父节点以及z节点的叔节点都是红色：将z节点的父节点以及叔节点置为黑色，z节点的祖父节点置为红色，继续循环z的祖父节点(z=z.p.p)

z.p.p　　　　　　　　　　z.p.p
y　　z.p　　→　　y　　z.p
z　　　　　　　　　　　　z

z可为z.p的左或右孩子

情况2：z节点的父节点为红色，叔节点为黑色，z为父节点的左孩子，对z的父节点做一次右旋，转为情况3：(旋转前后z所表示的关键字发生改变，但是z的祖父节点没有改变)

z.p.p　　　　　　　　　　z.p.p
y　　z.p　　→　　y　　z
z　　　　　　　　　　　　z.p--------->z’

情况3：z节点的父节点是红色，叔节点是黑色，z为父节点的右孩子，首先将父节点设为黑色，祖父节点设为红色，然后对祖父节点做一次左旋

z.p.p　　　　　　　　　　z.p
y　　z.p　　→　　z.p.p　　z
z　　　　　　　　　　　　y

旋转前插入红z不会导致性质5破坏:bh(z.p.p)=bh(z)=bh(z.p)=bh(y)+1

旋转后：由于bh(z)=by(y)+1保持不变，因此

bh(z.p)=bh(z.p.p)= bh(z)=by(y)+1 性质5依然成立

```
RB-INSERT-FIXUP(T,z)
while z.p.color==RED//由于 z.p 是红色，因此访问 z.p.p 的任何属性都是安全的
    if z.p==z.p.p.left
        y=z.p.p.right
        if y.color==RED
            z.p.color=BLACK
            y.color=BLACK
            z.p.p.color=RED
            z=z.p.p//继续循环
        else_if z==z.p.right
                z=z.p
                LEFT-ROTATE(T,z)
            z.p.color=BLACK
            z.p.p.color=RED
            RIGHT-ROTATE(T,z.p.p)//循环结束
    else z.p==z.p.p.right
        y=z.p.p.left
        if y.color==RED
            z.p.color=BLACK
            y.color=BLACK
            z.p.p.color=RED
            z=z.p.p//继续循环
        else_if z==z.p.left
                z=z.p
                RIGHT-ROTATE(T,z)
            z.p.color=BLACK
            z.p.p.color=RED
            LEFT-ROTATE(T,z.p.p) //循环结束
T.root.color=BLACK//针对第一个插入的 z，不会进入循环(性质 4 成立，但性质 2 破坏，这里纠正)
```

将v为根节点的子树代替u为根节点的子树

```
RB-TRANSPLANT(T,u,v)
if u.p==T.nil
    T.root=v
elseif u==u.p.left
    u.p.left=v
else u.p.right=v
v.p=u.p             //与搜索二叉树相比，这里没有判断，即使v是哨兵，也执行此句,对于移动到y位置的节点x(可能是哨兵)，会访问x.p，因此这里需要进行赋值
```

9. 4. 删除函数：

```
RB-DELETE(T,z)
1 y=z
2 y-original-color=y.color
3 if z.left==T.nil
4       x=z.right
5       RB-TRANSPLANT(T,z,z.right)
6 elseif z.right==T.nil
7       x=z.left
8       RB-TRANSPLANT(T,z,z.left)
9 else y=TREE-MINIMUM(z.right)
10      y-original-color=y.color
11      x=y.right
12      if y.p==z
13          x.p=y//使得x为哨兵节点时也成立
14      else RB-TRANSPLANT(T,y,y.right)//即使y.right是哨兵，也会指向y的父节点
15          y.right=z.right
16          y.right.p=y
17      RB-TRANSPLANT(T,z,y)
//17行运行之后，13、14行都会保证x指向原始y父节点的位置
18      y.left=z.left
19      y.left.p=y
20      y.color=z.color
21 if y-original-color==BLACK
22      RB-DELETE-FIXUP(T,x)
```

总结：

1) 删除最终转化为删除一个最多只有一个孩子的节点
   - 当被删除节点 z 最多只有只有一个孩子，满足该条规律
   - 当被删除节点 z 有两个孩子，那么找到该节点的后继节点 y，此时 y 节点必然最多只有一个右孩子，于是将其右孩子 y.right transplant 到到 y 节点处以删除 y 节点，然后再将 y 节点移动到 z 节点处，并保持 z 节点原来的颜色，那么等价于删除 y 节点
2) 当被删除节点的颜色为红色，那么不会破坏红黑树的性质
3) 当被删除的节点是黑色，那么transplant到该节点的节点x如果是黑色，那么为了保持黑高不变的性质，x必须含有双重黑色，此时又破坏了性质1，需要进行维护矫正

9.4.1. 删除辅助

```
RB-DELETE(T,z)
y=z
y-original-color=y.color
if z.left==T.nil
    x=z.right
    RB-TRANSPLANT(T,z,z.right)
elseif z.right==T.nil
    x=z.left
    RB-TRANSPLANT(T,z,z.left)
else y=TREE-MINIMUM(z.right)
    y-original-color=y.color
    x=y.right
    RB-TRANSPLANT(T,y,y.right)//即使y.right是哨兵，也会指向y的父节点
    y.right=z.right
    y.right.p=y
    RB-TRANSPLANT(T,z,y)
    y.left=z.left
    y.left.p=y
    y.color=z.color
if y-original-color==BLACK
    RB-DELETE-FIXUP(T,x)
```

蓝色部分为不同之处，即不用讨论(y.parent==z)也可以

y节点：**为删除节点z(z的孩子至少有一个为nil)**或者**将要移动到被删除节点z的节点(z有两个非nil的孩子)**

y-original-color：**y节点的原始颜色**

x：**指向将要移动到y节点的节点(x代表占有y原来位置的节点)**

1、**当y-original-color为红色时：不会违反红黑树的任何性质。**

①**当y为被删除节点时**：若y为红色，那么它的父节点为黑色，孩子节点也必为黑色，将孩子移植到该位置不会违反任何性质。

②**当y节点为z节点的后继时**：若y为红色，那么y节点的父节点以及y节点的右子节点(可能为哨兵)必为黑色，将y.right移植到y的位置，不会违反任何性质；**如果z节点是黑色的**，那么删除z节点后z的任意祖先的黑高将少一，但是由于将y的颜色设为黑色，做了补偿。**如果z节点是红色的**，将y节点也设为红色，那么删除z节点不会违反性质5

因此y-original-color为红色时，不会违反红黑树的任何性质

2、**当y-original-color为黑色时：可能会违反性质2或4或5。**

①如果 y 是根节点，而 y 的一个红色孩子成为新的根节点，违反了性质 2

②如果x和x.p是红色，违反了性质4

③在树中删除或移动y将导致先前包含y的简单路径上的黑色节点少1

若z节点的孩子均不为T.nil，会违反性质的部分是以y的原位置为根节点的子树(包括其父节点)

①当 x 是其父亲的左孩子时：x 为双重黑色

*情况1：*x的兄弟节点w是红色的(w必有两个黑色的<u>非哨兵子节点</u>，且父亲必为黑色)

将x.p置为红色，w置为黑色，对x.p做一次左旋并更新w，即可将情况1转为234的一种

B　　　　互换BD颜色　　　　D

A(x)　D(w)　　→　　B　E

C　E　　对B做一次左旋　　A(x)　C(w’)

*情况2：*x的兄弟节点w是黑色，并且w的两个子节点都是黑色(可以是哨兵)

由于x为双重黑色，为了取消x的双重性，将x与w都去掉一层黑色属性，因此x变为单黑，w变为红色，并更新x(将双重属性赋予x的父节点)，并继续循环

B　　　　除去x的双重特性，w置为红色　　　　B(x’)

A(x)　D(w)　　→　　A　D

C　E　　将双重特性移交给父节点　　C　E

*情况3：*x的兄弟节点w是黑色，w的左孩子是红色，右孩子是黑色

交换w与其左孩子的颜色，对w进行右旋，并更新w，即可转为情况4

B　　　　互换DC颜色　　　　B

A(x)　D(w)　　→　　A(x)　C(w’)

C　E　　对w做一次右旋　　D

E

*情况4：*x的兄弟节点是黑色，且w的右孩子是红色

交换BD颜色，将E置为黑色，并对B做一次左旋，即可退出循环

B　　　　互换BD颜色　　　　D

A(x)　D(w)　　→　　B　E

C　E　　对B做一次左旋　　A　C

②当 x 是其父亲的右孩子时：x 为双重黑色

情况1： x的兄弟节点w是红色的(w必有两个黑色的非哨兵子节点，且父亲必为黑色)

将x.p置为红色，w置为黑色，对x.p做一次右旋并更新w，即可将情况1转为234的一种

B　　　　互换BD颜色　　　　D

D(w)　A(x)　　→　　C　　B

C　E　　对B做一次右旋　　E(w')　A(x)

情况2： x的兄弟节点w是黑色，并且w的两个子节点都是黑色(可以是哨兵)

由于x为双重黑色，为了取消x的双重性，将x与w都去掉一层黑色属性，因此x变为单黑，w变为红色，并更新x(将双重属性赋予x的父节点)，并继续循环

B　　　　除去x的双重特性，w置为红色　　　　B(x')

D(w)　A(x)　　→　　D　　A

C　E　　将双重特性移交给父节点　　C　E

情况3： x的兄弟节点w是黑色，w的左孩子是黑色，右孩子是红色

交换w与其右孩子的颜色，对w进行左旋，并更新w，即可转为情况4

B　　　　互换DC颜色　　　　B

D(w)　A(x)　　→　　E(w')　A(x)

C　E　　对w做一次左旋　　D

C

情况4： x的兄弟节点是黑色，且w的左孩子是红色

交换BD颜色，将C置为黑色，并对B做一次右旋，即可退出循环

B　　　　互换BD颜色　　　　D

D(w)　A(x)　　→　　C　　B

C　E　　对B做一次右旋　　E　A

```
RB-DELETE-FIXUP(T,x)
while x≠T.root and x.color ==BLACK//若 x 是红色的，那么将 x 改为黑色即可
    if x==x.p.left//x可以是哨兵，访问x.p是合法的，因为在Delete中已经设置过
        w=x.p.right
        if w.color==RED
            w.color=BLACK
            x.p.color=RED
            LEFT-ROTATE(T,x.p)
            w=x.p.right
        if w.left.color==BLACK and w.right.color==BLACK
            w.color=RED
            x=x.p
        else_if w.right.color==BLACK
                w.left.color=BLACK
                w.color=RED
                RIGHT-ROTATE(T,w)
                w=x.p.right
            w.color=x.p.color
            x.p.color=BLACK
            w.right.color=BLACK
            LEFT-ROTATE(T,x.p)
            x=T.root
    elseif x==x.p.right
        w=x.p.left
        if w.color==RED
            w.color=BLACK
            x.p.color=RED
            RIGHT-ROTATE(T,x.p)
            w=x.p.left
        if w.left.color==BLACK and w.right.color==BLACK
            w.color=RED
            x=x.p
        else_if w.left.color==BLACK
                w.right.color=BLACK
                w.color=RED
                LEFT-ROTATE(T,w)
                w=x.p.left
            w.color=x.p.color
            x.p.color=BLACK
            w.left.color=BLACK
            RIGHT-ROTATE(T,x.p)
            x=T.root
x.color=BLACK
```

Chapter 10. AVLTree(版本 1)

10.1. 定义

1、节点的属性

1) val
2) left
3) right
4) parent
5) height

2、节点的性质

1) 每个节点的左子树与右子树的高度最多不超过 1
2) 节点的高度：从给定节点到其最深叶节点所经过的边的数量

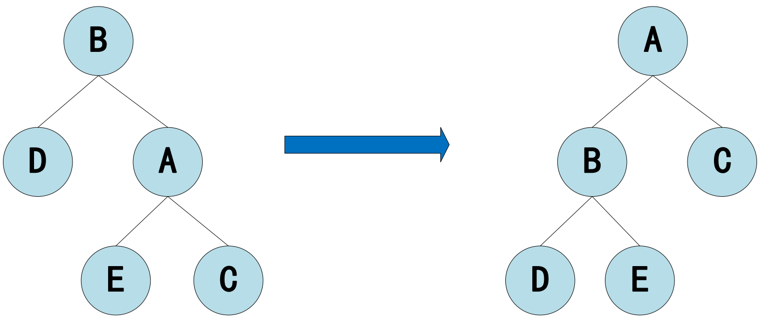
10.2. 平衡性破坏分析

1、调整前某节点 X 的高度记为 HX，调整后，该节点的高度记为 HX+

10.2.1. 可旋性分析

1、对于左旋，必须满足如下性质

- 首先，只有当右子树的高度大于左子树的高度才会有左旋的需求，因此 $H_D=H_A-1$ 或 $H_D=H_A-2$
- 其次，只有 $H_C>=H_E$ 时，旋转后该子树的所有节点才满足 AVL 树的性质，分析如下
- 旋转前，各节点高度如下
  - $H_D=H_A-1$ 或 $H_D=H_A-2$
  - $H_C=H_A-1$
  - $H_E=H_A-1$ 或 $H_E=H_A-2$
- 旋转后，各节点高度如下
  - $H_{D+}=H_D=H_A-1$ 或 H_A-2
  - $H_{E+}=H_E=H_A-1$ 或 H_A-2
  - $H_{C+}=H_C=H_A-1$
  - 旋转后，B 节点平衡，$H_{B+}=H_A$ 或 $H_{B+}=H_A-1$
  - 旋转后，A 节点平衡，$H_{A+}=H_A$ 或 $H_{A+}=H_A+1$



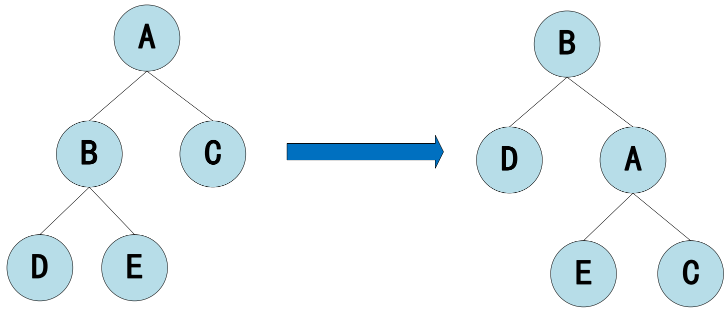
2、对于右旋，必须满足如下性质

- 首先，只有当左子树的高度大于右子树的高度才会有右旋的需求，因此 $H_C=H_B-1$ 或 $H_C=H_B-2$
- 其次，只有 $H_D>=H_E$ 时，旋转后该子树的所有节点才满足 AVL 树的性质，分析如下
- 旋转前，各节点高度如下
  - $H_C=H_B-1$ 或 $H_C=H_B-2$

- $H_D=H_B-1$
- $H_E=H_B-1$ 或 $H_E=H_B-2$

➢ 旋转后，各节点高度如下

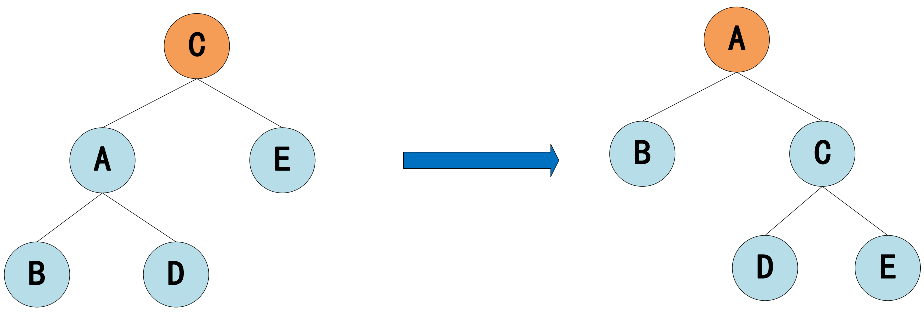
- $H_{D+}=H_D=H_B-1$
- $H_{E+}=H_E=H_B-1$ 或 H_B-2
- $H_{C+}=H_C=H_B-1$ 或 H_B-2
- 旋转后，A 节点平衡，且 $H_{A+}=H_B$ 或 $H_{A+}=H_B-1$
- 旋转后，B 节点平衡，且 $H_{B+}=H_B$ 或 $H_{B+}=H_B+1$



10.2.2. 平衡性破坏分析

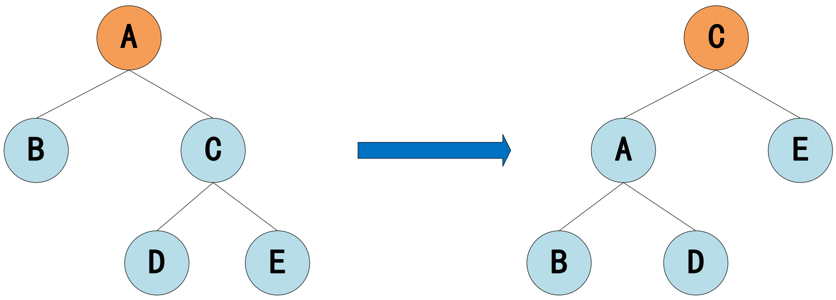
1、当 C 为平衡被破坏的节点，且 C 的左子树比右子树的高度大 2

➢ 需要对 C 进行一次右旋
➢ 右旋的前提是 $H_B>=H_D$
➢ 若不满足 $H_B>=H_D$，则需要首先对 A 进行一次左旋，而左旋又存在前提
➢ 一直往下递归，直至满足旋转条件



2、当 A 为平衡被破坏的节点，且 A 的右子树比左子树的高度大 2

➢ 需要对 A 进行一次左旋
➢ 左旋的前提是 $H_E>=H_D$
➢ 若不满足 $H_E>=H_D$，则需要首先对 C 进行一次右旋，而右旋又存在前提
➢ 一直往下递归，直至满足旋转条件



10.3. 基本操作

```
HEIGHT(T,x)
if x.left.height≥x.right.height     //左右节点均存在
        x.height=x.left.height+1
else x.height=x.right.height+1
```

```
TRANSPLANT(T,u,v)     //该函数与红黑树完全一致(都含有哨兵节点)
if u.p==T.nil
        T.root=v
elseif u==u.p.left
        u.p.left=v
else u.p.right=v
        v.p=u.p
```

10. 3. 1. 左旋

```
LEFT-ROTATE(T,x)
1 y=x.right
2 x.right=y.left
3 if y.left≠T.nil
4       y.left.p=x     //1-4行首先令节点b成为x的右孩子(改动两个指向：x.right 以及 b.p)
5 y.p=x.p
6 if x.p== T.nil
7       T.root=y
8 elseif x==x.p.left
9       x.p.left=y
10 else x.p.right=y     // 5-10行再令节点y代替x(改动两个指向：y.p 以及 x.p.left or x.p.right or root)
11 y.left=x
12 x.p=y            //11-12最后令x成为y的左孩子(改动两个指向：y.left 以及 x.p)
13 HEIGHT(T,x)
14 HEIGHT(T,y)     //13 14 两行顺序不得交换
15 return y     //返回旋转后的子树根节点
```

10. 3. 2. 右旋

```
RIGH-TROTATE(T,y)
1 x=y.left
2 y.left=x.right
3 if x.right≠T.nil
4         x.right.p=y      //1-4行首先令节点b成为y的左孩子(改动两个指向：y.l  以及  b.p)
5 x.p=y.p
6 if y.p==T.nil
7         root=x
8 elseif y==y.p.left
9         y.p.left=x
10 else y.p.right=x      // 5-10行再令节点x代替y(改动两个指向：x.p  以及  y.p.left or y.p.right or root)
11 x.right=y
12 y.p=x              //11-12最后令y成为x的右孩子(改动两个指向：x.right  以及  y.p)
13 HEIGHT(T,y)
14 HEIGHT(T,x)        //13 14 两行顺序不得交换
15 return x           //返回旋转后的子树根节点
```

10. 3. 3. 维护 AVL 树的性质

HOLD-ROTATE(T,x,orientation)

```
let stack1,stack2 be two stacks//不考虑实际用到的大小，直接用树的大小来分配堆栈空间大小
stack1.push(x)
stack2.push(orientation)
cur=Nil
rotateRoot=Nil //对 x 尝试旋转后，返回最终旋转后的根节点
curOrientation=INVALID;
while(!stack1.Empty())
    cur=stack1.top()
    curOrientation=stack2.top()
    if curOrientation==LEFT //需要对cur尝试进行左旋
        if cur.right.right.height≥cur.right.left.height
            stack1.pop()
            stack2.pop()
            rotateRoot=LEFT-ROTATE(T,cur)
        else
            stack1.push(cur.right)//否则cur右孩子需要尝试进行右旋来调整
            stack2.push(RIGHT);
    elseif curOrientation ==RIGHT//需要对cur尝试进行右旋
        if cur.left.left.height≥cur.left.right.height
            stack1.pop()
            stack2.pop()
            rotateRoot=RIGHT-ROTATE(T,cur)
        else
            stack1.push(cur.left) //否则cur左孩子需要尝试进行左旋来调整
            stack2.push(LEFT)
return rotateRoot
```

10.4. 插入

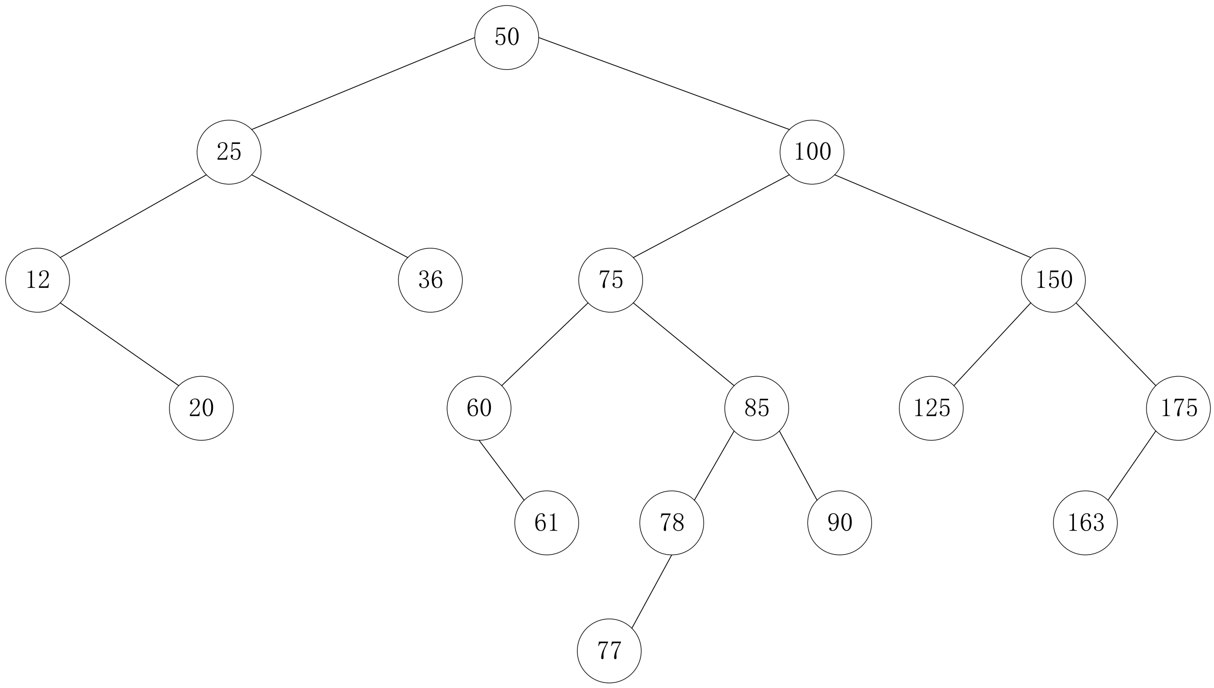
```
TREE-INSERT(T,z)
y=T.nil
x=T.root
while x≠T.nil//循环结束时 x 指向空，y 指向上一个 x
    y=x
    if z.key<x.key
        x=x.left
    else x=x.right
z.p=y//将这个叶节点作为 z 的父节点
if y==T.nil
    T.root=z
elseif z.key<y.key
    y.left=z
else y.right=z
z.left=T.nil
z.right=T.nil
FIXUP(T,z)
```

```
FIXUP(T,y)
if y==T.nil//为了使删除函数也能调用该函数，因为删除函数传入的参数可能是哨兵
    y=y.p
while(y≠T.nil) //沿着y节点向上遍历该条路径
    HEIGHT(y)
    if y.left.height==y.right.height+2 //左子树比右子树高2
        y= HOLD-ROTATE(T,y,2)
    elseif y.right.height=y.left.height+2
        y= HOLD-ROTATE(T,y,1)
    y=y.p
```

10. 5. 删除

```
TREE-DELETE(T,z)
y=z      //x 指向将要移动到 y 原本位置的节点，或者原本 y 节点的父节点
if z.left==T.nil
      x=y.right
      TRANSPLANT(T,z,z.right)
elseif z.right==T.nil
      x=y.left
      TRANSPLANT(T,z,z.left)
else y=TREE-MINIMUM(z.right) //找到z的后继，由于z存在左右孩子，故后继为右子树中的最小值
      x=y.right
      if y.p==z//如果y是z的右孩子，需要将x的parent指向y(使得x为哨兵节点也满足)
            x.p=y
      else TRANSPLANT(T,y,y.right)
            y.right=z.right
            y.right.p=y
        TRANSPLANT(T,z,y)
        y.left=z.left
        y.left.p=y
FIXUP(T,x)
```

参考函数HoldRotate思考该数数在插入77后的如何旋转以维持AVL性质



Chapter 11. AVLTree(版本 2)

11.1. 定义

1、节点的属性

6) val
7) left
8) right
9) parent
10) height

2、节点的性质

3) 每个节点的左子树与右子树的高度最多不超过 1
4) 节点的高度：从给定节点到其最深叶节点所经过的边的数量

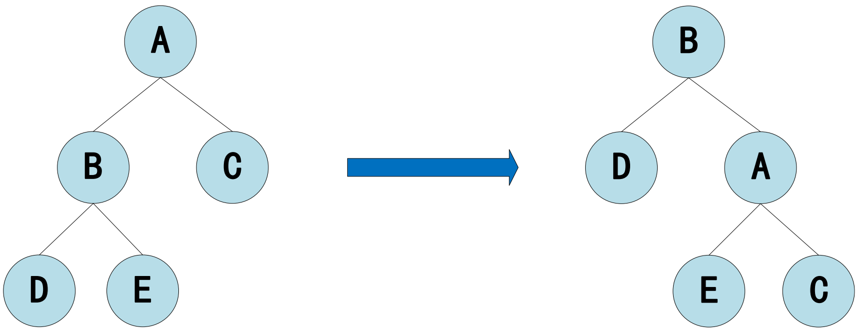
11.2. 平衡性破坏分析

1、与版本 1 的分析不同，这个版本的分析将会更加精确

2、当插入一个节点后，某节点 A 为从插入节点网上的第一个平衡性被破坏的节点，可以分为如下四种情况，又可分为两大类

1) 插入点位于 A 的左子节点的左子树--左左--第一类(外侧)
2) 插入点位于 A 的左子节点的右子树--左右--第二类(内侧)
3) 插入点位于 A 的右子节点的左子树--右左--第二类(内侧)
4) 插入点位于 A 的右子节点的右子树--右右--第一类(外侧)

11.2.1. 第一类不平衡(以左左为例)



1、调整前某节点 X 的高度记为 H_X，调整后，该节点的高度记为 H_{X+}

2、调整前，各节点的高度如下

- $H_B=H_C+2$
- $H_A=H_C+2$(为什么 A 和 B 高度相同，因为 B 的高度已经更新过了，而 A 仍然是是插入新节点之前的高度，即尚未维护 A 节点的 height 字段)
- $H_D=H_B-1=H_C+1$
- H_E 必定小于 H_D(否则在新节点插入到节点 D 为根节点的子树之前，A 节点就是不平衡的)，因此 $H_E=H_C$

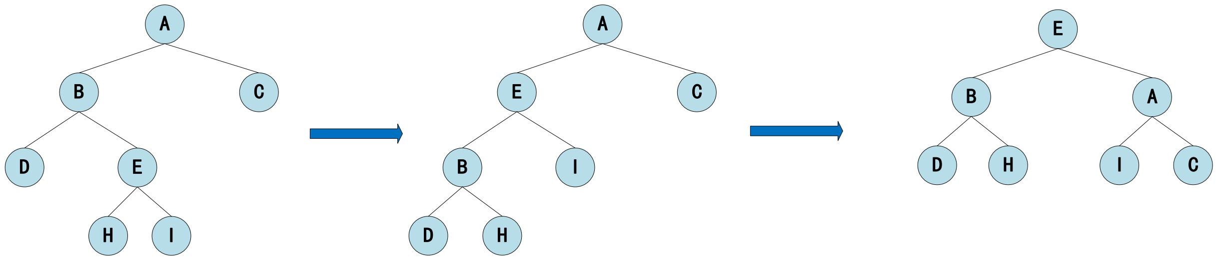
3、右旋调整后，各节点的高度如下

- $H_{D+}=H_D=H_C+1$

- $H_{E+}=H_E=H_C$
- $H_{C+}=H_C$
- 由于 $H_{E+}=H_{C+}$，于是 A 节点平衡，且 $H_{A+}=H_C+1$
- B 节点也是平衡的，且 $H_{B+}=H_C+2$

4、可以发现，调整前后子树根节点的高度都是 H_C+2，因此该节点上层的节点的平衡性不会被破坏，于是通过一次右旋，不平衡性即被消除

11.2.2. 第一类不平衡(以左右为例)



1、调整前某节点 X 的高度记为 H_X，第一次调整后，该节点的高度记为 H_{X+}，第二次调整后记为 H_{X++}

2、调整前，各节点的高度如下

- $H_B=H_C+2$
- $H_A=H_C+2$(为什么 A 和 B 高度相同，因为 B 的高度已经更新过了，而 A 仍然是是插入新节点之前的高度，即尚未维护 A 节点的 height 字段)
- $H_E=H_B-1=H_C+1$
- H_D 必定小于 H_E，因此 $H_D=H_C$
- H_H 与 H_I 至少有一个是 H_C，另一个可以是 H_C 或 H_C-1

3、对 B 节点进行一次左旋后，各节点高度如下

- $H_{D+}=H_D=H_C$
- $H_{H+}=H_H=H_C$ or H_C-1
- $H_{I+}=H_I=H_C$ or H_C-1
- $H_{C+}=H_C$
- $H_{A+}=H_A=H_C+2$
- $H_{B+}=H_C+1$
- 当 $H_{I+}=H_C-1$ 时，节点 E 可能是不平衡的，但是没关系，这只是个中间状态，$H_E=H_C+2$

4、对 A 节点进行一次右旋，各节点高度如下

- $H_{D++}=H_{D+}=H_C$
- $H_{H++}=H_{H+}=H_C$ or H_C-1
- $H_{I++}=H_{I+}=H_C$ or H_C-1
- $H_{C++}=H_{C+}=H_C$
- 旋转后，B 节点平衡，$H_{B++}=H_C+1$
- 旋转后，A 节点平衡，$H_{A++}=H_C+1$
- 因此旋转后，E 节点平衡，$H_{E++}=H_C+2$

5、可以发现，调整前后子树根节点的高度都是 H_C+2，因此该节点上层的节点的平衡性不会被破坏，于是通过一次右旋，不平衡性即被消除

11.3. 基本操作

```
HEIGHT(T,x)
if x.left.height≥x.right.height      //左右节点均存在
        x.height=x.left.height+1
else x.height=x.right.height+1
```

```
TRANSPLANT(T,u,v)      //该函数与红黑树完全一致(都含有哨兵节点)
if u.p==T.nil
        T.root=v
elseif u==u.p.left
        u.p.left=v
else u.p.right=v
        v.p=u.p
```

11.3.1. 左旋

```
LEFT-ROTATE(T,x)
1 y=x.right
2 x.right=y.left
3 if y.left≠T.nil
4       y.left.p=x     //1-4行首先令节点b成为x的右孩子(改动两个指向：x.right  以及  b.p)
5 y.p=x.p
6 if x.p== T.nil
7       T.root=y
8 elseif x==x.p.left
9       x.p.left=y
10 else x.p.right=y     // 5-10行再令节点y代替x(改动两个指向：y.p  以及  x.p.left or x.p.right or root)
11 y.left=x
12 x.p=y     //11-12最后令x成为y的左孩子(改动两个指向：y.left  以及  x.p)
13 HEIGHT(T,x)
14 HEIGHT(T,y)     //13 14 两行顺序不得交换
15 return y     //返回旋转后的子树根节点
```

11. 3. 2. 右旋

```
RIGH-TROTATE(T,y)
1 x=y.left
2 y.left=x.right
3 if x.right≠T.nil
4       x.right.p=y      //1-4行首先令节点b成为y的左孩子(改动两个指向：y.l 以及 b.p)
5 x.p=y.p
6 if y.p==T.nil
7       root=x
8 elseif y==y.p.left
9       y.p.left=x
10 else y.p.right=x      // 5-10行再令节点x代替y(改动两个指向：x.p 以及 y.p.left or y.p.right or root)
11 x.right=y
12 y.p=x             //11-12最后令y成为x的右孩子(改动两个指向：x.right 以及 y.p)
13 HEIGHT(T,y)
14 HEIGHT(T,x)        //13 14 两行顺序不得交换
15 return x          //返回旋转后的子树根节点
```

11. 4. 插入

```
TREE-INSERT(T,z)
y=T.nil
x=T.root
while x≠T.nil//循环结束时 x 指向空，y 指向上一个 x
    y=x
    if z.key<x.key
        x=x.left
    else x=x.right
z.p=y//将这个叶节点作为 z 的父节点
if y==T.nil
    T.root=z
elseif z.key<y.key
    y.left=z
else y.right=z
z.left=T.nil
z.right=T.nil
INSERT-FIX(T,z)
```

```
INSERT-FIX(T,z)
originHigh=z.h
HEIGHT(z)
r=z
if z.left.h==z.right.h+2
    if z.left.left.h>z.left.right.h    //第一类
        r=RIGHT-ROTATE(z)
    elseif z.left.left.h<z.left.right.h    //第二类
        LEFT-ROTATE(z.left)
        r=RIGHT-ROTATE(z)
    //不可能出现左右子树高度相同的情况，但是 DELETE-FIX 中可能出现，注意
elseif z.right.h==z.left.h+2
    if z.right.right.h>z.right.left.h    //第一类
        r=LEFT-ROTATE(z)
    elseif z.right.right.h<z.right.left.h    //第二类
        RIGHT-ROTATE(z.right)
        r=LEFT-ROTATE(z)
    //不可能出现左右子树高度相同的情况，但是 DELETE-FIX 中可能出现，注意
if r.h!=originHigh and r!=root
    INSERT-FIX(r.parent)
```

11.5. 删除

```
TREE-DELETE(T,z)
y=z     //x 指向将要移动到 y 原本位置的节点，或者原本 y 节点的父节点
p=y.parent     //p 为被删除节点的父节点
if z.left==T.nil
    TRANSPLANT(T,z,z.right)
elseif z.right==T.nil
    TRANSPLANT(T,z,z.left)
else y=TREE-MINIMUM(z.right) //找到z的后继，由于z存在左右孩子，故后继为右子树中的最小值
    if y==z.right     //这个边界判断必须，因为p必须定位到被删除节点的父节点
        p=y
    else
        p=y.parent
    TRANSPLANT(y,y.right)
    y.right=z.right
    y.right.parent=y
    y.left=z.left
    y.left.parent=y
    TRANSPLANT(T,z,y)
if p!=nil
    DELETE-FIX(p)
```

```
DELETE -FIX(T,z)
HEIGHT(z)
r=z
if z.left.h==z.right.h+2
    if z.left.left.h>=z.left.right.h     //第一类
        r=RIGHT-ROTATE(z)
    else     //第二类
        LEFT-ROTATE(z.left)
        r=RIGHT-ROTATE(z)
elseif z.right.h==z.left.h+2
    if z.right.right.h>=z.right.left.h     //第一类
        r=LEFT-ROTATE(z)
    else     //第二类
        RIGHT-ROTATE(z.right)
        r=LEFT-ROTATE(z)
if r!=root
    DELETE-FIX(r.parent)
```

Chapter 12.数据结构扩张

红黑树的扩展：每个节点带有另一个属性(以该节点为根节点的子树的节点个数(不包括Nil)

查找第i个顺序统计量(秩)

```
OS-SELECT(x,i)
r=x.left.size+1
if i==r
    return x
elseif i<r
    return OS-SELECT(x.left,i)
else return OS-SELECT(x,right,i-r)
```

```
OS-RANK(T,x)
r=x.left.size+1
y=x
while y≠T.root//循环不变式：每次迭代开始时，r为以y为根的子树中x节点的秩
    if y==y.p.right
        r=r+y.p.left.size+1
    y=y.p
return r
```

为了维护这个额外的属性，红黑树以下函数需要作出修改

左旋：

```
LEFT-ROTATE(T,x)
1 y=x.right
2 x.right=y.left
3 if y.left≠T.nil
4       y.left.p=x      //1-4行首先令节点b成为x的右孩子(改动两个指向：x.right  以及  b.p)
5 y.p=x.p
6 if x.p==T.nil
7       T.root=y
8 elseif x==x.p.left
9       x.p.left=y
10 else x.p.right=y // 5-10行再令节点y代替x(改动两个指向：y.p  以及  x.p.left or x.p.right or root)
11 y.left=x
12 x.p=y              //11-12最后令x成为y的左孩子(改动两个指向：y.left  以及  x.p)
13 y.size=x.size      //上述操作后，x.size未被改动，且改变子树的根节点不会导致节点数变化
14 x.size=x.left.size+x.right.size+1 //更新 x.size
```

右旋：

```
RIGHT-ROTATE(T,y)
1 x=y.left
2 y.left=x.right
3 if x.right≠T.nil
4       x.right.p=y     //1-4行首先令节点b成为y的左孩子(改动两个指向：y.l  以及  b.p)
5 x.p=y.p
6 if y.p==T.nil
7       root=x
8 elseif y==y.p.left
9       y.p.left=x
10 else y.p.right=x // 5-10行再令节点x代替y(改动两个指向：x.p  以及  y.p.left or y.p.right or root)
11 x.right=y
12 y.p=x              //11-12最后令y成为x的右孩子(改动两个指向：x.right  以及  y.p)
13 x.size=y.size    //上述操作后，y.size未被改动，且改变子树的根节点不会导致节点数变化
14 y.size=y.left.size+y.right.size+1 //更新 y.size
```

```
RB-INSERT(T,z)
y=T.nil
x=T.root
while x≠T.nil
    y=x
    y.size=y.size+1//新节点插入的路径上每一个父节点都需要将大小增加1
    if z.key<x.key
        x=x.left
    else x=x.right
z.p=y
if y==T.nil
    T.root=z
elseif z.key<y.key
    y.left=z
else y.right=z
z.left=T.nil
z.right=T.nil
z.colcor=RED
z.size=1//新插入的节点大小为1
RB-INSERT-FIXUP(T,z)
```

```
RB-DELETE(T,z)
1 y=z
2 y-original-color=y.color
3 if z.left==T.nil
4       x=z.right
5       RB-TRANSPLANT(T,z,z.right)
6 elseif z.right==T.nil
7       x=z.left
8       RB-TRANSPLANT(T,z,z.left)
9 else y=TREE-MINIMUM(z.right)
10      y-original-color=y.color
11      x=y.right
12      if y.p==z
13          x.p=y//使得x为哨兵节点时也成立
14      else RB-TRANSPLANT(T,y,y.right)//即使y.right是哨兵，也会指向y的父节点
15          y.right=z.right
16          y.right.p=y
17      RB-TRANSPLANT(T,z,y)
//17行运行之后，13、14行都会保证x指向原始y父节点的位置
18      y.left=z.left
19      y.left.p=y
20      y.color=z.color
21 p=x.p //由于x是挪到y原本位置的节点，因此x的属性未发生变动，x的所有父节点需要更新
22 while p≠T.nil
23       p.size=p.size-1
24       p=p.p
25 if y-original-color==BLACK
26      RB-DELETE-FIXUP(T,x)
```

Chapter 13. 动态规划 DP

13.1. 钢条切割问题

朴素递归

```
CUT-ROD(p,n)
1 if n==0
2     return 0
3 q=-∞
4 for i=1 to n    //i表示的是从左边切下的长度
5     q=max(q,p[i]+CUT-ROD(p,n-i))
6 return q
```

带备忘的自顶向下递归(因为需要有初始化的变量，因此需要两个函数!!!)

```
MEMOIZED-CUT-ROD(p,n)
1 let r[0...n] be a new array   //为什么要保存r[0]，见MEMOIZED-CUT-ROD-AUX第7行，会访问该元素
2 for i=0 to n
3     r[i]=-∞
4 return MEMOIZED-CUT-ROD-AUX(p,n,r)
```

```
MEMOIZED-CUT-ROD-AUX(p,n,r)
1 if r[n]⩾0//已存入备忘录，返回
2     return r[n]
3 if n==0//正常递归的返回
4     q=0
5 else q=-∞
6     for i=1 to n
7         q=max(q,p[i]+MEMOIZED-CUT-ROD-AUX(p,n-i,r)
8 r[n]=q
9 return q
```

自底向上非递归

```
BOTTOM-UP-CUT-ROD(p,n)
1 let r[0...n]be a new array //为什么要保存r[0]，见第6行，会访问r[0]
2 r[0]=0
3 for j=1 to n //按大小次序依次求解该问题以及其所有子问题
4     q=-∞
5     for i=1 to j //在求解子问题j之前，j的所有子问题必然已经求解出来
6         q=max(q,p[i]+r[j-i])//与CUT-ROD不同之处，直接访问结果而非递归调用
```

```
    r[j]=q
return r[n]
```

保存最优解的自底向上非递归

```
EXTENDED-BOTTOM-UP-CUT-ROD(p,n)
let r[0...n] and s[0...n] be new arrays
r[0]=0
for j=1 to n
    q=-∞
    for i=1 to j
        if q<p[i]+r[j-i]
            q=p[i]+r[j-i]
            s[j]=i//只保存第一段长度
    r[j]=q
return r and s
```

13. 2. 矩阵链相乘完全括号化问题

自底向上非递归法：

```
MATRIX-CHAIN-ORDER(p)
n=p.length-1//n为矩阵的个数
let m[1…n,1…n] and s[1…n-1,2…n] be new tables
for i=1 to n
    m[i,i]=0
for g=2 to n //依次计算长度为g的子链的最优括号化(不同的子链长度)
    for i=1 to n- g +1//i为该长为g的子链的起始索引(索引从1开始)(不同的起始位置)
        j=i+ g -1//长为g子链以i为起始索引时，终止索引为j
        m[i,j]=∞
        for k=i to j-1//子链[i,j]的分割点
            q=m[i,k]+m[k+1,j]+p_{i-1}p_kp_j//计算m[i,j]时，长度小于j-1+1的子链的最优括号化已求得
            if q<m[i,j]
                m[i,j]=q
                s[i,j]=k
return m and s
```

其中m[i,j]代表计算矩阵$A_{i…j}$所需标量乘法次数的最小值

其中s[i,j]代表满足$A_{i…j}$所需标量乘法次数的最小值时的分割点，故i≤s[i,j]<j

若矩阵为A1=30*35;　A2=35*15;　A3=15*5;　A4=5*10;　A5=10*20;　A6=20*25;

那么p=[30,35,15,5,10,20,25]　即A_i=p[i-1]*[i]

```
PRINT-OPTIMAL-PARENS(s,i,j)
if i==j
    print "A"_i
else print"("
    PRINT-OPTIMAL-PARENS(s,i,s[i,j])
    PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
    print")"
```

自带备忘的自顶向下法：

```
MATRIX-CHAIN-MEMOIZED(p)
int n=p.length-1
let m[1…n,1…n] and s[1…n-1,2…n]be new tables
for i=1 to n//该循环初始化，使得备忘录为特殊值
    for j=i to n
        m[i,j]=∞
MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,1,n)
return m and s
```

```
MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,i,j)
if m[i,j]<∞//若不为特殊值说明该情况已求得最优解
        return m[i,j]
if i==j
        m[i,j]=0
else for k=i to j-1
            q= MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,i,k)+
                MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,k+1,j)+ p_{i-1}p_kp_j
            if q<m[i,j]
                m[i,j]=q
                s[i,j]=k
return m[i,j]
```

自带备忘的自顶向下法的特点

1、需要两个函数，其中一个为递归函数

2、递归函数中，有3个返回点：备忘录中已有该问题的结果；平凡结果；非平凡结果

3、递归函数需要返回值：该问题的一个最优解

13.3. LCS-LENGTH

LCS(X,Y) longest common subsequence

```
m=X.length
n=Y.length
let b[1...m,1...n] and c[1...m,1...n]be new tables//c[i,j]表示 X1...i 与 Y1...i 的最长公共子序列的长度
//b[i,j]表示 c[i,j]分解成子问题的方式(存储即作出选择，该选择只有 3 种)
for i=1 to m
    c[i,0]=0
for j=0 to n
    c[0,j]=0
for i=1 to m
    for j=1 to n
        if x_i==y_j
            c[i,j]=c[i-1,j-1]+1
            b[i,j]= ↖
        elseif c[i-1,j]≥c[i,j-1]
            c[i,j]=c[i-1,j]
            b[i,j]= ↑
        else c[i,j]=c[i,j-1]
            b[i,j]= ←
return c and b
```

LMS-LENGTH(X) longest monotonous sunsequence

```
n=X.length
let c[1...n] b[1...n] be new tables //其中 c[i]表示以元素 X[i]结尾的最长单调子序列(子问题形式与原问题不同！ )
//b[i]表示以元素 X[i]结尾的最长单调子序列中第二大的元素的下标
for i=1 to n
    c[i]=1//至少为 1 嘛
for i=2 to n
    for j=1 to i-1
        if X[i]>X[j] and c[i]<c[j]+1
            c[i]=c[j]+1
            b[i]=j
```

13.4. OBST

```
OBST(p,q)
let e[1...n+1,0...n],w[1...n+1,0...n],and root[1...n,1...n]be new tables
//其中 p1...pn 代表关键字 k1...kn 的概率，q0...qn 代表伪关键字 d0...dn 的概率
//e[i,j]表示包含关键字 ki...kj 的子树的搜索期望代价，其中 e[i,i-1]=q[i-1]
//w[i,j]表示包含关键字 ki...kj 的子树的关键字以及伪关键字 di-1...dj 概率之和
//root[i,j]表示包含关键字 ki...kj 的子树的根节点
//包含关键字 ki...kj 的子树中必然包含伪关键字 di-1...dj
for i=1 to n+1
    e[i,i-1]=q[i-1];
    w[i,i-1]=q[i-1];
for L=1 to n //L 代表子树的长度
    for i=1 to n-L+1//i 代表长为 L 的子树的起始索引
        j=i+L-1//j 代表长为 L 的子树的终止索引
        e[i,j]=∞
        w[i,j]=w[i,j-1]+p[j]+q[j]
        for r=i to j//r 代表长为 L 的子树的分割点
            t=e[i,r-1]+e[r+1,j]+w[i,j]
            if t<e[i,j]
                e[i,j]=t
                root[i,j]=r
return e and root
```

```
PRINT_TREE(i,j,last)
cur=SUB_ROOT(i,j)//若 j<i 会返回 0
if i==1 and j==root.Length
    print("K"+cur+"为根")
elseif cur==0
    if j<last
        print("D"+j+"为"+"K"+last+"的左孩子)
    else print("D"+j+"为"+"K"+last+"的左孩子)
elseif index<last
    print("K"+index+"为"+"K"+last+"的左孩子)
    PRINT_TREE(i,index-1,index)
    PRINT_TREE(index+1,j,index)
else print("K"+index+"为"+"K"+last+"的右孩子)
    PRINT_TREE(i,index-1,index)
    PRINT_TREE(index+1,j,index)
```

```
SUB_ROOT(i,j)
if i<=j
    return root[i,j]
return 0 //当 i=1，j=0 时也能起作用，而 root[i,j]此时会异常
```

Chapter 14. 贪心算法

```
RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)
m=k+1
while m≤n and s[m]<f[k] //在 k 之后，n 之前的活动中，找到活动开始时间小于活动 k 结束时间的活动，由于活动结束时间已排序，因此第一个满足条件的活动一定是活动时间最早结束的活动
    m=m+1
if m≤n
    return {a_m}∪RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)
```

```
GREEDY-ACTIVITY-SELECTOR(s,f)
n=s.length
A={a_1} //由于活动按结束时间排序，第一个活动必定会选择
k=1
for m=2 to n
    if s[m]≥f[k]
        A=A∪{a_m}
        k=m
return A
```

14.1. 0-1 背包问题

MaxValue(p,w,V)

```
n=p.length
let dp[1...n][1...V] be a new array
for v=1 to V //初始化，对于不同的背包容量，对第一个商品的最大利益
    if w[1]<=v
        dp[1][v]=p[1] //装得下就装下
    else dp[1][v]=0 //装不下就舍弃
for i=2 to n
    for v=1 to V
        if v<w[i] //若大小为 v 的背包容量小于第 i 件商品的重量，那么第 i 件商品无法取得
            dp[i][v]=dp[i-1][v]
        else dp[i][v]=max(dp[i-1][v],dp[i-1][v-w[i]]+p[i])
return dp[n][V]
```

核心关系式：**dp[i][v]=max(dp[i-1][v],dp[i-1][v-w[i]]+p[i])**

子问题模式：dp[i][v]：对于前 i 个商品，给定背包容量 v 所能获取的最大收益(可以取可以不取)

14. 2. 哈夫曼编码

```
HUFFMAN(C)
n=|C|
Q=C//将 C 中的元素全部存入优先队列(优先队列用最小二叉堆实现)
for i=1 to n-1
    allocate a new node z
    z.left=x=EXTRACT-MIN(Q)//提取出优先队列中的第一项
    z.right=y=EXTRACT-MIN(Q) //提取出优先队列中的第一项
    z.freq=x.freq+y.freq
    INSERT(Q,z)
return EXTRACT-MIN(Q)
```

Q 是一个优先队列

Chapter 15. B 树

15.1. 定义

15.1.1. 节点

1、节点的属性

1) n：关键字个数
2) keys：关键字数组
3) children：孩子数组
4) isLeaf：是否为叶节点
5) color：颜色

2、每个节点具有以下性质

1) x.n：当前存储在节点 x 中的关键字个数
2) x.n 个关键字本身 $x.key_1, x.key_2, ..., x.key_{x.n}$，以非降序存放，使得

$$x.key_1 \le x.key_2 \le ... \le x.key_{x.n}$$

3) x.leaf：一个布尔值，如果 x 是叶节点，则为 TRUE，如果 x 为内部节点，则为 FALSE
4) 每个内部节点 x 还包含 x.n+1 个指向其孩子的指针，$x.c_1, x.c_2, ..., x.c_{x.n+1}$，叶节点没有孩子，所以他们的 c_i 属性没有定义
5) 关键字 $x.key_i$ 对存储在各子树中的关键字范围加以分割：如果 k_i 为任意一个存储在以 $x.c_i$ 为根的子树中的关键字，那么

$$k_1 \le x.key_1 \le k_2 \le x.key_2 \le ... \le x.key_{x.n} \le k_{x.n+1}$$

6) 每个叶节点都具有相同的深度，即树的高度 h
7) 每个节点所包含的关键字个数有上界和下界，用一个被称为 B 数的最小度数(minimum degree)的固定整数 t≥2 来表示这些界

- 除了根节点以外的每个节点必须至少有 t-1 个关键字，因此除了根节点以外的每个内部节点至少有 t 个孩子，如果树非空，根节点至少含有一个关键字
- 每个节点至多可包含 2t-1 个关键字，因此，一个内部节点最多可有 2t 个孩子，当一个节点恰好有 2t-1 个关键字时，称该节点是满的
- t=2 时的 B 数是最简单的，在实际中，t 的值越大，B 树的高度就越小

15.1.2. 树

1、属性

1) t：B 树的度
2) root：B 树的根节点

15. 2. 基本操作

对于节点 x ，关键字 $x.key_i$ 与子树指针 $x.c_i$ 的索引相同，就说 $x.c_i$ 是关键字 $x.key_i$ 对应的子树指针

子树 $x.c_i$ 的元素介于 $x.key_{i-1}$~$x.key_i$ 之间 $1 \le i \le x.n+1$，为保持一致性，记 $x.key_0= -\infty$，$x.key_{x.n+1}=+\infty$

```
B-TREE-SEARCH(x,k)
i=1
while i ≤ x.n and k > x.key_i
    i=i+1
if i ≤ x.n and k==x.key_i
    return (x,i)
elseif x.leaf
    return NIL
else DISK-READ(x,c_i)
    return B-TREE-SEARCH(s.c_i,k)
```

```
B-TREE-CREATE(T)
x=ALLOCATE-NODE()
x.leaf=TRUE
x.n=0
DISK-WRITE(x)
T.root=x
```

```
B-TREE-SPLIT-CHILD(x,i)//x.ci 是满节点，x 是非满节点
z=ALLOCATE-NODE()//z 是由 y 的一半分裂得到
y=x.c_i
z.leaf=y.leaf
z.n=t-1
for j=1 to t-1
    z.key_j=y.key_{j+t}//将 y 中[t+1…2t-1]总共 t-1 个关键字复制到节点 z 中作为[1…t-1]的关键字，其中第 t 个关键字会提取出来作为 x 节点的关键字
if not y.leaf//如果 y 不是叶节点，那么 y 还有 t 个指针需要复制到 z 中
    for j=1 to t
        z.c_j=y.c_{j+t}
y.n=t-1
for j=x.n+1 downto i+1//指针 y 和 z 必然是相邻的，并且他们所夹的关键字就是原来 y 中第 t 个
    x.c_{j+1}=x.c_j
x.c_{i+1}=z
for j=x.n downto i
    x.key_{j+1}=x.key_j
x.key_i=y.key_t
x.n=x.n+1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

15. 3. 插入

B-TREE-INSERT(T,k)

```
r=T.root
if r.n==2t-1 //需要处理根节点，若满了，则进行一次分裂，这是树增高的唯一方式
    s=ALLOCATE-NODE()//分配一个节点作为根节点
    T.root=s
    s.leaf=FLASE//显然由分裂生成的根必然是内部节点
    s.n=0
    s.c_1=r//之前的根节点作为新根节点的第一个孩子
    B-TREE-SPLIT-CHILD(s,1)
    B-TREE-INSERT-NONFULL(s,k)
else B-TREE-INSERT-NONFULL(r,k)
```

B-TREE-INSERT-NONFULL(x,k)

```
i=x.n
if x.leaf //如果是叶节点，保证是非满的，找到适当的位置插入即可
    while i ≥1 and k<x.key_i
        x.key_{i+1}=x.key_i
        i=i-1
    x.key_{i+1}=k
    x.n=x.n+1
    DISK-WRITE(x)
else while i ≥ 1 and k<x.key_i
        i=i-1
    i=i+1//转到对应的指针坐标
    DISK-READ(x.c_i)
    if x.c_i.n==2t-1
        B-TREE-SPLIT-CHILD(x,i)
        if k>x.key_i    //原来在 i 位置的关键字现在在 i+1 位置上，i 位置上是 y.keyt
            i=i+1
    B-TREE-INSERT-NONFULL(x.c_i,k)
```

从左往右遍历，第一个大于指定关键字的关键字的索引就是指针的索引
从右往左遍历，第一个小于指定关键字的关键字的索引+1 就是指针的索引

```
B-TREE-PRECURSOR(x,k)//得保证 k 必须存在于 B 树中
if !B-TREE-SEARCH(k) or k==B-TREE-MINIMUM(T.root)
throw error(no precursor)
B-TREE-PRECURSORAUX(T.root,k)
```

```
B-TREE-PRECURSORAUX(x,k)
i=1
if x.leaf//若为叶节点
    while i≤x.n and k>x.key_i ++i //找到第一个不小于 k 的关键字(大于或等于都可以)
    return x.key_{i-1}
else //若不为叶节点
    while i≤x.n and k>x.key_i ++i //找到第一个不小于 k 的关键字
    if k==x.key_i return B-TREE-MAXIMUM(x.c_i) //若这个关键字等于 k，那么在对应子树中找最大值
    if MINIMUM(x.c)≥k //如果该关键字对应的子树的最小值大于 k
        return x.k_{i-1} //那么前驱必然是当前节点中的前一个关键字
    return B-TREE-PRECURSORAUX(x.c_i,k)//否则在该关键字对应的子树中继续寻找
```

```
B-TREE-SEARCH-SUCCESSOR(x,k)
if !B-TREE-SEARCH(x,k) or k=B-TREE-MAXIMUM(T.root)
throw error (no successor)
B-TREE-SEARCH-SUCCESSORAUX(x,k)
```

```
B-TREE-SEARCH-SUCCESSORAUX(x,k)
i=x.n
if x.leaf
    while i≥1 and k<x.key_i --i
    return x.key_{i+1}
else
    while i≥1 and k<x.key_i --i
    if k==x.key_i return B-TREE-MINIMUMAUX(x.c_{i+1})
    if k≥B-TREE-MAXIMUM(x.c_{i+1})
        return x.key_{i+1}
    return B-TREE-SEARCH-SUCCESSORAUX(x.c_{i+1},k)
```

B-TREE-MINIMUM(x)

1 **if** x.leaf **return** x.key_1

2 **return** B-TREE-MIMIMUM(x.c_1)

B-TREE-MAXIMUM(x)

1 **if** x.leaf **return** x.$key_{x.n}$

2 **return** B-TREE-MAXIMUM(x.$c_{x.n+1}$)

15. 4. 删除

B-TREE-DELETE(T,k) **//以下都是 delete 会用到的函数**

```
r=T.root
if r.n==1
    DISK-READ(r.c_1)
    DISK-READ(r.c_2)
    y=r.c_1
    z=r.c_2
    if not r.leaf and y.n==z.n==t-1
        B-TREE-MERGE-CHILD(r,1,y,z)
        T.root=y
        FREE-NODE(r)
        B-TREE-DELETE-NOTNONE(y,k)
    else B-TREE-DELETE-NOTNONE(r,k)
else B-TREE-DELETE-NOTNONE(r,k)
```

B-TREE-MERGE(x,i,y,z)

```
y.n=2t-1
for j=t+1 to 2t-1
    y.key_j=z.key_{j-t}
y.key_t=x.key_i //the key from node x merge to node y as the tth key
if not y.leaf
    for j=t+1 to 2t
        y.c_j=z.c_{j-t}
for j=i+1 to x.n
    x.key_{j-1}=x.key_j
    x.c_j=x.c_{j+1}
x.n=x.n-1
Free(z)
```

B-TREE-SHIFT-TO-LEFT-CHILD(x,i,y,z)

```
y.n=y.n+1
y.key_{y.n}=x.key_i
x.key_i=z.key_1
z.n=z.n-1
j=1
while j≤z.n
    z.key_j=z.key_{j+1}
    j=j+1
if not z.leaf
    y.c_{y.n+1}=z.c_1
    j=1
    while j≤z.n+1
        z.c_j=z.c_{j+1}
        j++
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

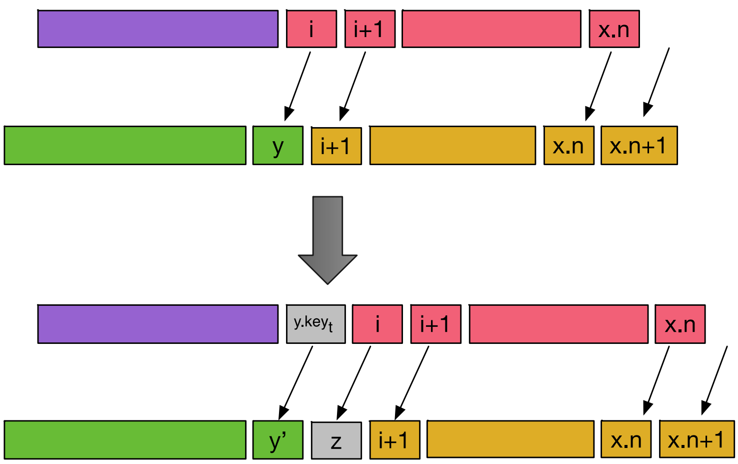
B-TREE-SHIFT-TO-RIGHT-CHILD(x,i,y,z)

```
z.n=z.n+1
j=z.n
while j>1
    z.key_j=z.key_{j-1}
    j--
z.key_1=x.key_i
x.key_i=y.key_{y.n}
if not z.leaf
    j=z.n
    while j>0
        z.c_{j+1}=z.c_j
        j--
    z.c_1=y.c_{y.n+1}
y.n=y.n-1
DISK-WRITE(y)
DISK-WRITE(z)
DISK-WRITE(x)
```

B-TREE-DELETE-NOTNONE(x,k)

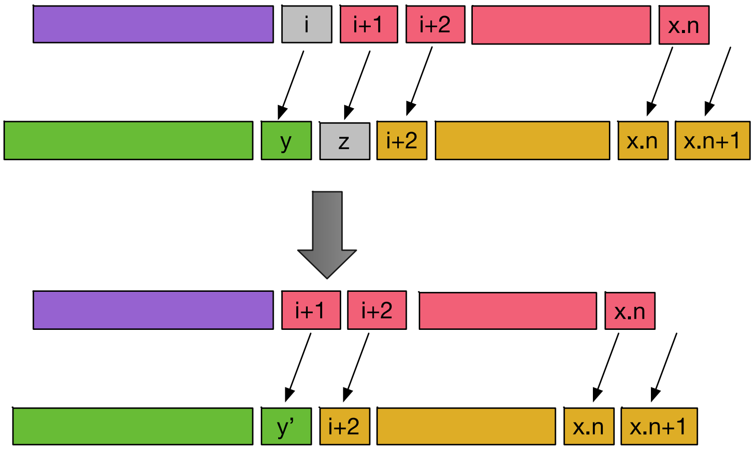
```
i=1
if x.leaf
    while i ≤ x.n and k>x.key_i
        i=i+1
    if k==x.key_i
        for j=i+1 to x.n
            x.key_{j-1}=x.key_j
        x.n=x.n-1
        DISK-WRITE(x)
    else error:"the key does not exist"
else while i ≤ x.n and k>x.key_i
        i=i+1
    DISK-READ(x.c_i)
    y=x.c_i
    if i ≤ x.n
        DISK-READ(x.c_{i+1})
        z=x.c_{i+1}
    if i ≤ x.n and k==x.key_i          //Cases 2
        if y.n>t-1       //Cases 2a
            k'=B-TREE-MIMIMUM(y)
            B-TREE-DELETE-NOTNONE(y,k')
            x.key_i=k'
        elseif z.n>t-1    //Case 2b
            k'=B-TREE-MAXIMUM(z)
            B-TREE-DELETE-NOTNONE(z,k')
            x.key_i=k'
        else B-TREE-MERGE-CHILD(x,i,y,z) //Cases 2c
            B-TREE-DELETE-NOTNONE(y,k)
    else      //Cases3
        if i>1
            DISK-READ(x.c_{i-1})
            p=x.c_{i-1}
        if y.n==t-1
            if i>1 and p.n>t-1    //Cases 3a
                B-TREE-SHIFT-TO-RIGHT-CHILD(x,i-1,p,y)
            elseif i ≤ x.n and z.n>t-1
                B-TREE-SHIFT-TO-LEFT-CHILD(x,i,y,z)
            elseif i>1    //Cases3b
                B-TREE-MERGE-CHILD(x,i-1,p,y)
                y=p
            else B-TREE-MERGE-CHILD(x,i,y,z) //Cases 3b
        B-TREE-DELETE-NOTNONE(y,k)
```

SPLID



i
i+1
x.n
y
i+1
x.n
x.n+1
y.key$_t$
i
i+1
x.n
y'
z
i+1
x.n
x.n+1

Merge



i
i+1
i+2
x.n
y
z
i+2
x.n
x.n+1
i+1
i+2
x.n
y'
i+2
x.n
x.n+1

Chapter 16. 斐波那契堆

16.1. 定义

16.1.1. 节点

1、key：节点的值
2、degree：节点的度???
3、left：左兄弟
4、right：右兄弟
5、p：父节点
6、child：第一个孩子节点
7、marked：是否被删除第一个孩子节点

16.2. 堆

1、n：堆节点的总数
2、min：最小节点，也就是斐波那契堆的根

16.3. 插入操作

```
FIB-HEAP-INSERT(H,x)
x.degree=0
x.p=NULL
x.child=NULL
x.mark=FLASE
if H.min==NULL
    create a root list for H containing just x
    H.min=x
else
    insert x into H's root list
    if x.key<H.min.key
        H.min=x
H.n=H.n+1
```

16.4. 堆合并

```
FIB-HEAP-UNION(H1,H2)
H=MAKE-FIB-HEAP
H.min=H1.min
concatenate the root list of H2 with the root list of H
if(H1.min==NULL) or (H2.min!=NULL and H2.min.key<H1.min.key)
        H.min=H2.min
H.n=H1.n+H2.n
return H
```

16. 5. 抽取最小节点

```
FIB-HEAP-EXTRACT-MIN(H)
z=H.min
if z!=NULL
    for each child x of z
        add x to the root list of H
        x.p=NULL
    remove z from the root list of H
    if z==z.right
        H.min=NULL
    else
        H.min=z.right
        CONSOLIDATE(H)
    H.n=H.n-1
return z
```

16. 6. 抽取最小节点时维护斐波那契堆的性质

```
CONSOLIDATE(H)
let A[0...D(H.n)] be a new array
for i=0 to D(H.n)
    A[i]=NULL
for each node w in the root list of H
    x=w
    d=x.degree
    while A[d]!=NULL
        y=A[d]
        if x.key>y.key
            exchange(x,y)
        FIB-HEAP-LINK(H,y,x)
        A[d]=NULL
        d=d+1
    A[d]=x
H.min=NULL
for i=0 to D(H.n)
    if A[i]!=NULL
        if H.min==NULL
            create a root list for H containing just A[i]
            H.min=A[i]
        else
            insert A[i] into H's root list
            if A[i].key<H.min.key
                H.min=A[i]
```

```
FIB-HEAP-LINK(H,y,x)
remove y from the root list of H
make y a child of x, incrementing x.degree
y.mark=FLASE
```

16. 7. 关键字减值和删除节点

```
FIB-HEAP-DECREASE-KEY(H,x,k)
if k>x.key
    error "new key is greater than current key"
x.key=k
y=x.p
if y!=NULL and x.key<y.key
    CUT(H,x,y)
    CASCADING-CUT(H,y)
if x.key<H.min.key
    H.min=x
```

```
CUT(H,x,y)
remove x from the child list of y, decrementing y.degree
add x to the root list of H
x.p=NIL
x.mark=FALSE
```

```
CASCADING-CUT(H,y)
z=y.p
if z!=NULL
    if y.mark==FALSE
        y.mark=TRUE
    else
        CUT(H,y,z)
        CASCADING-CUT(H,z)
```

Chapter 17. 基本图算法

17.1. BFS

BFS(G,s)

```
for each vertex u∈G.V-{s}
    u.color=WHITE
    u.d=+∞
    u. π=NIL
s.color=GRAY
s.d=0
s. π=NIL
let queue be a new Queue
queue.offer(s)
while not queue.isEmpty()
    u=queue.poll()
    for each v∈ G.Adj[u]
        if v.color==WHITE
            v.color=GRAY
            v.d=u.d+1
            v. π=u
            queue.offer(v)
    u.color=BLACK
```

17.2. DFS

DFS(G)

```
for each vertex u∈G.V
    u.color=WHITE
    u. π=NIL
time=0
for each vertex u∈G.V
    if u.color==WHITE
        DFS-VISIT(G,u)
```

DFS-VISIT(G,u)

```
time=time+1
u.d=time
u.color=GRAY
for each v∈G:Adj[u]
    if v.color==WHITE
    v. π=u
    DFS-VISIT(G,v)
u.color=BLACK
time=time+1
u.f=time
```

Chapter 18. 字符串匹配

18.1. KMP 算法

KMP-MATCHER(T,P)

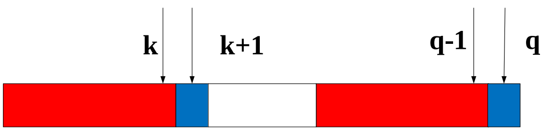
```
n=T.length
m=P.length
π=COMPUTE-PREFIX-FUNCTION(P)
k=0
for q=1 to n
    while k>0 and P[k+1]≠T[q]
        k=π[k]
    if P[k+1]==T[q]
        k=k+1
    if k==m
        print "Pattern occurs with shift" q-m
        k=π[k]
```

COMPUTE-PREFIX-FUNCTION(P)

```
1 m=P.length
2 let π[1...m] be a new array
3 π[1]=0
4 k=0
5 for q=2 to m
6     while k>0 and P[k+1]≠P[q]//若当前字符 q 与第 k+1 个不匹配，需要调整 k
7         k=π[k]
8     if P[k+1]==P[q]//如果
9         k=k+1
10    π[q]=k
11 return π
```

line 6：while 循环开始前，k 代表的是前一个 q 所对应的模式子串 P[1...q-1]的最大前后缀长度
即 k=π[q-1]

①若 k>0 也就是红色部分不为空，且 P[k+1]==P[q],那么 π[q]就等于 π[q-1]+1



②若 k>0 也就是红色部分不为空，且 P[k+1]≠P[q],那么 π[q]，那么需要在 P[1...k]中寻找是否存在包含 P[q]的最长前后缀，因此递归找出 P[1...k]的最大前后缀长度 π[k]，再看 P[k+1]是否与 P[q]相等