Chapter 2 插入、归并排序

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
```

归并排序（利用绝对索引）

```
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=
    MERGE-SORT(A,p,q)
    MERGE-SORT(A,q+1,r)
    MERGE(A,p,q,r)
```

Chapter 4 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=
    (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 6 堆排序

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)
```

构造最大堆

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

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

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 7 快速排序

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)
```

关键字存在重复时候的版本 1

```
PARTITION_REPEAT1(A,p,r)
x=A[r]
i=p-1      //小于 x 的最大索引
cnt=r-1    //以最后一个元素为主元，非主元的最大索引
for j=p to cnt
    if A[j]<x
        i=i+1
        EXCHANGE A[i] with A[j]
```

```
  elseif A[j]==x
        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 位置上已经是与 x 相同的数，因此循环边界递减
rn=r-cnt
for j=0 to rn-1
   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

```
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

关键字存在重复时候的版本 2

```
PARTITION_REPEAT2(A,p,r)
x=A[r]
i=p-1    //小于 x 的最大索引
k=p-1    //等于 x 的最大索引(不算最后一个)
for j=p to r-1
    if A[j]<x
        i=i+1
//A[j]位置的原值放在了 i 位置上,而 i 位置上的值 value（若 k≠i，说明存在于 x 相等的值，且 value=x，则 value 应该去 k+1 位置）
        EXCHANGE A[i] with A[j]
        k=k+1
        if i≠k//说明被换到 j 位置上的值与 x 相同，应该被至于[i+1,k]的区域
            EXCHANGE A[j] with A[k]
    elseif A[j]==x
        k=k+1
        EXCHANGE A[j] with A[k]
EXCHANGE A[k+1] with A[r]//此时与 x 相同的区域从[i+1,k]变为[i+1,k+1]
return i+1 and k+1
```

i+1 是与 x 值相同的区间内的开始，k+1 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,k+1]区间内的元素等于 x

[k+2,r]的元素大于 x

Chapter 8 线性时间排序

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
```

基数排序

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

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[]
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
```

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 9 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 10 基本数据结构

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）

```
PRESTACK(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
```

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

```
MIDSTACK(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
```

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
    1、**N1 的入栈计数为 2，访问该元素**
    2、**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）**

```
AFTERSTACK(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
```

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

1、 **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**

2、 **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

```
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)
```

树的前序遍历的非递归非栈算法：

```
Pre(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
    visit(cur)    //访问当前节点
    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
    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
    continue
  else//上一节点是当前节点的右孩子
    pre=cur
    cur=cur.p
    continue
```

访问出现在左孩子判断前

树的中序遍历的非递归非栈算法：

```
Mid(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
    visit(cur)   //访问当前节点
    if cur.right≠NULL
      pre=cur
      cur=cur.right
      continue
    pre=cur
    cur=cur.p
    continue
  elseif pre==cur.left//上一节点是当前节点的左孩子
    visit(cur)   //访问当前节点
    if cur.right≠Null
      pre=cur
      cur=cur.right
      continue
    pre=cur
    cur=cur.p
    continue
  else//上一节点是当前节点的右孩子
    pre=cur
    cur=cur.p
    continue
```

访问出现在右孩子判断前

树的后序遍历的非递归非栈算法：

```
After(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
    visit(cur)   //访问当前节点
    pre=cur
    cur=cur.p
    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
    visit(cur)   //访问当前节点
    pre=cur
    cur=cur.p
    continue
  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
    continue
```

访问出现在返回父节点之前

数的析构：

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)
```

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
```

数的遍历总结：

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 12 二叉搜索树

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
```

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
```

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

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)
1 if u.p==T.nil
2    T.root=v
3 elseif u==u.p.left
4    u.p.left=v
5 else u.p.right=v
6 if v≠T.nil
7    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 13 红黑树

- 每个节点或是红色的，或是黑色的
- 根节点是黑色的
- 每个叶节点(nil)是黑色的
- 如果一个节点是红色的，则它的两个子节点都是黑色的
- 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

y 右旋 **x**

x γ α **y**

α β 左旋 β γ

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

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)**

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

```
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：

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
```

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

```
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
```

AVLTree:（每个节点的左右子树高度差最多为 1）

每个节点记录一个额外属性：该节点的高度

该节点的高度最多比子节点大 2

当一个节点比其子节点大 3 时，需要通过左右旋来调整

①当 x 右子树比左子树的高度大 2 时，左旋（除 x 外，x 为根节点的其余节点均满足 AVLTree 性质）

x_0 y_1

α y_0 x_1 γ

β γ α β

由于 $h_{y0}=h_\alpha+2$,因此 $\max(h_\beta)= h_\alpha+1$ $\min(h_\beta)= h_\alpha$（且 β，γ 的高度至少有一个为 $h_\alpha+1$）

讨论左旋后 X 节点：$h_{x0}= h_\alpha+3$

㈠当 $h_\beta= h_\alpha+1$ 且 $h_\gamma= h_\alpha+1$：$h_{x1}= h_\alpha+2$，$h_{y1}= h_\alpha+3$**（与原 x 高相同）**

㈡当 $h_\beta= h_\alpha+1$ 且 $h_\gamma= h_\alpha$：$h_{x1}= h_\alpha+2$：y_1 又违反了性质（单独讨论）

㈢当 $h_\beta= h_\alpha$ 时：此时 $h_\gamma= h_\alpha+1$，$h_{x1}= h_\alpha+1$，$h_{y1}= h_\alpha+2$**（与原 x 高不同，需要继续向上维护性质）**

②当 y 的左子树比右子树高度大 2 时，右旋（除 y 外，y 为根节点的其余节点均满足 AVLTree 性质）

y_0 x_1

x_0 γ α y_1

α β β γ

由于 $h_{x0}=h_\gamma+2$,因此 $\max(h_\beta)= h_\gamma+1$ $\min(h_\beta)= h_\gamma$（且 α，β 的高度至少有一个为 $h_\gamma+1$）

讨论右旋后 Y 节点：$h_{y0}= h_\gamma+3$

㈠当 $h_\beta= h_\gamma+1$ 且 $h_\alpha= h_\gamma+1$：$h_{y1}= h_\gamma+2$，$h_{x1}= h_\gamma+3$**（与原 y 高相同）**

㈡当 $h_\beta= h_\gamma+1$ 且 $h_\alpha= h_\gamma$：$h_{y1}= h_\gamma+2$，x_1 又违反了性质（单独讨论）

㈢当 $h_\beta= h_\gamma$ 时：此时 $h_\alpha=h_\gamma+1$，$h_{y1}= h_\gamma+1$，$h_{x1}= h_\gamma+2$**（与原 y 高不同，需要继续向上维护性质）**

对于情况㈡产生原因的分析:

首先，AVL 数据结构中：

需要对一个节点进行左旋，那么必然该节点的右子树比左子树高 1 或 2

需要对一个节点进行右旋，那么必然该节点的左子树比右子树高 1 或 2

对于左旋，$H(y_0)=H(\alpha)+\varepsilon$，其中 $\varepsilon=1$ or $\varepsilon=2$。无论 ε 取值如何，**若 $H(\beta)>H(\gamma)$,那么旋转后势必破坏 y_1 节点的性质；若 $H(\beta)\leqslant H(\gamma)$，那么旋转后不会破坏 y_1 节点的性质。**

x_0 y_1

α y_0 x_1 γ

β γ α β

同理，对于右旋 $H(x_0)=H(\gamma)+\varepsilon$，其中 $\varepsilon=1$ or $\varepsilon=2$。无论 ε 取值如何，若 **$H(\beta)>H(\alpha)$，那么旋转后势必破坏 x_1 节点的性质；若 $H(\beta)\leqslant H(\alpha)$，那么旋转后不会破坏 x_1 节点的性质。**

y_0 x_1

x_0 γ α y_1

α β β γ

```
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）

```
1 if u.p==T.nil
2     T.root=v
3 elseif u==u.p.left
4     u.p.left=v
5 else u.p.right=v
6     v.p=u.p
```

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
```

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 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
```

HoldRotate(T,x,Type)

```
let S1,S2 be two STACKs sized T.size //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
S1.PUSH(x)
S2.PUSH(Type)
cur=Nil
CurRotateTop=Nil //对 x 尝试旋转后，返回最终旋转后的根节点
curType=-1;
while(!S1.Empty())
   cur=S1.TOP()
   curType=S2.TOP()
   if curType==1 //需要对 cur 尝试进行左旋
      if cur->right->right->height≥cur->right->left->height
         S1.POP()  S2.POP()
         CurRotateTop=LeftRotate(T,cur)
      else S1.PUSH(cur->right)//否则 cur 右孩子需要尝试进行右旋来调整
         S2.PUSH(2);
   elseif curType==2//需要对 cur 尝试进行右旋
      if cur->left->left->height≥cur->left->right->height
         S1.POP()  S2.POP()
         CurRotateTop=RightRotate(T,cur)
      else S1.PUSH(cur->left) //否则 cur 左孩子需要尝试进行左旋来调整
         S2.PUSH(1)
```

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= HoldRotate (T,y,2)
elseif y.right.height=y.left.height+2
   y= HoldRotate (T,y,1)
y=y.p
```

```
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 14 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)
1 y=T.nil
2 x=T.root
3 while x≠T.nil
4    y=x
5    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
6    if z.key<x.key
7       x=x.left
8    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 15 动态规划 DP

朴素递归

```
CUT-ROD(p,n)
if n==0
    return 0
q=-∞
for i=1 to n  //i 表示的是从左边切下的长度
    q=max(q,p[i]+CUT-ROD(p,n-i))
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
7     r[j]=q
8 return r[n]
```

保存最优解的自底向上非递归

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

矩阵链相乘完全括号化问题：

自底向上非递归法：

```
MATRIX-CHAIN-ORDER(p)
1 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 $A_{i\ldots j}$ 所需标量乘法次数的最小值

其中 **s[i,j]**代表满足 $A_{i\ldots j}$ 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

LCS-LENGTH(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
```

```
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 16 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
```

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）

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 18 B 树

1、定义

- 每个节点具有以下性质
  - x.n：当前存储在节点 x 中的关键字个数
  - x.n 个关键字本身 $x.key_1, x.key_2, \ldots, x.key_{x.n}$，以非降序存放，使得

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

  - 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
- 每个内部节点 x 还包含 x.n+1 个指向其孩子的指针，$x.c_1, x.c_2, \ldots, x.c_{x.n+1}$，叶节点没有孩子，所以他们的 c_i 属性没有定义
- 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

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

- 每个叶节点都具有相同的深度，即树的高度 h
- 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

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)
```

```
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)

```
if x.leaf return x.key_1
return B-TREE-MIMIMUM(x.c_1)
```

B-TREE-MAXIMUM(x)

```
if x.leaf return x.key_{x.n}
return B-TREE-MAXIMUM(x.c_{x.n+1})
```

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)
```

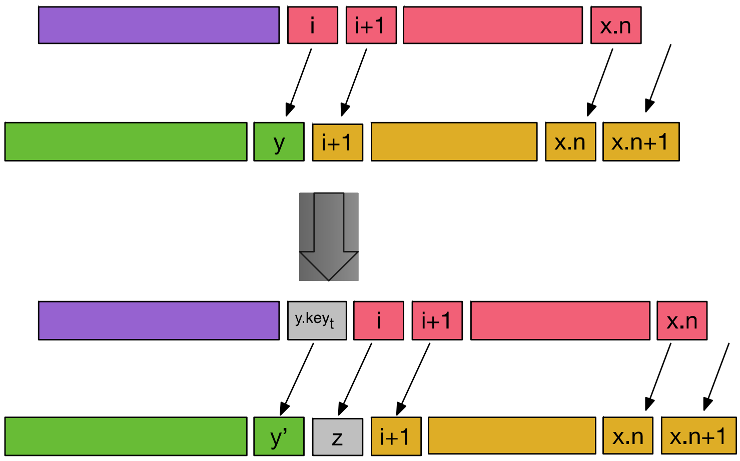
B-TREE-SHIFT-TO-RIGHT-CHILD(x,i,y,z)

```
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)
```

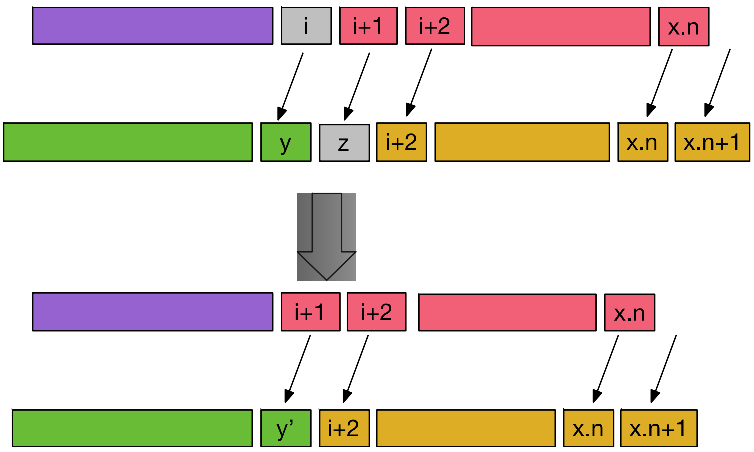
B-TREE-DELETE-NOTNONE(x,k)

```
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 22 基本图算法

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
```

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 32 字符串匹配

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