# 插入、归并排序

## 插入排序(稳定，原址)

INSERTION-SORT(A)

1 **for** j=2 **to** A.length

2 key=A[j]

3 **//Insert A[j] into the sorted sequence A[1…j-1]**

4 i=j-1

5 **while** i>0 and A[i]>key

6 A[i+1]=A[i]

7 i=i-1

8 A[i+1]=key

## 归并排序

MERGE(A,p,q,r)

1 n1=q-p+1

2 n2=r-q

3 let L[1…n1+1] and R[1…n2+1] be new arrays

4 **for** i=1 **to** n1

5 L[i]=A[p+i-1]

6 **for** j=1 **to** n2

7 R[j]=A[q+j]

8 L[n1+1]=∞

9 R[n2+1]=∞

10 i=1

11 j=1

12 **for** k=p **to** r

13 **if** L[i]≤R[j]

14 A[k]=L[i]

15 i=i+1

16 **else** A[k]=R[j]

17 j=j+1

MERGE-SORT(A,p,r)(稳定)

1 **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**

2 q=

3 MERGE-SORT(A,p,q)

4 MERGE-SORT(A,q+1,r)

5 MERGE(A,p,q,r)

# 最大和子数组

FIND-MAX-CROSING-SUBARRAY(A,low,mid,high)**//求包含mid的最大字数组**

1 left-sum=-∞**//mid左侧最大值，包括mid**

2 sum=0

3 **for** i=mid **downto** low//这里从mid算起，因此max-left最大为mid

4 sum=sum+A[i]

5 **if** sum>left-sum

6 left-sum=sum

7 max-left=i

8 right-sum=-∞**//mid 右侧最大值**

9 sum=0

10 **for** j=mid+1 **to** high //这里从mid+1算起，因此max-right最小为mid+1

11 sum=sum+A[j]

12 **if** sum>right-sum

13 right-sum=sum

14 max-right=j

15 **return** (max-left,max-right,left-sum+right-sum)

FIND-MAXIMUM-SUBARRAY(A,low,high)

1 **if** high==low**//递归终止**

2 **return**(low,high,A[low])

3 **else** mid=

4 (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**

5 (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**

6 (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**

7 **if** left-sum≥right-sum and left-sum≥cross-sum

8 **return** (left-low,left-high,left-sum)

9 **elseif** right-sum≥left-sum and right-sum≥cross-sum

10 **return**(right-low,right-high,right-sum)

11 **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]**

# 堆排序

## 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)

1 l=LEFT(i)

2 r=RIGHT(i)

3 **if** l≤A.heap-size and A[l]>A[i]

4 largest=l

5 **else** largest=i

6 **if** r≤A.heap-size and A[r]>A[largest]

7 largest=r

8 **if** largest≠i

9 exchange A[i] with A[largest]

10 MAX-HEAPIFY(A,largest)

## 构造最大堆

BUILD-MAX

1 A.heap-size=A.length**//这句什么用?**

2 **for** i= **downto** 1

3 MAX-HEAPIFY(A,i)

## 堆排序(非原址，非稳定)

HEAPSORT(A)

1 BUILD-MAX-HEAP(A)

2 **for** i=A.length **downto** 2

3 exchange A[1] with A[i]

4 A.heap-size=A.heap-size-1

5 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)

1 **return** A[1]

HEAP-EXTRACT-MAX(A)

1 **if** A.*heap-size*<1

2 error”heap underflow”

3 max=A[1]

4 A[1]=A[A.heap-size]//将最后一个数放置到第一个

5 A.*heap-size*=A.*heap-size*-1//减少堆的维度

6 MAX-HEAPIFY(A,1)//维护堆的性质

7 **return** max

HEAP-INCREASE-KEY(A,i,key)

1 **if** key<A[i]

2 error”new key is smaller than current key”

3 A[i]=key

4 **while** i>1 **and** A[PARENT(i)]<A[i]

5 exchange A[i] with A[PARENT(i)]

6 i=PARENT(i)

MAX-HEAP-INSERT(A,key)

1 A.heap-size=A.heap-size+1

2 A[A.heap-size]=-∞

3 HEAP-INCREASE-KEY(A,A.heap-size,key)

# 快速排序

## 快速排序(原址，非稳定)

QUICKSORT(A,p,r)

1 **if** p<r

2 q=PARTITION(A,p,r)

3 QUICKSORT(A,p,q-1)

4 QUICKSORT(A,q+1,r)

**PARTITION(A,p,r)//其实p=r的情况下也能运行**

1 x=A[r]

2 i=p-1

3 **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**

4 **if** A[j]≤x

5 i=i+1

6 exchange A[i] with A[j]

7 exchange A[i+1] with A[r]

8 **return** i+1

PARTITION的随机化版本

RANDOMIZED-PARTITION(A,p,r)

1 i=RANDOM(p,r)

2 exchang A[r] with A[i]**//必须将该值置于最后，才能调用PARTITION**

3 **return** PARTITION(A,p,r)

## 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)

1 x=A[r]

2 i=p-1 **//小于x的最大索引**

3 boundary=r-1  **//以最后一个元素为主元，非主元的最大索引**

4 **for** j=p **to** boundary

5 **if** A[j]<x

6 i=i+1

7 EXCHANGE A[i] with A[j]

8 **elseif** A[j]==x

9 EXCHANGE A[j] with A[boundary]  **//将于x相同的值先放到最后**

10 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)**

11 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**

12 n=r- boundary

13 **for** j=0 **to** n-1

14 EXCHANGE A[i+1+j] with A[r-j]

15 **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)

1 i=p-1

2 cnt=r **//非主元M的最大索引**

3 **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**

4 **if** A[j]<M

5 i=i+1

6 EXCHANGE A[i] with A[j]

7 **elseif** A[j]==M **//关键：将于M相同的值暂时放到A的最后边**

8 EXCHANGE A[j] with A[cnt] **//将于x相同的值先放到最后**

9 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)**

10 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**

11 rn=r-cnt

12 **for** j=0 **to** rn-1 //将等于M的区间挪到中间

13 EXCHANGE A[i+1+j] with A[r-j]

14 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

## 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)

1 x=A[r]

2 i1=p-1 **//小于x的最大索引**

3 i2=p-1 **//等于x的最大索引(算最后一个)**

4 **for** j=p **to** r-1

5 **if** A[j]<x

6 i1=i1+1

7 EXCHANGE A[i1] with A[j]

8 i2=i2+1

9 if i1≠i2

10 EXCHANGE A[i2] with A[j]

11 **elseif** A[j]==x

12 i2=i2+1

13 EXCHANGE A[i2] with A[k]

14 i2=i2+1

15 EXCHANGE A[i2] with A[r]

15 **return** i1+1 and i2

# 线性时间排序

## 计数排序(稳定，非原址)

COUNTING-SORT(A,B,k)

1 let C[0…k] be a new array

2 **for** i=0 **to** k

3 C[i]=0

4 **for** j=1 **to** A.length

5 C[A[j]]=C[A[j]]+1

6 //C[i] now contains the number of elements equal to i

7 **for** i=1 **to** k

8 C[i]=C[i]+C[i-1]

9 //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**

10 **for** j=A.length **down** to 1

11 B[C[A[j]]]=A[j]

12 C[A[j]]=C[A[j]]-1

## 基数排序

RADIX-SORT(A,d)

1 for i=1 to d

2 use a stable sort to sort array A on digit i

## 桶排序

BUCKET-SORT(A)

1 n=A.length

2 let B[0…n-1] be a new array

3 **for** i=0 **to** n-1

4 make B[i] an empty list

5 **for** i=1 **to** n

6 insert A[i] into list B[]

7 **for** i=0 **to** n-1

8 sort list B[i] with insertion sort

9 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)

1 if A[i]>A[j]

2 exchange A[i] with A[j]

INSERTION-SORT(A)

1 for j=2 to A.length**//循环不变式：A[1…j-1]是已排序的序列**

2 for i=j-1 down to 1

3 COMPARE-EXCHANGE(A,i,i+1)

# 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**

1 **if** p==r**//只有一个元素时，退出**

2 **return** A[p]

3 q=RANDOMIZED\_PARTITION(A,p,r);

4 k=q-p+1

5 **if** k==i**//若q就是要找的下标**

6 **return** A[**q**] **~~//别写成了A[i]~~**

7 **elseif** i<k

8 **return** RANDOMIZED\_SELECT(A,p,q-1,i)

9 **else** **return** RANDOMIZED\_SELECT(A,q+1,r,i-k)

# 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)**

PRE-ORDER-STACK(T)

1 let S be a STACK sized T.size

2 cur=T.root

3 **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**

4 **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**

5 visit(cur)

6 S.PUSH(cur)

7 cur=cur.left

8 **if** S.empty==Flase

9 cur=S.POP

10 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**

IN-ORDER-STACK(T)

1 let S be a STACK sized T.size

2 cur=T.root

3 **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**

4 **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**

6 S.PUSH(cur)

7 cur=cur.left

8 **if** S.empty==Flase

9 cur=S.POP

9 visit(cur)

10 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**

* 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)

1 let S be a STACK sized T.size

2 let every Node’s cnt be zero

3 cur=T.root

4 **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**

5 **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**

6 cur.cnt=cur.cnt+1

7 S.PUSH(cur)

8 cur=cur.left

9 **if** S.empty==Flase

10 cur=S.POP

11 **if** cur.cnt==2

12 visit(cur)

13 cur=NULL **//保证下一次循环直接跳过内层的while**

14 **else** cur.cnt=cur.cnt+1

15 S.PUSH(cur)

16 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**

* **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**

1. **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

1 let S be a STACK sized T.size

2 S.PUSH(T.root)

3 pre=cur=NULL

3 **while** S.empty==False**//栈不为空时进入循环**

4 cur=S.TOP**//获取栈顶元素(非弹出)**

5 **if** cur.left==NULL and cur.right==NULL **or** pre≠NULL **and** pre.p=cur

6 visit(cur)

6 S.POP**//弹出该元素**

7 pre=cur

14 **else\_if** cur.right≠NULL

15 S.PUSH(cur.right)

16 **if** cur.left≠NULL

17 S.PUSH(cur.left)

## 二叉树的前序遍历的非递归非栈算法

POST-ORDER-ELSE

1 pre=NULL**//前一节点初始化为空**

2 cur=T.root**//当前节点初始化为根节点**

3 **while**(cur≠NULL)

4 **if** pre==cur.p **//当前节点是上一节点的子节点**

5 visit(cur) **//访问当前节点**

6 pre=cur

7 **if** cur.left≠NULL

8 cur=cur.left

9 **elseif** cur.right≠NULL

10 cur=cur.right

11 **else**

12 cur=cur.p

13 **elseif** pre==cur.left**//上一节点是当前节点的左孩子**

14 pre=cur

15 **if** cur.right≠NULL

16 cur=cur.right

17 **else**

18 cur=cur.p

19 **else//上一节点是当前节点的右孩子**

20 pre=cur

21 cur=cur.p

**访问出现在左孩子判断前**

## 二叉树的中序遍历的非递归非栈算法

IN-ORDER-ELSE

1 pre=NULL**//前一节点初始化为空**

2 cur=T.root**//当前节点初始化为根节点**

3 **while**(cur≠NULL)

4 **if** pre==cur.p **//当前节点是上一节点的子节点**

5 pre=cur

6 **if** cur.left≠NULL

7 cur=cur.left

8 **elseif** cur.right≠NULL

9 visit(cur) **//访问当前节点**

10 cur=cur.right

11 **else**

12 visit(cur) **//访问当前节点**

13 cur=cur.p

14 **elseif** pre==cur.left**//上一节点是当前节点的左孩子**

15 pre=cur

16visit(cur) **//访问当前节点**

17 **if** cur.right≠NULL

18 cur=cur.right

19 **else**

20 cur=cur.p

21 **else//上一节点是当前节点的右孩子**

22 pre=cur

23 cur=cur.p

**访问出现在右孩子判断前**

## 二叉树的后序遍历的非递归非栈算法

POST-ORDER-ELSE

1 pre=NULL**//前一节点初始化为空**

2 cur=T.root**//当前节点初始化为根节点**

3 **while**(cur≠NULL)

4 **if** pre==cur.p **//当前节点是上一节点的子节点**

5 pre=cur

6 **if** cur.left≠NULL

7 cur=cur.left

8 **elseif** cur.right≠NULL

9 cur=cur.right

10 **else**

11 visit(cur) **//访问当前节点**

12 cur=cur.p

13 **elseif** pre==cur.left**//上一节点是当前节点的左孩子**

14 pre=cur

15 **if** cur.right≠NULL

16 cur=cur.right

17 **else**

18 visit(cur) **//访问当前节点**

19 cur=cur.p

20 **else//上一节点是当前节点的右孩子**

21 visit(cur) **//访问当前节点**

22 pre=cur

23 cur=cur.p

**访问出现在返回父节点之前**

## 二叉树树的析构：

### 通过后续遍历的栈算法2的变形来实现

~TREE(T)

1 let S be a STACK sized T.size

2 pre=cur=NULL

3 S.PUSH(T.root)

4 **while** S.empty==False**//栈不为空时进入循环**

5 cur=S.TOP**//获取栈顶元素(非弹出)**

**6 if cur.left==NULL and cur.right==NULL //与后续遍历的不同之处**

7 S.POP**//弹出该元素**

8 pre=cur

9 cur=cur.p

10 **if** cur≠NULL

11 **if** pre==cur.left

12 cur.left==NULL

13 **else** cur.right=NULL

14 delete pre**//释放被弹出的栈顶元素的内存**

15 **else\_if** cur.right≠NULL

16 S.PUSH(cur.right)

17 **if** cur.left≠NULL

18 S.PUSH(cur.left)

### 通过指针路径算法的变形来实现

~TREE(T)

1 pre=NULL**//前一节点初始化为空**

2 cur=T.root**//当前节点初始化为根节点**

3 **while**(true)

4 **if** cur==NULL

5 break **//当前节点为空时，退出循环**

6 **if** pre==cur.p **//当前节点是上一节点的子节点**

7 **if** cur.left≠NULL

8 pre=cur

9 cur=cur.left

10 continue

11 **if** cur.right≠NULL

12 pre=cur

13 cur=cur.right

14 continue

15 pre=cur

16 cur=cur.p

17 **if** cur≠NULL and pre==cur.left

18 i=1

19 **elseif** cur≠NULL and pre==cur.right

20 i=2

21 delete pre continue

22 **elseif** pre==cur.left**//上一节点是当前节点的左孩子**

23 **if** cur.right≠Null

24 pre=cur

25 cur=cur.right

26 continue

27 pre=cur

28 cur=cur.p

29 **if** cur≠NULL and pre==cur.left

30 i=1

31 **elseif** cur≠NULL and pre==cur.right

32 i=2

33 delete pre continue

34 **elseif** pre==cur.right**//上一节点是当前节点的右孩子**

35 pre=cur

36 cur=cur.p

37 **if** cur≠NULL and pre==cur.left

38 i=1

39 **elseif** cur≠NULL and pre==cur.right

40 i=2

41 delete pre continue

42 else switch(i)

43 case 1: **if** cur.right≠Null

44 pre=cur

45 cur=cur.right

46 continue

47 pre=cur

48 cur=cur.p

49 **if** cur≠NULL and pre==cur.left

50 i=1

51 **elseif** cur≠NULL and pre==cur.right

52 i=2

53 delete pre continue

54 case 2: pre=cur

55 cur=cur.p

56 **if** cur≠NULL and pre==cur.left

57 i=1

58 **elseif** cur≠NULL and pre==cur.right

59 i=2

60 delete pre continue

## 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**

# 二叉搜索树

## 插入二叉搜索树

TREE\_INSERT(T,z)

1 y=T.nil

2 x=T.root

3 **while** x≠T.nil**//循环结束时x指向空，y指向上一个x**

4 y=x

5 **if** z.key<x.key

6 x=x.left

7 **else** x=x.right

8 z.p=y**//将这个叶节点作为z的父节点**

9 **if** y==T.nil

10 T.root=z

11 **elseif** z.key<y.key

12 y.left=z

13 **else** y.right=z

TREE-SEARCH(x,k) x指向根节点

1 **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**

2 **if** k<x.key

3 x=x.left

4 **else** x=x.right

5 **return** x

以x为根节点的子树的最大值

TREE-MAXIMUM(x)

1 **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**

2 x=x.right

3 **return** x

以x为根节点的子树的最小值

TREE-MINIMUM(x)

1 **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**

2 x=x.left

3 **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)

1 **if** x.right≠T.nil

2 **return** TREE-MINIMUM(x.right)**//找到以右孩子为根节点的最大值**

3 y=x.p

4 **while** y≠T.nil **and** x≠y.left**//循环结束时x为y的左孩子**

5 x=y

6 y=y.p

7 **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)

1 **if** x.left≠T.nil

2 **return** TREE-MAXIMUM(x.left)**//找到以左孩子为根节点的最大值**

3 y=x.p

4 **while** y≠T.nil **and** x≠y.right**//循环结束时x为y的右孩子**

5 x=y

6 y=y.p

7 **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

**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

# 红黑树

## 定义

### 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

### 树

1、属性

1. nil：哨兵节点
2. root：根节点

## 旋转

**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**

## 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：**

### 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**

孩子

**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**

孩子

**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**

孩子

**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**

孩子

**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**

孩子

**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**

孩子

**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)

1 **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**

2 **if** z.p==z.p.p.left

3 y=z.p.p.right

4 **if** y.color==RED

5 z.p.color=BLACK

6 y.color=BLACK

7 z.p.p.color=RED

8 z=z.p.p**//继续循环**

9 **else**\_**if** z==z.p.right

10 z=z.p

11 LEFT-ROTATE(T,z)

12 z.p.color=BLACK

13 z.p.p.color=RED

14 RIGHT-ROTATE(T,z.p.p)**//循环结束**

15 **else** z.p==z.p.p.right

16 y=z.p.p.left

17 **if** y.color==RED

18 z.p.color=BLACK

19 y.color=BLACK

20 z.p.p.color=RED

21 z=z.p.p**//继续循环**

22 **else\_if** z==z.p.left

23 z=z.p

24 RIGHT-ROTATE(T,z)

25 z.p.color=BLACK

26 z.p.p.color=RED

27 LEFT-ROTATE(T,z.p.p) **//循环结束**

28 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)

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 **//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

1. 当被删除节点的颜色为红色，那么不会破坏红黑树的性质
2. 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

### 删除辅助

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 RB-TRANSPLANT(T,y,y.right)//即使y.right是哨兵，也会指向y的父节点**

**13 y.right=z.right**

**14 y.right.p=y**

15 RB-TRANSPLANT(T,z,y)

16 y.left=z.left

17 y.left.p=y

18 y.color=z.color

19 **if** y-original-color==BLACK

20 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**

孩子

**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**

孩子

**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’)**

孩子

**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**

孩子

**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**

孩子

**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**

孩子

**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)**

孩子

**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**

孩子

**C E 对B做一次右旋 E A**

RB-DELETE-FIXUP(T,x)

1 **while** x≠T.root **and** x.color ==BLACK**//若x是红色的，那么将x改为黑色即可**

2 **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**

3 w=x.p.right

4 **if** w.color==RED

5 w.color=BLACK

6 x.p.color=RED

7 LEFT-ROTATE(T,x.p)

8 w=x.p.right

9 **if** w.left.color==BLACK and w.right.color==BLACK

10 w.color=RED

11 x=x.p

12 **else\_if** w.right.color==BLACK

13 w.left.color=BLACK

14 w.color=RED

15 RIGHT-ROTATE(T,w)

16 w=x.p.right

17 w.color=x.p.color

18 x.p.color=BLACK

19 w.right.color=BLACK

20 LEFT-ROTATE(T,x.p)

21 x=T.root

22 **elseif** x==x.p.right

23 w=x.p.left

24 **if** w.color==RED

25 w.color=BLACK

26 x.p.color=RED

27 RIGHT-ROTATE(T,x.p)

28 w=x.p.left

29 **if** w.left.color==BLACK and w.right.color==BLACK

30 w.color=RED

31 x=x.p

32 **else\_if** w.left.color==BLACK

33 w.right.color=BLACK

34 w.color=RED

35 LEFT-ROTATE(T,w)

36 w=x.p.left

37 w.color=x.p.color

38 x.p.color=BLACK

39 w.left.color=BLACK

40 RIGHT-ROTATE(T,x.p)

41 x=T.root

42 **x**.color=BLACK

# AVLTree(版本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

## 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+

### 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HD=HA-1或HD=HA-2
* 其次，只有HC>=HE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
* HD=HA-1或HD=HA-2
* HC=HA-1
* HE=HA-1或HE=HA-2
* 旋转后，各节点高度如下
* HD+=HD=HA-1或HA-2
* HE+=HE=HA-1或HA-2
* HC+=HC=HA-1
* 旋转后，B节点平衡，HB+=HA或HB+=HA-1
* 旋转后，A节点平衡，HA+=HA或HA+=HA+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HC=HB-1或HC=HB-2
* 其次，只有HD>=HE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
* HC=HB-1或HC=HB-2
* HD=HB-1
* HE=HB-1或HE=HB-2
* 旋转后，各节点高度如下
* HD+=HD=HB-1
* HE+=HE=HB-1或HB-2
* HC+=HC=HB-1或HB-2
* 旋转后，A节点平衡，且HA+=HB或HA+=HB-1
* 旋转后，B节点平衡，且HB+=HB或HB+=HB+1



### 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HB>=HD
* 若不满足HB>=HD，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HE>=HD
* 若不满足HE>=HD，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



## 基本操作

HEIGHT(T,x)

1 **if** x.left.height≥x.right.height **//左右节点均存在**

2 x.height=x.left.height+1

3 **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

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**

### 维护AVL树的性质

**HOLD-ROTATE(T,x,orientation)**

1 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**

2 stack1.push(x)

3 stack2.push(orientation)

4 cur=Nil

5 rotateRoot=Nil **//对x尝试旋转后，返回最终旋转后的根节点**

6 curOrientation=INVALID;

7 **while**(!stack1.Empty())

8 cur=stack1.top()

9 curOrientation=stack2.top()

10 **if** curOrientation==LEFT **//需要对cur尝试进行左旋**

11 **if** cur.right.right.height≥cur.right.left.height

12 stack1.pop()

13 stack2.pop()

14 rotateRoot=LEFT-ROTATE(T,cur)

15 **else**

16 stack1.push(cur.right)**//否则cur右孩子需要尝试进行右旋来调整**

17 stack2.push(RIGHT);

18 **elseif** curOrientation ==RIGHT**//需要对cur尝试进行右旋**

19 **if** cur.left.left.height≥cur.left.right.height

20 stack1.pop()

21 stack2.pop()

22 rotateRoot=RIGHT-ROTATE(T,cur)

23 **else**

24 stack1.push(cur.left) **//否则cur左孩子需要尝试进行左旋来调整**

25 stack2.push(LEFT)

26 return rotateRoot

## 插入

TREE-INSERT(T,z)

1 y=T.nil

2 x=T.root

3 **while** x≠T.nil**//循环结束时x指向空，y指向上一个x**

4 y=x

5 **if** z.key<x.key

6 x=x.left

7 **else** x=x.right

8 z.p=y**//将这个叶节点作为z的父节点**

9 **if** y==T.nil

10 T.root=z

11 **elseif** z.key<y.key

12 y.left=z

13 **else** y.right=z

14 z.left=T.nil

15 z.right=T.nil

**16 FIXUP(T,z)**

**FIXUP(T,y)**

1 **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**

2 y=y.p

3 **while**(y≠T.nil) **//沿着y节点向上遍历该条路径**

4 HEIGHT(y)

5 **if** y.left.height==y.right.height+2 **//左子树比右子树高2**

6 y= HOLD-ROTATE(T,y,2)

7 **elseif** y.right.height=y.left.height+2

8 y= HOLD-ROTATE(T,y,1)

9 y=y.p

## 删除

TREE-DELETE(T,z)

1 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**

2 **if** z.left==T.nil

3 x=y.right

4 TRANSPLANT(T,z,z.right)

5 **elseif** z.right==T.nil

6 x=y.left

7 TRANSPLANT(T,z,z.left)

8 **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**

9 x=y.right

10 **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)**

11 x.p=y

12 **else** TRANSPLANT(T,y,y.right)

13 y.right=z.right

14 y.right.p=y

15 TRANSPLANT(T,z,y)

16 y.left=z.left

17 y.left.p=y

**18 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**



# AVLTree(版本2)

## 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

## 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)

### 第一类不平衡(以左左为例)



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+

2、调整前，各节点的高度如下

* HB=HC+2
* HA=HC+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)
* HD=HB-1=HC+1
* HE必定小于HD(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HE=HC

3、右旋调整后，各节点的高度如下

* HD+=HD=HC+1
* HE+=HE=HC
* HC+=HC
* 由于HE+=HC+，于是A节点平衡，且HA+=HC+1
* B节点也是平衡的，且HB+=HC+2

4、可以发现，调整前后子树根节点的高度都是HC+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

### 第一类不平衡(以左右为例)



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

2、调整前，各节点的高度如下

* HB=HC+2
* HA=HC+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)
* HE=HB-1=HC+1
* HD必定小于HE，因此HD=HC
* HH与HI至少有一个是HC，另一个可以是HC或HC-1

3、对B节点进行一次左旋后，各节点高度如下

* HD+=HD=HC
* HH+=HH=HC or HC-1
* HI+=HI=HC or HC-1
* HC+=HC
* HA+=HA=HC+2
* HB+=HC+1
* 当HI+=HC-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，HE=HC+2

4、对A节点进行一次右旋，各节点高度如下

* HD++=HD+=HC
* HH++=HH+= HC or HC-1
* HI++=HI+=HC or HC-1
* HC++=HC+=HC
* 旋转后，B节点平衡，HB++=HC+1
* 旋转后，A节点平衡，HA++=HC+1
* 因此旋转后，E节点平衡，HE++=HC+2

5、可以发现，调整前后子树根节点的高度都是HC+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

## 基本操作

HEIGHT(T,x)

1 **if** x.left.height≥x.right.height **//左右节点均存在**

2 x.height=x.left.height+1

3 **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

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**

## 插入

TREE-INSERT(T,z)

1 y=T.nil

2 x=T.root

3 **while** x≠T.nil//循环结束时x指向空，y指向上一个x

4 y=x

5 **if** z.key<x.key

6 x=x.left

7 **else** x=x.right

8 z.p=y//将这个叶节点作为z的父节点

9 **if** y==T.nil

10 T.root=z

11 **elseif** z.key<y.key

12 y.left=z

13 **else** y.right=z

14 z.left=T.nil

15 z.right=T.nil

16 INSERT-FIX(T,z)

INSERT-FIX(T,z)

1 originHigh=z.h

2 HEIGHT(z)

3 r=z

4 **if** z.left.h==z.right.h+2

5 **if** z.left.left.h>z.left.right.h //第一类

6 r=RIGHT-ROTATE(z)

7 **elseif** z.left.left.h<z.left.right.h //第二类

8 LEFT-ROTATE(z.left)

9 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

10 **elseif** z.right.h==z.left.h+2

11 **if** z.right.right.h>z.right.left.h //第一类

12 r=LEFT-ROTATE(z)

13 **elseif** z.right.right.h<z.right.left.h //第二类

14 RIGHT-ROTATE(z.right)

15 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

16 **if** r.h!=originHigh **and** r!=root

17 INSERT-FIX(r.parent)

## 删除

TREE-DELETE(T,z)

1 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**

2 **p=y.parent** //p为被删除节点的父节点

3 **if** z.left==T.nil

4 TRANSPLANT(T,z,z.right)

5 **elseif** z.right==T.nil

6 TRANSPLANT(T,z,z.left)

7 **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**

8 **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

9 p=y

10 **else**

11 p=y.parent

12 TRANSPLANT(y,y.right)

13 y.right=z.right

14 y.right.parent=y

15 y.left=z.left

16 y.left.parent=y

17 TRANSPLANT(T,z,y)

18 if p!=nil

19 DELETE-FIX(p)

DELETE -FIX(T,z)

1 HEIGHT(z)

2 r=z

3 if z.left.h==z.right.h+2

4 if z.left.left.h**>=**z.left.right.h //第一类

5 r=RIGHT-ROTATE(z)

6 else //第二类

7 LEFT-ROTATE(z.left)

8 r=RIGHT-ROTATE(z)

9 elseif z.right.h==z.left.h+2

10 if z.right.right.h**>=**z.right.left.h //第一类

11 r=LEFT-ROTATE(z)

12 else //第二类

13 RIGHT-ROTATE(z.right)

14 r=LEFT-ROTATE(z)

15 if r!=root

16 DELETE-FIX(r.parent)

# 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)

1 r=x.left.size+1

2 if i==r

3 return x

4 elseif i<r

5 return OS-SELECT(x.left,i)

6 else return OS-SELECT(x,right,i-r)

OS-RANK(T,x)

1 r=x.left.size+1

2 y=x

3 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**

4 if y==y.p.right

5 r=r+y.p.left.size+1

6 y=y.p

7 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

9 z.p=y

10 **if** y==T.nil

11 T.root=z

12 **elseif** z.key<y.key

13 y.left=z

14 **else** y.right=z

15 z.left=T.nil

16 z.right=T.nil

17 z.colcor=RED

18 z.size=1**//新插入的节点大小为1**

19 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)

# 动态规划DP

## 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**

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**

2 let m[1…n,1…n] and s[1…n-1,2…n] be new tables

3 for i=1 to n

4 m[i,i]=0

5 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)**

6 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)**

7 j=i+ g -1**//长为g子链以i为起始索引时，终止索引为j**

8 m[i,j]=∞

9 for k=i to j-1**//子链[i,j]的分割点**

10 q=m[i,k]+m[k+1,j]+pi-1pkpj**//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**

11 if q<m[i,j]

12 m[i,j]=q

13 s[i,j]=k

14 return m and s

**其中m[i,j]代表计算矩阵Ai…j所需标量乘法次数的最小值**

**其中s[i,j]代表满足Ai…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Ai=p[i-1]\*[i]**

PRINT-OPTIMAL-PARENS(s,i,j)

1 if i==j

2 print “A”i

3 else print”(”

4 PRINT-OPTIMAL-PARENS(s,i,s[i,j])

5 PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)

6 print”)”

**自带备忘的自顶向下法：**

MATRIX-CHAIN-MEMOIZED(p)

1 int n=p.length-1

2 let m[1…n,1…n] and s[1…n-1,2…n]be new tables

3 for i=1 to n//该循环初始化，使得备忘录为特殊值

4 for j=i to n

5 m[i,j]=∞

6 MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,1,n)

7 return m and s

MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,i,j)

1 if m[i,j]<∞//若不为特殊值说明该情况已求得最优解

2 return m[i,j]

3 if i==j

4 m[i,j]=0

5 else for k=i to j-1

6 q= MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,i,k)+

MATRIX-CHIAN-MEMOIZED-AUX(p,m,s,k+1,j)+ pi-1pkpj

7 if q<m[i,j]

8 m[i,j]=q

9 s[i,j]=k

10 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

LCS(X,Y) longest common subsequence

1 m=X.length

2 n=Y.length

3 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)

4 **for** i=1 **to** m

5 c[i,0]=0

6 **for** j=0 **to** n

7 c[0,j]=0

8 **for** i=1 **to** m

9 **for** j=1 **to** n

10 **if** xi==yj

11 c[i,j]=c[i-1,j-1]+1

12 b[i,j]= ↖

13 else**if** c[i-1,j]≥c[i,j-1]

14 c[i,j]=c[i-1,j]

15 b[i,j]= ↑

16 **else** c[i,j]=c[i,j-1]

17 b[i,j]= ←

18 **return** c and b

**LMS-LENGTH**(X) longest monotonous sunsequence

1 n=X.length

2 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

3 **for** i=1 **to** n

4 c[i]=1//至少为1嘛

5 **for** i=2 **to** n

6 **for** j=1 **to** i-1

7 **if** X[i]>X[j] **and** c[i]<c[j]+1

8 c[i]=c[j]+1

9 b[i]=j

## 0BST

0BST(p,q)

1 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

2 **for** i=1 **to** n+1

3 e[i,i-1]=q[i-1];

4 w[i,i-1]=q[i-1];

5 **for** L=1 **to** n //L代表子树的长度

6 **for** i=1 **to** n-L+1//i代表长为L的子树的起始索引

7 j=i+L-1//j代表长为L的子树的终止索引

8 e[i,j]=∞

9 w[i,j]=w[i,j-1]+p[j]+q[j]

10 **for** r=i **to** j//r代表长为L的子树的分割点

11 t=e[i,r-1]+e[r+1,j]+w[i,j]

12 **if** t<e[i,j]

13 e[i,j]=t

14 root[i,j]=r

15 **return** e and root

PRINT\_TREE(i,j,last)

1 cur=SUB\_ROOT(i,j)//若j<i会返回0

2 **if** i==1 **and** j==root.Length

3 print(“K”+cur+”为根”)

4 **elseif** cur==0

5 **if** j<last

6 print(“D”+j+”为”+”K”+last+”的左孩子)

7 else print(“D”+j+”为”+”K”+last+”的左孩子)

8 **elseif** index<last

9 print(“K”+index+”为”+”K”+last+”的左孩子)

10 PRINT\_TREE(i,index-1,index)

11 PRINT\_TREE(index+1,j,index)

12 **else** print(“K”+index+”为”+”K”+last+”的右孩子)

13 PRINT\_TREE(i,index-1,index)

14 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

# 贪心算法

RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n)

1 m=k+1

2 **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

3 m=m+1

4 **if** m≤n

5  **return** {am}URECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)

GREEDY-ACTIVITY-SELECTOR(s,f)

1 n=s.length

2 A={a1} **//由于活动按结束时间排序，第一个活动必定会选择**

3 k=1

4 **for** m=2 **to** n

5 **if** s[m]≥f[k]

6 A=A∪{am}

7 k=m

8 **return** A

## 0-1背包问题

**MaxValue(p,w,V)**

1 n=p.length

2 let dp[1...n][1...V] be a new array

3 **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

4 if w[1]<=v

5 dp[1][v]=p[1] //装得下就装下

6 else dp[1][v]=0 //装不下就舍弃

7 **for** i=2 **to** n

8 for v=1 **to** V

9  **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

10 dp[i][v]=dp[i-1][v]

11 **else** dp[i][v]=max(dp[i-1][v],dp[i-1][v-w[i]]+p[i])

12 **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)

1 n=|C|

2 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)

3 **for** i=1 **to** n-1

4 allocate a new node z

5 z.left=x=EXTRACT-MIN(Q)//提取出优先队列中的第一项

6 z.right=y=EXTRACT-MIN(Q) //提取出优先队列中的第一项

7 z.freq=x.freq+y.freq

8 INSERT(Q,z)

9 **return** EXTRACT-MIN(Q)

Q是一个优先队列

# B树

## 定义

### 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.key1, x.key2, ..., x.keyx.n，以非降序存放，使得

x.key1≤x.key2≤...≤x.keyx.n

1. 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
2. 每个内部节点x还包含x.n+1个指向其孩子的指针，x.c1, x.c2, ..., x.cx.n+1，叶节点没有孩子，所以他们的ci属性没有定义
3. 关键字x.keyi对存储在各子树中的关键字范围加以分割：如果ki为任意一个存储在以x.ci为根的子树中的关键字，那么

k1≤x.key1≤k2≤x.key2≤…≤x.keyx.n≤kx.n+1

1. 每个叶节点都具有相同的深度，即树的高度h
2. 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

### 树

1、属性

1. t：B树的度
2. root：B树的根节点

## 基本操作

**对于节点x ，关键字x.keyi 与子树指针x.ci 的索引相同，就说x.ci是关键字x.keyi对应的子树指针**

**子树x.ci的元素介于 x.keyi-1~x.keyi之间 1≤i≤x.n+1，为保持一致性，记x.key0= -∞，x.keyx.n+1=+∞**

**B-TREE-SEARCH(x,k)**

1 i=1

2 **while** i ≤ x.n **and** k > x.keyi

3 i=i+1

4 **if** i ≤ x.n and k==x.keyi

5 **return** (x,i)

6 **elseif** x.leaf

7 **return** NIL

8 **else** DISK-READ(x,ci)

9 **return** B-TREE-SEARCH(s.ci,k)

**B-TREE-CREATE(T)**

1 x=ALLOCATE-NODE()

2 x.leaf=TRUE

3 x.n=0

4 DISK-WRITE(x)

5 T.root=x

**B-TREE-SPLIT-CHILD(x,i)**//x.ci是满节点，x是非满节点

1 z=ALLOCATE-NODE()//z是由y的一半分裂得到

2 y=x.ci

3 z.leaf=y.leaf

4 z.n=t-1

5 **for** j=1 **to** t-1

6 z.keyj=y.keyj+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

7 **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

8 **for** j=1 **to** t

9 z.cj=y.cj+t

10 y.n=t-1

11 **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

12 x.cj+1=x.cj

13 x.ci+1=z

14 **for** j=x.n **downto** i

15 x.keyj+1=x.keyj

16 x.keyi=y.keyt

17 x.n=x.n+1

18 DISK-WRITE(y)

19 DISK-WRITE(z)

20 DISK-WRITE(x)

## 插入

**B-TREE-INSERT(T,k)**

1 r=T.root

2 **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

3 s=ALLOCATE-NODE()//分配一个节点作为根节点

4 T.root=s

5 s.leaf=FLASE//显然由分裂生成的根必然是内部节点

6 s.n=0

7 s.c1=r//之前的根节点作为新根节点的第一个孩子

8 B-TREE-SPLIT-CHILD(s,1)

9 B-TREE-INSERT-NONFULL(s,k)

10**else** B-TREE-INSERT-NONFULL(r,k)

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

1 i=x.n

2 **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

3 **while** i ≥1 **and** k<x.keyi

4 x.keyi+1=x.keyi

5 i=i-1

6 x.keyi+1=k

7 x.n=x.n+1

8 DISK-WRITE(x)

9 **else** **while** i ≥ 1 **and** k<x.keyi

10 i=i-1

11 i=i+1//转到对应的指针坐标

12 DISK-READ(x.ci)

13 **if** x.ci.n==2t-1

14 B-TREE-SPLIT-CHILD(x,i)

15 **if** k>x.keyi //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

16 i=i+1

17 B-TREE-INSERT-NONFULL(x.ci,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

1 **if** !B-TREE-SEARCH(k) **or** k==B-TREE-MINIMUM(T.root)

2 throw error(no precursor)

3 B-TREE-PRECURSORAUX(T.root,k)

**B-TREE-PRECURSORAUX(x,k)**

1 i=1

2 **if** x.leaf//若为叶节点

3 while i≤x.n **and** k>x.keyi ++i //找到第一个不小于k的关键字(大于或等于都可以)

4 **return** x.keyi-1

5 **else** //若不为叶节点

6 **while** i≤x.n **and** k>x.keyi ++i //找到第一个不小于k的关键字

7 **if** k==x.keyi return B-TREE-MAXIMUM(x.ci) //若这个关键字等于k，那么在对应子树中找最大值

8 **if** MINIMUM(x.c)≥k **//如果该关键字对应的子树的最小值大于k**

9 **return** x.ki-1 **//那么前驱必然是当前节点中的前一个关键字**

10 **return** B-TREE-PRECURSORAUX(x.ci,k)//否则在该关键字对应的子树中继续寻找

**B-TREE-SEARCH-SUCCESSOR(x,k)**

1 **if** !B-TREE-SEARCH(x,k) **or** k=B-TREE-MAXIMUM(T.root)

2 throw error (no successor)

3 B-TREE-SEARCH-SUCCESSORAUX(x,k)

**B-TREE-SEARCH-SUCCESSORAUX(x,k)**

1 i=x.n

2 **if** x.leaf

3 **while** i≥1 **and** k<x.keyi --i

4 **return** x.keyi+1

5 **else**

6 **while** i≥1 **and** k<x.keyi --i

7 **if** k==x.keyi **return** B-TREE-MINIMUMAUX(x.ci+1)

8 **if** k≥B-TREE-MAXIMUM(x.ci+1)

9 **return** x.keyi+1

9 **return** B-TREE-SEARCH-SUCCESSORAUX(x.ci+1,k)

**B-TREE-MINIMUM(x)**

1 **if** x.leaf **return** x.key1

2 **return** B-TREE-MIMIMUM(x.c1)

**B-TREE-MAXIMUM(x)**

1 **if** x.leaf **return** x.keyx.n

2 **return** B-TREE-MAXIMUM(x.cx.n+1)

## 删除

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

1 r=T.root

2 **if** r.n==1

3 DISK-READ(r.c1)

4 DISK-READ(r.c2)

5 y=r.c1

6 z=r.c2

7 **if** **not** r.leaf **and** y.n==z.n==t-1

8 B-TREE-MERGE-CHILD(r,1,y,z)

9 T.root=y

10 FREE-NODE(r)

11 B-TREE-DELETE-NOTNONE(y,k)

12 **else** B-TREE-DELETE-NOTNONE(r,k)

13 **else** B-TREE-DELETE-NOTNONE(r,k)

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

1 y.n=2t-1

2 **for** j=t+1 **to** 2t-1

3 y.keyj=z.keyj-t

4 y.keyt=x.keyi //the key from node x merge to node y as the tth key

5 **if** **not** y.leaf

6 **for** j=t+1 **to** 2t

7 y.cj=z.cj-t

8 **for** j=i+1 **to** x.n

9 x.keyj-1=x.keyj

10 x.cj=x.cj+1

11 x.n=x.n-1

12 Free(z)

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

1 y.n=y.n+1

2 y.keyy.n=x.keyi

3 x.keyi=z.key1

4 z.n=z.n-1

5 j=1

6 **while** j≤z.n

7 z.keyj=z.keyj+1

8 j=j+1

9 **if** **not** z.leaf

10 y.cy.n+1=z.c1

11 j=1

12 **while** j≤z.n+1

13 z.cj=z.cj+1

14 j++

15 DISK-WRITE(y)

16 DISK-WRITE(z)

17 DISK-WRITE(x)

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

1 z.n=z.n+1

2 j=z.n

3 **while** j>1

4 z.keyj=z.keyj-1

5 j--

6 z.key1=x.keyi

7 x.keyi=y.keyy.n

8 **if** **not** z.leaf

9 j=z.n

10 **while** j>0

11 z.cj+1=z.cj

12 j--

13 z.c1=y.cy.n+1

14 y.n=y.n-1

15 DISK-WRITE(y)

16 DISK-WRITE(z)

17 DISK-WRITE(x)

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

1 i=1

2 **if** x.leaf

3 **while** i ≤ x.n **and** k>x.keyi

4 i=i+1

5 **if** k==x.keyi

6 **for** j=i+1 **to** x.n

7 x.keyj-1=x.keyj

8 x.n=x.n-1

9 DISK-WRITE(x)

10 **else** error:”the key does not exist”

11 **else** **while** i ≤ x.n **and** k>x.keyi

12 i=i+1

13 DISK-READ(x.ci)

14 y=x.ci

15 **if** i ≤ x.n

16 DISK-READ(x.ci+1)

17 z=x.ci+1

18 **if** i ≤ x.n **and** k==x.keyi //Cases 2

19 **if** y.n>t-1 //Cases 2a

20 k’=B-TREE-MIMIMUM(y)

21 B-TREE-DELETE-NOTNONE(y,k’)

22 x.keyi=k’

23 **elseif** z.n>t-1 //Case 2b

24 k’=B-TREE-MAXIMUM(z)

25 B-TREE-DELETE-NOTNONE(z,k’)

26 x.keyi=k’

27 **else** B-TREE-MERGE-CHILD(x,i,y,z) //Cases 2c

28 B-TREE-DELETE-NOTNONE(y,k)

29 **else** //Cases3

30 **if** i>1

31 DISK-READ(x.ci-1)

32 p=x.ci-1

33 **if** y.n==t-1

34 **if** i>1 **and** p.n>t-1 //Cases 3a

35 B-TREE-SHIFT-TO-RIGHT-CHILD(x,**i-1**,p,y)

36 **elseif** i ≤ x.n **and** z.n>t-1

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

38 **elseif** i>1 //Cases3b

39 B-TREE-MERGE-CHILD(x**,i-1**,p,y)

40 y=p

41 **else** B-TREE-MERGE-CHILD(x,i,y,z) //Cases 3b

42 B-TREE-DELETE-NOTNONE(y,k)

**SPLID**



**Merge**



# 斐波那契堆

## 定义

### 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

## 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

## 插入操作

FIB-HEAP-INSERT(H,x)

1 x.degree=0

2 x.p=NULL

3 x.child=NULL

4 x.mark=FLASE

5 if H.min==NULL

6 ***create a root list for H containing just x***

7 H.min=x

8 else

9 ***insert x into H's root list***

10 if x.key<H.min.key

11 H.min=x

12 H.n=H.n+1

## 堆合并

FIB-HEAP-UNION(H1,H2)

1 H=MAKE-FIB-HEAP

2 H.min=H1.min

3 ***concatenate the root list of H2 with the root list of H***

4 if(H1.min==NULL) or (H2.min!=NULL and H2.min.key<H1.min.key)

5 H.min=H2.min

6 H.n=H1.n+H2.n

7 return H

## 抽取最小节点

FIB-HEAP-EXTRACT-MIN(H)

1 z=H.min

2 if z!=NULL

3 for each child x of z

4 ***add x to the root list of H***

5 x.p=NULL

6 ***remove z from the root list of H***

7 if z==z.right

8 H.min=NULL

9 else

10 H.min=z.right

11 CONSOLIDATE(H)

12 H.n=H.n-1

13 return z

## 抽取最小节点时维护斐波那契堆的性质

CONSOLIDATE(H)

1 let A[0...D(H.n)] be a new array

2 for i=0 to D(H.n)

3 A[i]=NULL

4 for each node w in the root list of H

5 x=w

6 d=x.degree

7 while A[d]!=NULL

8 y=A[d]

9 if x.key>y.key

10 exchange(x,y)

11 FIB-HEAP-LINK(H,y,x)

12 A[d]=NULL

13 d=d+1

14 A[d]=x

15 H.min=NULL

16 for i=0 to D(H.n)

17 if A[i]!=NULL

18 if H.min==NULL

19 ***create a root list for H containing just A[i]***

20 H.min=A[i]

21 else

22 ***insert A[i] into H's root list***

23 if A[i].key<H.min.key

24 H.min=A[i]

FIB-HEAP-LINK(H,y,x)

1 ***remove y from the root list of H***

2 ***make y a child of x, incrementing x.degree***

3 y.mark=FLASE

## 关键字减值和删除节点

FIB-HEAP-DECREASE-KEY(H,x,k)

1 if k>x.key

2 error "new key is greater than current key"

3 x.key=k

4 y=x.p

5 if y!=NULL and x.key<y.key

6 CUT(H,x,y)

7 CASCADING-CUT(H,y)

8 if x.key<H.min.key

9 H.min=x

CUT(H,x,y)

1 remove x from the child list of y, decrementing y.degree

2 add x to the root list of H

3 x.p=NIL

4 x.mark=FALSE

CASCADING-CUT(H,y)

1 z=y.p

2 if z!=NULL

3 if y.mark==FALSE

4 y.mark=TRUE

5 else

6 CUT(H,y,z)

7 CASCADING-CUT(H,z)

# 基本图算法

## BFS

**BFS(G,s)**

1 **for** each vertex u∈G.V-{s}

2 u.color=WHITE

3 u.d=+∞

4 u. π=NIL

5 s.color=GRAY

6 s.d=0

7 s. π=NIL

8 let queue be a new Queue

9 queue.offer(s)

10 **while** **not** queue.isEmpty()

11 u=queue.poll()

12 **for** each v∈ G.Adj[u]

13 **if** v.color==WHITE

14 v.color=GRAY

15 v.d=u.d+1

16 v. π=u

17 queue.offer(v)

18 u.color=BLACK

## DFS

**DFS(G)**

1 **for** each vertex u∈G.V

2 u.color=WHITE

3 u. π=NIL

4 time=0

5 **for** each vertex u∈G.V

6 **if** u.color==WHITE

7 DFS-VISIT(G,u)

DFS-VISIT(G,u)

1 time=time+1

2 u.d=time

3 u.color=GRAY

4 **for** each v∈G:Adj[u]

5 **if** v.color==WHITE

6 v. π=u

7 DFS-VISIT(G,v)

8 u.color=BLACK

9 time=time+1

10 u.f=time

# 字符串匹配

## KMP算法

**KMP-MATCHER(T,P)**

1 n=T.length

2 m=P.length

3 π=COMPUTE-PREFIX-FUNCTION(P)

4 k=0

5 **for** q=1 **to** n

6 **while** k>0 **and** P[k+1]≠T[q]

7 k=π[k]

8 **if** P[k+1]==T[q]

9 k=k+1

10  **if** k==m

11 print “Pattern occurs with shift” q-m

12 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

 