

## 第8章 常用数据结构与算法

一个衣柜若未分层，则所有的衣服、鞋子就会堆放在一起，显得十分凌乱。对衣柜进行分层可以方便我们按类别高效地管理存放在衣柜里的衣服或鞋子。计算机的内存就像是一个衣柜，我们在内存上设计不同的数据结构，为的就是高效管理数据；有时，虽然衣柜分了层，但是我们在摆放衣服时会把常穿的衣服放在靠外的位置，而把不常穿的衣服压在箱底，这类似于计算机内存的数据存取方法。采用有效的存取方法，可以提高我们访问数据的效率，而通过数据对问题进行求解的思路与过程称为算法。

要学习算法，就必须先掌握常用的数据结构。常用的数据结构包括数组、栈、队列、链表、树、堆、哈希表、图等，本章讲解除图外的所有数据结构。

学习目标：

- 常用栈、队列、二叉树、堆、红黑树、哈希表等数据结构的增删查改。
- 通过排序算法掌握时间复杂度与空间复杂度。
- 掌握常用的查找算法。
- 算法学习方法讲解。

### 8.1 数据结构

#### 8.1.1 栈（运行展示）

栈（stack）又称堆栈，是一种运算受限的线性表。栈的限制是仅允许在表的一端进行插入和删除操作，被允许操作的一端被称为栈顶，另一端则被称为栈底，如图 8.1.1 所示。向一个栈中插入新元素又称入栈或压栈，它把新元素放到栈顶元素的上面，使之成为新的栈顶元素。从一个栈中删除元素又称出栈或退栈，它把栈顶元素删除，使其相邻的元素成为新的栈顶元素。由于堆栈只允许在一端进行操作，所以遵循后进先出（Last In First Out, LIFO）的操作原则。

栈可以用数组实现，也可以用链表实现，这里用链表来实现栈。第 7 章介绍了链表的增删查改，这里采用链表的头部插入法（头插法）、头部删除法来实现先进后出的效果。数据结构包括逻辑结构、存储结构和对数据的运算。栈的逻辑结构前面已经介绍，由于采用链表实现，所以具体的存储结构如例 8.1.1 所示。

【例 8.1.1】实现一个栈。

```
typedef struct tag
{
    int m_ival;
    struct tag* next;
}Node, *pNode;
```

```
typedef struct tagstack
{
    pNode phead; //栈顶指针
    int size; //栈中的元素个数
}Stack, *pStack;
```

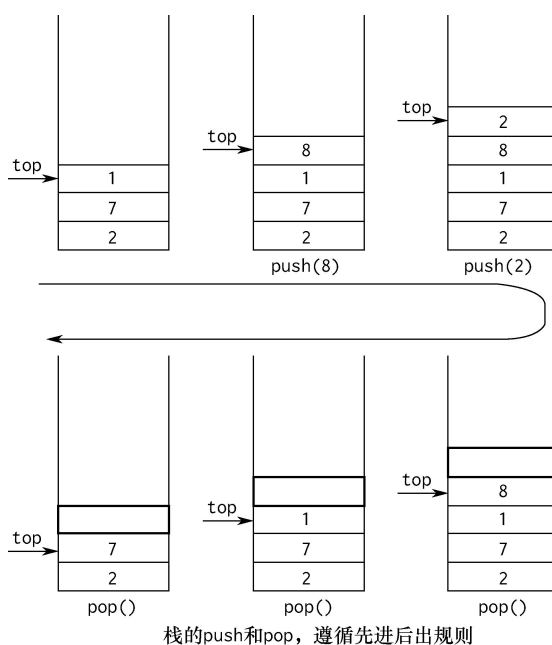


图 8.1.1 栈

要求栈具有下列功能, 函数是对数据的运算方法:

```
void init_stack(pStack stack);
void pop(pStack stack)           //出栈
void push(pStack stack, int val) //入栈
int top(pStack stack)           //返回栈顶元素
int empty(pStack stack)         //判断栈是否为空
int size(pStack stack)          //返回栈中数据的元素个数
```

首先要初始化栈, 将头指针赋值为 `NULL`, 将栈大小初始化为零, 具体实现如下:

```
void init_stack(pStack p)
{
    p->phead=NULL; //链表头指针设置为 NULL
    p->size=0;
}
```

因为栈是空的, 所以首先要做压栈操作, 压栈函数的具体实现如下:

```
void push(pStack p,int val)
```

```
{
    pNode pnew=(pNode)calloc(1,sizeof(Node)); //为压栈的新结点申请空间
    pnew->m_val=val; //将要压栈的值放入对应结点空间
    pnew->pnext=p->head; //头插法
    p->head=pnew;
    p->size++; //栈的元素加 1
}
```

压栈之后可以返回栈顶元素。下面是 *top* 函数的实现，注意 *top* 函数只获得栈顶元素值，而不会弹栈，详细实现如下：

```
int top(pStack p)
{
    return p->head->m_val; //返回栈顶元素值
}
```

我们通过 *p->size* 存储了栈的大小，当栈为空时，该变量的值为零，因此我们对其取反。这样，当栈为空时，*empty* 函数返回 1；栈不为空时，*empty* 函数返回 0。详细实现如下：

```
int empty(pStack p)
{
    return !p->size;
}
```

判断栈是否为空，如果不为空就返回栈的大小。因为我们存储了栈的大小，所以直接返回对应的变量值即可。详细代码如下：

```
int size(pStack p)
{
    return p->size; //将栈大小存储到 size 中
}
```

下面是一个简单的测试程序，读者可在 VS 中自行编写并测试：

```
int main()
{
    Stack s;
    init_stack(&s);
    push(&s,10);
    push(&s,5);
    printf("the top of the stack val=%d\n",top(&s));
    pop(&s);
    printf("the top of the stack val=%d\n",top(&s));
    pop(&s);
    printf("stack is empty?=%c\n",empty(&s)?'Y':'N');
    return 0
}
```

```
}
```

### 8.1.2 队列（运行展示）

队列是先进先出（First-In-First-Out, FIFO）的线性表。在具体应用中，通常用链表或数组来实现队列。队列只允许在后端（称为 *rear*）进行插入操作，在前端（称为 *front*）进行删除操作。通过对链表进行头部删除、尾部插入，可以实现每个元素先进先出的效果。由于代码与之前类似，因此这里不再赘述。

**循环队列：**如果通过链表实现，那么让链表尾指针的 *pnext* 指向头指针，即可用链表实现循环队列。如果通过数组实现，那么对数组进行遍历，当 *i* 等于最后一个元素时，将 *i* 赋值为 0，就可重新回到数组元素的起始点，如例 8.1.2 所示。

【例 8.1.2】循环队列实现。

```
#include <stdio.h>
#include <stdlib.h>

#define MaxSize 5
typedef int ElemType;
typedef struct{
    ElemType data[MaxSize]; //数组，存储 MaxSize-1 个元素
    int front,rear; //队列头、队列尾下标
}SqQueue;

//初始化队列时，让队列头和队列尾都指向数组的 0 号元素，数组下标从零开始
void InitQueue(SqQueue *Q)
{
    Q->rear=Q->front=0;
}

//判空，如果队列头部与队列尾部指向的下标相同，那么说明循环队列为空
int isEmpty(SqQueue *Q)
{
    //不需要为零，因为具体循环队列是不断地入队、出队的，出队一个元素时，Q->front 加 1，
    //头部和尾部相等时，代表循环队列为空
    if(Q->rear==Q->front)
        return 1;
    else
        return 0;
}

//入队
int EnQueue(SqQueue *Q,ElemType x)
```

```

{
//循环队列，Q->rear指向的元素是我们将要放置的元素位置，为了能够区分头部与尾部，对
//于循环队列我们需要空出一个元素作为分割，因此数组的长度为5，实际可以放入的元素个
//数为4。因此，我们判断队列是否满的策略是，Q->rear加1后，判断是否等于Q->front，
//因为要考虑到循环的问题，所以需要除以MaxSize得余数。
    if((Q->rear+1)%MaxSize==Q->front)
        return 0;
    Q->data[Q->rear]=x;//3 4 5 6
    //每放入一个元素，需要对Q->rear增1，因为数组可以访问的最大下标为4，为防止
    //到达数组的末尾，需要对MaxSize取模
    Q->rear=(Q->rear+1)%MaxSize;
    return 1;
}
//出队
int DeQueue(SqQueue *Q,ElemType *x)
{
    //如果front与rear相等，那么代表循环队列为空，这时直接返回
    if(Q->rear==Q->front)
        return 0;
    *x=Q->data[Q->front]; //先进先出
    Q->front=(Q->front+1)%MaxSize; //出队一个元素后，front加1
    return 1;
}

```

下面是一个测试的例子。首先定义一个循环队列Q，然后对其初始化，初始化后判断是否为空，这时会打印循环队列为空。再后，依次入队3、4、5、6、7共五个元素，发现入队7时失败，因为队列已满。这时，我们首先出队两个元素，然后入队元素8，就可以成功。在实际运行代码的过程中，可以通过内存窗口观察内存的变化过程，以便进一步理解循环队列。

```

int main()
{
    SqQueue Q;
    int ret; //存储返回值
    ElemType element; //存储出队元素
    InitQueue(&Q);
    ret=isEmpty(&Q);
    if(ret)
    {
        printf("队列为空\n");
    }else{

```

```
        printf("队列不为空\n");
    }
    EnQueue(&Q,3);
    EnQueue(&Q,4);
    EnQueue(&Q,5);
    ret=EnQueue(&Q,6);
    //因为队列最大长度为 MaxSize-1 个元素，因此 7 入队时失败
    ret=EnQueue(&Q,7);
    if(ret)
    {
        printf("入队成功\n");
    }else{
        printf("入队失败\n");
    }
    ret=DeQueue(&Q,&element);
    if(ret)
    {
        printf("出队成功,元素值为 %d\n",element);
    }else{
        printf("出队失败\n");
    }
    ret=DeQueue(&Q,&element);
    if(ret)
    {
        printf("出队成功,元素值为 %d\n",element);
    }else{
        printf("出队失败\n");
    }
    ret=EnQueue(&Q,8);
    if(ret)
    {
        printf("入队成功\n");
    }else{
        printf("入队失败\n");
    }
    return 0;
}
```

### 8.1.3 二叉树

在计算机科学中，二叉树是指每个结点最多有两个子树的树结构。一般来说，子树分为左子树（left subtree）和右子树（right subtree）。二叉树常被用来实现二叉查找树和二叉堆。

二叉树的每个结点至多有两棵子树（不存在度大于2的结点），二叉树的子树有左右之分，次序不能颠倒。

二叉树的第 $i$ 层至多有 $2^{i-1}$ 个结点；深度为 $k$ 的二叉树至多有 $2^k - 1$ 个结点；对任何一棵二叉树 $T$ ，如果其终端结点数为 $n_0$ ，度为2的结点数为 $n_2$ ，那么 $n_0 = n_2 + 1$ 。

深度为 $k$ 且有 $2^k - 1$ 个结点的树称为满二叉树；深度为 $k$ 且有 $n$ 个结点的二叉树称为完全二叉树，当且仅当每个结点都与深度为 $k$ 的满二叉树中序号为1至 $n$ 的结点对应，如图8.1.2所示。

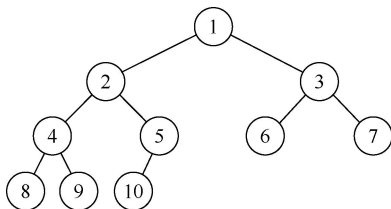


图 8.1.2 完全二叉树

完全二叉树的特点如下：

- (1) 只允许最后一层有空缺结点且空缺在右边，即叶子结点只能在层次最大的两层上出现。
- (2) 对任意一个结点，如果其右子树的深度为 $j$ ，那么其左子树的深度必为 $j$ 或 $j+1$ ，即度为1的点只有1个或0个。

二叉树的存储可以是顺序存储（顺序存储将在介绍堆排序时学习），也可以是链式存储。采用链式存储实现二叉树的例子如例8.1.3所示。

【例 8.1.3】二叉树的实现。

```
#include <stdio.h>
#include <stdlib.h>
//首先要将二叉树结点的数据结构定义清楚
typedef struct node{
    char c; //结点内存的元素类型
    struct node *left; //指向左子结点的指针
    struct node *right; //指向右子结点的指针
}Node,*pNode;

void preOrder(pNode p)
{
    if(p!=NULL)
    {
        putchar(p->c); //打印当前结点
```

```

        preOrder(p->left); //打印左子结点
        preOrder(p->right); //打印右子结点
    }
}

void midOrder(pNode p)
{
    if(p!=NULL)
    {
        midOrder(p->left); //打印左子结点
        putchar(p->c); //打印当前结点
        midOrder(p->right); //打印右子结点
    }
}

void latOrder(pNode p)
{
    if(p!=NULL)
    {
        latOrder(p->left); //打印左子结点
        latOrder(p->right); //打印右子结点
        putchar(p->c); //打印当前结点
    }
}

```

这里建立二叉树的方法是最常用的层次建树法。如下面的代码所示，首先进树的元素是从 A 到 J，共 10 个元素。采用层次建树法，一层放满后，才放下一层，所以采用层次建树法建立的二叉树一定是一棵满二叉树。为了建树方便，我们首先为每个结点申请空间，将每个结点的指针值存入一个指针数组，这样通过 *for* 循环就可以依次得到进树的元素。进树时，第 0 个结点（根部结点）的左子结点为空，因此先放左边，接着放右边。右边也放置结点后，就需要将 *j* 加 1，这样进树的结点将放入下一个结点的左子结点，以此类推，最终实现二叉树的层次建树。

```

#define N 10
int main()
{
    char c[N+1]="ABCDEFGHIJ";
    int i,j;
    pNode a[N];
    //通过 for 循环，为每个要进树的结点申请空间，并将结点值放入

```



```

for(i=0;i<N;i++)
{
    a[i]=(pNode)calloc(1,sizeof(Node)); //申请空间
    a[i]->c=c[i]; //申请空间后, 将对应的元素值填入
}
for(j=0,i=1;i<N;i++) //通过下标 i 控制要进入树的元素
{
    if(NULL==a[j]->left) //没有左子结点, 就放入左子结点
    {
        a[j]->left=a[i];
    }else if(NULL==a[j]->right) //否则放入右子结点
    {
        a[j]->right=a[i];
        j++; //放入右子结点后, 当前结点满, 因此对 j 加 1, 从而下次进树时,
            //放置到下一个结点, 我们通过下标 j 记录进树的位置
    }
}
//通过打印树的前序、中序、后序遍历结果来判断二叉树是否创建正确
printf("前序遍历\n");
preOrder(a[0]);
printf("\n 中序遍历\n");
midOrder(a[0]);\
printf("\n 后序遍历\n");
latOrder(a[0]);
return 0;
}

```

下面来看前序遍历。前序遍历首先打印根结点, 然后打印左结点, 接着打印右结点。对于如图 8.1.3 所示的二叉树, 其前序遍历结果为 **ABC**。

在图 8.1.3 所示的二叉树中添加两个结点, 如图 8.1.4 所示, 单独看 **B** 结点的子树时, 打印顺序为 **BDE**, 这时如果与上面的遍历结果结合, 那么当前二叉树的遍历结果是 **ABDEC**, 遵守的原则是子结点要与自己的父结点相邻。

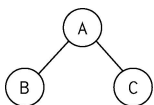


图 8.1.3 二叉树 1

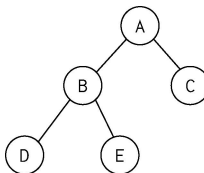


图 8.1.4 二叉树 2

根据上面的推演, 可知依次遍历的结果如下:

```
ABDECFG      //对于 C 结点
ABDHIECFG    //对于 D 结点
ABDHIEJCFG   //对于 E 结点
```

其中, *E* 结点的二叉树如图 8.1.5 所示。

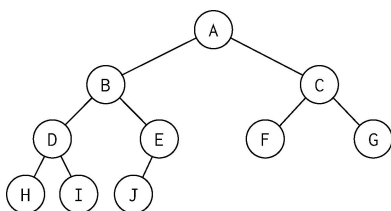


图 8.1.5 二叉树 3

二叉树的建树过程总结如下: 由于需要将每个字母作为一个二叉树的结点进行存储, 因此首先要通过循环为每个结点申请空间, 然后用指针数组存储每个结点对应的指针值。在建树过程中, 循环控制要进入树的元素, 内部判断找到要添加的位置。

**思考题:** 如果建树时, 结点数量的多少不确定, 那么如何修改代码? 在实际工作遇到的问题中, 结点数量大部分时间是不确定的。

上面进行层次建树时, 提前使用了一个指针数组。这样, 元素进树时, 若某个结点已满, 则 *j* 加 1, 得到下一个要放置元素的结点位置。按照这一原理, 如果在结点数量不确定时进行层次建树, 那么就需要使用一个辅助队列, 如图 8.1.6 所示。往树中放一个元素时, 需要将其放入队列尾部, 如图中的元素 *C*。当元素 *C* 放入树中后, 发现元素 *A* 的左右子结点都放满了, 这时辅助队列的头 *queHead* 始终指向要放的结点位置, 因此如图 8.1.6 中的箭头所示, 需要将 *queHead* 移动并指向元素 *B*。这样, 放入元素 *D* 时, 就把元素 *D* 作为元素 *B* 的左子结点放入。

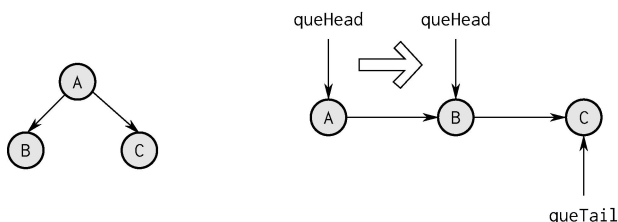


图 8.1.6 层次建树法

例 8.1.4 是具体的代码实现, 代码实现使用了函数 *buildBinaryTree*。每次往树中添加一个元素, 就调用一次 *buildBinaryTree* 函数。当然, 树建好了而不需要辅助队列时, 可以销毁辅助队列。掌握了链表操作的读者应了解销毁辅助队列的方法。

**【例 8.1.4】** 元素个数不限的二叉树建树。

```
#include <stdio.h>
#include <stdlib.h>
```

```

typedef char ElemType; //便于在树中放任意类型的元素

typedef struct node_t{
    ElemType c;
    struct node_t *pleft;
    struct node_t *pright;
}Node_t,*pNode_t;

typedef struct queue_t{
    pNode_t insertPos;
    struct queue_t *pNext;
}Queue_t,*pQueue_t; //辅助队列数据结构，辅助队列中每个元素放的是树中元素结点的
                        //地址值，这样才能快速确定树中的结点

void buildBinaryTree(pNode_t* treeRoot,pQueue_t* queHead,pQueue_t* queTail,ElemType
val)
{
    pNode_t treeNew=(pNode_t)calloc(1,sizeof(Node_t));
    pQueue_t queNew=(pQueue_t)calloc(1,sizeof(Queue_t));
    pQueue_t queCur=*queHead;
    treeNew->c=val;
    queNew->insertPos=treeNew;
    if(NULL==*treeRoot) //当树为空时，新结点作为树根，同时新结点既作为队列头，
                        //又作为队列尾
    {
        *treeRoot=treeNew;
        *queHead=queNew;
        *queTail=queNew;
    }else{
        (*queTail)->pNext=queNew; //新结点通过尾插法放入队列尾
        *queTail=queNew;
        if(NULL==queCur->insertPos->pleft)
        {
            queCur->insertPos->pleft=treeNew;
        }else if(NULL==queCur->insertPos->pright)
        {
            queCur->insertPos->pright=treeNew;
            *queHead=queCur->pNext; //某个结点放满时，队列头后移一个结点，删除

```

```

//原有头部
    free(queCur);
    queCur=NULL;
}
}
}
int main()
{
    ElemType val;
    pNode_t treeRoot=NULL;
    pQueue_t queHead=NULL,queTail=NULL;
    while(scanf("%c",&val)!=EOF)
    {
        if(val=='\n')
        {
            break;
        }
        buildBinaryTree(&treeRoot,&queHead,&queTail,val);
    }
    preOrder(treeRoot); //与之前的例子一致，所以未列出代码，请参考上一个实例
    printf("\n-----\n");
    midOrder(treeRoot);
    printf("\n-----\n");
    lastOrder(treeRoot);
    system("pause");
}

```

### 8.1.4 红黑树

红黑树（Red-Black tree，简称 RB 树）是一种自平衡二叉查找树，是计算机科学中常见的一种数据结构，其典型用途是实现关联数组。红黑树于 1972 年由鲁道夫·贝尔发明，被称为对称二叉 B 树，今天的名字源于 Leo J. Guibas 和 Robert Sedgewick 于 1978 年撰写的一篇论文。红黑树的结构复杂，但其操作有着良好的最坏情况运行时间，且在实践中有着较高的效率：它可以在  $O(\log_2 n)$  时间内完成查找、插入和删除操作，其中的  $n$  是树中结点的数量。

红黑树（RB 树）是二叉树中重要的知识点，因为在操作系统的内核中、C++ 的 STL 和 Java 的数据结构中大量使用了红黑树，红黑树的增删查改的复杂度均为  $\log_2 n$ 。有的读者会说使用自平衡二叉查找树也可以。下面我们来看二者的区别。

以 AVL 树为例，AVL 树是最早被发明的自平衡二叉查找树。在 AVL 树中，任意一个结点对应的两棵子树的最大高度差为 1，因此它也被称为高度平衡树，对其进行查找、插入和删除在平

均和最坏情况下的时间复杂度都是  $O(\log_2 n)$ ，但是增加和删除元素的操作可能需要借助一次或多次树旋转才能实现树的重新平衡。

红黑树相对于 AVL 树的时间复杂度是一样的，但优势是在插入或删除结点时，红黑树实际的调整次数更少，旋转次数更少，因此红黑树插入/删除的效率要高于 AVL 树。大量的中间件产品中使用了红黑树。

红黑树相对于 AVL 树来说，牺牲了部分平衡性以换取插入/删除操作时少量的旋转操作，整体来说性能要优于 AVL 树。

红黑树是每个结点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树的一般要求以外，对于任何有效的红黑树，我们额外增加了如下性质。

- (1) 结点是红色的或黑色的。
- (2) 根是黑色的。
- (3) 所有叶子结点都是黑色的（叶子是 NIL 结点）。
- (4) 每个红色结点必须有两个黑色的子结点（从每个叶子到根的所有路径上不能有两个连续的红色结点）。
- (5) 从任意一个点到其每个叶子的所有简单路径都包含相同数量的黑色结点。

图 8.1.7 所示为一棵具体的红黑树（图中浅色圆圈代表红色，黑色圆圈代表黑色，如结点 13 是黑色的，结点 8 是红色的）。

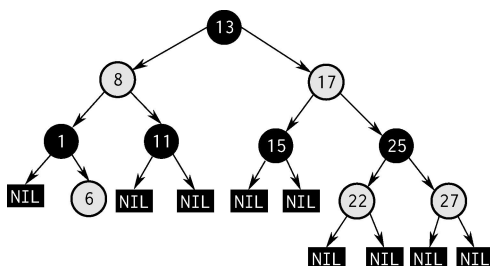


图 8.1.7 红黑树

NIL 结点可以理解为空结点。因此，对于红黑树，重点掌握性质（4）和性质（5）即可，因为在插入和删除操作中，调整的目的主要是为了保证性质（4）和性质（5）。我们可以按照如下方式来辅助记忆：红色代表发脾气，发脾气的人不能在一起，对应性质（4）；黑色代表冷静，世界上冷静的人是均衡的，所以才有世界和平，对应性质（5）。只要从根结点到任意叶子结点的黑色结点数量相等，从任意一个结点到叶子结点的黑色结点数量就是相等的。

这些性质确保了红黑树的关键特性，即从根到叶子的最长可能路径不多于最短可能路径的 2 倍，因此红黑树大致上是平衡的。由于插入、删除和查找某个值的操作的最坏情况时间都要求与树的高度成比例，因此这个高度上的理论上限允许红黑树在最坏情况下都是高效的，这是不同于普通二叉查找树的地方。

要想知道为什么这些性质确保了这个结果，看到性质（4）导致路径不能有两个连续的红色结点就清楚了。最短的可能路径上都是黑色结点，最长的可能路径上交替出现红色和黑色结点。因为根据性质（5），所有最长的路径都有相同数量的黑色结点，这就表明没有路径的长度能多于

任何其他路径的 2 倍。

因为每棵红黑树也是一棵特殊的二叉查找树,因此红黑树上的只读操作与普通二叉查找树上的只读操作是相同的。然而,在红黑树上进行插入操作和删除操作会导致不再匹配红黑树的性质,恢复红黑树的性质需要少量  $[O(\log_2 n)]$  的颜色变更(实际是非常快速的)和不超过三次树旋转(对于插入操作是两次)。虽然插入和删除很复杂,但操作时间仍可以保持为  $O(\log_2 n)$ 。

## 1. 红黑树的插入

首先按二叉查找树的方法增加结点并标记为红色(即任何新增结点都要标记为红色后才添加到树中),在下面的多幅示意图中,将要插入的结点标为  $N$ ,将  $N$  的父结点标为  $P$ ,将  $N$  的祖父结点标为  $G$ ,将  $N$  的叔父结点标为  $U$ 。

针对红黑树的插入,我们分以下 5 种情形(注意不要和红黑树本身的性质混为一谈),情形 1 和情形 2 比较简单,重点是情形 3、情形 4 和情形 5。

**情形 1:** 新结点  $N$  位于树的根上,没有父结点。这种情况下,红黑树没有其他结点,只有根结点,于是直接标记为黑色,或者经过调整后红黑树的根结点已变为红色,就直接把它标记为黑色以满足性质 2。因为在每个路径上对黑结点数量增 1,即满足性质 5。

**情形 2:** 新结点的父结点  $P$  是黑色的,即满足性质 4(新结点是红色的)。在这种情形下,树仍是有效的。同时也满足性质 5: 尽管新结点  $N$  有两个黑色叶子结点,但由于新结点  $N$  是红色的,通过它的每个子结点的路径就都有与通过它取代的黑色叶子的路径同样数量的黑色结点,所以依然满足这个性质。

**情形 3:** 如果父结点  $P$  和叔父结点  $U$  二者都是红色的(此时新插入结点  $N$  作为  $P$  的左子结点或右子结点都属于情形 3,这里仅显示  $N$  作为  $P$  的左子结点的情形),那么我们可以将它们两个重绘为黑色,并重绘祖父结点  $G$  为红色(用来保持性质 5)。现在我们的新结点  $N$  有了一个黑色的父结点  $P$ ,因为通过父结点  $P$  或叔父结点  $U$  的任何路径都必定通过祖父结点  $G$ ,在这些路径上的黑色结点数量没有改变。但是,红色的祖父结点  $G$  可能是根结点,这就违反了性质 2;也有可能祖父结点  $G$  的父结点是红色的,这就违反了性质 4。为了解决这个问题,我们在祖父结点  $G$  上递归地实现情形 1 的整个过程(把  $G$  视为新加入的结点进行各种情形的检查),具体调整如图 8.1.8 所示。

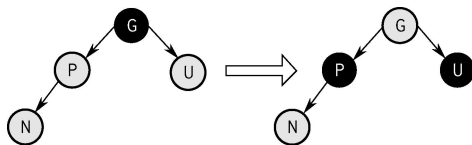


图 8.1.8 情形 3

有的读者可能会有这样的疑问: 如果  $G$  是根结点,那么通过情形 1 直接变为黑色即可;但是,如果  $G$  不是根结点,那么通过  $G$  的父结点也是红色的,此时怎么办?

当  $G$  的父结点也是红色时,就需要重新把  $G$  作为子结点,重新来看  $G$  的父结点、叔父结点、祖父结点的颜色与位置,确认到底是情形 3 还是情形 4 或情形 5,符合哪种情形,就按对应情形进行调整。

**情形 4:** 父结点  $P$  是红色的而叔父结点  $U$  是黑色的或缺少,并且新结点  $N$  是其父结点  $P$  的右子结点,而父结点  $P$  又是其父结点的左子结点。在这种情形下,我们进行一次左旋转调换新结点和其父结点的角色;接着,我们按情形 5 处理以前的父结点  $P$  以解决性质 4 未满足的问题。因为没有增加黑色结点的数量,所以性质 5 仍有效。具体调整如图 8.1.9 所示。

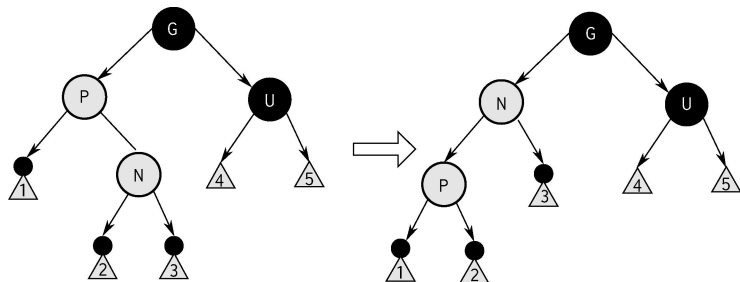


图 8.1.9 情形 4

图 8.1.9 中标注 1、2、3 的位置有 3 个小圆,这是为了便于说明和右边的总黑色结点数量一致,不用过于关注,目的是为了让大家看到  $N$  结点左旋转后  $N$  的孩子、 $P$  的孩子发生的变化。

**情形 5:** 父结点  $P$  是红色的而叔父结点  $U$  是黑色的或缺少,新结点  $N$  是其父结点的左子结点,而父结点  $P$  又是其父结点  $G$  的左子结点。在这种情形下,需要将祖父结点  $G$  右旋转一次,在旋转产生的树中,以前的父结点  $P$  现在是新结点  $N$  和以前的祖父结点  $G$  的父结点。可以知道以前的祖父结点  $G$  是黑色的,否则父结点  $P$  就不可能是红色的(如果  $P$  和  $G$  都是红色的就违反了性质 4,所以  $G$  必须是黑色的)。我们切换以前的父结点  $P$  和祖父结点  $G$  的颜色,结果树满足性质 4。性质 5 也仍然满足:因为通过这三个结点中任何一个的所有路径以前都通过祖父结点  $G$ ,现在它们都通过以前的父结点  $P$ 。在各自的情形下,这都是三个结点中唯一的黑色结点,如图 8.1.10 所示。

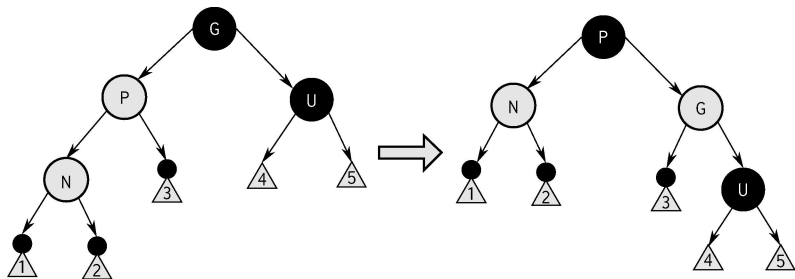


图 8.1.10 情形 5

通过图 8.1.10 中的变化可以看出,对  $G$  也就是祖父结点进行右旋转,同时交换祖父结点  $G$  与父结点  $P$  的颜色,不仅可以满足性质 5 从根结点到任意叶子结点的黑色结点数量不变,同时

满足性质 4，即没有红色结点相邻。

由于情形 3 到情形 5 相对复杂一些，针对上面的情况，图 8.1.11 进行了总结。图 8.1.11（图中浅色代表红色结点，深色代表黑色结点）中的情形 3，叔父结点  $U$  是红色的，无论  $N$  结点是  $P$  结点的左孩子结点还是右孩子结点，都采用变色机制；针对情形 4，叔父结点不存在或是黑色的，就需要对父结点  $P$  进行相应的旋转，旋转后刚好变成了对应自己下方情形 5 的样子，然后对祖父结点  $G$  进行相应的旋转即可。

理解图 8.1.11 以后，再看例 8.1.5 的实现，就会觉得一目了然。核心代码的每部分都进行了详细的注释，分别对应于图 8.1.11 中的情形 3、情形 5、情形 5，`rbtree_insert_fixup` 是实现红黑树的关键函数。

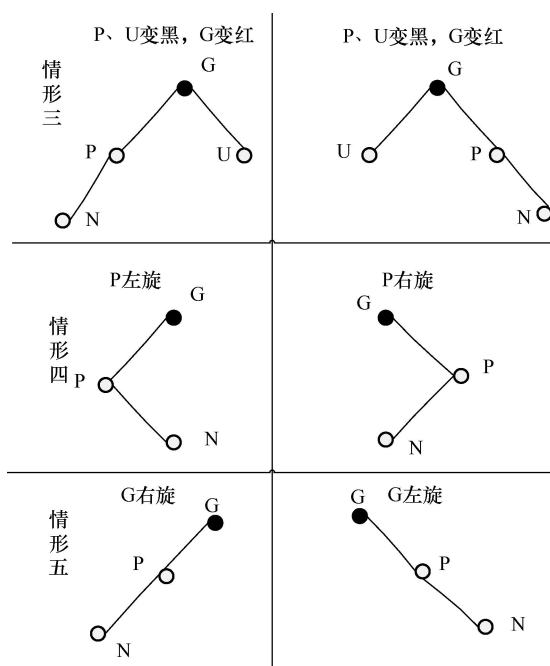


图 8.1.11 红黑树插入总结

【例 8.1.5】红黑树的插入修正函数。

```
/*
 * 红黑树插入修正函数
 *
 * 向红黑树中插入结点后（失去平衡），再调用该函数；
 * 目的是将它重新塑造成一棵红黑树。
 *
 * 参数说明：
 *   root 红黑树的根
 *   node 插入的结点    //对应《算法导论》中的 z
```



```

*/
static void rbtree_insert_fixup(RBRoot *root, Node *node)
{
    Node *parent, *gparent;
    // 若“父结点存在，并且父结点的颜色是红色的”，就说明需要调整！
    while ((parent = rb_parent(node)) && rb_is_red(parent))
    {
        gparent = rb_parent(parent);
        //若“父结点”是“祖父结点的左孩子结点”
        if (parent == gparent->left)
        {
            //Case 1 条件：叔父结点存在且是红色的，这里是情形 3
            {
                Node *uncle = gparent->right;
                if (uncle && rb_is_red(uncle))//没有结点进入该分支，如何构造？
                {
                    rb_set_black(uncle);
                    rb_set_black(parent);
                    rb_set_red(gparent);
                    node = gparent;
                    continue;
                }
            }
            //Case 2 条件：叔父结点是黑色的或不存在，且当前结点是右孩子结点，这里是情形 4
            if (parent->right == node)
            {
                Node *tmp;
                rbtree_left_rotate(root, parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }
            //Case 3 条件：叔父结点是黑色的，且当前结点是左孩子结点。这里是情形 5
            rb_set_black(parent); //旋转前设置好颜色
            rb_set_red(gparent); //旋转前设置好颜色
            rbtree_right_rotate(root, gparent);
        }
        else//若父结点是祖父结点的右孩子结点，则下面三种和上面的是对称的

```

```

{
    //Case 1 条件: 叔父结点是红色的。这里是情形 3
    {
        Node *uncle = gparent->left;
        if (uncle && rb_is_red(uncle))
        {
            rb_set_black(uncle);
            rb_set_black(parent);
            rb_set_red(gparent);
            node = gparent;
            continue; //继续进行调整
        }
    }
    //Case 2 条件: 叔父结点是黑色的, 且当前结点是左孩子结点。这里是情形 4
    if (parent->left == node)
    {
        Node *tmp;
        rbtree_right_rotate(root, parent);
        tmp = parent;
        parent = node;
        node = tmp;
    }
    //Case 3 条件: 叔父结点是黑色的, 且当前结点是右孩子结点。这里是情形 5
    rb_set_black(parent); //旋转前设置好颜色
    rb_set_red(gparent); //旋转前设置好颜色
    rbtree_left_rotate(root, gparent);
}
}
//将根结点设为黑色
rb_set_black(root->node);
}

```

## 2. 红黑树的删除

如果需要删除的结点有两个子结点, 那么问题可以转化成删除另一个只有一个子结点的问题 (为了表述方便, 这里所指的子结点是叶子结点的子结点)。对于二叉查找树, 在删除带有两个非叶子结点的子结点时, 我们要么找到其左子树中的最大元素, 要么找到其右子树中的最小元素,

并把它值转移到要删除的结点中。例如，如果要删除图 8.1.12 中的结点 13，那么可以把左子树的结点 11 或右子树的结点 15 与结点 13 交换。

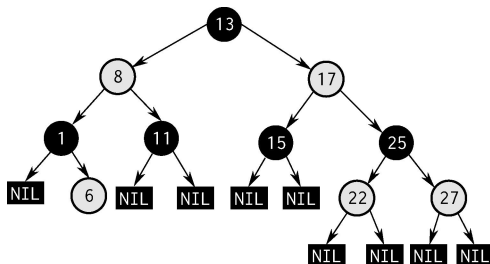


图 8.1.12 红黑树的删除情形 1

接着删除我们从中复制出值的那个结点，它必定有少于两个非叶子结点的子结点（即最多只有一个子结点，例如结点 15 有一个红色的右子结点 16）。因为只复制了一个值（没有复制颜色），不破坏任何性质，这就把问题简化为如何删除最多有一个儿子结点的问题，而不需要关心这个结点是最初要删除的结点还是我们从中复制出值的那个结点。

在本文余下的部分中，我们只需讨论如何删除只有一个子结点的结点（如果它的两个子结点都为空，即均为叶子结点，那么我们将其中的任意一个结点视为它的子结点）。如果我们删除一个红色结点（此时该结点的子结点将都为叶子结点），那么它的父结点和子结点一定是黑色的，所以我们可以简单地用它的黑色子结点替换它，而不会破坏性质 3 和性质 4。通过被删除结点的所有路径只是少了一个红色结点，这样可以继续保证性质 5。另一种简单的情况是，在被删除结点是黑色的而其子结点是红色的时候，如果只删除这个黑色结点，用它的红色子结点顶替，那么会破坏性质 5，但是，如果我们将它的儿子结点重绘为黑色，那么曾经通过它的所有路径将通过它的黑色子结点，这样就可以继续满足性质 5。

需要进一步讨论的是，在要删除的结点和它的子结点都是黑色的时候，这是一种复杂的情况（这种情况下该结点的两个子结点都是叶子结点，否则若其中一个子结点是黑色的非叶子结点，另一个儿子是叶子结点，则从该结点通过非叶子结点的子结点的路径上的黑色结点数最小为 2，而从该结点到另一个叶子结点的子结点的路径上的黑色结点数为 1，违反了性质 5）。我们首先把要删除的结点替换为它的子结点，为方便起见，我们称这个子结点为 *N*（在新的位置上），称它的兄弟结点（它的父结点的另一个子结点）为 *S*。在下面的示意图中，我们仍使用 *P* 作为 *N* 的父结点，使用 *SL* 作为 *S* 的左子结点，使用 *SR* 作为 *S* 的右子结点，并使用如下函数找到兄弟结点：

**如果 *N* 和它初始的父结点是黑色的，那么删除它的父结点会导致通过 *N* 的路径都比不通过它的路径少一个黑色结点，这违反了性质 5，树需要被重新平衡。有 6 种情形（注意不要和红黑树的性质混为一谈）需要考虑。在下列情形的示意图中，我们用 *child* 替换了 *N*，因此示意图中 *N* 所在的分支比不通过 *N* 的分支少一个黑色结点。**

**情形 1：***N* 是新的根。在这种情形下，我们从所有路径删除了一个黑色结点，而新根是黑色的。

**注意：**在情形 2、5 和 6 下，我们假定 *N* 是其父结点的左子结点；如果它是右子结点，那么

在这些情形下的左子结点和右子结点应当对调。

**情形 2:**  $S$  是红色的。如图 8.1.13 所示, 在这种情形下我们在  $N$  的父结点上做左旋转, 把红色兄弟结点转换成  $N$  的祖父结点, 接着对调  $N$  的父结点和祖父结点的颜色。完成这两个操作后, 尽管所有路径上黑色结点的数量没有改变, 但现在  $N$  有了一个黑色的兄弟结点和一个红色的父结点 (它的新兄弟结点是黑色的, 因为它是红色结点  $S$  的一个子结点), 所以我们接下来可以按情形 4、情形 5 或情形 6 来处理。

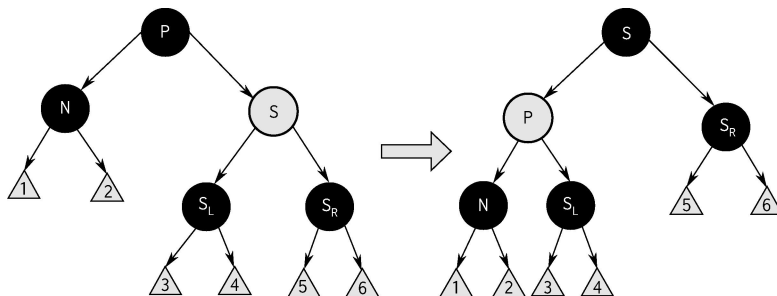


图 8.1.13 红黑树的删除情形 2

**注意:** 图 8.1.13 中并未显示  $N$  是删除了黑色结点 (假如删除的黑色结点为  $X$ ) 后替换上来的子结点, 所以这个过程由  $P \rightarrow X \rightarrow N$  变成了  $P \rightarrow N$ , 实际上少了一个黑色结点, 也可以理解为  $Parent(Black)$  和  $Silbing(Red)$ , 于是它们的黑色子结点的数量肯定不等, 让它们做兄弟结点肯定是不平衡的, 还需继续处理。如果  $N$  结点与  $S_L$  结点做兄弟, 那么通过  $N$  结点的分支肯定比通过  $S_L$  结点的分支要少一个黑色结点, 本来通过  $N$  结点数量与  $S_L$  结点的黑色结点数量是相等的, 因为我们的图是  $N$  结点的分支, 是  $N$  结点的 *child* 将  $N$  结点替换后的, 所以比  $S_L$  结点的分支这边少一个黑色结点。

**情形 3:**  $N$  的父结点、 $S$  结点和  $S$  结点的子结点都是黑色的, 如图 8.1.14 所示。在这种情形下, 我们简单地将  $S$  重绘为红色, 结果是通过  $S$  的所有路径 (即以前不通过  $N$  结点的那些路径) 都少了一个黑色结点。因为删除  $N$  结点的初始父结点使通过  $N$  结点的所有路径少了一个黑色结点, 使得所有结点都得到了平衡。但是, 若把通过  $P$  结点的整体视为一棵右子树或左子树, 则通过  $P$  结点的所有路径现在比不通过  $P$  结点的路径 (另外一边的子树) 少了一个黑色结点, 所以仍然违反性质 5。要修正这个问题, 我们就要从情形 1 开始, 在  $P$  结点上重新做平衡处理, 也就是把  $P$  结点视为一个子结点, 即把  $P$  结点视为  $N$  结点重新做平衡处理。

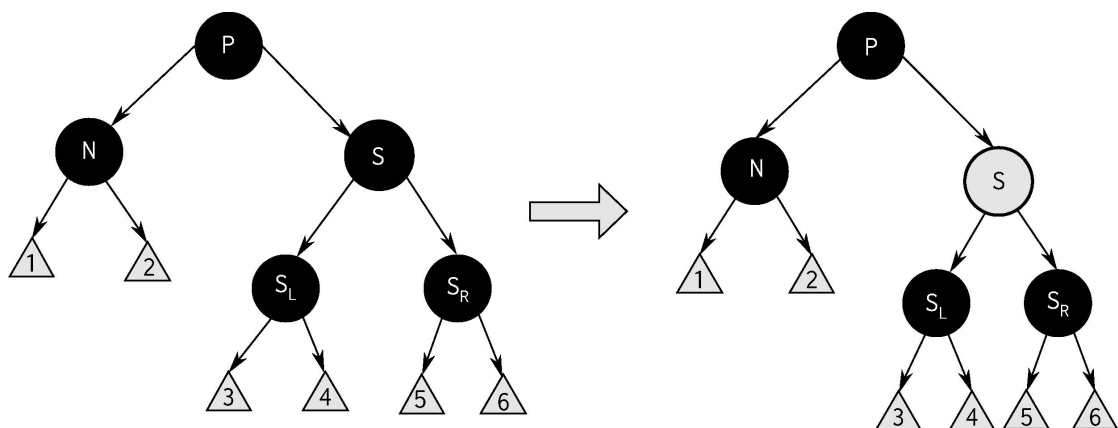


图 8.1.14 红黑树的删除情形 3

如果不符合情形 3，那么就接着匹配情形 4。

**情形 4：**  $S$  结点和  $S$  结点的子结点都是黑色的，但是  $N$  结点的父结点是红色的，如图 8.1.15 所示。在这种情形下，我们简单地交换  $N$  结点的兄弟结点和父结点的颜色。这不影响不通过  $N$  的路径的黑色结点的数量，但是它在通过  $N$  结点的路径上对黑色结点的数量增 1，添加了在这些路径上删除的黑色结点。

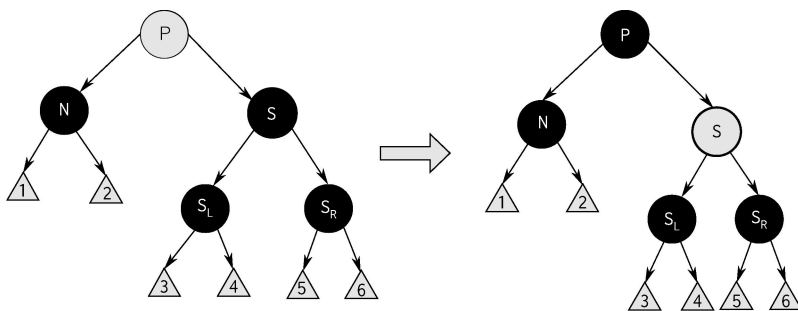


图 8.1.15 红黑树的删除情形 4

**情形 5：**  $S$  结点是黑色的， $S$  结点的左子结点是红色的， $S$  结点的右子结点是黑色的，而  $N$  结点是其父结点的左子结点，如图 8.1.16 所示。在这种情形下，于结点  $S$  上做右旋转，使得  $S$  结点的左子结点成为  $S$  结点的父结点和  $N$  结点的新兄弟结点。接着交换  $S$  结点和其新父结点的颜色，所有路径仍有同样数量的黑色结点，但现在  $N$  结点有了一个黑色兄弟结点，它的右子结点是红色的，所以我们进入了情形 6， $N$  结点和其父结点都不受这个变换的影响。

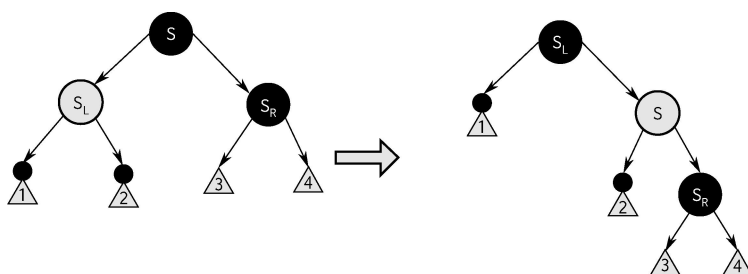


图 8.1.16 红黑树的删除情形 5

**情形 6:**  $S$  结点是黑色的,  $S$  结点的右子结点是红色的, 而  $N$  结点是其父结点的左子结点, 如图 8.1.17 所示。在这种情形下, 我们在  $N$  的父结点上做左旋转, 使  $S$  成为  $N$  的父结点 ( $P$ ) 及其右子结点的父结点。接着交换  $N$  的父结点  $P$  (无论  $P$  是黑色的还是红色的都可以, 所以图 8.1.17 中标注  $R/B$  (red 或 black 可) 和  $S$  结点的颜色, 并使  $S$  结点的右子结点为黑色。子树在其根上仍有同样的颜色, 所以未违反性质 3。但是,  $N$  结点现在增加了一个黑色的祖先结点: 要么  $N$  结点的父结点变成黑色, 要么它是黑色的而  $S$  结点增加一个黑色的祖父结点。所以, 通过  $N$  结点的路径都增加了一个黑色结点。

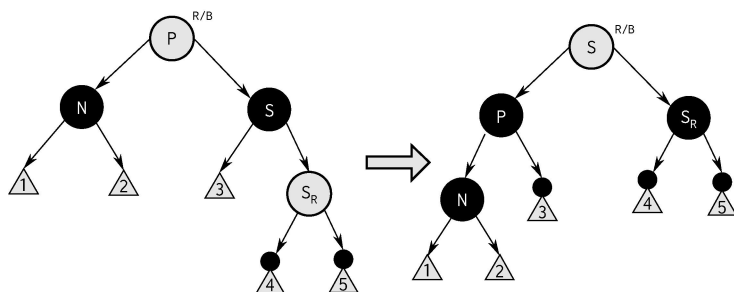


图 8.1.17 红黑树的删除情形 6

在任何情况下, 在这些路径上的黑色结点数量都未改变, 所以满足性质 4。图 8.1.17 中的  $R/B$  结点可以是红色的或黑色的, 但在变换前后都必须指定相同的颜色。

通过上面 6 种情形的分析, 由于情形 2~6 相对复杂, 所以我们进行了如图 8.1.18 所示的总结分析 (图中黑色代表黑色结点, 浅色代表红色结点, 方形代表红色或黑色结点), 由该图可以发现如下规律: 对于情形 2,  $N$  结点的兄弟结点  $S$  如果为红色, 那么对父结点  $P$  进行左旋转, 这时由于  $N$  结点与  $S_L$  结点两边的黑色结点数量不相等, 需要再次进行情形 4、5、6 中的处理。

情形 2 分析的  $S$  结点是红色的, 如果  $S$  结点不是红色的而是黑色的, 那么就要分为情形 4、情形 5、情形 6 三种情况:  $S$  结点的两个孩子结点都是黑色的,  $S$  的左子结点是红色的 (右子结点的颜色任意或不存在),  $S$  的右子结点是红色的 (左子结点的颜色任意或不存在)。情形 3 和情形 4 差别不大, 父结点  $P$  分别是黑色的或红色的。针对情形 3, 把  $S$  结点变为红色后, 通过  $N$  结点和通过  $S$  结点的左右子树黑色结点数相等。但是, 如果  $P$  结点不是根结点, 那么通过  $P$  结

点这边子树的黑色结点数会少 1，所以将  $P$  结点作为  $N$  结点重新再做调整即可。情形 5 的目的是变为情形 6，情形 6 让  $N$  结点的分支多了一个黑色结点，但是  $S_R$  结点分支的黑色结点数量不变。情形 5 和情形 6 中，结点画成矩形代表黑色或红色，情形 6 旋转后，保持  $S$  结点的颜色和原有  $P$  结点的颜色一致即可。

图 8.1.18 中的浅色方框既可表示黑色又可表示红色，只要保证旋转前后对应位置的颜色一致即可（情形 5 中  $P$  结点的颜色要与  $S$  结点右旋后的颜色保持一致；情形 6 中  $S$  结点的颜色要和  $P$  结点左旋之前的颜色一致）。

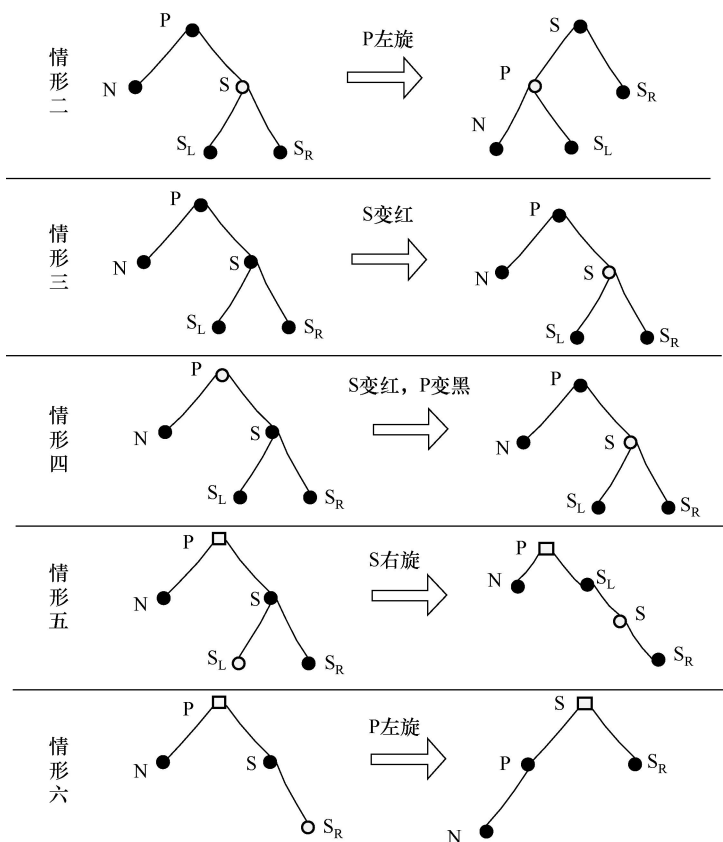


图 8.1.18 红黑树的删除情形总结图

要学习任何数据结构或算法，理解其原理后阅读核心代码必不可少。例 8.1.6 是红黑树删除的核心代码。当然，下面的代码近 100 行，需要在理解图 8.1.18 的基础上进行阅读，只有这样才能快速理解，同时在面试时顺利地写出代码。面试依靠记忆写出代码是不靠谱的，一定要理解，要凭自己的理解写出代码。例如，可以首先画出图 8.1.18，然后根据图形来写代码。如果实际面试时笔试时间不够，那么可以直接画出对应的图形，进入面对面交谈时，向面试官讲解即可。

红黑树的整体代码较多，加上测试代码总计约有 700 行，全部附在书中不合适，需要全部代码及讲解视频的读者可通过 QQ 群获取。

【例 8.1.6】红黑树删除修正函数。

```

/*
 * 红黑树删除修正函数部分的核心代码
 *
 * 从红黑树中删除插入结点后（红黑树失去平衡），再调用该函数；
 * 目的是将它重新塑造成一棵红黑树。
 *
 * 参数说明：
 *     root 红黑树的根
 *     node 待修正的结点
 */
static void rbtree_delete_fixup(RBRoot *root, Node *node, Node *parent)
{
    Node *other; //other 代表图中的 S 结点
    //下面的 N 结点说的是 node 结点
    while ((!node || rb_is_black(node)) && node != root->nnode)
    {
        if (parent->left == node)
        {
            //删除
            other = parent->right;
            if (rb_is_red(other))
            {
                // Case 1: N 结点的兄弟结点 S 是红色的，对应前面图中的情形 2
                rb_set_black(other);
                rb_set_red(parent);
                rbtree_left_rotate(root, parent);
                other = parent->right; //旋转后 other 结点变为图中的 SL 结点
            }
            if ((!other->left || rb_is_black(other->left)) &&
                (!other->right || rb_is_black(other->right)))
            {
                // Case 2: N 结点的兄弟结点 S 是黑色的，且 S 结点的两个孩子结点也是黑色的
                //对应图中的情形 3 和情形 4，针对情形 4 的父结点变为黑色是在 while
                //循环外进行的
                rb_set_red(other);
                node = parent;
                parent = rb_parent(node);
            }
        }
    }
}

```



```

else
{
    if (!other->right || rb_is_black(other->right))
    {
        // Case 3: N 结点的兄弟结点 S 是黑色的，并且 S 结点的右子结点是黑色的
        // 对应图中的情形 5
        rb_set_black(other->left);
        rb_set_red(other);
        rbtree_right_rotate(root, other);
        other = parent->right;
    }
    // Case 4: N 结点的兄弟结点 S 是黑色的，且 S 结点的右子结点是红色
    // 的，左子结点为任意颜色，对应图中的情形 6
    rb_set_color(other, rb_color(parent));
    rb_set_black(parent);
    rb_set_black(other->right);
    rbtree_left_rotate(root, parent);
    node = root->node;
    break;
}
}
else
    // 相对于上面的 if 语句，下面的 case1-case4 是上面 case1-case4 的对称场景
    // 删除
    other = parent->left;
    if (rb_is_red(other))
    {
        // Case 1: N 结点的兄弟结点 S 是红色的
        rb_set_black(other);
        rb_set_red(parent);
        rbtree_right_rotate(root, parent);
        other = parent->left;
    }
    if ((!other->left || rb_is_black(other->left)) &&
        (!other->right || rb_is_black(other->right)))
    {
        // Case 2: N 结点的兄弟结点 S 是黑色的，且 S 结点的两个孩子结点也是黑色的
        rb_set_red(other);
    }
}

```

```

        node = parent;
        parent = rb_parent(node);
    }
    else
    {
        if (!other->left || rb_is_black(other->left))
        {
            // 删除 99 时, S 结点的左子结点应是黑色的
            // Case 3: N 结点的兄弟结点 S 是黑色的, 且 S 结点的左子结点是黑色的
            rb_set_black(other->right);
            rb_set_red(other);
            rbtree_left_rotate(root, other);
            other = parent->left;
        }
        // Case 4: N 结点的兄弟结点 S 是黑色的, 且 S 结点的右子结点是红
        // 色的, 左子结点为任意颜色
        rb_set_color(other, rb_color(parent));
        rb_set_black(parent);
        rb_set_black(other->left);
        rbtree_right_rotate(root, parent);
        node = root->node;
        break;
    }
}
}
if (node)
    rb_set_black(node);
}

```

### 删除图形总结:

s 为红其他为黑, 父亲要旋转变色

全黑时, s 变红, p 红 s 黑, 颜色互换 (情形三, 情形四)

sl 为红时, s 要右旋 (p s sl 为折线), 颜色交换 (与新增的类似)

P, s, sr 在一条线, p 左旋, p 和 s 颜色互换 (与新增的类似)

### 8.1.5 数据结构学习技巧

下面介绍学习数据结构和算法的技巧。学习红黑树时, 不仅要了解上面的原理, 而且要自己画图总结其中的原理, 同时下载并运行红黑树的源码。插入一个结点时, 可以首先自己画图, 看

看最终的图会变成什么样子，然后运行代码，了解代码的实际输出效果是否与自己预期的图形效果一致。如果一致，那么再单步调试，看看代码运行了哪部分，以便将对应部分的代码实现与原理讲解对应起来；如果不一致，那么打印实际运行的代码，看看对应原理的那部分，如果对应得上，那么接着看代码的运行效果；如果对应不上，那么要用到前面的调试方法，在调试窗口中，每次单步运行，以了解数据的变化。如果是数组等大块连续内存，那么采用内存窗口进行查看；如果是链表、红黑树等链式实现的数据结构，那么通过监视逐级打开来查看。

在阅读代码的过程中，要及时为每块能看懂的代码添加备注，以便再次使用、阅读时能够迅速看懂。其实，不仅是学习数据结构时要这样做，以后学习各种中间件的源码时，也可以这样做。在阅读源码时，可以针对各个模块的关系画图进行梳理，或者画出某块核心功能的流程图。这些手段都是帮助我们学习中间件的好方法。当然，这样做还可以帮助我们在以后的面试过程中，非常有条理地回答面试官的问题。

## 8.2 算法

### 8.2.1 时间复杂度与空间复杂度

在计算机科学中，算法的时间复杂度是一个函数，它定量地描述了该算法的运行时间，这是一个代表算法输入值的字符串的长度的函数。时间复杂度常用大  $O$  符号表述，不包括这个函数的低阶项和首项系数。使用这种方式时，时间复杂度被称为可渐近的，即考查输入值大小趋近无穷时的情况。例如，如果一个算法对于任何大小为  $n$ （必须比  $n_0$  大）的输入，至多需要  $5n^3 + 3n$  的时间运行完毕，那么它的渐近时间复杂度是  $O(n^3)$ 。为了计算时间复杂度，我们通常会估计算法的操作单元数量，每个单元运行的时间都是相同的。因此，总运行时间和算法的操作单元数量最多相差一个常量系数。

相同大小的不同输入值仍可能造成算法的运行时间不同，因此我们通常使用算法的最坏情况复杂度，记为  $T(n)$ ，其他的算法时间复杂度如图 8.2.1 所示。

若对于一个算法， $T(n)$  的上界与输入大小无关，则称其具有常数时间，记为  $O(1)$ 。

若算法的  $T(n) = O(\log n)$ ，则称其具有对数时间。由于计算机使用二进制数，因此对数常以 2 为底，即  $\log_2 n$ ，有时写为  $\text{lb } n$ 。具有对数时间的常用算法有二叉树的相关操作和二分搜索。

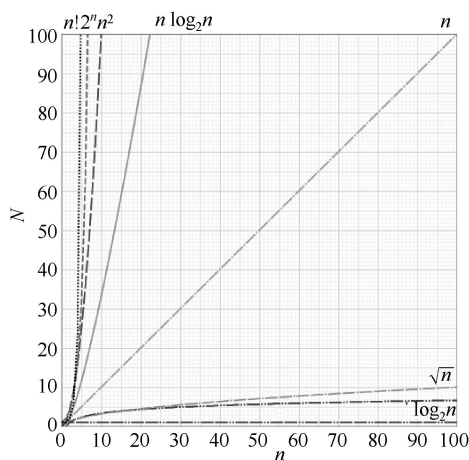


图 8.2.1 时间复杂度

如果一个算法的时间复杂度为  $O(n)$ ，那么称这个算法具有线性时间或  $O(n)$  时间，如无序数组的搜索。

若一个算法的时间复杂度  $T(n) = O(n \log n)$ ，则称这个算法具有线性对数时间，如快速排序、堆排序。

空间复杂度 (Space Complexity) 是对一个算法在运行过程中临时占用存储空间大小的量度 (输入数据本身所占用的空间不计算)，记为  $S(n) = O(f(n))$ ，例如插入排序的空间复杂度为  $O(1)$ 。

我们要能够快速算出时间复杂度，因为这是面试的高频问题。空间复杂度相对面试得少一些。下面我们通过学习排序算法，来掌握时间复杂度及空间复杂度的计算。

## 8.2.2 排序算法

一般来说，我们需要掌握 8 种排序算法，依次是冒泡排序、选择排序、插入排序、希尔排序、快速排序、堆排序、归并排序和计数排序。

排序算法分为以下 5 类。

- 插入类：插入排序，希尔排序
- 选择类：选择排序，堆排序
- 交换类：冒泡排序，快速排序
- 归并类：归并排序
- 分配类：基数排序、计数排序、桶排序，用额外的空间来“分配”和“收集”，继而实现排序，它们的时间复杂度可达到线性阶  $O(n)$ 。

实际面试得最多的排序是快速排序、堆排序和计数排序，因此我们要非常熟练这三种排序算法！

学习排序算法时，可以结合动画提升学习效果，这里推荐的动画网址是 [https://www.cs.usfca.edu/~galles/visualization/Algorithms.html?tdsourcetag=s\\_pcqq\\_aiomsg](https://www.cs.usfca.edu/~galles/visualization/Algorithms.html?tdsourcetag=s_pcqq_aiomsg)。

表 8.2.1 给出了各种排序算法的时间复杂度、空间复杂度、稳定性和复杂性。

表 8.2.1 各种排序算法的时间复杂度、空间复杂度、稳定性和复杂性

排序方式	时间复杂度			空间复杂度	稳 定 性	复 杂 性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n\log_2n)$	$O(n^2)$	$O(n\log_2n)$	$O(\log_2n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n\log_2n)$	$O(n\log_2n)$	$O(n\log_2n)$	$O(1)$	不稳定	较复杂
归并排序	$O(n\log_2n)$	$O(n\log_2n)$	$O(n\log_2n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定	较复杂

不稳定并不是指排序不成功，而是指相同大小的数字在排序后发生了位置交换，因此称为不稳定。

下面来看各种算法实现的代码实例。首先在头文件中定义

```
#define N 10
#define SWAP(a,b) {int tmp;tmp=a;a=b;b=tmp;}
```

即数组的长度为 10，也可修改该值来增加长度，同时 SWAP 交换采用的是宏替换，宏替换不仅比函数的效率高，而且避免了重复代码。

1. 冒泡排序（运行展示）

假设有 10 个无序的数，要求从小到大排序。冒泡法排序的思想是首先对 10 个数通过一轮比较，把最大的数放在最后面，然后对剩下的 9 个数再进行一次比较，把最大数的放在最后面，以此类推，最终实现数组有序。我们通过外层循环控制无序数的数量，通过内层循环控制比较，每次拿 j 和 j+1 进行比较，也就是拿相邻的两个数进行比较，如果发现数 j 大于数 j+1，那么就进行交换。表 8.2.2 中列出了第一轮发生交换时数的变化情况（未列出未发生交换的情况）。

表 8.2.2 冒泡排序过程

最初	3 8 7 2 9 3 7 8 5 6 1 3 8 12 40
第 1 次交换	3 2 8 7 9 3 7 8 5 6 1 3 8 12 40
第 2 次交换	3 2 8 7 7 8 9 3 5 6 1 3 8 12 40
第 3 次交换	3 2 8 7 7 8 5 6 9 3 1 3 8 12 40
第 4 次交换	3 2 8 7 7 8 5 6 6 1 9 3 8 12 40
第 5 次交换	3 2 8 7 7 8 5 6 6 1 3 8 9 3 12 40
第 6 次交换	3 2 8 7 7 8 5 6 6 1 3 8 12 9 3 40
第 7 次交换	3 2 8 7 7 8 5 6 6 1 3 8 12 40 9 3

如何分析确定内层的判断边界呢？是 j<i 还是 j<=i？首先 i 是 N-1，即 i 的值最大值为 9，因为内层是 j 和 j+1 进行比较，数组下标从零开始，最后两个元素依次是 a[8]和 a[9]，因此 j 最大取到 8 即可，所以边界是 j<i；针对外层边界到底是 i>0 还是 i>=0 的问题，当 i 等于 1 时，

对内层是  $j < 1$ ，然后刚好使得  $a[0]$  和  $a[1]$  发生比较，也就是  $i = 1$  时，已经确保了数组最后两个元素发生了比较，因此选择  $i > 0$  即可。

时间复杂度其实就是程序实际的运行次数，可以看到内层是  $j < i$ ，外层  $i$  的值是从  $N-1$  到 1，所以程序的总运行次数是  $1+2+3+\dots+(N-1)$ ，即从 1 一直加到  $N-1$ ，这是等差数列求和，得到的结果是  $N(N-1)/2$ ，即总计运行了这么多次，忽略了低阶项和高阶项的首项系数，因为时间复杂度为  $O(n^2)$ 。因为未使用额外的空间（额外空间必须与输入元素的个数  $N$  相关），所以空间复杂度为  $O(1)$ 。具体代码如下所示：

```
void arr_bubble(int a[])
{
    int i, j;
    for(i=N-1; i>0; i--) //外层是无序数的数量
    {
        for(j=0; j<i; j++) //设定比较策略
        {
            if(a[j]>a[j+1])
            {
                SWAP(a[j], a[j+1])
            }
        }
    }
}
```

**思考题：**如果要按从大到小的顺序排列，那么需要怎么修改代码？

## 2. 选择排序（运行展示）

上面的冒泡排序在内层比较时，我们发现较大的数就交换，这会造成交换的频次过高。如果通过一个变量来记录最大值，一轮比较下来，将最大值再和最后的元素交换，那么这样一轮比较就只需要交换一次。选择排序使用这种思想对冒泡排序进行了改进，下面的代码依然实现的是从小到大排序，外层循环依然记录的是无序数的情况，但这次我们每轮遍历找到最小值，让最小值和最前面的元素发生交换，循环进行，最终使得数组有序。

首先假定第零个元素是最小的，把下标 0 赋值给 `min_pos`，内层比较时，从 1 号元素一直比较到 9 号元素，谁更小，就把它的下标赋给 `min_pos`，一轮比较结束后，将 `min_pos` 对应位置的元素与元素  $i$  交换，如表 8.2.3 所示。第一轮确认 2 最小，将 2 与数组开头的元素 3 交换。第二轮我们最初认为 87 最小，经过一轮比较，发现 3 最小，这时将 87 与 3 交换。持续进行，最终使数组有序。

表 8.2.3 选择排序过程

最初	3 87 2 93 78 56 61 38 12 40
第 1 轮比较后标记	3 87 2 93 78 56 61 38 12 40

第 1 次交换	2 87 <b>3</b> 93 78 56 61 38 12 40
第 2 轮比较后标记	2 87 <b>3</b> 93 78 56 61 38 12 40
第 2 次交换	2 <b>3</b> 87 93 78 56 61 38 12 40
第 3 轮比较后标记	2 3 87 93 78 56 61 38 <b>12</b> 40
第 3 次交换	2 3 <b>12</b> 93 78 56 61 38 <b>87</b> 40
.....	

经验表明有 10 个数时，选择排序的效率最佳。然而，实际工作中需要排序的数的数量较大，只有 10 个数时我们并不在意使用哪种排序算法，因为时间很短。

选择排序虽然减少了交换次数，但是循环比较的次数依然和冒泡排序的数量是一样的，都是从 1 加到  $N-1$ ，总运行次数为  $N(N-1)/2$ 。我们忽略了循环内语句的数量，因为我们在计算时间复杂度时，主要考虑与  $N$  有关的循环，如果循环内交换得多，例如有 5 条语句，那么最终得到的无非是  $5n^2$ ；循环内交换得少，例如有 2 条语句，那么得到的就是  $2n^2$ ，但是时间复杂度计算是忽略首项系数的，因此最终还是  $O(n^2)$ 。因此，选择排序的时间复杂度依然为  $O(n^2)$ 。因为未使用额外的空间（额外空间必须与输入元素的个数  $N$  相关），所以空间复杂为  $O(1)$ 。具体代码如下所示：

```
void arr_select(int *a)
{
    int i,j,min_pos;
    for(i=0;i<N-1;i++) //控制无序数的数量
    {
        min_pos=i;
        for(j=i+1;j<N;j++) //设定比较策略
        {
            if(a[min_pos]>a[j])
            {
                min_pos=j;
            }
        }
        SWAP(a[i],a[min_pos]);
    }
}
```

**思考题：**如果要按从大到小的顺序排列，那么需要怎么修改代码？

### 3. 插入排序（运行展示）

如果一个序列只有一个数，那么该序列自然是有序的。插入排序首先将第一个数视为有序序列，然后把后面 9 个数视为要依次插入的序列。首先，我们通过外层循环控制要插入的数，用 `insertVal` 保存要插入的值 87，这时 `j` 的值为 0，我们比较 `arr[0]` 是否大于 `arr[1]`，即 3 是否大于 87，由于不大于，因此不发生移动，这时有序序列是 3, 87。接着，将数值 2 插入有序序列，首先将 2 赋给 `insertVal`，这时判断 87 是否大于 2，因为 87 大于 2，所以将 87 向后移动，将 2 覆盖，

然后判断 3 是否大于 2, 因为 3 大于 2, 所以 3 移动到 87 所在的位置, 这时  $j$  已经等于零, 再将  $j$  减 1, 得  $j$  为 -1, 内层循环结束, 这时将 2 赋给  $arr[j+1]$ , 即  $arr[0]$  的位置, 得到表 8.2.4 中第二次插入后的效果。

表 8.2.4 插入排序过程

	有序序列	要插入的数的序列
最初	3	87 2 93 78 56 61 38 12 40
第 1 次插入	3 87	2 93 78 56 61 38 12 40
第 2 次插入	2 3 87	93 78 56 61 38 12 40
.....	.....	.....

继续循环会将数依次插入有序序列, 最终使得整个数组有序。插入排序主要用在部分数有序的场景, 例如手机通讯录时时刻刻都是有序的, 新增一个电话号码时, 以插入排序的方法将其插入原有的有序序列, 这样就降低了复杂度。

随着有序序列的不断增加, 插入排序比较的次数也会增加, 插入排序的执行次数也是从 1 加到  $N-1$ , 总运行次数为  $N(N-1)/2$ , 时间复杂度依然为  $O(n^2)$ 。因为未使用额外的空间 (额外空间必须与输入元素的个数  $N$  相关), 所以空间复杂为  $O(1)$ 。具体代码如下所示:

```
void arrInsert(int *arr)
{
    int i,j,insertVal;
    for(i=1;i<N;i++) //控制要插入的数
    {
        insertVal=arr[i];
        for(j=i-1;j>=0&&arr[j]>insertVal;j--=1) //内层控制比较
        {
            arr[j+1]=arr[j];
        }
        arr[j+1]=insertVal;
    }
}
```

思考题: 如果要按从大到小的顺序排列, 那么如何修改代码?

#### 4. 希尔排序 (跳过)

希尔排序 (Shell's Sort) 是插入排序的一种, 又称缩小增量排序 (Diminishing Increment Sort), 是直接插入排序算法的一种更高效的改进算法。希尔排序是非稳定排序算法, 以其设计者希尔 (Donald Shell) 的名字命名, 该算法在 1959 年首次被提出。

通过代码对比可以发现, 相对于插入排序, 希尔排序增加了一层循环, 也就是步长  $gap$ ,  $i$  每次以  $gap$  位置作为插入的起始点, 步长  $gap$  一开始是数组的长度除以 2, 也就是 5, 后面每次按除以 2 进行递减, 直到  $gap$  为 1, 当  $gap$  等于 1 时, 内层就是原有的插入排序。

表 8.2.5 中给出的是  $gap$  为 5 时的比较过程, 第一次是 3 和 56 比较, 这时由于 3 小于 56, 因此



不发生交换，接着比较 87 与 61，因为 87 大于 61，因此 87 覆盖 61 的位置，最内层循环结束，61 放到 87 的位置；接着比较的是 2 与 38，由于 2 小于 38，因此不发生交换，然后是 93 与 12 进行比较，因为 93 大于 12，因此 93 放到 12 的位置，12 放到 93 的位置。如表中第 4 次比较所示。再后是 78 与 40 进行比较，因为 78 大于 40，因此 78 放到 40 的位置，40 放到 78 的位置，如表中第 5 次比较所示。这时 *gap* 为 5 的步长进行完毕，*gap* 为 2 时与 *gap* 为 5 的原理相同，读者可自行推算。

表 8.2.5 希尔排序过程

<i>gap</i> 为 5 时的第 1 次比较	3 87 2 93 78 <b>56</b> 61 38 12 40
<i>gap</i> 为 5 时的第 2 次比较——发生交换	3 <b>61</b> 2 93 78 56 <b>87</b> 38 12 40
<i>gap</i> 为 5 时的第 3 次比较	3 61 2 93 78 56 87 <b>38</b> 12 40
<i>gap</i> 为 5 时的第 4 次比较——发生交换	3 61 2 <b>12</b> 78 56 87 38 <b>93</b> 40
<i>gap</i> 为 5 时的第 5 次比较——发生交换	3 61 2 12 <b>40</b> 56 87 38 93 <b>78</b>
.....	.....

希尔排序推算的时间复杂度为  $O(n^{1.3})$ ，比插入排序要快，甚至在小数组中比快速排序和堆排序还快，但是在涉及大量数据时希尔排序还是比快速排序慢，因为希尔排序编写复杂，且只有在数量为 6000 多以内时比快速排序和堆排序快，因此优势不大。因为未使用额外的空间（额外空间必须与输入元素的个数  $N$  相关），所以空间复杂为  $O(1)$ 。具体代码如下所示：

```
void shell_sort(int arr[], int len) {
    int gap, i, j; //gap 是步长
    int insertVal;
    for (gap = len >> 1; gap > 0; gap >>= 1)
    {
        for (i = gap; i < len; i++)
        {
            insertVal = arr[i];
            for (j = i - gap; j >= 0 && arr[j] > insertVal; j -= gap)
                arr[j + gap] = arr[j];
            arr[j + gap] = insertVal;
        }
    }
}
```

## 5. 快速排序

快速排序的核心是分治思想：假设我们的目标依然是按从小到大的顺序排列，我们找到数组中的一个分割值，把比分割值小的数都放在数组的左边，把比分割值大的数都放在数组的右边，这样分割值的位置就被确定。数组一分为二，我们只需排前一半数组和后一半数组，复杂度直接减半。采用这种思想，不断地进行递归，最终分割得只剩一个元素时，整个序列自然就是有序的。

*arr\_quick* 是我们的快速排序的递归函数，其关键是 *partition*，也就是分割函数的编写，

分割函数返回分割点的下标值。再次递归 `arr_quick` 数组的前半部分和递归 `arr_quick` 数组的后半部分，编辑递归时要注意对返回的分割点的下标减 1 和加 1，递归的结束条件是 `left` 等于 `right` 时，即被分割后的数组只剩一个元素。

接下来看一下 `partition` 函数，我们用数组最右边的值 40 作为分割点，通过下标 `i` 来遍历数组，下标 `k` 始终记录比 40 小的元素将要放的位置，当放置一个比 40 小的元素时，`k` 加 1。首先我们比较 3 和 40，发现 3 小于 40，这时交换 `a[i]` 和 `a[k]`，由于指向同一个位置，就是自己与自己交换，这时 `k` 加 1，`i` 加 1，如表 8.2.6 的第一行所示。然后比较 87 与 40 的大小，87 不小于 40，这时只让 `i` 加 1，然后比较 2 与 40 的大小，2 小于 40，这时交换 `a[i]` 与 `a[k]`，然后对 `k` 加 1，如表 8.2.6 的第 4 行所示；这时 `a[i]` 不断地与 40 比较，一直到达 38 时，发现 38 小于 40，于是 `a[k]` 与 `a[i]` 发生交换，`k` 加 1，如表 8.2.6 的第 5 行和第 6 行所示；这时 12 小于 40，因此 `a[k]` 与 `a[i]` 发生交换，如表 8.2.6 的第 7 行所示，此时 `i` 已经遍历到最后，循环结束。循环结束后，我们将 `a[right]` 与 `a[k]` 交换，如表 8.2.6 的第 8 行所示，这时发现元素 40 的位置已经确定，比 40 小的元素被放在了 40 的左边，比 40 大的元素被放在了 40 的右边，返回 `k`，也就是分割点的下标值，接下来分别对左右两边的两个数组继续进行快速排序。

表 8.2.6 快速排序过程

1	最初	$k$ $i$ 3 87 2 93 78 56 61 38 12 <b>40</b>
2	第 1 次比较后	$k$ $i$ 3 87 2 93 78 56 61 38 12 <b>40</b>
3	第 2 次比较后	$k$ $i$ 3 87 <b>2</b> 93 78 56 61 38 12 <b>40</b>
4	第 3 次比较发生交换后效果	$k$ $i$ 3 2 87 93 78 56 61 38 12 <b>40</b>
5	第 3 次将要发生交换	$k$ $i$ 3 2 87 93 78 56 61 <b>38</b> 12 <b>40</b>
6	第 3 次发生交换后	$k$ $i$ 3 2 38 93 78 56 61 87 12 <b>40</b>
7	第 4 次发生交换后	$k$ $i$ 3 2 38 12 78 56 61 87 93 <b>40</b>
8	最后一次交换	$k$

		<i>i</i>
	3	2 38 12 40 56 61 87 93 78

假如每次快速排序数组都被平均地一分为二,那么可以得出 `arr_quick` 递归的次数是  $\log_2 n$ , 第一次 `partition` 遍历次数为  $n$ , 分成两个数组后, 每个数组遍历  $n/2$  次, 加起来还是  $n$ , 因此时间复杂度是  $O(n\log_2 n)$ , 因为计算机是二进制的, 所以在面试回答复杂度或与人交流时, 提到复杂度时一般直接讲  $O(n\log n)$ , 而不带下标 2。快速排序最差的时间复杂度为什么是  $n^2$  呢? 因为数组本身从小到大有序时, 如果每次我们仍然用最右边的数作为分割值, 那么每次数组都不会二分, 导致递归  $n$  次, 所以快速排序最坏时间复杂度为  $n$  的平方。当然, 为了避免这种情况, 有时会首先随机选择一个下标, 先将对应下标的值与最右边的元素交换, 再进行 `partition` 操作, 从而极大地降低出现最坏时间复杂度的概率, 但是仍然不能完全避免。

在快速排序过程中, 由于我们不断地递归, 每次使用的形参 `left`、`right` 所占用的空间是与递归次数相关的, 所以快速排序的空间复杂度是  $O(\log_2 n)$ , 我们的快速排序是采用递归实现的, 如例 8.2.1 所示, 因为递归实现的快速排序比较简单, 而非递归的快速排序难度较大, 但好处是执行速度要比非递归的快速排序快 (注意非递归的执行效率更优), 同时递归的快速排序在排序的数特别多时, 会出现 `Stack Overflow` 报错, 因为函数递归调用是有上限的, 这时有的读者可能会问排序比较多的数时应怎么办? 其实, 这里可以使用 `qsort`, 也就是标准库提供的快速排序函数, 其内部使用的是非递归的快速排序算法。当然, 如果排序的数上亿, 那么最好还是使用下面介绍的堆排序, 因为针对上亿的数, 一旦快速排序出现最差或最坏的时间复杂度, 那么排序的时间就会特别长。

【例 8.2.1】快速排序的递归实现。

```
int partition(int a[],int left,int right)
{
    int i;
    int k=left;
    for(i=left;i<right;i++)
    {
        if(a[i]<a[right])
        {
            SWAP(a[k],a[i]);
            k++;
        }
    }
    SWAP(a[right],a[k]);
    return k;
}
```

```
void arr_quick(int *a,int left,int right)
```

```
{
    int pivot;
    if(left<right)
    {
        pivot=partiton(a,left,right);
        arr_quick(a,left,pivot-1);
        arr_quick(a,pivot+1,right);
    }
}
```

*qsort* 的函数形式如下所示:

```
#include <stdlib.h> void qsort(void *buf, size_t num, size_t size,
    int (*compare)(const void *, const void *));
```

其功能是对 *buf* 指向的数据（包含 *num* 项，每项的大小为 *size*）进行快速排序。如果函数 *compare* 的第一个参数小于第二个参数，那么返回负值；如果等于第二个参数，那么返回零值；如果大于第二个参数，那么返回正值。函数对 *buf* 指向的数据按升序排列。*qsort* 的本质是其内部实现了快速排序，但是需要我们告诉它任意两个元素如何比较，因为 *qsort* 可以排序各种类型的数组，因此我们通过 *compare* 函数将比较规则告诉 *qsort*。

使用 *qsort* 的关键是编写 *compare* 函数，*qsort* 的第四个参数是一个函数指针。我们知道，传入参数如果是一个函数指针，那么说明我们要传入的是一个函数名。*qsort* 为什么要我们传递一个函数给它呢？因为 *qsort* 需要一个比较规则，*qsort* 函数是可以排序任意类型的数组的，当我们排序整型数组和排序结构体数组时，排序的规则自然不一样。

下面先来看 *qsort* 排序整型数组的代码。一开始数组元素随机生成，排序的代码是 *qsort(arr,N,sizeof(int),compare);*，其中 *arr* 是数组的起始地址，*N* 是数组中元素的个数，*sizeof(int)* 是让 *qsort* 排序的每个元素的大小。接下来我们解析 *compare* 函数，*compare* 函数有两个 *void\** 类型的指针，我们起的名字是 *left* 和 *right*，因为我们把 *compare* 函数传递给 *qsort*，所以实际调用 *compare* 函数的是 *qsort*，*qsort* 调用 *compare* 函数时，传递的实参是任意两个元素的地址值，因为我们排序的是整型元素，*left* 和 *right* 接收的地址值是整型，因此将 *left* 和 *right* 转换为整型指针。*qsort* 规定，第一个参数小于第二个参数时返回负值；第一个参数等于第二个参数时返回零值；第一个参数大于第二个参数时返回正值，于是默认为升序。如果把 *return \*p1-\*p2* 改为 *return \*p2-\*p1*，那么就是降序排列。例 8.2.2 中的代码同时演示了如何实现随机数生成，以及排序的数很多时统计排序时间的方法。读者可以把 *N* 改大，比如千万或上亿，来看一下实际排序所花的时间。注意在统计实际排序时间时，要注释掉打印函数，否则屏幕将会不停地打印。

【例 8.2.2】*qsort* 的使用。

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <time.h>

#define N 10
#define M 100
int compare(const void* left,const void* right)
{
    int* p1=(int*)left;
    int* p2=(int*)right;
    return *p1-*p2;
}
void arrPrint(int *arr)
{
    int i;
    for( i = 0; i <N; i++ )
    {
        printf("%3d",arr[i]);
    }
    printf("\n");
}
int main()
{
    int i;
    int *arr=(int*)malloc(N*sizeof(int)); //这里申请堆空间的目的是当排序的数超过百万
                                         //时，不会发生 Stack Overflow

    time_t start,end;
    srand(time(NULL)); //生成 10 个随机数
    for( i = 0; i <N; i++ )
    {
        arr[i]=rand()%M;
    }
    arrPrint(arr);
    printf("rand success\n");
    start=time(NULL);
    qsort(arr,N,sizeof(int),compare); //实现排序
    end=time(NULL);
    arrPrint(arr);
    printf("use time=%d\n",end-start); //打印排序使用的时间
    system("pause");
```

```
}
```

下面通过 *qsort* 排序结构体数组，完成如下题目。

有一个学生结构体，其数据成员有学号、姓名和3门课程。从键盘上输入5名学生的信息。要求：（1）按照学号递增输出全部学生信息，每名学生的信息一行（格式：学号 姓名 分数1 分数2 分数3 总分）；（2）输出每门课程最高分的学生的信息；（3）输出每门课程的平均分；（4）按照总分输出学生排名。

下面用 *qsort* 来完成第（1）问和第（4）问，例8.2.3中的代码实现了按学号递增排序和按总分从低到高排序。按学号递增排序时，将 *qsort* 的第四个参数改为 *compareNum*，可以看到在 *compareNum* 内我们把指针转换为了结构体指针，因为排序的每个元素都是结构体，所以将 *void\** 类型的指针转换为结构体指针，但是实际上我们比较的是学号的大小，因此 *return p1->num-p2->num*；也就是用学号相减返回正值、负值或零；对总分进行排序时，使用 *compareScore* 函数作为 *qsort* 的第四个参数，我们依然将指针转换为结构体指针，但这时比较的是三门课加起来的总分，当总分是浮点数时，比较大小时不再像前面的整型数那样直接调用 *return*，而必须通过大于、小于来解决。相信通过 *compareNum* 函数和 *compareScore* 函数，读者对如何使用 *qsort* 的第四个参数已有清晰的理解。为方便读者输入测试数据，下面列出了5组学生信息，编写代码运行的读者可以直接输入下面5名学生的数据进行测试：

```
1001 lele 96.5 97.3 88.1
1007 lili 92.5 99.3 85.1
1003 xiongda 93.5 94.3 94.1
1005 guangtougang 91.5 98.3 96.1
1002 xionger 88.5 92.3 91.1
```

【例8.2.3】*qsort* 排序结构体数组。

```
#include <stdio.h>
#include <stdlib.h>

typedef struct{
    int num;
    char name[20];
    float chinese;
    float math;
    float english;
}Student_t,*pStudent_t;
#define N 5
//按学号从小到大排序
int compareNum(const void* pleft,const void* pright)
{
```

```
pStudent_t p1=(pStudent_t)pleft;
pStudent_t p2=(pStudent_t)pright;
return p1->num-p2->num;
}
//按总分从小到大排序
int compareScore(const void* pleft,const void* pright)
{
    pStudent_t p1=(pStudent_t)pleft;
    pStudent_t p2=(pStudent_t)pright;
    if(p1->chinese+p1->math+p1->english-p2->chinese-p2->math-p2->english>0)
    {
        return 1;
    }else
    if(p1->chinese+p1->math+p1->english-p2->chinese-p2->math-p2->english<0)
    {
        return -1;
    }else{
        return 0;
    }
}
int main()
{
    Student_t sArr[N];
    int i=0;
    //输入 5 名学生的学号、姓名、3 门课的成绩信息
    for(i=0;i<N;i++)
    {
        scanf("%d%s%f%f%f",&sArr[i].num,sArr[i].name,&sArr[i].chinese,&sArr[i].math,
            &sArr[i].english);
    }
    qsort(sArr,N,sizeof(Student_t),compareScore);
    //打印排序后的结果
    for(i=0;i<N;i++)
    {
        printf("%d %15s %5.2f %5.2f %5.2f %6.2f\n",sArr[i].num,sArr[i].name,
            sArr[i].chinese,sArr[i].math,sArr[i].english,
            sArr[i].chinese+sArr[i].math+sArr[i].english);
    }
}
```

```
system("pause");  
}
```

在硕士研究生复试的机试或校招的 OnlineJudge 中（如浙大 PAT），如果遇到的题目中需要使用排序，那么应尽量使用 *qsort*，也就是库函数提供的排序功能，这样既节省时间，又不容易出错。当然，如果题目特别要求手写某种排序算法，那么不可以使用 *qsort* 等排序 API。

我们在淘宝网上购物时，在搜索框中输入“鞋子”，可以看到图 8.2.2 所示的效果，每款鞋子的信息包含图片、价格、购买人数、名字、店铺、店铺地址等，它们在服务器的内存中就像一个结构体信息，当不同用户需要这些信息时，就把这些结构体信息发送给用户。但是，存在一个问题，即不同用户需要的排序效果是不一样的，有些人想以销量排序，有些人想以价格排序，有些人想以信用排序，每个用户需要一种新的排序时，如果我们都交换内存里的结构体，那么交换成本会很高，而且这些商品往往是以链表形式存储到内存中的，交换时需要的代码逻辑多于数组。那么有没有好的解决办法呢？答案是当然有，那就是使用索引的原理（不理解索引是什么没有关系）。

在链表的实际使用中，链表中有多少个元素是已经统计好的。假设链表中有 5 个结点，那么通过 *calloc* 申请 5 个指针大小的空间，相当于使用动态的指针数组，然后遍历链表，把链表中每个结点的地址存储到指针数组中。*qsort* 不仅可以排序整型数组、浮点型数组、结构体数组，而且可以排序指针数组。根据用户的排序需求，无论是价格、销量还是信用，我们只需交换对应的指针即可，这样交换的成本将极大地降低，客户需要排序的结果时，我们可以通过访问有序的指针数组把数据从服务器传输给客户。

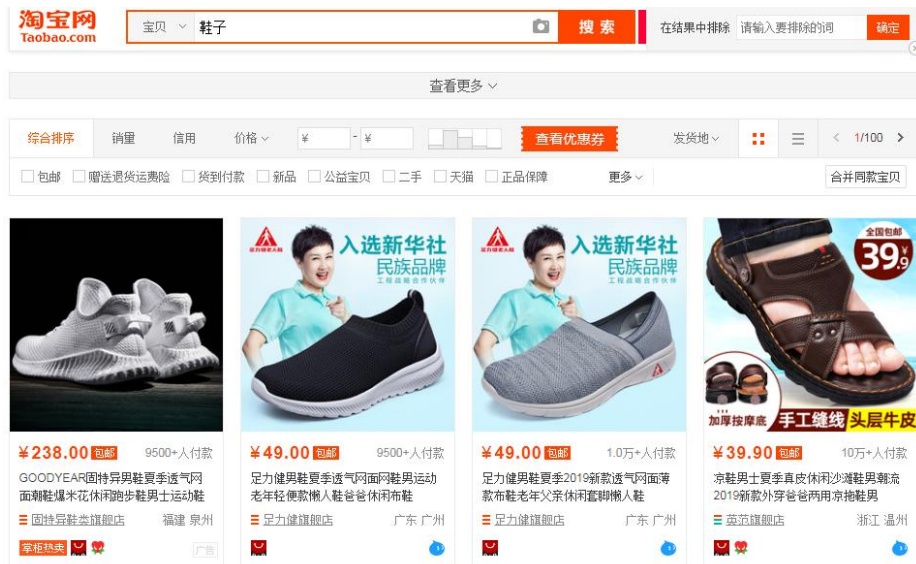


图 8.2.2 淘宝网搜索界面

下面我们看一下 *qsort* 排序指针数组的代码，如例 8.2.4 中的代码所示。首先采用头插法新建一个链表，输入 3、6、9、2、7 创建链表后，链表中存储了 5 个整型元素。通过 *listLen* 统计链表的长度，然后申请 5 个结构体指针大小的空间。因为我们要在申请的内存空间内存储结构体指针，



因此需要将 `calloc` 的返回值强制转换为二级指针，并赋给 `pArr`（在指针章节的二级指针部分讲过），然后遍历链表，将链表中每个结点的地址依次赋给 `pArr[0]`到 `pArr[4]`，再后开始通过 `qsort` 排序指针数组 `pArr`。元素个数是 `listLen`，每个元素的大小是 `sizeof(pStudent_t)`，注意有的读者易写错为 `sizeof(Student_t)`。下面，我们来看最难的 `comparePointer` 函数，因为 `pleft` 和 `pright` 是任意两个元素的地址值，而现在我们排序的是指针数组，每个元素本身是一级结构体指针，如果再对元素取地址，那么得到的就是二级指针，因此需要将 `void*` 类型 `pleft` 和 `pright` 强制转换为二级结构体指针。因为实际比较的是指针指向的结构体的学号数据，因此首先要取值(`*p1`)变为一级结构体指针，然后通过成员访问对应的学号，最终依据 `qsort` 要求的正值、负值、零三种情况，`return (*p1)->num-(*p2)->num;`打印指针数组时，可以得到有序的效果，但是实际链表的顺序未改变，具体执行结果如图 8.2.3 所示。

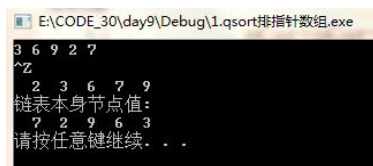


图 8.2.3 执行结果

【例 8.2.4】`qsort` 排序指针数组。

```

#include <stdlib.h>
#include <stdio.h>

typedef struct student_t{
    int num; //为了输入方便，我们仅为学生添加学号信息
    struct student_t *pNext;
}Student_t,*pStudent_t;

int comparePointer(const void* pleft,const void* pright)
{
    pStudent_t* p1=(pStudent_t*)pleft;
    pStudent_t* p2=(pStudent_t*)pright;
    return (*p1)->num-(*p2)->num;
}

#define N 10
int main()
{
    int i,listLen=0; //listLen 用于记录链表中元素的个数
    pStudent_t phead,ptail,pNew,pCur;
  
```

```
pStudent_t *pArr;
phead=ptail=NULL;
//通过头插法新建链表
while(scanf("%d",&i)!=EOF)
{
    pNew=(pStudent_t)calloc(1,sizeof(Student_t));
    pNew->num=i;
    if(NULL==phead)
    {
        phead=pNew;
        ptail=pNew;
    }else{
        pNew->pNext=phead;
        phead=pNew;
    }
    listLen++;
}
pArr=(pStudent_t*)calloc(listLen,sizeof(pStudent_t)); //存储 listLen 个数的指针
pCur=phead;
//遍历链表, 将链表的每个结点的地址存入指针数组 pArr
for(i=0;i<listLen;i++)
{
    pArr[i]=pCur;
    pCur=pCur->pNext;
}
//排序指针数组
qsort(pArr,listLen,sizeof(pStudent_t),comparePointer);
//打印排序后的效果
for(i=0;i<listLen;i++)
{
    printf("%3d",pArr[i]->num);
}
printf("\n");
printf("链表本身结点值:\n");
pCur=phead;
while(pCur!=NULL)
{
    printf("%3d",pCur->num);
```

```
        pCur=pCur->pNext;
    }
    printf("\n");
    //链表的销毁
    while(phead!=NULL)
    {
        pCur=phead;
        phead=phead->pNext;
        free(pCur);
        pCur=NULL;
    }
    ptail=NULL;
    free(pArr); //释放指针数组空间
    system("pause");
}
```

上面的代码中针对链表进行了销毁，即对指针数组申请的空间进行了释放。工作时，高频查询的数据通常会常驻内存，或者会将排序后的结果存入磁盘，从而降低下次用户提取结果的时间（第9章中将讲解如何将数据存入磁盘）。

## 6. 堆排序

如果排序的数据较多，而且要求空间复杂度为  $O(1)$ ，那么就要优先考虑堆排序，因为相对于快速排序，堆排序的最坏和平均时间复杂度都是  $O(n\log n)$ 。

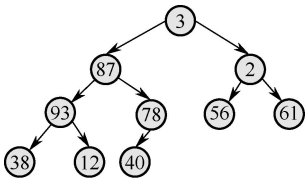
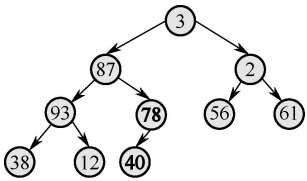
堆（Heap）是计算机科学中的一种特殊的树状数据结构。若满足以下特性，则可称为堆：“给定堆中任意结点  $P$  和  $C$ ，若  $P$  是  $C$  的父结点，则  $P$  的值小于等于（或大于等于） $C$  的值。”若父结点的值恒小于等于子结点的值，则该堆称为最小堆（min heap）；反之，若父结点的值恒大于等于子结点的值，则该堆称为最大堆（max heap）。堆中最顶端的那个结点称为根结点（root node），根结点本身没有父结点（parent node）。平时在工作中，我们将最小堆称为小根堆或小顶堆，把最大堆称为大根堆或大顶堆。堆始于 J. W. J. Williams 于 1964 年发表的堆排序（heap sort），当时他提出了二叉堆树作为此算法的数据结构。

假设我们有 3, 87, 2, 93, 78, 56, 61, 38, 12, 40 共 10 个元素，我们将这 10 个元素建成一棵完全二叉树，这里采用层次建树法，虽然只用一个数组存储元素，但是我们能将二叉树中任意一个位置的元素对应到数组下标上，我们将二叉树中每个元素对应到数组下标的这种数据结构称为堆，比如最后一个父元素的下标是  $N/2-1$ ，也就是  $a[4]$ ，对应的值为 78。为什么下标一定是  $N/2-1$ ？因为这是层次建立一棵完全二叉树的特性。可以这样记忆：如果父结点的下标是  $dad$ ，那么父结点对应的左子结点的下标值是  $2*dad+1$ ，因为树每多一层，元素个数就会翻倍并多 1，这很容易得出。接着，依次将每棵子树都调整为父结点最大，最终将整棵树变为一个大根堆。

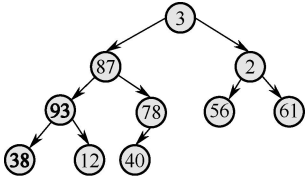
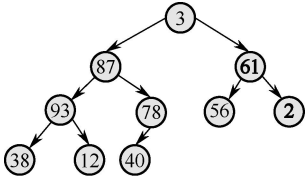
如表 8.2.7 所示，第 2 行中的 `adjustMaxHeap` 函数对 `arr[4]` 与 `arr[9]` 即 78 和 40 进行调

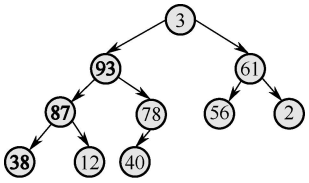
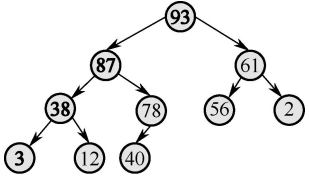
整, 因为 78 大于 40, 所以无须交换; 通过一个循环对  $i$  减 1, 结果是  $arr[3]$ , 得到前面一棵子树的父结点, 也就是 93, 如第 3 行所示。首先, 38 大于 12, 所以将 38 与 93 比较, 93 大于 38, 无须调整; 接着如第 4 行所示,  $arr[2]$  与  $arr[6]$  进行比较, 因为 2 小于 61, 所以交换  $arr[2]$  与  $arr[6]$ , 交换效果如第 4 行所示; 然后对  $arr[1]$  和  $arr[3]$  也就是 87 和 93 进行比较, 87 小于 93, 所以发生交换, 但是这时 87 即  $arr[3]$  本身是一个父结点, 有可能交换会导致 87, 38, 12 这棵子树不满足最大堆的特性, 因此在 `adjustMaxHeap` 函数中使用了一个循环 `while`, 发生交换时, 我们让子结点重新作为父结点, 再次进行一轮调整, 当然, 前提是调整的结点的确是一个父结点, 这时因为 87 大于 38, 所以无须调整, `while` 循环结束。

表 8.2.7 堆排序

1	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树	
2	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树, $arr[4]$ 与 $arr[9]$ 进行比较	

(续表)

3	数组元素情况	3 87 2 93 78 56 61 38 12 40
	数组元素对应的二叉树, $arr[3]$ 与 $arr[7]$ 进行比较	
4	数组元素情况	3 87 61 93 78 56 2 38 12 40
5	数组元素对应的二叉树, $arr[2]$ 与 $arr[6]$ 进行比较, 发生交换	
	数组元素情况	3 93 61 87 78 56 2 38 12 40

	数组元素对应的二叉树，arr[1]与 arr[3]进行比较，发生交换，发生交换后 arr[3]重新作为父结点，与孩子结点比较	
6	数组元素情况	93 87 61 38 78 56 2 3 12 40
	数组元素对应的二叉树，arr[0]与 arr[1]进行比较，发生交换，发生交换后 arr[1]重新作为父结点，与孩子结点比较	

接着我们看第 6 行，这时 arr[0]和 arr[1]进行比较，3 小于 93，发生交换，交换后，arr[1]重新作为父结点，与子结点 arr[3]进行比较，3 小于 87，因此 arr[1]和 arr[3]发生交换，这时我们再次将 arr[3]作为父结点，与自己的子结点 arr[7]进行比较，即让 3 和 38 比较，3 小于 38，因此发生交换，最终得到的效果如第 6 行的图形所示。到此，我们将数组调整为了大根堆，任何一棵子树的父结点都是大于子结点的，从而可以确定根结点是最大的。

接着将顶部元素与数组的最后一个元素进行交换，这样，最大的元素就在数组的最后，然后将剩余的 9 个元素重新调整为大根堆。因为这时只有数组的第一个元素不满足大根堆的特性，所以只需调用 adjustMaxHeap 函数调整顶部元素，调整完毕后，再次将顶部元素和数组倒数的第二个元素交换，以此类推，最终使数组有序。堆排序的具体代码如例 8.2.5 所示。

【例 8.2.5】堆排序。

```
void adjustMaxHeap(int *arr,int adjustPos,int arrLen)
{
    int dad=adjustPos; //adjustPos 是要调整的结点位置
    int son=2*dad+1;
    while(son<arrLen) //arrLen 代表数组的长度
    {
        if(son+1<arrLen&&arr[son]<arr[son+1]) //比较左子结点和右子结点，如果右子
            //结点大于左子结点，son 加 1，下一步对右子结点与父结点进行比较
        {
            son++;
        }
        if(arr[dad]<arr[son])
        {
            SWAP(arr[dad],arr[son]);
            dad=son;
            son=2*dad+1;
        }else{
```

```

        break;
    }
}
void arrHeap(int *arr)
{
    int i;
    //调整为大根堆
    for(i=N/2-1;i>=0;i--)
    {
        adjustMaxHeap(arr,i,N);
    }
    SWAP(arr[0],arr[N-1]); //交换顶部与最后一个元素
    for(i=N-1;i>1;i--)
    {
        adjustMaxHeap(arr,0,i);    //将剩余 9 个元素再次调整为大根堆，不断交换根部
                                   //元素与数组尾部元素，然后将堆元素个数减 1
        SWAP(arr[0],arr[i-1]);    //交换顶部元素与堆中的最后一个元素，已在尾部
                                   //排好的不算堆中的元素
    }
}

```

**思考题：**如果要按从大到小的顺序排列，那么应该如何修改代码？

如果将上面介绍的堆排序封装为像 *qsort* 一样的函数，那么应该如何做呢？下面通过例 8.2.6 的代码，学习如何实现自己的 *heapSort* 函数。实现自己的 *heapSort* 函数既可让我们更加深刻地理解 *qsort*，同时也可让我们进一步熟练使用函数指针。要熟悉 C/C++ 语言，函数指针是必须掌握的。*heapSort* 函数的编写与我们上面的堆排序结构一致，关键是 *heapMax*。我们给 *heapMax* 传入的参数是要调整的父结点的地址值，同时要让 *heapMax* 能够算出父结点对应的子结点的地址值。在比较父结点和子结点时，要使用 *compare* 函数进行比较，同时在交换父结点和子结点时，通过内存交换的方式进行，因为结点的类型可能是任何类型，所以 *swap* 交换时，在传递两个结点的起始地址的同时，还要把每个结点所占用的空间大小传递进去。

**【例 8.2.6】**堆排序的封装。

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define N 10
//每个参数的意义与 qsort 是一致的
void heapSort(void *arr, size_t num, size_t size, int(*compare)(const void *a,

```

```

        const void *b)) {
    int i;
    for (i = num / 2 - 1; i >= 0; i--) {
        heapMax(arr, (char*)arr + i * size, num, i, size, compare);
    }
    swap((char*)arr, (char*)arr + (num - 1) * size, size);
    for (i = num - 1; i > 1; i--) {
        //heapMax的形参分别是数组首地址，开始判断的起始地址，长度，结点的编号，
        //单个元素大小
        heapMax(arr, arr, i, 0, size, compare);
        swap((char*)arr, (char*)arr + (i - 1) * size, size);
    }
}

//dadStartAddr 是要调整的父结点的起始地址，len 是数组长度
//code_Num 是父结点在数组中的下标值，size 是每个元素的大小
void heapMax(void *arr, void *dadStartAddr, int len, int code_Num, int size,
             int(*compare)(const void *a, const void *b)) {
    char *dad = (char*)dadStartAddr;
    char *son = (char*)arr + (code_Num * 2 + 1) * size;
    code_Num = code_Num * 2 + 1;
    while (son <= (char*)arr + (len - 1) * size) {
        if (son + size <= (char*)arr + (len - 1) * size && compare(son, son + size) < 0) {
            son = son + size;
            code_Num++;
        }
        if (compare(dad, son) < 0) {
            swap(dad, son, size);
            dad = son;
            son = (char*)arr + (code_Num * 2 + 1) * size;
            code_Num = code_Num * 2 + 1;
        }
        else {
            break;
        }
    }
}

//因为不知道每个元素的大小，因此交换时只能逐个字节地移动，交换两个元素的内存
void swap(char *a, char *b, int len) {

```

```
char temp;
while (len--) {
    temp = *a;
    *a++ = *b;
    *b++ = temp;
}
}
//比较规则与 qsort 的原理一致
int cmp(const void *a, const void *b) {
    return *(int*)a - *(int*)b;
}
int main() {
    int arr[N];
    int i;
    //生成随机数
    srand((unsigned)time(NULL));
    for (i = 0; i < N; ++i) {
        arr[i] = rand() % 100;
        printf(" %d", arr[i]);
    }
    printf("\n");
    //测试 heapSort 是否可以排序正确
    heapSort(arr, N, sizeof(int), cmp); //堆排序
    //打印输出排序后的结果
    for (i = 0; i < N; ++i) {
        printf(" %d", arr[i]);
    }
    printf("\n");
    system("pause");
    return 0;
}
```

**思考题：**如果要让 `heapSort` 按从大到小的顺序排列，那么应该如何修改 `cmp` 函数？

## 7. 归并排序

归并排序的时间复杂度是  $O(n\log n)$ ，空间复杂度是  $O(n)$ 。因此，正常单机排序时应优先使用堆排序，但在多机排序时会用到归并排序的思想。下面介绍归并排序。

归并排序的代码如例 8.2.7 所示，它是采用递归思想实现的。非递归的归并排序难度较大，面试时只需掌握如下递归的二路归并即可。首先，最小下标值和最大下标值相加并除以 2，得到



中间下标值 *mid*，用 *MergeSort* 对 *low* 到 *mid* 排序，然后用 *MergeSort* 对 *mid+1* 到 *high* 排序。当数组的前半部分和后半部分都排好后，使用 *Merge* 函数。*Merge* 函数的作用是合并两个有序数组。为了提高合并有序数组的效率，在 *Merge* 函数内定义了 *B[N]*。首先，我们通过循环把数组 *A* 中从 *low* 到 *high* 的元素全部复制到 *B* 中，这时游标 *i*（工作中把遍历的变量称为游标）从 *low* 开始，游标 *j* 从 *mid+1* 开始，谁小就将谁先放入数组 *A*，对其游标加 1，并在每轮循环时对数组 *A* 的计数游标 *k* 加 1，当 *i* 到达 *mid* 或 *j* 到达 *high* 时，说明有一个有序数组已经全部放完，这时下面的两个 *while* 循环负责将某个有序数组的剩余部分放入数组 *A*。通过下面的代码可以看出，编写归并排序的核心是合并两个有序数组。针对 *MergeSort* 的递归结束条件，当 *low<high* 时，若 *low* 和 *high* 相邻，则相当于合并两个元素，因此 *low* 等于 *high* 就无须再归并，因为只有一个元素的序列自然是有序的，所以递归的结束条件是 *low<high*。

当一台计算机的内存不足以放置所有数据时，若不采用外部排序方法，则多机同时排序提高效率是常用的办法。多机排序后的结果最终仍然需要以归并排序的思想进行合并。

【例 8.2.7】归并排序实现。

```
#include <stdio.h>
#include <stdlib.h>

#define N 7
typedef int ElemType;
//49,38,65,97,76,13,27
void Merge(ElemType A[],int low,int mid,int high)
{
    ElemType B[N];
    int i,j,k;
    for(k=low;k<=high;k++) //复制元素到 B 中
        B[k]=A[k];
    for(i=low,j=mid+1,k=i;i<=mid&& j<=high;k++) //合并两个有序数组
    {
        if(B[i]<=B[j])
            A[k]=B[i++];
        else
            A[k]=B[j++];
    }
    while(i<=mid) //如果某个有序部分还有剩余元素，接着放入即可
        A[k++]=B[i++];
    while(j<=high)
        A[k++]=B[j++];
}
```

//归并排序不限定为二路归并（也可称为两两归并），还是多种归并，因为两两归并编写简单，  
//同时面试官也不会要求编写其他数量的归并，所以大家掌握两两归并即可

```
void MergeSort(ElemType A[],int low,int high) //递归分割
```

```
{
    if(low<high)
    {
        int mid=(low+high)/2;
        MergeSort(A,low,mid);
        MergeSort(A,mid+1,high);
        Merge(A,low,mid,high);
    }
}

void print(int* a)
{
    for(int i=0;i<N;i++)
    {
        printf("%3d",a[i]);
    }
    printf("\n"); //为了打印在同一排，在最后打印换行
}

// 归并排序

int main()
{
    int A[7]={49,38,65,97,76,13,27}; //数组，7 个元素
    MergeSort(A,0,6);
    print(A);
    system("pause");
}
```

## 8. 计数排序

上面的快速排序、堆排序在对 100000000 个数排序时，需要的时间约为 38 秒，当然，这是在作者的计算机上测试的结果。如果计算机配置更好，那么所需的时间自然会比上面的时间更短。然而，30 多秒的耗时在工作中的很多时候是我们无法接受的，有没有更好的办法呢？答案是有，计数排序就采用空间换时间的方式来提高排序效率。如果用计数排序对 0~99 之间的随机数排序，总计 100000000 个随机数，那么排序所需要的时间不到 1 秒。之所以会这么快，是因为计数排序的复杂度是  $O(n+m)$ ，其中  $m$  是下面代码中的  $M$ ，即数的数值范围。因为数的变化范围是 0~99，所以计数排序的执行次数是 100000000 + 100 数量级，但是堆排序的复杂度是  $O(n\log n)$ ，排序的执行次数是 2700000000 次。在实际应用中，商品的价格、销量及人的年龄、身

高都在一个有限的范围内，因此为了提高排序效率，使用计数排序是很常见的。

在例 8.2.8 中，我们通过 `arrCount` 来实现计数排序。首先，我们遍历一次整个数组，统计 100000000 个数中 0 出现多少次，1 出现多少次，一直到 99 出现多少次，并把每个元素出现的次数统计到数组 `count` 中。接着，开始依次填入数组 `arr`。例如，0 出现了 122 次，把数组 `arr` 中从下标 0 到下标 121 的元素全部填写为 0，1 出现了 308 次，把数组 `arr` 中从下标 122 到下标 429 的元素全部填写为 1，以此类推，最终使 `arr` 数组有序。

编写计数排序代码时，如果要测试编写的代码是否正确，可以把 `N` 改为 10，将注释掉的 `arrPrint` 函数打开，以了解排序后的结果是否正确。如果多次执行排序后结果正确，那么说明编写的计数排序代码没有问题，这时就可以测试 100000000 个数的排序。

计数排序的空间复杂度是  $O(m)$ 。如果是整型数，其变化范围是 -2100000000 多到 +2100000000 多，那么这时就不适合使用计数排序，因为时间复杂度是  $O(n+m)$ ，相当于执行次数为 100000000 + 4200000000 量级，大于上面堆排序的执行次数 2700000000 多，同时需要额外的 16GB 空间（4G\*4 字节）来存储 `count` 数组，这显然是不合适的。然而，如果面试过程中遇到去重的问题，那么就要使用计数排序思想，因为我们用位来做标识，也就是所说的位图，针对 4200000000 次量级整型数的变化范围，用位图处理所需要的空间是 512MB。

【例 8.2.8】计数排序实现。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 1000000000
#define M 100
#define SWAP(a,b) {int tmp;tmp=a;a=b;b=tmp;}
void arrCount(int *arr)
{
    int count[M]={0};
    int i,j,k;
    //遍历数组后，统计数组中每个元素出现的次数
    for(i=0;i<N;i++)
    {
        count[arr[i]]++;
    }
    k=0; //k 用来记录数组中哪些元素已经填入值
    //将每个数值出现的次数，依次从前到后填入数组 arr
    for(i=0;i<M;i++)
    {
        for(j=0;j<count[i];j++)
```

```
        {
            arr[k++]=i;
        }
    }
}

int main()
{
    int i;
    int *arr=(int*)malloc(N*sizeof(int));
    time_t start,end;
    srand(time(NULL));
    for( i = 0; i < N; i++ )
    {
        arr[i]=rand()%M;
    }
    //arrPrint(arr);
    printf("rand success\n");
    start=time(NULL);
    arrCount(arr);
    end=time(NULL);
    //arrPrint(arr);
    printf("use time=%d\n",end-start);
    system("pause");
}
```

分配类的排序包含计数排序、基数排序和桶排序，由于基数排序是按照数据的每一位进行比较的，编写代码相对复杂一些，因此在面试中出现得较少。如果待排序的数的范围较大，例如从 0 到 10000000，那么桶排序的思想是划分若干区间，比如划分 10 个区间，这时遍历一次数组将 0 到 1000000 放入第一个桶，将 1000000 到 2000000 放入第二个桶，以此类推，最后将 9000000 到 10000000 放入最后一个桶，然后对每个桶分别进行排序，最后合并起来即可。

前面介绍了八大排序算法。对于重要的快速排序、堆排序，务请读者要非常熟练地编写出代码。下面我们来看面试时关于排序的几个具体问题。

#### 1. 对于 $N$ 个无序数的数组，如何找出第 $K$ 大的数？

如果这时你的回答是对数组排序，那么面试到此就结束了。之前我们在找第一大和第二大的数时，只需遍历一次数组。这个问题同样如此。首先，我们对前  $K$  个数建立小根堆。为什么是小根堆？因为对于  $K$  个元素，小根堆的根部就是第  $K$  大的元素，这时拿  $K+1$  个元素依次与堆顶进行比较，如果大于堆顶，就删除堆顶并放入新元素，重新调整为小根堆，直到遍历到数组末尾，堆顶即为第  $K$  大的元素。这样做的复杂度只有  $O(N\log K)$ ，远低于排序的复杂度。针对这个问题还有其他的解法吗？大家可以把自己的想法发到 QQ 群交流。

## 2. 如果内存无法存储 $K$ 个元素，那么应怎么办？

此时面试官考查的是逆向思维。首先一定要做内存能存储  $N-K$  个元素的假设，因为面试官并没有说这样做不行。如果内存可以存下  $N-K+1$  个元素，那么我们首先建立  $N-K+1$  个元素的大根堆，然后依次遍历数组，如果发现某个元素小于堆顶，就删除堆顶并将新元素放入，重新调整为大根堆，遍历到数组末尾，堆顶元素即为第  $N-K+1$  小的，所以也就是第  $K$  大的。

3. 上面找第  $K$  大的元素时未考虑重复元素的问题。如果现在要找不含重复元素的第  $K$  大元素，那么应该怎么解决？

虽然我们可以采用在建立小根堆时不放入重复元素的解法，但只要面试官问的是去重问题，我们首先就要回答采用位图（我们自己来实现一下），因为位图是最快的去重方法（以空间换时间）。例如，对于一个 64 位的程序，我们首先定义一个长整型指针并为其申请 512MB 的空间，即 `long *arr=(long*)malloc(512*1024*1024)`，写为 `malloc(1<<29)` 也可，然后通过 `memset(arr,0,1<<29)` 将申请的空间全部置为 0。这时，遍历一次数组得到一个数后，将对应的位设置为 1 即可。比如，若来的数是 129， $129/64$  的值为 2，则需要对 `arr[2]` 进行操作， $129\%64$  的余数为 1，于是将 `arr[2]` 按位或 `1<<1` 即可，总公式为 `arr[129/64]=arr[129/64]|1<<129%64`。也就是说，如果输入的数为  $m$ ，那么公式为 `arr[m/64]=arr[m/64]|1<<m%64`，位图建立后，依次遍历位图，找到第  $K$  个为 1 的位置，然后逆向算出对应的值。如果程序是 32 位的，那么将公式中的 64 改为 32 即可。

4. 如果单台机器的内存既不能存储  $K$  个元素，又无法存储  $N-K$  个元素，那么如何使用多机排序找出第  $K$  大的元素？

如果内存放不下，又不可以多机排序，那么这时只能使用外部排序。外部排序首先将数据划分为  $k$  块，每块可以放入内存，排序后重新写回，然后针对  $k$  块数据采用多路归并（败者树的多路归并思想），所用到的思想和之前的类似，读者可以自行寻找源码进行学习。下面介绍多机如何找出第  $K$  大的元素。这里先要了解两台机器如何找出第  $K$  大的元素，多机的原理与此类似。我们将两台机器分别称为 A 机和 B 机，如果首先用小根堆方法得出 A 机  $N$  个元素中的前  $K/2$  大的元素并将其排序，得出 B 机  $N$  个元素中的前  $K/2$  大的元素并将其排序，然后将 A 机的第  $k/2$  个元素（即 `A[k/2-1]`）和 B 机的第  $k/2$  个元素（即 `B[k/2-1]`）进行比较，那么有以下三种情况（为简单起见，这里先假设  $k$  为偶数，所得到的结论对于  $k$  是奇数也成立）：

- `A[k/2-1] == B[k/2-1]`
- `A[k/2-1] > B[k/2-1]`
- `A[k/2-1] < B[k/2-1]`

如果 `A[k/2-1] > B[k/2-1]`，那么 `A[0]` 到 `A[k/2-1]` 肯定在  $A \cup B$  的 top  $k$  元素范围内，换句话说，`A[k/2-1]` 不可能大于  $A \cup B$  的第  $k$  大元素。因此，我们可以放心地删除 A 数组的这  $k/2$  个元素。同理，当 `A[k/2-1] < B[k/2-1]` 时，可以删除 B 数组的  $k/2$  个元素。

`A[k/2-1] == B[k/2-1]` 时，说明找到了第  $k$  大的元素，直接返回 `A[k/2-1]` 或 `B[k/2-1]` 即可。

如果删除了 `A[0]` 到 `A[k/2-1]`，那么这时我们寻找的是 A 中剩余的元素与 `B[0]` 到

$B[k/2-1]$ 中的第  $k/2$  大元素, 这时我们将  $A$  中剩余元素的前  $k/4$  个元素 (有序, 因此一开始设计有序数组时尽量充分占用了内存) 与  $B[k/4-1]$  进行比较, 原理与上面的类似, 体现了递归的思想。

下面我们直接写一下找出两个升序数组中第  $k$  小的元素的代码, 如例 8.2.9 所示。我们可以打开代码中的注释, 同时去掉  $i=8$ , 以查看从第 1 小到第 22 小的结果是否正确。代码中确保  $len1$  小于  $len2$  的目的是, 首先拿短的数组与  $k/2$  进行比较, 同时注意递归结束条件, 当  $k$  等于 1 时递归结束。这里没有第零小, 最小的就是第 1 小, 第 1 小的数为 0; 当其中一个数组没有  $k/2$  个元素时也没有关系, 这时我们针对另外一个数组, 下标取为  $k-num1$ , 取得更大一些也可。当然, 在调用 *FindKthInTwoSortedArray* 时, 要确保  $k$  的值小于两个数组的长度之和。

【例 8.2.9】找出两个有序数组中第  $k$  小的数。

```
#include <stdio.h>
#include <stdlib.h>
/*在两个升序排列的数组中找到第 k 小的元素*/
int FindKthInTwoSortedArray(int array1[], int len1, int array2[], int len2, int k)
{
    if( k < 0 )
    {
        printf("Invalid %d \n", k);
        return -1;
    }
    /*保证 len1 <= len2*/
    if( len1 > len2 )
        return FindKthInTwoSortedArray(array2, len2, array1, len1, k);
    if( len1 == 0 )
        return array2[k-1];
    if( k == 1 )
        return ((array1[0] >= array2[0]) ? array2[0] : array1[0]);

    /*不一定每个数组都有 k/2 个元素*/
    int num1 = (len1 >= k/2) ? k/2 : len1;
    int num2 = k - num1;

    if( array1[num1-1] == array2[num2-1] )
        return array1[num1-1];
    else if( array1[num1-1] > array2[num2-1] )
    {
        return FindKthInTwoSortedArray(array1, len1, &array2[num2], len2-num2,
```

```

k-num2);
    }
    else if( array1[num1-1] < array2[num2-1] )
    {
        return FindKthInTwoSortedArray(&array1[num1], len1-num1, array2, len2,
k-num1);
    }
}

int main()
{
    int ret;
    int i;
    int array1[11] = {0,1,2,3,4,5,6,7,8,9,17};
    int array2[11] = {3,4,5,6,7,8,9,10,11,12,29};

    printf("sorted two array:\n");
    //for(i=1; i<=22; i++)
    //{
        i=8;//假如我们找第 8 小的元素
        ret=FindKthInTwoSortedArray(array1, 11, array2, 11, i);
        printf("第%d 大: %d\n",i,ret);
    //}
    system("pause");
}

```

下面再来看一道面试题：总计有 25 匹马和 5 条赛道，一匹马一条赛道。比赛只能得到 5 匹马之间的快慢程度，请问至少需要比多少次才能确定前三名的马？请写出分析思路（阿里面试题）。这个题目网易、头条都面试过，只是马的数量或赛道数不同而已。分析思路如下。

给所有的马标号，分成 5 组：

A 组：A1, A2, A3, A4, A5

B 组：B1, B2, B3, B4, B5

C 组：C1, C2, C3, C4, C5

D 组：D1, D2, D3, D4, D5

E 组：E1, E2, E3, E4, E5

假设每组的马的速度是  $X_1 > X_2 > X_3 > X_4 > X_5$  ( $X=A, B, C, \dots$ )。

首先，每组 5 匹马的比赛是避免不了的，根据每组的快慢并找出最快的前 3 匹马，则 5 场比赛下来，剩下的马如下：

A组: A1, A2, A3

B组: B1, B2, B3

C组: C1, C2, C3

D组: D1, D2, D3

E组: E1, E2, E3

再将每组的第一名 A1, B1, C1, D1, E1 五匹马拿出来比赛, 假设结果为 A1>B1>C1>D1>E1。得出全场最佳为 A1, 于是确定了第一名。接下来确定第二名和第三名, 这时最后两名的 D 组和 E 组都可以淘汰。同理, C 组的 C1 在第六次比赛拿第三, 则 C 组除了 C1, 其他的马不可能进入前三; B 组的 B1 在第六次比赛中拿第二, B2 有可能成为全场第三, 可以把 B3 淘汰。

A组: A1, A2, A3

B组: B1, B2, B3

C组: C1, C2, C3

第二名的可能性为 A 组的第二名 A2 与第六次比赛的第二名 B1。

第三名的可能性为 A 组的第三名 A3 与 B 组的第二名 B2 和第六次比赛的第三名 C1。

接下来把 A2, B1, A3, B2, C1 这五匹马再进行一场比赛即可确定全场第二与第三。

总共进行 7 场比赛就可以确定最快的前三匹马。

完成这种类型的题目时, 要注意通过画图分析, 全面思考后再向面试官讲解。这类问题考查的主要是分组思想与淘汰思想。

### 8.2.3 二分查找算法

前面我们学习了排序算法, 掌握了时间复杂度及空间复杂度的计算。排序的目的除了把排序的结果展示给用户, 更多的是为了做增删查改操作。针对有序数组, 我们经常使用的查找算法是二分查找算法。比如有一个跟朋友玩的游戏: 让你的朋友心里想一个 0~1000 之间的数, 我们只需要 10 次就可以猜到这个数是几。这是怎么做到的? 首先问你的朋友他心里的那个数是比 500 大还是比 500 小, 如果比 500 小, 那么接着问是比 250 大还是比 250 小, 这样通过 10 次询问就可以得到这个数。这一过程中运用了二分查找的思想。二分查找的时间复杂度是  $O(\log_2 n)$ , 而 2 的 10 次幂是 1024, 也就是 1024 个数, 因此我们查到任意一个数只需要 10 次, 而 1000 小于 1024, 因此 10 次内自然肯定可以查到。对于有序数组, 无论数的范围是多少, 查找的时间复杂度都是  $O(\log_2 n)$ , 因为  $n$  是数的个数而不是数的范围。例 8.2.10 是二分查找的代码实现, 实现过程和上面描述的原理一致。

【例 8.2.10】二分查找。

```
#include <stdio.h>
#include <stdlib.h>
#define N 10
int binarySearch(int *arr,int low,int high,int target)
```



```
{
    int mid;
    while(low<=high)//high 是我们传入的值 9，能够访问到这个下标，所以这里小于等于 high
    {
        mid=(low+high)/2;
        if(arr[mid]>target)
        {
            high=mid-1;
        }else if(arr[mid]<target)
        {
            low=mid+1;
        }else{
            return mid;
        }
    }
    return -1;
}

int main()
{
    int a[]={2,14,18,31,32,46,71,82,85,99};
    int pos; //存储查找到的元素位置
    pos=binarySearch(a,0,N-1,14);
    printf("pos=%d\n",pos);
    system("pause");
}
```

该代码实例找到对应元素值后，返回对应的数组下标。二分查找的时间复杂度为  $O(\log n)$ ，学习二分查找的目的是学习分解思想。在计算机问题处理中，分解思想极其重要，我们通过分解来降低问题的复杂度，进而降低计算机的运算时间。工作中，大家可以直接使用 `bsearch` 来实现二分查找，其接口描述如下：

```
#include <stdlib.h> void *bsearch( const void *key, const void *buf, size_t num,
                                size_t size, int (*compare)(const void *, const void *) );
```

功能是用折半查找法从数组元素 `buf[0]` 到 `buf[num-1]` 匹配参数 `key`，如果函数 `compare` 的第一个参数小于第二个参数，那么返回负值；如果等于第二个参数，那么返回零值；如果大于第二个参数，那么返回正值。数组 `buf` 中的元素应以升序排列，函数 `bsearch()` 的返回值指向匹配项，如果没有发现匹配项，那么返回 `NULL`。

**思考题：**对于一个有序的单向链表，应如何使用二分查找？如果用 `bsearch`，那么如何使用

才能找到对应的元素？

### 8.2.4 哈希查找算法

二分查找的时间复杂度是  $O(\log n)$ ，但其在一些对查找性能要求极高的场合无法满足要求，比如淘宝网的负载均衡服务器。如果需要时间复杂度为  $O(1)$  的查找，那么就必须使用哈希查找，哈希又称哈希表、哈希表（Hash table），是根据键（Key）直接访问内存存储位置的数据结构。也就是说，它通过计算一个关于键值的函数，将所需查询的数据映射到表中的一个位置来访问记录，这加快了查找速度。这个映射函数称为哈希函数，存放记录的数组称为哈希表。

一个通俗的例子是，为了查找电话簿中某人的号码，可以创建一个按照人名首字母顺序排列的表（即建立人名  $\{x\}$  到首字母  $\{F(x)\}$  的一个函数关系），在首字母为 W 的表中查找“王”姓的电话号码，显然要比直接查找快得多。这里使用人名作为关键字，“取首字母”是这个例子中哈希函数的函数法则  $\{F(x)\}$ ，存放首字母的表对应哈希表。关键字和函数法则理论上可以任意确定。

给定表  $M$ ，存在函数  $f(\text{key})$ ，将任意给定的关键字值  $\text{key}$  代入函数后，若能得到包含该关键字的记录在表中的地址，则称表  $M$  为哈希表，函数  $f(\text{key})$  为哈希函数。所谓地址，就是一个整型数，我们的哈希表是一个数组， $f(\text{key})$  返回的值是一个整型数，这样就可以按照复杂度  $O(1)$  找到数组对应的下标位置。

哈希函数能使对一个数据序列的访问过程更加迅速、有效，通过哈希函数，数据元素将被更快地定位。

实际工作中需要视不同的情况采用不同的哈希函数，通常考虑的因素如下：

- 计算哈希函数所需的时间。
- 关键字的长度。
- 哈希表的大小。
- 关键字的分布情况。
- 记录的查找频率。

构造哈希函数的方法通常有以下几种。

（1）直接寻址法。取关键字或关键字的某个线性函数值为哈希地址，即  $H(\text{key}) = \text{key}$  或  $H(\text{key}) = a \cdot \text{key} + b$ ，其中  $a$  和  $b$  为常数（这种哈希函数称为自身函数）。若  $H(\text{key})$  中已有值，就往下一个找，直到  $H(\text{key})$  中没有值了就放进去。

（2）数字分析法。分析一组数据，比如一组员工的出生年、月、日，这时我们发现出生年、月、日的前几位数字大体相同，因此出现冲突的概率很大。但是，我们发现年、月、日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成哈希地址，那么冲突的概率明显降低。因此，数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突概率较低的哈希地址。

（3）平方取中法。无法确定关键字中哪几位分布较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为哈希地址。这是因为平方后中间几位和关键字中的每一位都相关，因此不同关键字会以较高的概率产生不同的哈希地址。

（4）折叠法。将关键字分割成位数相同的几部分，最后一部分的位数可以不同，然后取这几部分的叠加和（去掉进位）作为哈希地址。数位叠加分为移位叠加和间界叠加两种方法。移位叠

加将分割后的每一部分的最低位对齐，然后相加；间界叠加从一端向另一端沿分割界来回折叠，然后对齐相加。

(5) 随机数法。选择一个随机函数，取关键字的随机值作为哈希地址，通常用于关键字长度不同的场合。

(6) 除留余数法。取关键字被某个不大于哈希表表长  $m$  的数  $p$  除后，所得的余数为哈希地址，即  $H(\text{key}) = \text{key} \bmod p, p \leq m$ 。不仅可以对关键字直接取模，而且可以在折叠、平方取中等运算之后取模。对  $p$  的选择很重要，一般取素数或  $m$ ，若  $p$  选得不好，则容易产生同义词。

例 8.2.11 通过折叠法外加除留余数法来计算 5 个字符串的哈希值，并将其存入 `hashTable`，这样再次查找时就可以直接找到。代码中只是打印了每个字符串的哈希值，并将其存入 `hashTable`。这里没有编写查找某个字符串的代码，读者可以自行添加。同时在存入 `hashTable` 时，我们也可以再次自行申请空间，工作时 `hashTable` 是一个结构体指针数组，比如查找某个人的名字得到 `key` 值后，在对应下标处得到的是一个结构体指针，其中有对应人的年龄、性别、联系方式等信息。

【例 8.2.11】哈希实现。

```
#include <stdio.h>
#include <stdlib.h>

#define MAXKEY 1000
int hash(char *key)
{
    int h = 0, g;
    while (*key)
    {
        h = (h << 4) + *key++;
        g = h & 0xf0000000;
        if (g)
            h ^= g >> 24;
        h &= ~g;
    }
    return h % MAXKEY;
}

int main()
{
    char *pStr[5]={"xiongda","lele","hanmeimei","wangdao","fenghua"};
    char *hashTable[MAXKEY]={NULL};
    int i;
```

```
for(i=0;i<5;i++)
{
    printf("%10s hashValue=%d\n",pStr[i],hash(pStr[i]));
    hashTable[hash(pStr[i])]=pStr[i];
}
system("pause");
}
```

上面用的 *hash* 函数是经典的 *elf hash* 函数，但当数据量较大时，会发生哈希冲突：对不同的关键字可能得到同一哈希地址，即  $k_1 \neq k_2$ ，而  $f(k_1) = f(k_2)$ ，这种现象称为哈希冲突。解决哈希冲突有以下几种方法。

(1) 开放寻址法。 $H_i = (H(\text{key}) + d_i) \bmod m, i = 1, 2, \dots, k, k \leq m - 1$ ，其中  $H(\text{key})$  为哈希函数， $m$  为哈希表长， $d_i$  为增量序列，可有下列三种取法：

- ①  $d_i = 1, 2, \dots, m - 1$ ，称为线性探测再哈希。
- ②  $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \dots, \pm k^2, k \leq m/2$ ，称为二次探测再哈希。
- ③  $d_i$  为伪随机数序列，称为伪随机探测再哈希。

(2) 再哈希法。 $H_i = RH_i(\text{key}), i = 1, 2, \dots, k$ ， $RH_i$  均是不同的哈希函数，即在同义词产生地址冲突时计算另一个哈希函数的地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间。

(3) 单独链表法。将哈希到同一个存储位置的所有元素保存到一个链表中。

除了二分查找、哈希查找，针对上面讲的二叉树、红黑树，可以进行二叉树、红黑树查找，红黑树查找的时间复杂度依然是  $O(\log_2 n)$ ，相对于二分查找的好处是其新增和删除的操作次数少；另外，还有 B 树查找，B 树的增删查改将在 Linux 系统编程讲解 MySQL 的索引实现原理时介绍。

### 8.2.5 其他算法

计算机的算法还有很多，如贪心算法、动态规划等。准备考研复试的同学，在掌握了我们讲解的基本算法后，即可参考《计算机考研机试指南》进行算法准备。如果关于数据结构的知识薄弱，那么可以首先参考王道的数据结构考研图书，编写代码，完成对应数据结构的实现，然后学习王道的《计算机考研机试指南》。