

第 8 章 常用数据结构与算法

(视频讲解: 8 小时)

当衣柜不分格子时,把所有衣服都堆在一起,好像没有人这么做,为什么衣柜要分格子,为了更加高效的管理衣服。内存就像衣柜,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,虽然有了结构,但是我们摆放衣服时,我们会把经常穿的衣服放在靠外,把基本不穿的衣服压箱底,这就是存取方法,通过有效的存取方法可以提高我们访问数据的效率,我们称之为算法。

学习算法的必须先掌握常用的数据结构,而常用的数据结构有哪些呢,有数组、栈、队列、链表、树、堆、散列表、图等,除了图,其他数据结构下面我们讲一一进行讲解。

学习本章,你将掌握:

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

8.1 数据结构

8.1.1 栈

栈(stack)又名堆栈,它是一种运算受限的线性表。其限制是仅允许在表的一端进行插入和删除运算,这一端被称为栈顶,相对地,把另一端称为栈底。如图 8.1.1-1,向一个栈插入新元素又称作进栈、入栈或压栈,它是把新元素放到栈顶元素的上面,使之成为新的栈顶元素;从一个栈删除元素又称作出栈或退栈,它是把栈顶元素删除掉,使其相邻的元素成为新的栈顶元素。由于堆栈数据结构只允许在一端进行操作,因而按照后进先出(LIFO, Last In First Out)的原理运作

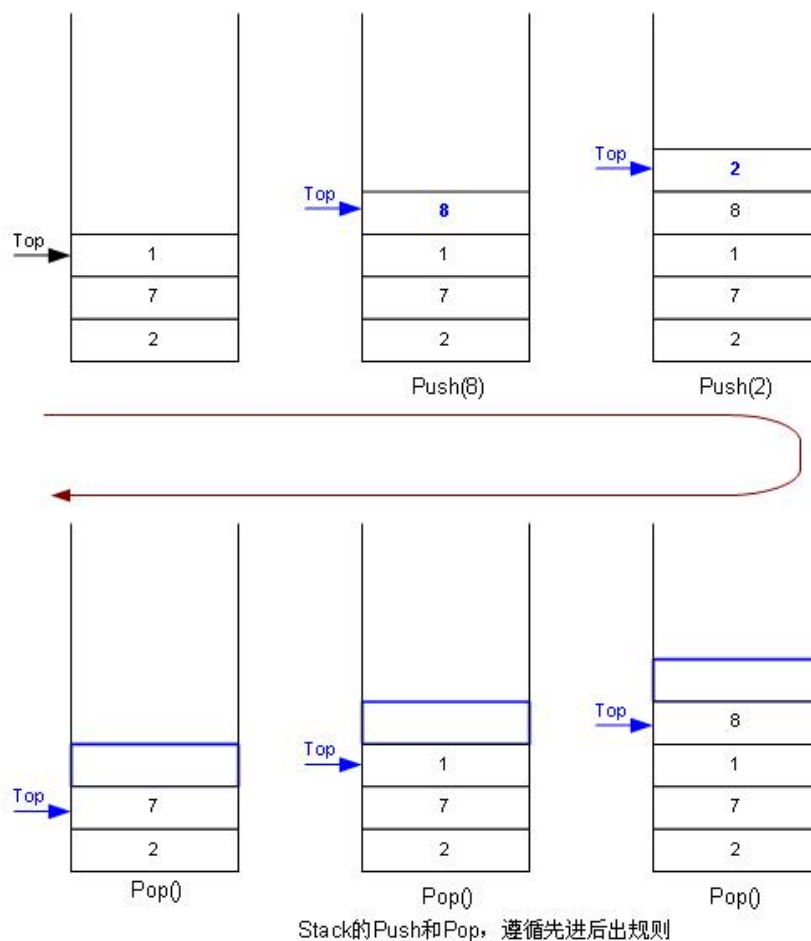


图 8.1.1-1

栈可以用**数组**实现，也可以用**链表**实现，这里我们将通过链表结构来实现栈，之前在前一章我们掌握链表的增删改查，这里我们可以通过链表的头部插入法，头部删除法，实现先进后出的效果，考研的同学都知道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讲明，因为我们通过链表实现，存储结构如例 8.1.1-1：

【例 8.1.1-1】实现一个 stack

```
typedef struct tag
{
    int m_val;
    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;
```

要求 Stack 具有下列功能，下面函数就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
```

首先我们需要初始化栈，将头指针赋值为 NULL，栈大小初始化为零，具体实现如下：

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

因为栈里边是空的，所以首先要做压栈操作，我们先实现压栈函数，具体实现如下：

```
void push(pStack p, int val)
{
    pNode pnew=(pNode) calloc(1, sizeof(Node)); //为压栈的新节点申请空间
    pnew->m_val=val; //将要压栈的值放入对应节点空间
    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，接下来我们实现 top 函数，注意 top 只是获得栈顶元素值，并不会弹栈，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，栈为空时，empty 函数返回 1，栈不空时，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，就是返回栈的大小，因为我们存储了栈的大小，所以直接返回对应变量值即可，详细代码如下：

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

这里写了一个简单的测试程序，小伙伴可以在自己的集成开发环境 VS 中，自行编写进行测试，如果想获取完整代码的小伙伴，可以加入书籍前言的 QQ 群进行获取哦，实践是提升能力的最快路径！

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

```

8.1.2 队列

队列：是先进先出（FIFO, First-In-First-Out）的线性表。在具体应用中通常用链表或者数组来实现。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，实现每个元素达到先进先出的效果。由于代码与之前类似，所以不再赘述。

循环队列：如果通过链表实现，让链表尾指针的 **pnext**，指向头指针，即可用链表实现循环队列，这也是链表当天的作业，难度较小。如果通过数组实现，对数组进行遍历，当 **i** 等于最后一个元素时，将 **i** 赋值为 0，重新回到数组元素起始点，见例 8.1.2-1。

【例 8.1.2-1】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，我们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，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的入队出队的，当出队一个元素时，Q->front就会加1，当头部和尾部相等时，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，因此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，实际我们可以放入的元素个数为4个，因此我们判断队列是否满的策略是，对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MaxSize得余数。

```

    if((Q->rear+1)%MaxSize==Q->front)
        return 0;
    Q->data[Q->rear]=x;//3 4 5 6
    //每次放入一个元素有，需要对Q->rear进行增1，因为数组可以访问的最大下标为
    4，为了防止到达数组的末尾，因此我们需要模MaxSize
    Q->rear=(Q->rear+1)%MaxSize;
    return 1;
}
//出队
int DeQueue(SqQueue *Q, ElemType *x)
{
    //如果front与rear相等，代表循环队列为空，这时候直接返回
    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，对front加1
    return 1;
}

```

//下面我们写一个测试的例子，首先我们定义一个循环队列Q，然后对其进行初始化，初始化后判断是否为空，这时候会打印循环队列为空，然后我们依次入队，3, 4, 5, 6, 7，总计五个元素，发现当入队7时，入队失败，因为这时队列已满，这时我们出队两个元素，然后再次入队元素8，就可以入队成功。小伙伴在实际运行代码的过程中，可以通过内存窗口，观察内存变化过程，从而更深一步理解循环队列。

```

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入队时将入队失败
}

```

```

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

```

8.1.3 二叉树

在计算机科学中，**二叉树是每个节点最多有两个子树的树结构**。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。二叉树的每个节点至多只有二棵子树(不存在度大于 2 的节点)，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} 个节点；深度为 k 的二叉树至多有 $2^k - 1$ 个节点；对任何一棵二叉树 T ，如果其终端节点数为 n_0 ，度为 2 的节点数为 n_2 ，则 $n_0 = n_2 + 1$ 。

一棵深度为 k ，且有 $2^k - 1$ 个节点称之为满二叉树；深度为 k ，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中，序号为 1 至 n 的节点对应时，称之为完全二叉树。如图 8.1.3-1 所示

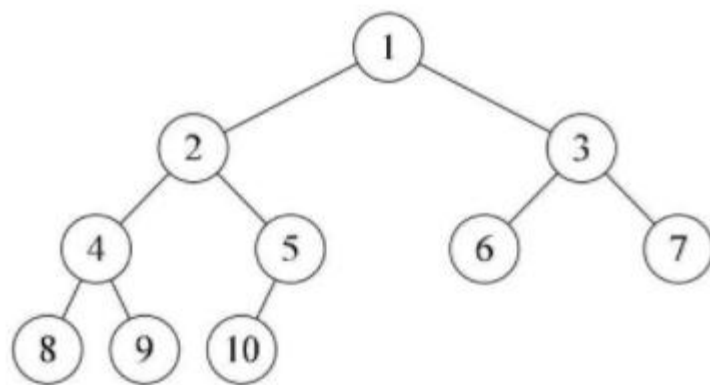


图 8.1.3-1

完全二叉树的特点是：

- 1) 只允许最后一层有空缺节点且空缺在右边，即叶子节点只能在层次最大的两层上出现；
- 2) 对任一节点，如果其右子树的深度为 j ，则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（顺序存储在堆排序中进行学习），也可以是链式存储，下面我们来看通过链式存储来实现一颗二叉树存储，见【例 8.1.3-1】。

【例 8.1.3-1】二叉树的建树

```

#include <stdio.h>
#include <stdlib.h>
//首先我们需要将二叉树节点的数据结构定义清晰
typedef struct node{
    char c;//节点内存的元素类型
    struct node *left;//指向左子节点的指针
    struct node *right;//指向右子节点的指针
}Node,*pNode;

void preOrder(pNode p)
{
    if(p!=NULL)
    {
        putchar(p->c);//打印当前节点
        preOrder(p->left); //打印左子节点
        preOrder(p->right); //打印右子节点
    }
}

void midOrder(pNode p)
{
    if(p!=NULL)
    {
        midOrder(p->left); //打印左子节点
        putchar(p->c); //打印当前节点
    }
}
  
```

```

        midOrder(p->right); //打印右子节点
    }
}

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

```

//我们建立二叉树的方法是业界最常见的的层次建树，首先我们进树的元素是从A到J，总计10个元素，层次建树，一层放满后，才会放下一次，所以我们通过层次建树，建立的二叉树，一定是一颗满二叉树。为了建树方便，我们首先为每一个节点申请空间，将每一个节点的指针值，存入一个指针数组，这样通过for循环就可以依次拿到进树的元素，进树时，首先第0个节点(根部节点)的左子节点为空，我们就放左边，接着放右边，当右边也放置节点后，就需要将j进行加1，这样进树的节点将放入下一个节点的左子节点，依次循环往复，最终实现二叉树的层次建树。

```

#define N 10
int main()
{
    char c[N+1]="ABCDEFGHJIJ";
    int i, j;
    pNode a[N];
    //通过for循环，为每一个要进树的节点申请空间，并将节点值放入
    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) //没有左子节点，就放入左子节点
        {
            a[j]->left=a[i];
        } else if(NULL==a[j]->right) //否则放入右子节点
        {
            a[j]->right=a[i];
            j++; //放入右子节点后，当前节点满了，因此对j进行加1，从而下次进树时，
            //放置到下一个节点，我们通过下标j记录进树的位置
        }
    }
}

```



```

    }
}
//接下来我们通过打印树的前序，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]);
system("pause");
return 0;
}

```

下面我们来看下前序遍历，前序遍历是首先打印根节点，然后打印左节点，然后打印右节点，如果我们的二叉树是图 8.1.3-2，前序遍历结果为 ABC

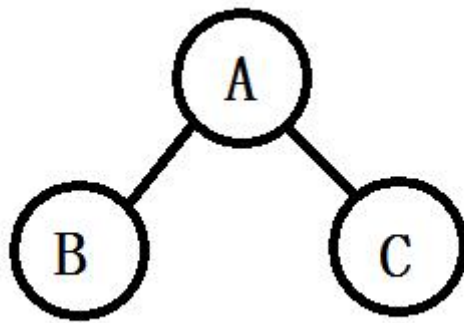


图 8.1.3-2

这个时候我们又放入两个节点，如图 8.1.3-3 所示，单独看 B 节点的子树，打印顺序为 BDE，这个时候再与上面的结合，就是 ABDEC，遵守的原则是子节点要跟自己的父亲相邻。

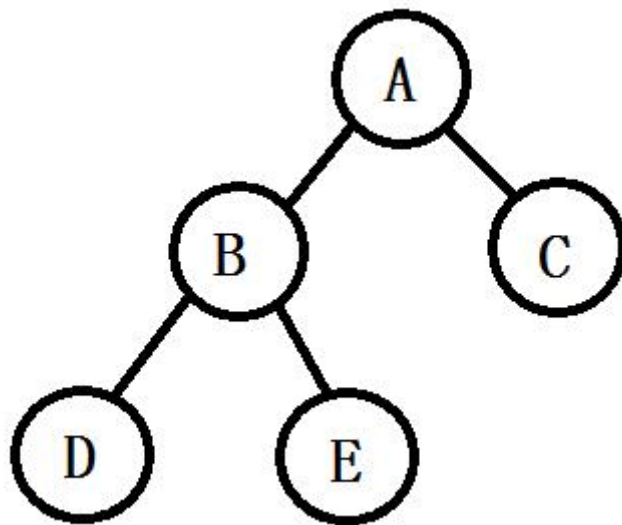


图 8.1.3-3

根据上面的推演，那么依次推演的结果是

ABDECFG //对于 C 节点

ABDHIECFG //对于 D 节点

ABDHIEJCFG //对于 E 节点, 也就是图 8.1.3-4

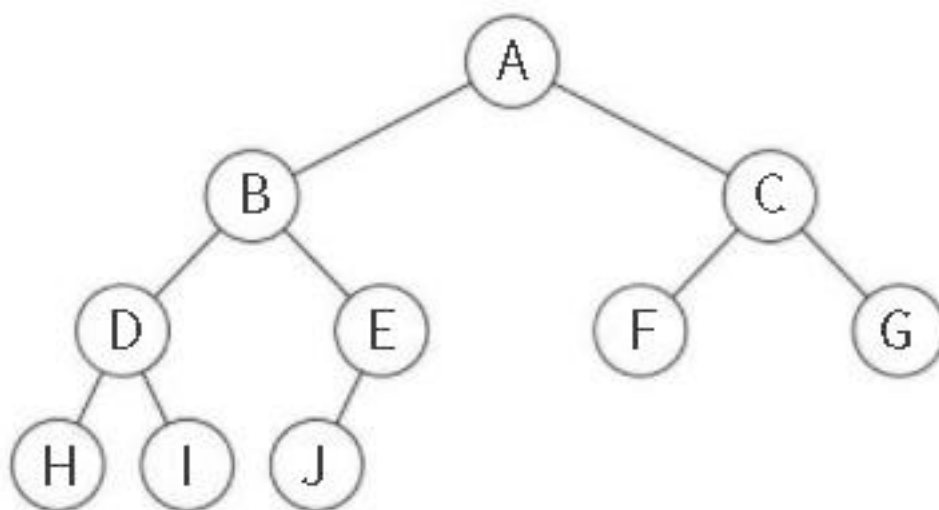


图 8.1.3-4

对于二叉树的建树，我们进行一个总结，首先我们需要将每一个字母作为一个二叉树的节点进行存储，所以首先通过循环，为每一个节点申请空间，通过指针数组存储每一个节点对应的指针值。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，节点数目多少不确定，代码如何进行修改？因为工作中我们遇到的问题是节点数目不确定的情况。

上面层次建树时，我们是提前使用了一个指针数组，这样元素进树时，当某个节点满时，我们对 j 进行加 1，就得到下一个要放置元素的节点位置，通过这个原理，如果是节点数目不确定的情况下，要进行层次建树，我们就需要使用一个辅助队列，如图 8.1.3-5 所示，每往树中放一个元素时，我们需要将其放入队列尾部，比如图中的 C 元素，当 C 元素放入树中后，我们发现元素 A 的左右子节点都放满了，这时辅助队列的头 `queHead` 始终指向要放的节点位置，因此如图 8.1.3-5 中的箭头所示，我们需要将 `queHead` 移动指向元素 B，这样当放入元素 D 时，我们是把 D 作为元素 B 的左子节点进行放入。

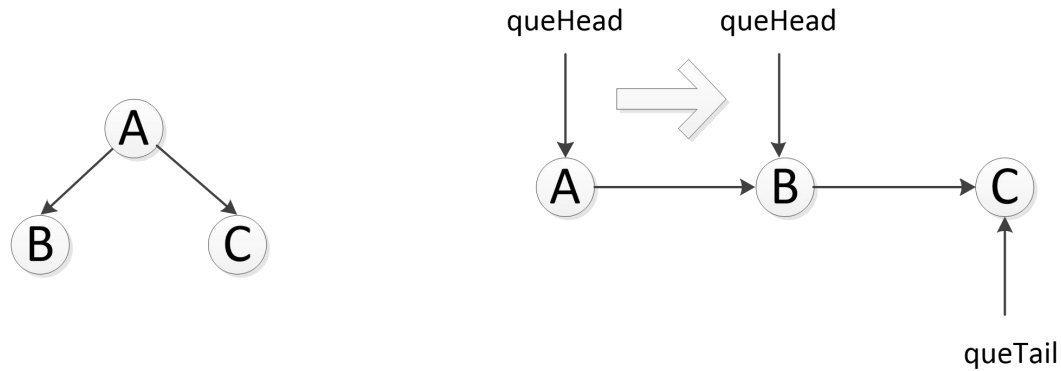


图 8.1.3-5

【例 8.1.3-2】是具体的代码实现，代码实现是采用 buildBinaryTree，每次往树中添加一个元素时，我们调用一次 buildBinaryTree，当然有时如果树建好了，辅助队列不需要使用，那么我们需要销毁辅助队列，销毁辅助队列的工作具体如何实现，相信掌握了前面链表操作的同学一定能够自己实现。

【例 8.1.3-2】元素个数不限的二叉树建树

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

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

```
typedef char ElemType; //为了树中放任意类型元素方便
```

```
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，辅助队列每个元素放的是树中元素节点的地址
值，这样才能快速确定树中节点
```

```
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，新节点作为树根，同时新节点即作为队列头，也
    作为队列尾
    {
```

```

        *treeRoot=treeNew;
        *queHead=queNew;
        *queTail=queNew;
    }else{
        (*queTail)->pNext=queNew; //新节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; //当某个节点放满时，队列头向后移动一个节点，
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); //与之前例子一致，所以代码未列出，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（英语：**Red-black tree**）是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它在 1972 年由鲁道夫·贝尔发明，被称为"对称二叉 B 树"，它现代的名字源于 Leo J. Guibas 和 Robert Sedgwick 于 1978 年写的一篇文章。

红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在 $O(\log_2 n)$ 时间内完成查找，插入和删除，这里的 n 是树中元素的数目。

二叉树中很重要是掌握红黑树（RB 树），因为操作系统内核中，C++ 的 STL，包括 Java 的数据结构中都大量用到红黑树，红黑树的增删查改的复杂度均为 $\log_2 n$ ，可能有同学会说自平衡二叉查找树也可以啊，AVL 树是最早被发明的自平衡二叉查找树。在 AVL 树中，任一节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。查找、插入和删除在平均和最坏情况下的时间复杂度都是 $O(\log_2 n)$ 。增加和删除元素的操作则可能需要借由一次或多次树旋转，以实现树的重新平衡。

红黑树相对于 AVL 树的时间复杂度是一样的，但是优势是当插入或者删除节点时，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高于 AVL 树，大量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。

红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

1. 节点是红色或黑色。
2. 根是黑色。
3. 所有叶子都是黑色（叶子是 NIL 节点）。
4. 每个红色节点必须有两个黑色的子节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
5. 从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

图 8.1.4-1 是一个具体的红黑树的图例（下面圆圈浅色背景代表红色，圆圈背景纯黑代表黑色，例如 13 结点是黑色，8 结点是红色）：

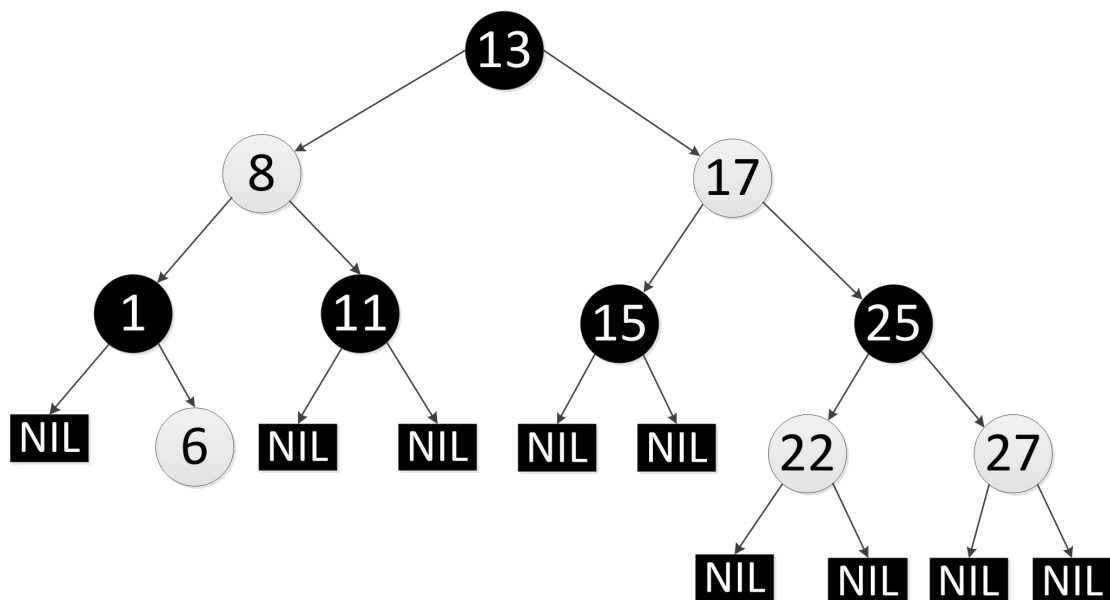


图 8.1.4-1

NIL 节点可以理解为没有，不存在，因此，对于红黑树，记住的是第四条和第五条特性即可，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。可以这样来

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，这个世界冷静的人都是均衡的，所以才有世界和平，对应性质 5。只要从根节点到任意叶子节点的黑色节点数目相等，那么从任一节点到叶子节点的黑色节点数目也是相等的。

这些约束确保了红黑树的关键特性：**从根到叶子的最长的可能路径不多于最短的可能路径的两倍长**。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

要知道为什么这些性质确保了这个结果，注意到性质 4 导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质 5 所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。

因为每一个红黑树也是一个特化的二叉查找树，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相同。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 $O(\log_2 n)$ 次。

8.1.4.1 红黑树的插入

我们首先以二叉查找树的方法增加节点并标记它为红色（就是新增任何结点，我们把它标记为红色后，放入），在下面的示意图中，将要插入的节点标为 **N**，**N** 的父节点标为 **P**，**N** 的祖父节点标为 **G**，**N** 的叔父节点标为 **U**。

对于每一种情形，我们将使用示例代码来展示。通过下列函数，可以找到一个节点的叔父和祖父节点：

```
node* grandparent(node *n) {
    return n->parent->parent;
}

node* uncle(node *n) {
    if(n->parent == grandparent(n)->left)
        return grandparent(n)->right;
    else
        return grandparent(n)->left;
}
```

针对红黑树的插入，我们分为以下五种情形（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，重点在情形 3，情形 4，情形 5。**

情形 1: 新节点 **N** 位于树的根上，没有父节点，这种情况就是红黑树没有其他节点，只有根节点，那就直接标记为黑色，或者时经过调整后，红黑树的根节点被变为红色了，那就直接把它图为黑色。在这种情形下，我们把它重绘为黑色以满足性质 2。因为它在每个路径上对黑节点数目增加一，性质 5 匹配。

```
void insert_case1(node *n) {
    if(n->parent == NULL)
        n->color = BLACK;
    else
        insert_case2 (n); //代表请看下面的情形 2
}
```

情形 2:新节点的父节点 **P** 是黑色，所以性质 4 没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质 5 也未受到威胁，尽管新节点 **N** 有两个黑色叶子子节点；但由于新节点 **N** 是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

```
void insert_case2(node *n) {
    if(n->parent->color == BLACK)
        return; /* 树仍旧有效*/
    else
        insert_case3 (n); //代表请看下面的情形 3
}
```

情形 3:如果父节点 **P** 和叔父节点 **U** 二者都是红色，（此时新插入节点 **N** 做为 **P** 的左子节点或右子节点都属于情形 3，这里右图仅显示 **N** 做为 **P** 左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点 **G** 为红色（用来保持性质 5）。现在我们的新节点 **N** 有了一个黑色的父节点 **P**。因为通过父节点 **P** 或叔父节点 **U** 的任何路径都必定通过祖父节点 **G**，在这些路径上的黑节点数目没有改变。但是，红色的祖父节点 **G** 可能是根节点，这就违反了性质 2，也有可能祖父节点 **G** 的父节点是红色的，这就违反了性质 4。为了解决这个问题，我们在祖父节点 **G** 上递归地进行情形 1 的整个过程。（把 **G** 当成是新加入的节点进行各种情形的检查），具体调整如图 8.1.4.1-1 所示：

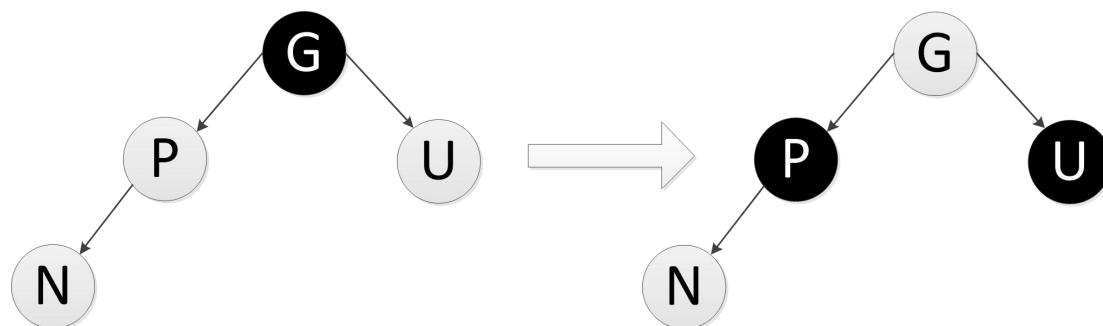


图 8.1.4.1-1 情形 3

关键代码如下：

```
void insert_case3(node *n) {
    if(uncle(n) != NULL && uncle(n)->color == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(grandparent(n));
    }
```

```

    }
    else
        insert_case4(n); //代表请看下面的情形 4
    }

```

有同学会有这样的疑问，如果 G 是根节点，通过情形一直接变为黑色即可，但是如果 G 不是根节点，通过 G 的父亲也是红色怎么办，当 G 的父亲也是红色时，我们就需要重新把 G 作为子节点，重新来看 G 的父亲，叔叔，爷爷的颜色与位置，确认到底是情形 3，还是下面的情形 4，或者情形 5，符合那种情形，就按对应情形进行调整即可。

情形 4:父节点 P 是红色而叔父节点 U 是黑色或缺少，并且新节点 N 是其父节点 P 的右子节点而父节点 P 又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色；接着，我们按情形 5 处理以前的父节点 P 以解决仍然失效的性质 4。因为没有增加黑色节点的数目，所以性质 5 仍有效。，具体调整如图 8.1.4.1-2 所示：

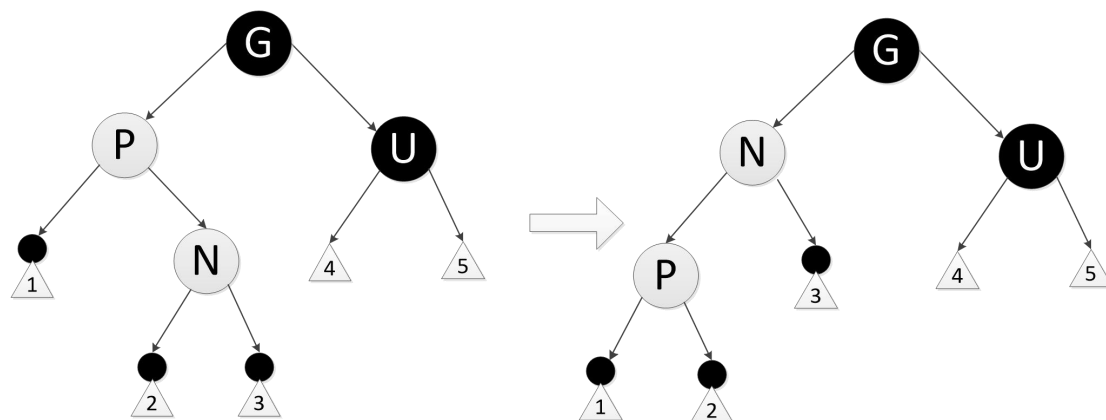


图 8.1.4.1-2 情形 4

图 8.1.4.1-1 中 1,2,3 的位置有 3 个小圆是为了说明和右边的总计的黑色节点数目一致，不用过于关注，标注 1,2,3 的目的是为了让大家看到 N 节点左旋后，N 的孩子，P 的孩子发生的变化。

代码如下：

```

void insert_case4(node *n) {
    if(n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(n->parent);
        n = n->left;
    } else if(n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(n->parent); //这种是图 8.1.4.1-2 当中的对称情况，父亲在爷爷的
        // 右边，孩子在父亲的左边
        n = n->right;
    }
    insert_case5(n); //代表请看下面的情形 5
}

```

情形 5: 父节点 P 是红色而叔父节点 U 是黑色或缺少，新节点 N 是其父节点的左子节点，而

父节点 P 又是其父节点 G 的左子节点。在这种情形下，我们进行针对祖父节点 G 的一次右旋转；在旋转产生的树中，以前的父节点 P 现在是新节点 N 和以前的祖父节点 G 的父节点。我们知道以前的祖父节点 G 是黑色，否则父节点 P 就不可能是红色（如果 P 和 G 都是红色就违反了性质 4，所以 G 必须是黑色）。我们切换以前的父节点 P 和祖父节点 G 的颜色，结果的树满足性质 4。性质 5 也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点 G，现在它们都通过以前的父节点 P。在各自的情形下，这都是三个节点中唯一的黑色节点。如图 8.1.4.1-3 所示：

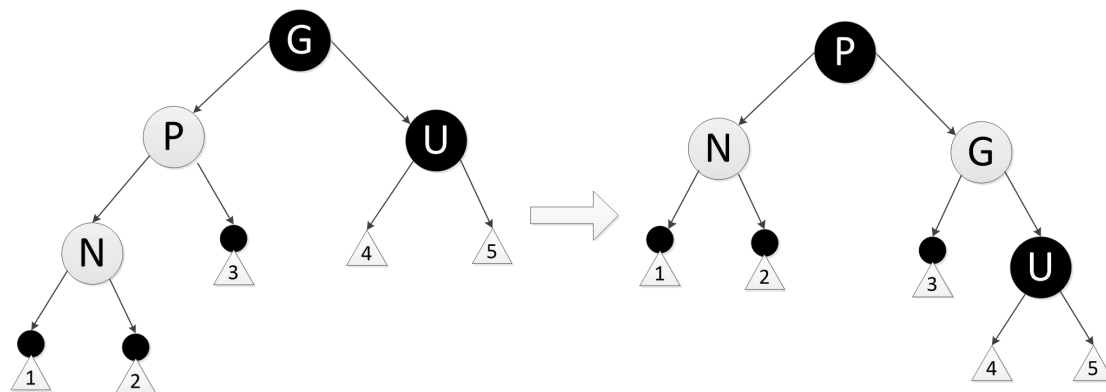


图 8.1.4.1-3 情形 5

通过图 8.1.4.1-3 变化可以看出，对 G 也就是爷爷节点进行右旋，同时交换爷爷 G 与父亲 P 的颜色，不仅可以满足性质 5，就是从根节点到任意叶子节点的黑色节点数目不变，同时满足性质 4，没有红色节点数目相邻。

代码如下：

```
void insert_case5(node *n) {
    n->parent->color = BLACK;
    grandparent (n)->color = RED;
    if(n == n->parent->left && n->parent == grandparent(n)->left) {
        rotate_right(grandparent(n));
    } else {
        /* Here, n == n->parent->right && n->parent == grandparent (n)->right */
        rotate_left(grandparent(n)); //对称情况，如果父亲 P 在爷爷 G 的右边，同时
        //孩子 N 在父亲 P 的右边，那么直接对爷爷进行左旋
    }
}
```

由于情形 3 到情形 5 相等复杂一些，针对上面的情况，通过图 8.1.4.1-4 为大家进行了总结，图 8.1.4.1-4（图中浅色代表红色结点，深色代表黑色结点）的情形三，叔叔 U 是红色，无论 N 是 P 的左孩子还是右孩子，都是采用变色机制。针对情形四，叔叔不存在或者是黑色，就需要对父亲 P 进行相应的旋转，旋转后刚好变成了对应自己下方情形五的样子，然后再对爷爷 G 进行相应的旋转即可。

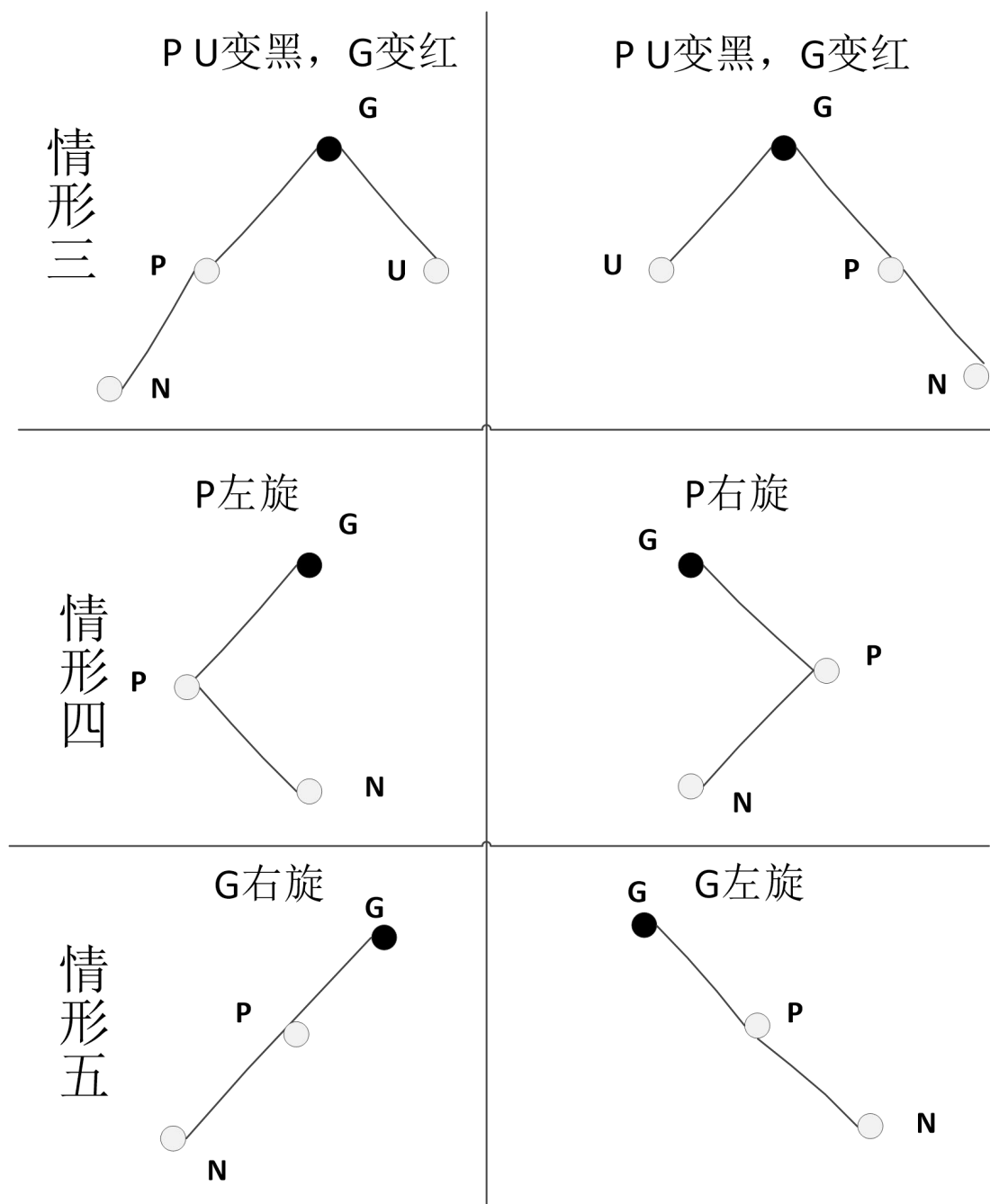


图 8.1.4.1-4 红黑树插入总结

理解了图 8.1.4.1-4 以后，再看下方核心的代码【例 8.1.4.1-1】实现，就会觉的一目了然，核心代码每一部分都进行了详细的注释，分别对应图 8.1.4.1-4 中的情形三，情形四，情形五，rbtree_insert_fixup 是实现红黑树的关键函数，针对不理解代码的同学，可以加入 QQ 群，获取完整代码（包含测试用例），运行协助理解，群内可以获取讲解视频，理解有困难的同学可以通过视频协助进行理解。红黑树已经是目前校招面试大公司必须掌握的数据结构！而很多书籍没有这部分的讲解，一些书籍结构书籍对这部分的讲解也不够详细透彻。（龙哥在这里给大家划重点了！）

【例 8.1.4.1-1】红黑树的插入修正函数

```
/*
```

```
* 红黑树插入修正函数
```

```

*
* 在向红黑树中插入节点之后(失去平衡)，再调用该函数；
* 目的是将它重新塑造成一颗红黑树。
*
* 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“父节点存在，并且父节点的颜色是红色”，就说明需要调整啦！
    while ((parent = rb_parent(node)) && rb_is_red(parent))
    {
        gparent = rb_parent(parent);
        //若“父节点”是“祖父节点的左孩子”
        if (parent == gparent->left)
        {
            // Case 1条件：叔叔节点存在且是红色，这里是情形三
            {
                Node *uncle = gparent->right;
                if (uncle && rb_is_red(uncle))//没有节点进入该分支，如何构造？
                {
                    rb_set_black(uncle);
                    rb_set_black(parent);
                    rb_set_red(gparent);
                    node = gparent;
                    continue;
                }
            }
            // Case 2条件：叔叔是黑色或不存在，且当前节点是右孩子，这里是情形四
            if (parent->right == node)
            {
                Node *tmp;
                rbtree_left_rotate(root, parent);
                tmp = parent;
                parent = node;
                node = tmp;
            }
            // Case 3条件：叔叔是黑色，且当前节点是左孩子。这里是情形五
            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//若父节点是祖父节点的右孩子，下面三种和上面的是对称的
{
    // Case 1条件：叔叔节点是红色，这里是情形三
    {
        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：叔叔是黑色，且当前节点是左孩子，这里是情形四
    if (parent->left == node)
    {
        Node *tmp;
        rbtree_right_rotate(root, parent);
        tmp = parent;
        parent = node;
        node = tmp;
    }
    // Case 3条件：叔叔是黑色，且当前节点是右孩子，这里是情形五
    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);
}
}
// 将根节点设为黑色
rb_set_black(root->node);
}

```

8.1.4.2 红黑树的删除

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，并它的值转移到要删除的节点中（比如如果要删除图 8.1.4.2-1 中的 13，我们可以把左子树的 11 或者右子树的 15 与 13 进行交换）。

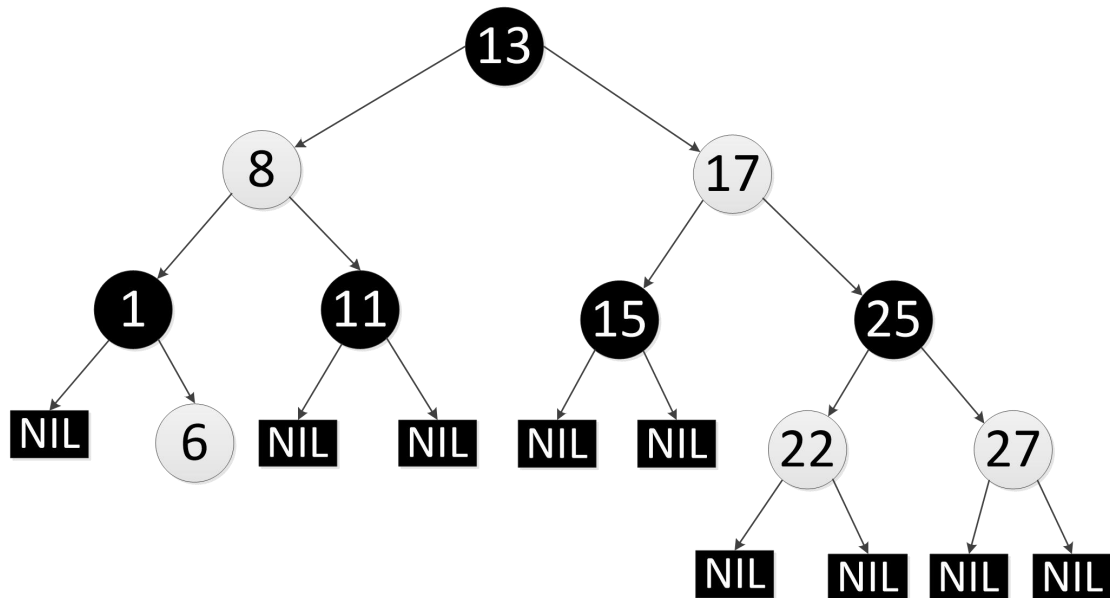


图 8.1.4.2-1

我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子（即最多只会会有一个儿子，假如 15 有一个红色的右子结点 16）。因为只是复制了一个值（没有复制颜色），不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

在本文余下的部分中，我们只需要讨论删除只有一个儿子的节点（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质 3 和性质 4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质 5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质 5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质 5。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况（这种情况下该结点的两个儿子都是叶子结点，否则若其中一个儿子是黑色非叶子结点，另一个儿子是叶子结点，那么从该结点通过非叶子结点的儿子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儿子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节点替换为它的儿子。出于方便，称呼这个儿子为 **N**（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为 **S**。在下面的示意图中，我们还是使用 **P** 称呼 **N** 的父亲，**SL** 称呼 **S** 的左儿子，**SR** 称呼 **S** 的右儿子。我们将使用下述函数找到兄弟节点：

```
struct node *
sibling(struct node *n)
{
    if(n == n->parent->left)
        return n->parent->right;
    else
```

```

        return n->parent->left;
    }

```

我们可以使用下列代码进行上述的概要步骤,这里的函数 `replace_node` 替换 `child` 到 `n` 在树中的位置,我们删除的是 `n`, 因为 `n` 只有一个孩子, 用 `n` 的 `child` 替换掉 `n`, 下面的讨论我们假定 `n` 一定是有一个孩子的, 代码如下:

```

void
delete_one_child(struct node *n)
{
    /*
     * Precondition: n has at most one non-null child.
     */
    struct node *child = is_leaf(n->right)? n->right: n->left;

    replace_node(n, child);
    if(n->color == BLACK) {
        if(child->color == RED)
            child->color = BLACK;
        else
            delete_case1 (child); //如果孩子是黑色的, 请看下面的情形
1 的分析
    }
    free (n);
}

```

如果 `N` 和它初始的父亲是黑色, 则删除它的父亲导致通过 `N` 的路径都比不通过它的路径少了一个黑色节点。因为这违反了性质 **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不要混到一谈)需要考虑, 下面情形的示意图中, 我们已经用 `child` 替换掉了 `N`, 因此示意图中 `N` 所在的分支比不通过 `N` 的分支少一个黑色节点:

情形 1: `N` 是新的根。在这种情形下, 我们就做完了。我们从所有路径去除了一个黑色节点, 而新根是黑色的, 所以性质都保持着。

```

void
delete_case1(struct node *n)
{
    if(n->parent != NULL)
        delete_case2 (n); //如果删除的节点不是根
}

```

注意: 在情形 2、5 和 6 下, 我们假定 `N` 是它父亲的左儿子。如果它是右儿子, 则在这些情形下的左和右应当对调。

情形 2: `S` 是红色。如图 8.1.4.2-2 所示, 在这种情形下我们在 `N` 的父亲上做左旋转, 把红色兄弟转换成 `N` 的祖父, 我们接着对调 `N` 的父亲和祖父的颜色。完成这两个操作后, 尽管所有

路径上黑色节点的数目没有改变，但现在 N 有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色 S 的一个儿子），所以我们可以接下去按情形 4、情形 5 或情形 6 来处理。

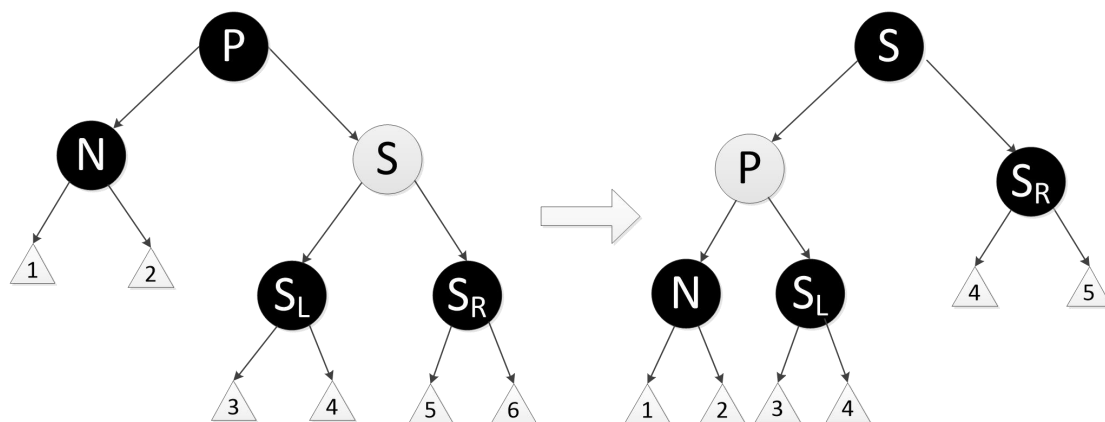


图 8.1.4.2-2 删除情形 2

（注意：这里的图 8.1.4.2-2 中没有显示出来，N 是删除了黑色节点（假如删除的黑色节点为 X）后替换上来的子节点，所以这个过程中由 P→X→N 变成了 P→N，实际上是少了一个黑色节点，也可以理解为 Parent (Black) 和 Silbling (Red) 那么他们的孩子黑色节点的数目肯定不等，让他们做新兄弟肯定是不平衡的，还需后面继续处理。N 与 S_L 做兄弟，那么通过 N 这边的分支肯定比通过 S_L 的要少一个黑色节点，本来通过 N 和 S_L 的黑色节点相等的，因为我们的图 N 的分支是 N 的 child 把 N 替换后的，所以比 S_L 的分支这边少一个黑色节点）

代码如下：

```
void
delete_case2(struct node *n)
{
    struct node *s = sibling (n);

    if(s->color == RED) {
        n->parent->color = RED;
        s->color = BLACK;
        if(n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent); //考虑到对称情况
    }
    delete_case3 (n); //请看情形 3
}
```

情形 3： N 的父亲、S 和 S 的儿子都是黑色的，如图 8.1.4.2-3 所示。在这种情形下，我们简单的重绘 S 为红色。结果是通过 S 的所有路径，它们就是以前不通过 N 的那些路径，都少了一个黑色节点。因为删除 N 的初始的父亲使通过 N 的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，把通过 P 的整体看做一颗右子树或者左子树，那么通过 P 的所有路径

现在比不通过 P 的路径（另外一边的子树）少了一个黑色节点，所以仍然违反性质 5。要修正这个问题，我们要从**情形 1** 开始，在 P 上做重新平衡处理。也就是把 P 看成一个孩子，也就是把 P 看成 N，重新做平衡处理。

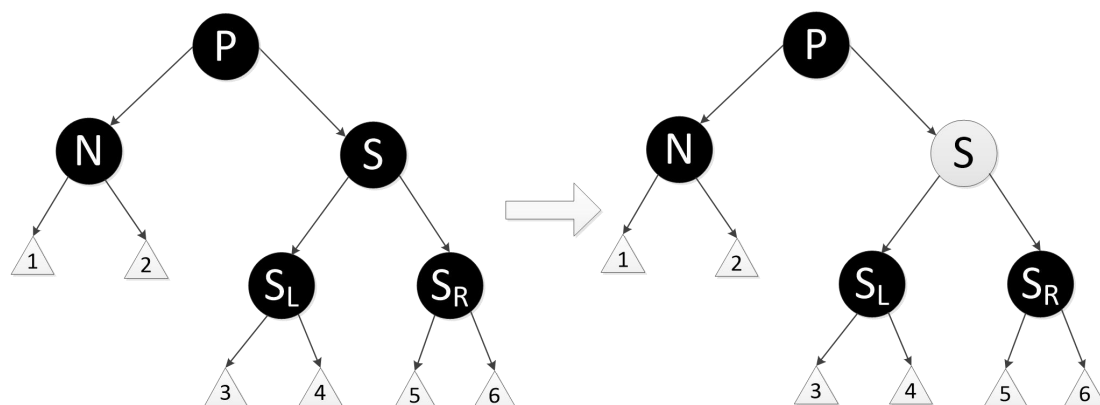


图 8.1.4.2-3 删除情形 3

如果不符合情形 3，那么就接着匹配情形 4，具体代码如下：

```
void
delete_case3(struct node *n)
{
    struct node *s = sibling (n);

    if((n->parent->color == BLACK)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
        s->color = RED;
        delete_case1(n->parent); //重新进行情形 1 匹配,我们传入的是 N 的父
        亲 P，也就是要调整的节点变为 P
    } else
        delete_case4 (n); //如果不符合情形 3，请看情形 4
}
```

情形 4: S 和 S 的儿子都是黑色，但是 N 的父亲是红色，如图 8.1.4.2-4 所示，在这种情形下，我们简单的交换 N 的兄弟和父亲的颜色。这不会影响不通过 N 的路径的黑色节点的数目，但是它在通过 N 的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

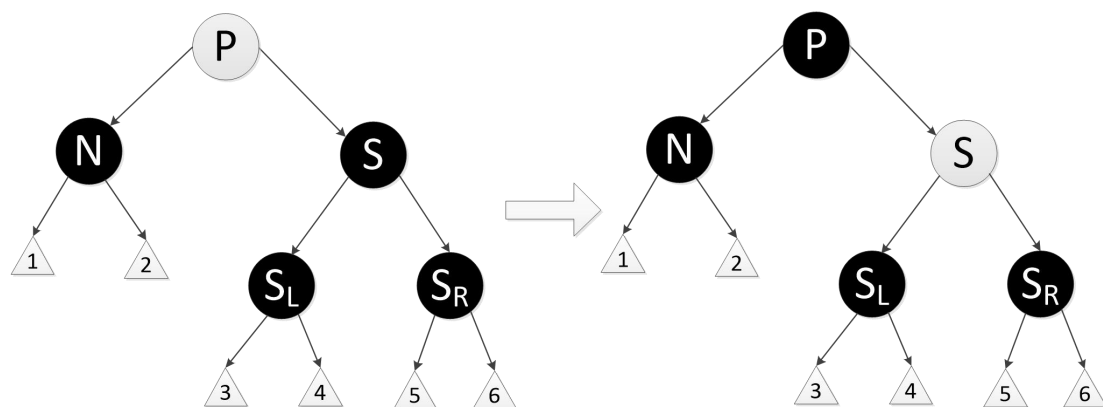


图 8.1.4.2-4 删除情形 4

操作代码如下：

```
void
delete_case4(struct node *n)
{
    struct node *s = sibling (n);

    if ((n->parent->color == RED)&&
        (s->color == BLACK)&&
        (s->left->color == BLACK)&&
        (s->right->color == BLACK)) {
        s->color = RED; // N 的兄弟 S 设置为红色
        n->parent->color = BLACK; // N 的父亲设置为黑色
    } else
        delete_case5 (n); // 如果不符合情形 4，请看情形 5
}
```

情形 5：S 是黑色，S 的左儿子是红色，S 的右儿子是黑色，而 N 是它父亲的左儿子，如图 8.1.4.2-5 所示。在这种情形下我们在 S 上做右旋转，这样 S 的左儿子成为 S 的父亲和 N 的新兄弟。我们接着交换 S 和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在 N 有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了**情形 6**。N 和它的父亲都不受这个变换的影响。

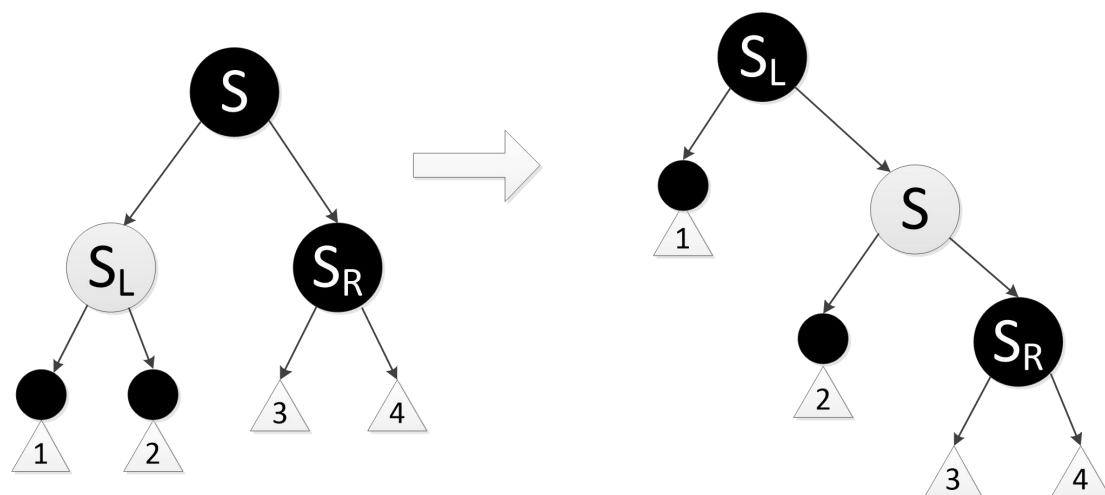


图 8.1.4.2-5 删除情形 5

具体操作代码如下：

```
void
delete_case5(struct node *n)
{
    struct node *s = sibling (n);

    if (s->color == BLACK) {
        if ((n == n->parent->left)&&
            (s->right->color == BLACK)&&
            (s->left->color == RED)) { // 这种是 n 是其父亲的左边，S 是其父亲的右边
            s->color = RED;
            s->left->color = BLACK;
            rotate_right (s);
        } else if ((n == n->parent->right)&&
            (s->left->color == BLACK)&&
            (s->right->color == RED)) { // 这种情况是 n 是其父亲的右边，S 是其父亲的左边
            s->color = RED;
            s->right->color = BLACK;
            rotate_left (s);
        }
    }
    delete_case6 (n); // 调整完毕后，再通过情形 6 进行调整
}
```

情形 6： S 是黑色，S 的右儿子是红色，而 N 是它父亲的左儿子，如图 8.1.4.2-6 所示。在这种情形下我们在 N 的父亲上做左旋转，这样 S 成为 N 的父亲（P）和 S 的右儿子的父亲。我们接着交换 N 的父亲 P（无论 P 是黑色还是红色都可以，所以图 8.1.4.2-6 中标注 R/B，即 red 或者 black 都可以）和 S 的颜色，并使 S 的右儿子为黑色。子树在它的根上的仍是同样的颜

色，所以性质 3 没有被违反。但是，N 现在增加了一个黑色祖先：要么 N 的父亲变成黑色，要么它是黑色而 S 被增加为一个黑色祖父。所以，通过 N 的路径都增加了一个黑色节点。

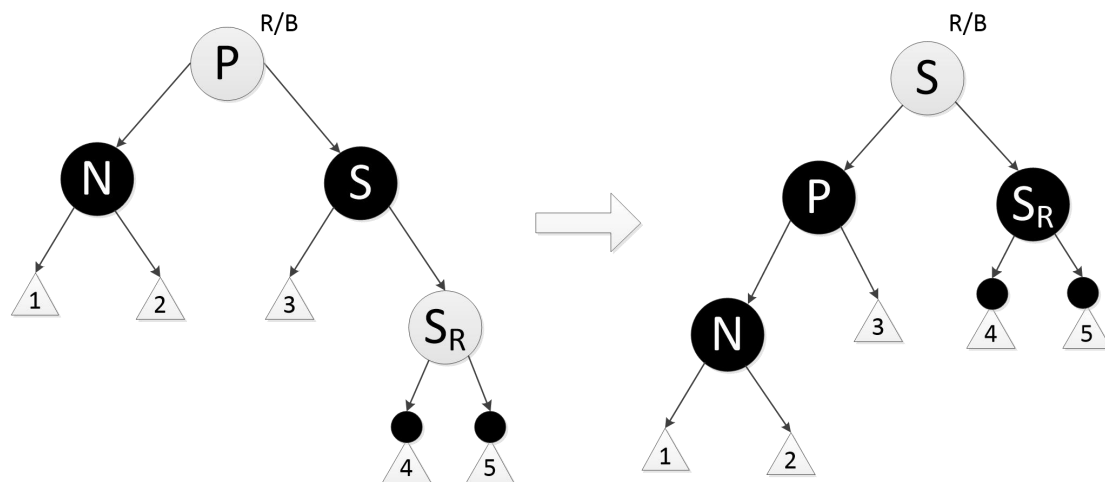


图 8.1.4.2-6 删除情形 6

此时，如果一个路径不通过 N，则有两种可能性：

- 它通过 N 的新兄弟。那么它以前和现在都必定通过 S 和 N 的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。
- 它通过 N 的新叔父，S 的右儿子。那么它以前通过 S、S 的父亲和 S 的右儿子，但是现在只通过 S，它被假定为它以前的父亲的颜色，和 S 的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质 4。在示意图 8.1.4.2-6 中的 R/B 节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。

详细代码如下：

```
void
delete_case6(struct node *n)
{
    struct node *s = sibling (n);

    s->color = n->parent->color;
    n->parent->color = BLACK;

    if(n == n->parent->left){
        s->right->color = BLACK;
        rotate_left(n->parent);
    } else {
        s->left->color = BLACK;
        rotate_right(n->parent);
    }
}
```

同样的，函数调用都使用了尾部递归，所以算法是原地算法。此外，在旋转之后不再做递归调用，所以进行了恒定数目（最多 3 次）的旋转。

通过上面六种情况的分析，由于情形 2-情形 6 相对复杂，我们进行了下面整体图 8.1.4.2-7 的分析（请看下面的整体图，图中颜色黑的代表黑色结点，偏浅的代表红色结点，方形节点代表红色或者黑色都可以），通过总结图我们可以发现规律，首先是情形二，N 的兄弟 S 如果为红色，对父亲 P 进行左旋，这时由于 N 与 S_L 两边的黑色节点数目不等，还需要再次进行情形 4, 5, 6 中的处理。为什么通过 N 和 S_L 的黑色节点数目不等呢，通过图中可以看出，S 是红色时，最初通过 N 和 S_L 的肯定是相等的，但是由于 N 是删除了原位置黑色节点，child 替代上来的，所以通过 N 的分支肯定比通过 S_L 分支的黑色节点数目少一个。这样情形 2，就转换为了情形 4, 5, 6 中的某一种。

情形 2 分析的是 S 是红色，那么如果 S 不是红色，便是黑色，S 是黑色时，就分为情形四，情形五，情形六 总计分别 3 种情况，分别是，S 的两个孩子都是黑色，S 的左孩子为红色（右孩子任意颜色或者没有），S 的右孩子为红色（左孩子任意颜色或者没有）。情形三和情形四差别不大，分别是父亲 P 是黑色或者红色。针对情形三，把 S 变为红色后，通过 N 和通过 S 的左右子树黑色节点数相等，但是如果 P 不是根节点，那么通过 P 这边子树的黑色节点会少 1，所以将 P 作为 N 重新再做调整即可。情形五的目的是变为情形六，情形六让 N 的分支多了一个黑色节点，但是的 S_R 分支黑色数目节点不变。情形五和情形六，节点画成透明的方形代表黑色或者红色都可以，情形六旋转后，保持 S 的颜色和原有 P 的颜色保持一致即可。

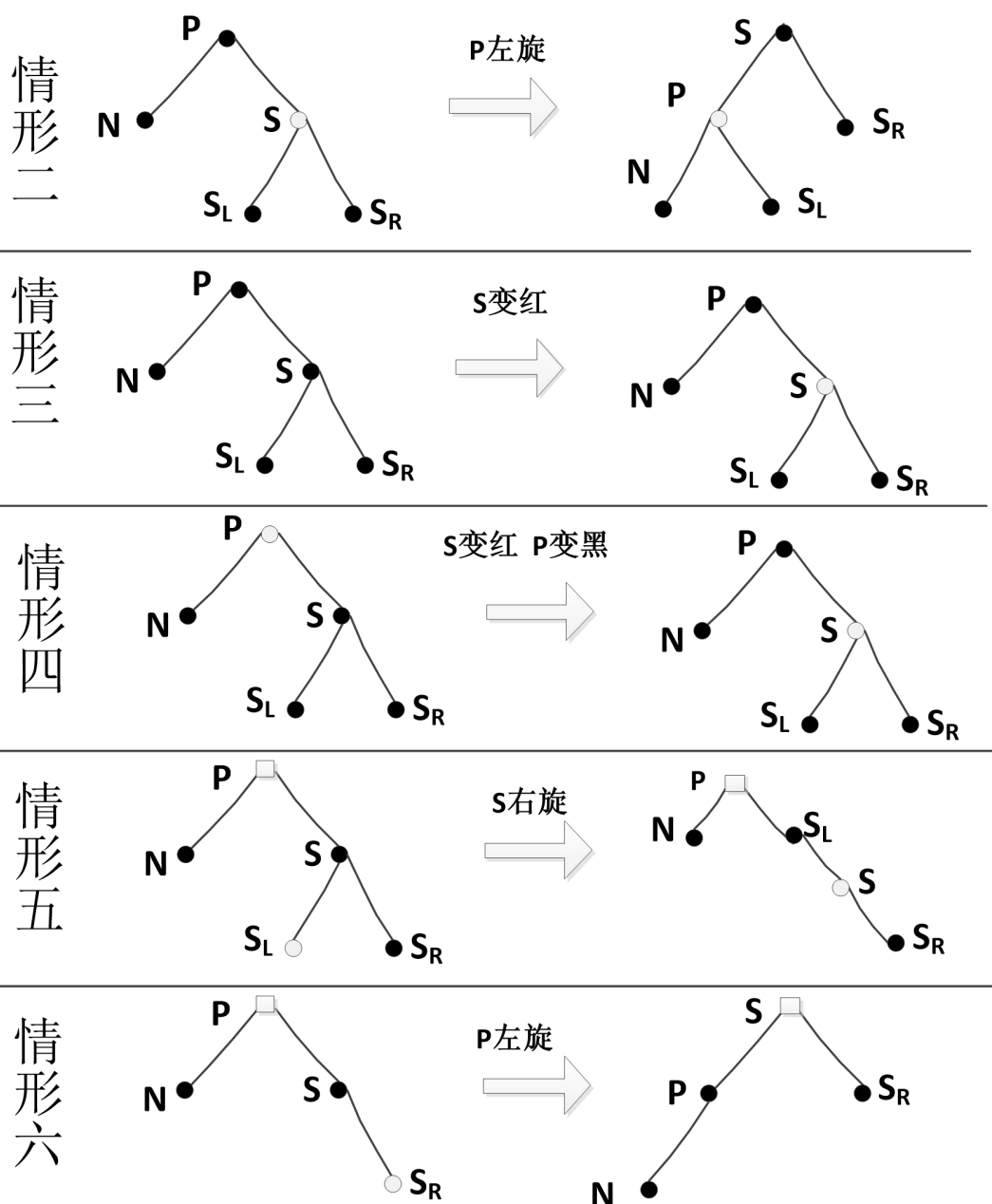


图 8.1.4.2-7 删除总结图

图 8.1.4.2-7 中浅色方框代表无论是黑色还是红色都可以，只要保证旋转前后对应位置颜色一致即可。（情形五，P 在 S 右旋后的颜色一致，情形六，S 结点的颜色要和 P 左旋之前 P 结点的颜色一致）

学习任何数据结构或者算法，图形理解原理后，阅读核心代码是必不可少的，下面【例 8.1.4.2-1】是红黑树删除的核心代码，面试时写出如下代码，面试官将会更加喜欢你，当然下面的代码有接近 100 行，需要在理解图 8.1.4.2-7 原理的基础上进行阅读源码，才能够快速理解，同时也能够在面试时顺利编写出来，面试依靠记忆代码是不靠谱的，一定要理解，凭自己的理解把代码写出来，比如面试时，先画一下 8.1.4.2-7 图，然后再根据图形来写代码，那么就会顺理成章，代码就会自然写出来。而且如果实际面试时笔试时间不够，就可以直接将对应图形画出，在进入当面交谈时，讲解给面试官听即可。

红黑树整体的代码较多，加上测试代码总计 700 行，全部附在书籍内不太合适，因此需要全部代码的同学可以加入前言的 QQ 群进行获取，同时在群内可以获取本书籍的视频讲

解。

【例 8.1.4.2-1】红黑树删除修正函数

```

/*
 * 红黑树删除修正函数，删除部分的核心代码
 *
 * 在从红黑树中删除插入节点之后(红黑树失去平衡)，再调用该函数；
 * 目的是将它重新塑造成一颗红黑树。
 *
 * 参数说明：
 *     root 红黑树的根
 *     node 待修正的节点
 */
static void rbtree_delete_fixup(RBRoot *root, Node *node, Node *parent)
{
    Node *other; // other代表图中的S
    // 下面的N说的是node
    while ((!node || rb_is_black(node)) && node != root->node)
    {
        if (parent->left == node)
        {
            // 删除
            other = parent->right;
            if (rb_is_red(other))
            {
                // Case 1: N的兄弟S是红色的，对应上面图中的情形2
                rb_set_black(other);
                rb_set_red(parent);
                rbtree_left_rotate(root, parent);
                other = parent->right; // 旋转后other节点变为图中的SL
            }
            if ((!other->left || rb_is_black(other->left)) &&
                (!other->right || rb_is_black(other->right)))
            {
                // Case 2: N的兄弟S是黑色，且S的两个孩子也都是黑色的
                // 对应上图中的 情形3和情形4，针对情形4的父亲变为黑色是在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的兄弟S是黑色的，并且S的右孩子为黑色。
        //对应上图中的 情形5
        rb_set_black(other->left);
        rb_set_red(other);
        rbtree_right_rotate(root, other);
        other = parent->right;
    }
    // Case 4: N的兄弟S是黑色的；并且S的右孩子是红色的，左孩子任意颜色。、
    //对应上图中的 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，下面的case1-case4是上面的case1-case4的对称场景
{//删除
    other = parent->left;
    if (rb_is_red(other))
    {
        // Case 1: N的兄弟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的兄弟S是黑色，且S的两个孩子也都是黑色的
        rb_set_red(other);
        node = parent;
        parent = rb_parent(node);
    }
    else
    {
        if (!other->left || rb_is_black(other->left))
        {
            //删除99时，应该是S的左孩子是黑色
            // Case 3: N的兄弟S是黑色的，并且S的左孩子是为黑色。

```

```

        rb_set_black(other->right);
        rb_set_red(other);
        rbtree_left_rotate(root, other);
        other = parent->left;
    }
    // Case 4: N的兄弟S是黑色的；并且S的右孩子是红色的，左孩子任意颜色。
    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);
}

```

8.1.5 数据结构学习技巧

最后龙哥再跟大家传授一下学习数据结构和算法的技巧，龙哥在学习红黑树时，不仅看上面的原理，自己总结画图总结其中的原理，同时会下载一份可运行的红黑树的源码，把源码运行起来，插入一个节点时，我会自己画图，看看最终的图会变成什么样子，然后运行代码看代码实际输出的效果，是否跟自己预期的图形效果一致，如果一致，就在单步调试，看看代码走了那一部分，这样就可以将对应部分的代码实现与原理讲解对应起来。如果不一致，通过实际的代码运行打印，看看对应原理的那一部分，这会就会分为两种情况，对应的上，还有就是对不上，对应的上，那么接着再看看代码的运行效果，如果对应不上，这时候就需要用到我们前面的调试方法，在调试窗口中，每次单步来看数据变化，如果是数组等大块连续内存，采用内存窗口进行查看，如果是链表，红黑树等通过链式实现的数据结构，那么适合通过监视逐级点开来看。

同时在阅读代码的过程中，针对每一块看懂的代码及时加入备注，这样以后再次使用，或者阅读，都可以迅速的捡起来，其实不仅学习数据结构，以后学习各种中间件的源码时，由于记忆力是有限的，因此在阅读代码时，及时加入备注，同时可以用画图软件，在阅读源码时，针对各个模块的关系，进行画图梳理，也可以用流程图针对某一块核心功能，进行画图，都是帮助自己学习中间件的好方法。也可以帮助自己在以后的面试过程中，非常有条理的回答面试官的问题。

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，时间复杂度可被称为是渐近的，亦即考察

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,它至多需要 $5n^3 + 3n$ 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，我们通常会估算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 $T(n)$ 。

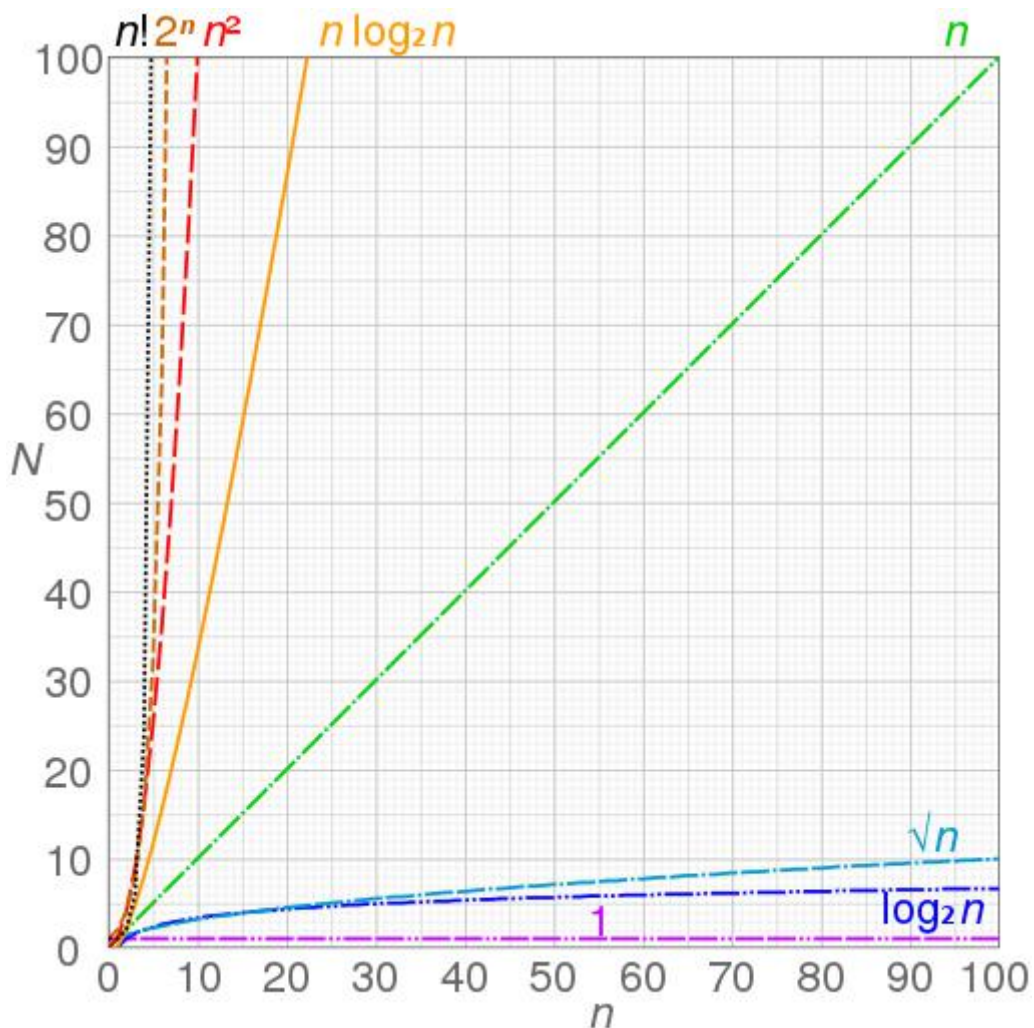


图 8.2.1-1

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，记作 $O(1)$ 时间

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的记数系统，对数常常以 2 为底（即 $\log_2 n$ ，有时写作 $\lg n$ ），常见的具有对数时间的算法有二叉树的相关操作和二分搜索。

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

若一个算法时间复杂度 $T(n) = O(n \log n)$ ，则称这个算法具有线性对数时间，例如快排，堆排。

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空间不计算），记做 $S(n)=O(f(n))$ ，例如插入排序的空间复杂度为

$O(1)$ 。

时间复杂度大家要能够快速计算得出，因为这是面试的高频问题，空间复杂度相对面的少一些。下面我们通过学习排序算法掌握时间复杂度及空间复杂度的计算。

8.2.2 排序算法

针对排序算法，掌握通常的 8 种排序算法即可，依次是 冒泡，选择，插入，希尔，快排，堆排，归并，计排。

对于排序算法分为以下 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)，通过用额外的空间来“分配”和“收集”来实现排序，它们的时间复杂度可达到线性阶： $O(n)$

实际面试面的最多的是快速排序，堆排序，还有计数排序，因为在下面的排序算法中，大家要对这三种排序算法非常熟练！

这里推荐一个动画演示排序效果的网站，在下面学习排序算法的过程中，可以结合动画 https://www.cs.usfca.edu/~galles/visualization/Algorithms.html?tdsourcetag=s_pcqq_aiomsg

下面表 8.2.1-1 是各种排序算法的时间复杂度，空间复杂度，稳定性

| 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 | $O(n\log_2 n)$ | $O(n^2)$ | $O(n\log_2 n)$ | $O(\log_2 n)$ | 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 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(1)$ | 不稳定 | 较复杂 |
| 归并排序 | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n\log_2 n)$ | $O(n)$ | 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 |

表 8.2.1-1

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

下面来看一下各种算法实现的代码实例：

首先我们在头文件中定义了

```
#define N 10
```

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

也就是数组的长度为 10，当然要增加长度，直接修改该值即可，同时 SWAP 交换我们采用的是宏替换，比函数的效率高一些，而且避免了重复代码。

8.2.2.1 冒泡排序

假如我们有 10 个无序的数，假如我们从小到大排序，冒泡法排序的思想是通过一轮比较，把最大的放在最后面，然后将剩余的 9 个元素再进行一趟比较，再把最大的放在最后面，这

样不断进行，最终实现数组有序。我们通过外层循环控制无序数的数目，内层控制比较，每次我们拿j和j+1进行比较，也就是相邻的两个元素进行比较，如果发现元素j大于元素j+1，那么就进行交换，下表8.2.2.1-1中列出了第一趟每次发生交换时，元素的变化情况（没有发生交换的情况，没有列出）。

| | |
|-------|---|
| 最初 | 3 87 2 93 78 56 61 38 12 40 |
| 第1次交换 | 3 2 87 93 78 56 61 38 12 40 |
| 第2次交换 | 3 2 87 78 93 56 61 38 12 40 |
| 第3次交换 | 3 2 87 78 56 93 61 38 12 40 |
| 第4次交换 | 3 2 87 78 56 61 93 38 12 40 |
| 第5次交换 | 3 2 87 78 56 61 38 93 12 40 |
| 第6次交换 | 3 2 87 78 56 61 38 12 93 40 |
| 第7次交换 | 3 2 87 78 56 61 38 12 40 93 |

表8.2.2.1-1

如何分析确定内层的判定边界呢，是j<i还是j<=i呢？首先i是N-1，也就是i最大为9，那么因为内层是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呢，当i等于1时，对于内层是j<1，然后刚好使得a[0]和a[1]发生了比较，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。

时间复杂度如何计算呢，就是程序实际的运行次数，我们可以看到内层是j<i，外层i的值是从N-1一直到1，所以程序总计的运行次数是1+2+3+... (N-1)，就是从1一直加到N-1，因此等差数列求和，得到的是N(N-1)/2，总计运行这么多次，忽略低阶项和高阶项的首项系数，因为时间复杂度为O(n²)。因为没有使用额外的空间（所谓的额外空间必须跟输入元素个数N相关），所以空间复杂为O(1)。

```
void arr_bubble(int a[])
{
    int i, j;
    for(i=N-1; i>0; i--) //外层是无序数的数目
    {
        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：如果我们要从大到小排序，代码中哪个地方修改一下就可以实现呢？

8.2.2.2 选择排序

上面的冒泡排序在内层比较时，我们发现较大的数就交换，会造成交换的频次过高，如果我们通过一个变量记录时刻记录最大值，一轮比较下来，将最大值再和最后的元素交换，

这样一轮比较就只需要交换一次。选择排序正是使用这种思想对冒泡进行了改进，下面的代码依然是实现从小到大排序，外层循环依然记录的是无序数的情况，但是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，最终数组有序。

首先我们假定第零个元素是最小的，所以把下标0赋值给min_pos，内层比较时，从1号一直比较到9号元素，发现谁更小，就把它的下标给min_pos，一轮比较结束后，就将min_pos对应位置的元素与元素i发生交换，如下表8.2.2.2-1所示，第一轮我们确认2为最小的，将2与数组开头元素3发生交换。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发生交换。持续进行，最终使数组有序。

经验验证在10个数时，选择排序效率最佳，但是往往工作时排序的数的数目都较大，只有10个数我们也不在意使用哪种排序算法，因为时间很短。

| | |
|----------|-----------------------------|
| 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 3 93 78 56 61 38 12 40 |
| 第2轮比较后标记 | 2 87 3 93 78 56 61 38 12 40 |
| 第2次交换 | 2 3 87 93 78 56 61 38 12 40 |
| 第3轮比较后标记 | 2 3 87 93 78 56 61 38 12 40 |
| 第3次交换 | 2 3 12 93 78 56 61 38 87 40 |
| 省略 | |

表8.2.2.2-1

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目是一样的，就是从1加到N-1，总计运行了 $N(N-1)/2$ ，针对循环内语句的数目我们是忽略的，因为我们在计算时间复杂度时，主要考虑与N有关的循环，而循环内交换的多，有5条语句，最终得到的无法是 $5n^2$ ，循环内交换的少，有两条语句，得到的是 $2n^2$ ，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 $O(n^2)$ 。因为没有使用额外的空间（所谓的额外空间必须跟输入元素个数N相关），所以空间复杂为 $O(1)$ 。

```
void arr_select(int *a)
{
    int i, j, min_pos;
    for(i=0; i<N-1; i++)//控制无序数的数目
    {
        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：如果我们要从大到小排序，代码中哪个地方修改一下就可以实现呢？

8.2.2.3 插入排序

假如只有一个数，那自然有序，插入排序首先就是将第一个数看成有序的序列，然后把后面9个数看成要依次插入的序列，首先我们通过外层循环控制要插入的数，首先用 insertVal 把要插入的值87保存，这时j的值为0，我们比较arr[0]是否大于arr[1]，也就是3是否大于87，没有大于，因此不发生移动；这时有序序列是3 87，接下来我们要将数值2插入到有序序列中，首先我们将2赋值给insertVal，这时判断87是否大于2，因为87大于2，所以我们将87向后移动，把2覆盖掉，接着判断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，然后再次对j减一后，j为-1，内层循环结束，这时将2赋值给 arr[j+1]，也就是arr[0]的位置，从得到下表8. 2. 2. 3-1中第二次插入后的效果。

循环进行，将数依次插入有序序列，最终整个数组有序，插入排序主要用在部分数有序的场景，比如你的手机通讯录，时刻是有序的，当新增一个电话号码后，我们以插入排序的方法将其插入原有有序序列，这样复杂度较低。

| | 有序序列 | 要插入的数的序列 |
|-------|--------|---------------------------|
| 最初 | 3 | 87 2 93 78 56 61 38 12 40 |
| 第一次插入 | 3 87 | 2 93 78 56 61 38 12 40 |
| 第二次插入 | 2 3 87 | 93 78 56 61 38 12 40 |
| 省略 | | |

表8. 2. 2. 3-1

随着有序序列的不断增加，插入排序比较的次数依次增加，因此插入排序的执行次数，也是从1一直加到N-1, 总计运行次数为 $N(N-1)/2$ ，时间复杂度依然为 $O(n^2)$ 。因为没有使用额外的空间（所谓的额外空间必须跟输入元素个数N相关），所以空间复杂为 $O(1)$ 。

```
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：如果我们要从大到小排序，代码中哪个地方修改一下就可以实现呢？

8.2.2.4 希尔排序

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版本。希尔排序是非稳定排序算法。希尔排序按其设计者希尔（Donald Shell）的名字命名，该算法由1959年公布。

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，然后内层的循环中，j原有跟1有关的都改为gap，同时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2，也就是5，后面每次以除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。

下表8.2.2.4-1中首先gap为5时的比较，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覆盖到61的位置，最内层循环结束，61放到87的位置，如下表8.2.2.4-1所示，接着比较的是2与38，由于2小于38，因此不发生交换，接着是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，如下表8.2.2.4-1第4次比较所示，接着是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，如下表8.2.2.4-1第5次比较所示。这时gap为5的步长进行完毕，接下来是gap为2，再次进行，因为与gap为5原理相同，大家自行推算一下。

希尔排序（经验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，因为希尔排序编写较复杂，且只有在数目为6千多以内时，比快排和堆排快，因此优势不大，所以面试面的较少。因为没有使用额外的空间（所谓的额外空间必须跟输入元素个数N相关），所以空间复杂度为 $O(1)$ 。

| | |
|-------------------|-----------------------------|
| gap为5时的第1次比较 | 3 87 2 93 78 56 61 38 12 40 |
| gap为5时的第2次比较—发生交换 | 3 61 2 93 78 56 87 38 12 40 |
| gap为5时的第3次比较 | 3 61 2 93 78 56 87 38 12 40 |
| gap为5时的第4次比较—发生交换 | 3 61 2 12 78 56 87 38 93 40 |
| gap为5时的第5次比较—发生交换 | 3 61 2 12 40 56 87 38 93 78 |
| 省略 | |

表8.2.2.4-1

```
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;
        }
    }
}
```

8.2.2.5 快速排序

快排的核心思想是分治，首先我们的目标依然是从小到大排序，我们找到数组中一个分割值，把比分割值小的数都放在数组的左边，把分割值大的数放在数组的右边，这样分割值的位置就确定下来了，数组一分为二，我们只需要排前一半数组，后一半数组，复杂度直接

减半，通过这种思想，不断的进行递归，最终分割的只剩余一个元素，就自然有序。

arr_quick是我们的快排的递归函数，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，这样我们通过再次递归，arr_quick数组的前半部分，然后arr_quick数组的后半部分，递归编写时注意到对返回的分割点的下标减一和加一，递归的结束条件，当left等于right时，说明被分割后的数组只剩余一个元素，这时递归结束。

接下来我们来看一下partition函数，首先我们拿数组的最右边的值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就加1，首先我们拿3和40比较，发现小于40，这是交换a[i]和a[k],由于指向同一个位置，就是自己跟自己交换，这时k加1，i加1（如下表8.2.2.5-1第一行所示），然后比较87与40的大小，87没有小于40，这是只是i加1（如下表8.2.2.5-1第3行所示），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进行加1（如下表8.2.2.5-1第4行所示）；这时a[i]不断的与40比较，一直到达38时，发现38小于40，这时a[k]与a[i]发生交换，这时k加1（如下表8.2.2.5-1第5,6行所示）；这时12小于40，因此a[k]与a[i]发生交换（如下表8.2.2.5-1第7行所示），这时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（如下表8.2.2.5-1第8行所示），这时可以发现元素40的位置已经确定，比40小的元素被放在了左边，比40大的元素被放在了40的右边，我们return k，也就是分割点的下标值，接下来再分别对左右两边的两个数组继续进行快排即可。

| | | |
|---|--------------|---------------------------------------|
| 1 | 最初 | k i 3 87 2 93 78 56 61 38 12 40 |
| 2 | 第一次比较后 | k i 3 87 2 93 78 56 61 38 12 40 |
| 3 | 第二次比较后 | k i 3 87 2 93 78 56 61 38 12 40 |
| 4 | 第二次比较发生交换后效果 | k i 3 2 87 93 78 56 61 38 12 40 |
| 5 | 第三次将要发生交换 | k i 3 2 87 93 78 56 61 38 12 40 |
| 6 | 第三次发生交换后 | k i 3 2 38 93 78 56 61 87 12 40 |
| 7 | 第四次发生交换后 | k i 3 2 38 12 78 56 61 87 93 40 |
| 8 | 最后一次交换 | k i 3 2 38 12 40 56 61 87 93 78 |

表8.2.2.5-1

假如每次快排数组都被平均的一分为二，我们可以得出arr_quick递归的次数是log₂n，第一次partition遍历数目为N，分成两个数组后，每个遍历n/2,但加起来还是n，因此时间

复杂度计算得出是 $O(n\log_2 n)$ ，因为计算机是二进制的，因此我们面试时回答复杂度，或者与人交流说复杂度，直接讲 $O(n\log n)$ ，不会带2字。快排最差的时间复杂度为什么是 n^2 呢？因为如果数组本身从小到大有序，如果每次我们仍然拿最右边的作为分割值，这时每次数组不会二分，就会造成递归 n 次，所以快排最坏时间复杂度为 n 的平方，当然为了避免这种情况，一些写快排的，首先随机一个下标，先将对应下标的值与最右边的元素交换后，再进行partition操作，这样可以极大降低出现最坏时间复杂度的概率，但是仍然不能避免。

快排由于我们不断的递归，每次使用的形参left, right所占有的空间是跟递归次数相关的，所以快排的空间复杂度是 $O(\log_2 n)$ ，我们的快排是递归实现的（见【例8.2.2.5-1】），递归实现的快排比较简单，面试时大家写递归的快排即可，非递归的快排编写难度大，但好处是执行速度比非递归的快排要快（这个我们前面讲递归讲过，非递归的执行效率更优），同时递归的快排在排序的数目特别大时，会出现Stack Overflow报错，因为函数递归调用是有上线的，这时有同学可能就担心排序比较大的数怎么办呢，我们可以使用qsort，也就是标准库给我们提供的快排接口，其内部是使用非递归的快排算法。当然如果排序的数目上亿，最好还是使用我们下面的堆排序，因为针对上亿的数，一旦快排出现最差，或者接近最坏的时间复杂度，排序的时间就会特别久。

【例8.2.2.5-1】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);
    }
}
```

首先我们来看一下 qsort 的函数形式

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



```
(*compare)(const void *, const void *) );
```

功能： 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, 返回负值; 如果等于返回零值; 如果大于返回正值。函数对 *buf* 指向的数据按升序排序。**qsort 的本质是其内部实现了快排, 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 关键是编写 compare 函数, qsort 的第四个参数是一个函数指针, 前面我们学习指针章节知道, 入参如果是一个函数指针, 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, 当我们排整型数组和排结构体数组, 排序的规则自然是不一样的。

首先我们先来看 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);` 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, 接下来我们来解析 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, 所以 left 和 right 接收的地址值是整型类型, 因此我们将 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, 返回负值; 如果等于返回零值; 如果大于返回正值, 这样默认是升序, 如果我们把 `return *p1-*p2` 改为 `return *p2-*p1`, 那么就是降序排列。下面【例 8.2.2.5-2】代码同时演示了如何实现随机数生成, 以及排序的数目很多时, 进行排序时间统计的方法, 各位同学可以把 N 改大, 比如千万, 或者上亿, 来看一下实际排序所耗费的时间, 注意在统计实际排序时间时, 注释掉打印函数, 否则屏幕将会不停的打印。

【例 8.2.2.5-2】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)); //这里申请堆空间目的是当排序超过百万
    时, 不会发生Stack Overflow
    time_t start,end;
    srand(time(NULL)); //通过随机数, 生成10个随机的数
    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); //打印排序使用时间
    system("pause");
}

```

下面我们来看通过 qsort 排序结构体数组, 接下来我们通过 qsort 来完成上一个章节的课后作业的第一题, 题目如下:

1. 有一个学生结构体, 其数据成员有: 学号, 姓名, 3 门课程。从键盘上输入5 个学生的信息。要求输出:

(1) 按照学号递增输出全部学生信息, 每个学生的信息一行。(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

接下来我们通过 qsort 来完成第 1 小问, 还有第 4 小问, 下面【例 8.2.2.5-3】代码实现了按学号递增排序和按总分从低到高排序, 当要按学号递增排序时, 将 qsort 的第四个参数, 改为 **compareNum**, 可以看到 **compareNum** 内我们把指针转换为结构体指针, 因为我们排序的每一个元素是结构体, 所以将 **void*** 类型的指针转为结构体指针, 但是实际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, 负值或者零; 当我们要对总分进行排序时, 使用 **compareScore** 函数作为 qsort 的第四个参数, 我们依然是将指针转换为结构体指针, 但是这时我们比较的是三门课加起来的总分, 当是浮点数时, 我们比较大小不是像前面整型数一样直接一个 **return** 搞定, 必须通过大于, 小于来进行。相信通过 **compareNum** 和 **compareScore** 函数, 大家对于 qsort 第四个参数如何使用, 有了清晰的理解。为了方便大家输入测试数据, 下面我列了 5 组学生信息, 编写代码运行的小伙伴可以直接输入下面 5 个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 guangtouqiang 91.5 98.3 96.1
1002 xionger 88.5 92.3 91.1

```

【例8.2.2.5-3】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;
    //输入五名学生的学号，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].c
hinese, sArr[i].math, sArr[i].english, sArr[i].chinese+sArr[i].math+sArr[i].englis
h);
    }
    system("pause");
}

```

在研究生复试机试，或者校招 OnlineJudge 中（如浙大 PAT），如果遇到的题目中需要使用到排序，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，也不容易出错，当然如果题目特别要求手写某种排序算法，那么就不可以使用 qsort 等排序 API。

我们在淘宝购物时，在搜索框中输入“鞋子”，可以看到如图 8.2.2.5-1 的效果，每一款鞋子的信息都是图片，价格，购买人数，名字，店铺，店铺地址等，在服务器的内存中就像一个一个的结构体信息，当不同的用户需要时，我们就把这些结构体信息发给用户，但存在一个问题，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，如果每个用户需要一种新的排序时，我们都交换内存里的结构体，交换成本将会很高，而且这些商品往往是以链表形式存在内存中的，交换时需要的代码逻辑多于数组，那有没有好的解决办法呢，当然有，那就是使用索引的原理（不理解索引是什么没关系）。

在实际链表的使用中，链表中有多个元素，已经统计好，假设链表中有 5 个节点，那么我们通过 calloc 申请 5 个指针大小的空间，相当于使用动态的指针数组（这个在指针章节讲过），然后遍历链表，把链表中每一个节点的地址存储在指针数组中，qsort 不仅可以排序整型数组，浮点型数组，结构体数组，也可以排序指针数组，**根据用户的排序需求，无论是价格，销量，还是信用，我们交换对应的指针，这样交换的成本将极大的降低，客户需要排序的结果时，我们通过访问有序的指针数组，把数据从服务器传输给他即可。**

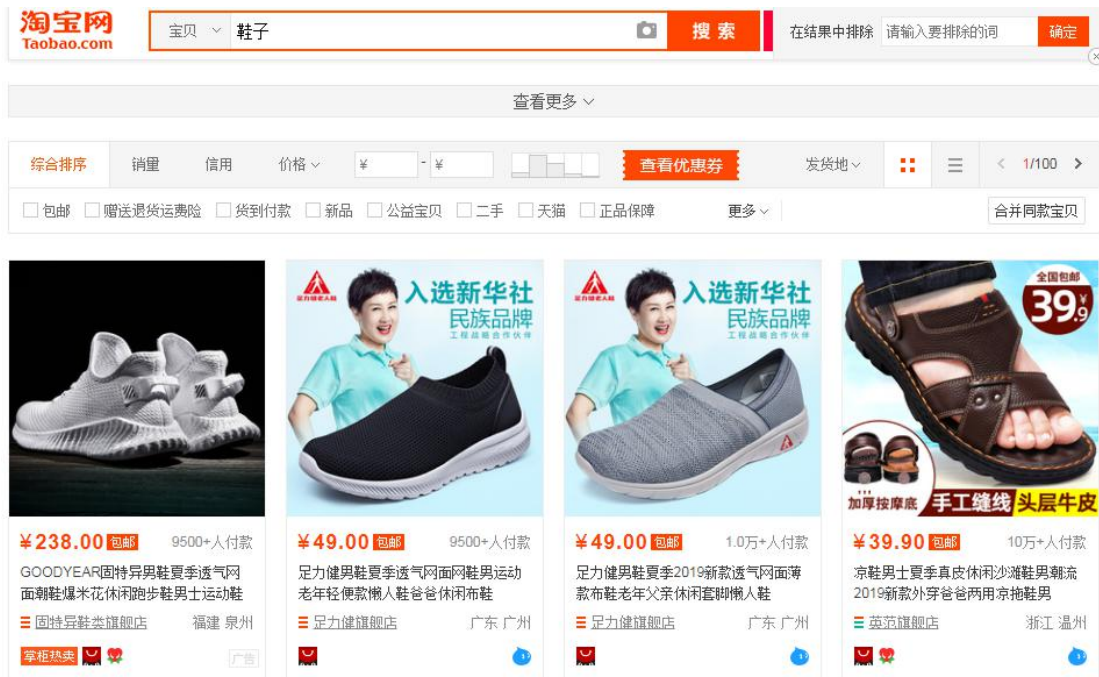


图 8.2.2.5-1

下面我们来看一下 qsort 排序指针数组的代码，见【例 8.2.2.5-4】代码，首先我们通过头插法新建一条链表，我们通过输入 3 6 9 2 7 创建链表，链表里存储 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存储空间内存储结构体指针，因此我们需要将 calloc 的返回值强转为二级指针然后赋值给 pArr（在指针章节的二级指针部分已经讲过），然后遍历链表，将链表中每一个节点的地址依次赋值给 pArr[0] 到 pArr[4]，然后开始通过 qsort 排序指针数组 pArr，元素个数是 listLen，每一个元素的大小是 `sizeof(pStudent_t)`，部分同学容易写错为 `sizeof(Student_t)`，然后我们来看一下最难的 `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，每一个元素本身是一级结构体指针，如果再对元素取地址，得到就是二级指针，因此我们需要将 void* 类型 pleft 和 pright 强转为二级结构体指针，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) 变为一二级结构体指针，然后再通过成员访问拿到对应的学号，最终依据 qsort 要求的正值，负值，零三种情况的规则，`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没有发生改变，具体执行结果见图 8.2.2.5-2。

【例 8.2.2.5-4】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，我们仅为学生添加了学号信息
    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，将链表的每个节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("链表本身节点值:\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");
}

```

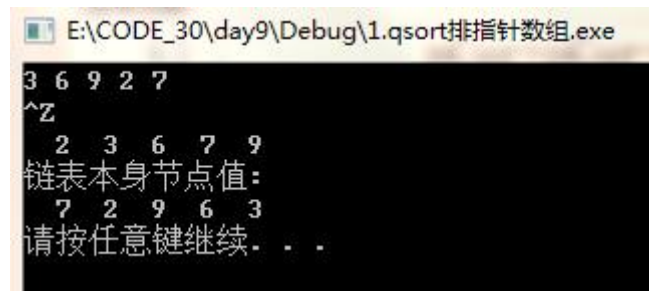


图 8.2.2.5-2

上面的代码中，针对链表进行了销毁，指针数组申请的空间进行了释放，工作时针对高频查询的数据会常驻内存，或者会将排序后的结果存入磁盘，从而降低下次用户需要我们对数据进行提取的时间（下一章节讲解如何将数据存入磁盘）。

8.2.2.6 堆排

如果排序的数据较多，而且又要求空间复杂度为 $O(1)$ ，那么我们会使用堆排，因为相对于快排，堆排的时间复杂度是最坏和平均都是 $O(n\log n)$ 。

堆（英语：Heap）是计算机科学中的一种特别的树状数据结构。若是满足以下特性，即可称为堆：“给定堆中任意节点 P 和 C，若 P 是 C 的母节点，那么 P 的值会小于等于（或大于等于）C 的值”。若母节点的值恒小于等于子节点的值，此堆称为**最小堆**（min heap）；反之，若母节点的值恒大于等于子节点的值，此堆称为**最大堆**（max heap）。在堆中最顶端的一个节点，称作根节点（root node），根节点本身没有母节点（parent node）。平时在工作中，我们会把最小堆，叫**小根堆**，或者**小顶堆**，把最大堆称为**大根堆**，或者**大顶堆**，

堆始于 J. W. J. Williams 在 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。

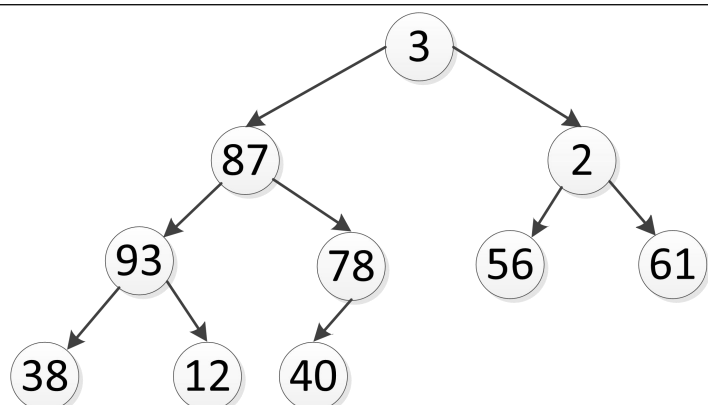
假如我们有以下 10 个元素，3 87 2 93 78 56 61 38 12 40，我们将这 10 个元素建成一颗完

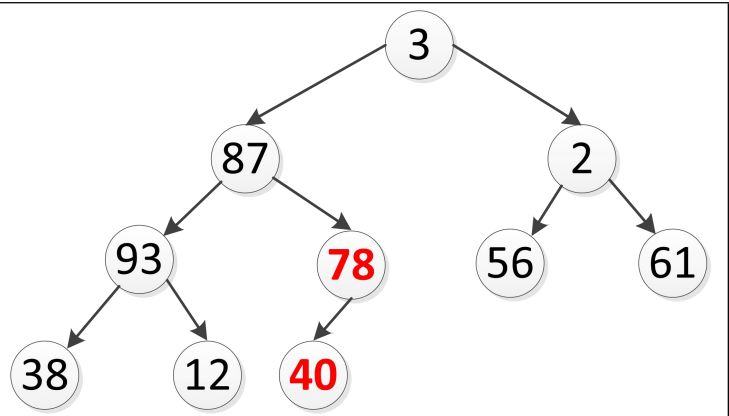
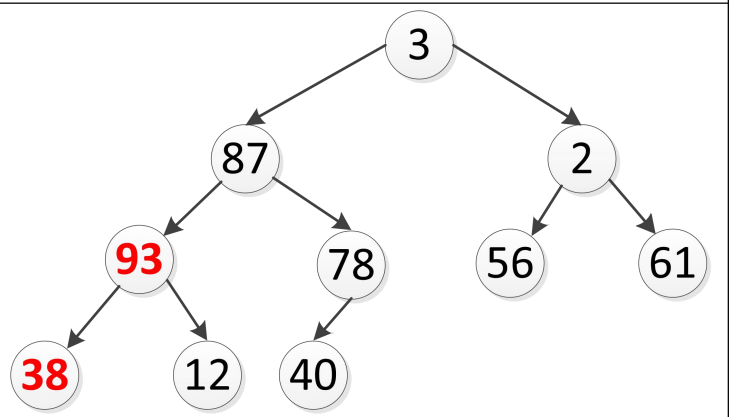
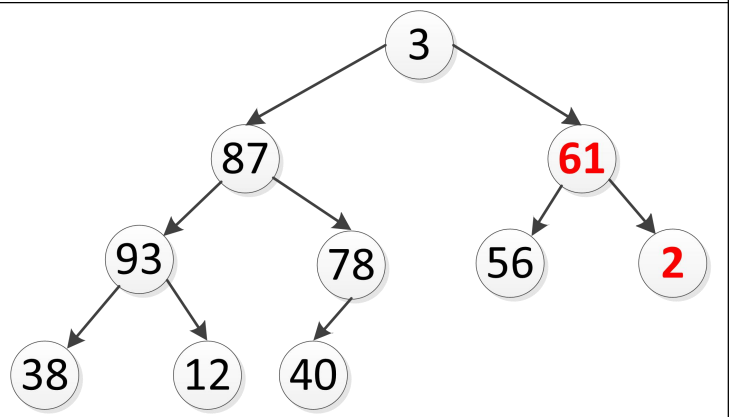
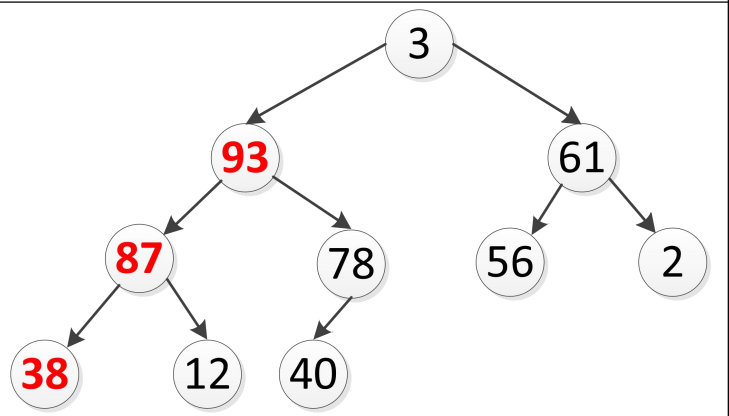
全二叉树，因为我们采用层次建树，虽然只有一个数组存储元素，但是我们能将二叉树中的任意一个位置的元素，对应到数组下标上，我们把二叉树每个元素对应到数组下标这种数据结构称之为 堆，也可以说是心里有棵树，比如最后一个父亲元素的下标是 $N/2-1$ ，也就是 $a[4]$ ，对应的值为78，为什么下标一定是 $N/2-1$ ，这是层次建立一颗完全二叉树的特性，可以记住，假如父亲节点的下标是 dad ，那么父亲节点对应的左孩子的下标值是 $2*dad+1$ ，这个特性是因为树每多一层，元素个数就会翻倍多1，这个很容易观察出来，接着我们开始依次将每一颗子树都调整为父节点最大，最终将整棵树变为一个大根堆。

下表8.2.2.6-1所示，第2列，我们`adjustMaxHeap`函数对 $arr[4]$ 与 $arr[9]$ ，也就是78和40进行调整，因为78大于40，所以无需交换；我们通过一个循环，对 i 进行减1，是 $arr[3]$ ，得到前面一颗子树的父亲节点，也就是93，如第3列所示，首先38大于12，所以我们拿38与93进行比较，93大于38，无需调整；接着如第4列所示， $arr[2]$ 与 $arr[6]$ 比较大小，因为2小于61，所以交换 $arr[2]$ 与 $arr[6]$ ，交换效果第4列所示；接着 $arr[1]$ 与 $arr[3]$ ，也就是87与93比较，87小于93，所以发生交换，但是这时87，也就是 $arr[3]$ ，本身是一个父节点，有可能我们的交换导致87, 38, 12这颗子树不是最大堆的特性，因此`adjustMaxHeap`函数中，我们使用了一个循环`while`，当发生交换时，我们让孩子重新作为父亲，再次进行一轮调整，当然前提是调整的节点的确是一个父亲，当然这时因为87大于38，所以无需调整，`while`循环结束；

接着我们看第6列，这时 $arr[0]$ 和 $arr[1]$ 进行比较，3小于93，发生交换，发生交换后， $arr[1]$ 重新作为父节点，与孩子 $arr[3]$ 进行比较，3小于87，因此 $arr[1]$ 和 $arr[3]$ 发送交换，这时我们将 $arr[3]$ 再次作为父亲，与自己的孩子 $arr[7]$ 进行比较，3与38比较，小于38，因此发送交换，最终得到效果如第6列的图形所示，到此，我们将数组调整为大根堆，任何一颗子树的父亲节点都是大于孩子的，从而可以确定，根部节点为最大。

这时我们将顶部元素与数组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，因为这时候只有数组第一元素不满足大根堆的特性，所以只需要调用`adjustMaxHeap`函数将顶部元素调整即可，调整完毕后，再次将顶部元素和数组倒数第二个元素交换，这样循环往复，最终使数组有序。

| | | |
|---|------------|--|
| 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 |  |
| 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 |
| | 数组元素对应的二叉树，arr[2]与arr[6]进行比较，发生交换 |  |
| 5 | 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]重新作为父节点，与孩子比较 |  |
| 6 | 数组元素情况 | 93 87 61 38 78 56 2 3 12 40 |

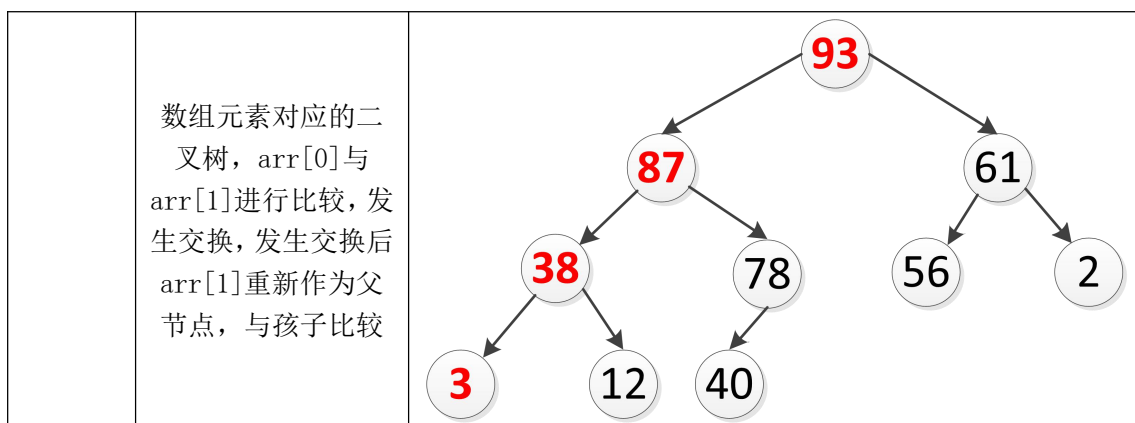


表8.2.2.6-1

【例8.2.2.6-1】堆排序

```

void adjustMaxHeap(int *arr, int adjustPos, int arrLen)
{
    int dad=adjustPos;//adjustPos是要调整的节点位置
    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]) //比较左孩子和右孩子，如果右孩子大于左孩子，将son加1，从而下一步拿右孩子与父亲比较
        {
            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，然后调整的堆元素个数减一
```

```
    SWAP(arr[0], arr[i-1]); //交换顶部元素与堆中最后一个元素，已经在尾部排好的不算堆中元素
```

```
    }
}
```

思考题：如果要从大到小排序，代码如何修改？改好代码的同学可以把代码发到QQ群和大家分享

如果让你将我们上面的堆排，封装成像qsort一样的接口，如何进行呢？通过【例8.2.2.6-2】的代码我们来学习一下如何实现我们自己的heapSort接口，实现自己的heapSort可以让我们更加深刻理解qsort，同时也会进一步熟练使用函数指针，要熟练C/C++语言，函数指针是必须掌握的。heapSort的编写与我们上面的堆排结构一致，关键是heapMax，我们给heapMax传入的参数是要调整的父亲节点的地址值，同时要让heapMax能够计算出父亲节点对应的儿子节点的地址值，在比较父亲节点和儿子节点时，我们要使用compare函数进行比较，同时在交换父亲节点和孩子节点时，通过内存交换的方式进行，因为节点的类型可能是任何类型，所以swap交换时，我们需要传递两个节点的起始地址的同时，还需要把每个节点所占的空间大小传递进去。

【例8.2.2.6-2】堆排的封装

```
#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，节点的编号，单个元素大小
        heapMax(arr, arr, i, 0, size, compare);
        swap((char *)arr, (char *)arr + (i - 1) * size, size);
    }
}
//dadStartAddr是要调整的父节点的起始地址，len是数组长度
//code_Num是父节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;
    }
}
}

```

//因为我们不知道每个元素的大小，因此交换时只能每个字节的移动，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：如果要想让heapSort实现从大到小的排序，如何修改cmp函数？

8.2.2.7 归并排序

归并排序的时间复杂度是 $O(n\log n)$ ，但是空间复杂度是 $O(n)$ ，因此正常单机排序时，优先使用堆排，所以归并面试的比较少，但是一旦多机排序，就会用到归并排序的思想，下面我们来学习一下归并排序。

归并排序下面的代码【例8.2.2.7-1】是采用递归方式实现的，非递归的归并排序难度较高，面试时只需完成下面的递归的二路归并即可，首先最小下标和最大下标值相加除以2，从而得到中间值下标mid，这时通过递归MergeSort对low到mid实现排序，然后再用MergeSort排序从mid+1到high即可，当数组的前半部分和后半部分都排好后，我们再用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，因此在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，就先放入数组A中，并对其游标加1，并对数组A的计数游标k，每轮循环进行加1，当i到达mid，或者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中，通过下面代码可以看出**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，假如low和high相邻，这时相当于合并两个元素，因此low等于high就无需再归并，因为一个元素自然有序，所以递归结束条件是low<high。

当一台电脑的内存不足以放置所有的数据时，如果这时不采用外部排序手法，那么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需以归并排序的思想进行合并。

【例8.2.2.7-1】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++];
}
//归并排序不限制是二路归并（也可以叫两两归并），还是多个归并，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目的归并，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");//这里为了打印在同一排，所以在最后打印换行
}
// 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");
}

```

8.2.2.8 计数排序

上面的快排堆排在排序1亿个数时，需要的时间在38秒，当然这是我的电脑的测试结果，你的电脑配置更好，自然会比我的更快一些，但是30多秒在工作中很多时候是我们无法接受的，有没有更好的办法呢？答案是有的，计数排序就是用空间换时间的方式来提高排序效率，如果我们用计数排序排我们随机的0-99的数，总计1亿个，排序的时间不到1秒钟，为什么会

这么快，因为计数排序的复杂度是 $O(n+m)$ ， m 就是下面代码中的大 M ，数的数值范围，因为数的变化返回是0-99，所以计数排序的执行次数是1亿+1百的数量级，但是堆排是 $O(n\log n)$ ，得到的是27亿次，实际应用中，商品的价格，销量，人的年龄，身高，都是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。

下面的代码【例8.2.2.8-1】，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1亿个数中，0出现多少次，1出现多少次，一直到99出现多少次，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，比如0出现了122次，那么把数组arr的从下标0到下标121全部填写为0，然后1出现了308次，那么从下标122到下标429，全部填写为1，这样不断进行，最终使arr数组有序。

一开始编写计数排序，要先测试编写的对不对时，可以把N改为10，将注释掉的arrPrint函数打开，看下排序后的结果是不是正确的，如果多次执行排序结果正确，说明写的计数排序是没有问题的，这时就可以测试排序1亿个数。

计数排序的空间复杂度是 $O(M)$ ，假如是整型数，变化范围是-21亿多到正的21亿多，这时就不适合用计数排序，因为时间复杂度是 $O(n+m)$ ，相当于1亿+42亿，大于我们上面计算的堆排的27亿多，同时我们需要额外16G的空间（4G*4个字节），来存储count，这显然是不合理的。但是如果面试过程中遇到去重的问题，我们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42亿的整型数int的变化范围，用位图处理所需要的空间是512M即可，因为是（4G/8，每个字节是8位）。

【例8.2.2.8-1】计数排序实现

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

#define N 100000000
#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，将数组中每个元素出现的次数进行统计
    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中即可
    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，还有桶排序，有与基数排序是按照数据的每一位进行比较，编写相对复杂一些，因此面试较少，假如数的范围较大，比如是 0 到 1 千万，桶排序的思想是划分若干个区间，比如划分 10 个区间，这时遍历一边数组，将 0 到 100 万的放入第一个桶，100 万至 200 万的放入第二个桶，依次类推，最后 900 万到 1 千万，放到最后一个桶，这样对每个桶分别进行排序，然后再合并起来即可。

8.2.2.9 面试题训练

通过上面的学习，我们已经掌握了八大排序算法，当然对于重要的**快排**，**堆排**，大家一定要能够自己非常熟练的编写出来，下面我们来看一道微信的排序的面试题：

1、N 个无序数的数组，让找出第 K 大的数？

如果这时你回答对数组排序，那么这里面试就挂了，之前做过找第一大，和第二大的数，只需遍历一次数组即可，这里也是这样的，首先我们对前 K 个数，建立小根堆，为什么要小根堆，因为对于 K 个元素，小根堆的根部就是第 K 大的元素，这时我们拿 K+1 个元素，依次跟堆顶进行比较，如果大于堆顶，就移除堆顶，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，这样的复杂度只有 $O(N\log K)$ ，比排序复杂度降低了很多。针对这个问题还有其他的解法吗？大家可以把自己的思考发到群里交流

2、接着面试官又问，如果内存无法存储 K 个元素怎么办？

这时面试官考的是一个逆向思维，首先一定要做内存能存下 N-K 元素的假设，因为面试官并没有讲不行，假如内存可以存下 N-K+1 个元素，我们首先建立 N-K+1 元素的大根堆，这时依次遍历数组，如果发现某个元素小于堆顶，这时移除堆顶，将新元素放入，重新调整为大根堆，这时遍历到数组末尾，堆顶元素即为第 N-K+1 小的，所以也就是第 K 大的。

3、接着面试官又问，刚才找的第 K 大的元素是没有考虑重复元素的问题，现在要找的第 K 大是不含有重复元素的第 K 大？

虽然我们可以讲建立小根堆时，重复的元素不放入，这是一种解决方法，但是面试官一旦问的是去重问题，我们首先要回答的是位图，因为位图是最快的去重方法（采用空间换时间），假如是 64 位的程序，我们定义一个长整型指针，并为其申请 512M 的空间，

`long *arr=(long*)malloc(512*1024*1024)`，写成 `malloc(1<<29)` 也可以，然后通过 `memset(arr,0,1<<29)`，把申请的空间全部置位 0，这时遍历一边数组，得到一个数后，将对应的位设置为 1 即可，比如来的数是 129，那么 $129/64$ ，得到的值为 2，那么我们需要对 `arr[2]` 进行操作，然后 $129\%64$ 的余数为 1，那么我们拿 `arr[2]` 按位或上 $1<<1$ 即可，总公式即为 `arr[129/64]=arr[129/64]|1<<129%64` 即可，也就是如果输入的数为 m ，公式即为 `arr[m/64]=arr[m/64]|1<<m%64`，位图建立完毕后，依次遍历位图，找到第 K 个位 1 的位置，然后逆向算出对应的值即可。如果程序是 32 位的，把公式中的 64 改为 32 即可。

4、如果单台机器的内存即不能存储 K 个元素，也无法存储 $N-K$ 个元素，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，这时只能使用外部排序，外部排序，首先是将数据切分为 k 块，每一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），因为所用到的思想和之前的类型，可以自行百度搜索源码进行学习。下面我们来说一下多机如何找第 K 大元素，这里我们就先搞清两台机器如何找第 K 大的元素，那么多机的原理类似，我们把两台机器分别称为 A 机，和 B 机，如果我们首先用小根堆方法，得出 A 机 N 个元素，前 $K/2$ 大的元素并将其排序，得出 B 机 N 个元素，前 $K/2$ 大的元素并将其排序，我们将 A 的第 $k/2$ 个元素（即 `A[k/2-1]`）和 B 的第 $k/2$ 个元素（即 `B[k/2-1]`）进行比较，有以下三种情况（为了简化这里先假设 k 为偶数，所得到的结论对于 k 是奇数也是成立的）：

- `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]`，意味着 `A[0]` 到 `A[k/2-1]` 的肯定在 `A ∪ B` 的 top k 元素的范围内，换句话说，`A[k/2-1]` 不可能大于 `A ∪ B` 的第 k 大元素。

因此，我们可以放心的删除 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。

当 `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。

假如我们删除了 `A[0]` 到 `A[k/2-1]`，这时我们寻找的是 A 中剩余元素，与 `B[0]` 到 `B[k/2-1]` 中的第 $k/2$ 大的元素，这时我们拿 A 剩余元素的前 $k/4$ （有序的，因此一开始设计有序数组时，尽量充分占用内存），与 `B[k/4-1]` 进行比较，原理与上面类似，是一种递归的手法。

下面我们直接写了一下找出两个升序数组第 K 小的元素的代码【例 8.2.2.9-1】，可以打开代码中的注释，同时去除 `i=8`，可以查看从第 1 小到第 22 小的结果是否正确，代码中我们确保 `len1` 小于 `len2` 的目的是我们首先拿短的数组与 $k/2$ 进行比较，同时注意递归结束条件，当 k 等于 1 时，递归结束，我们这里没有第零小，最小的就是第 1 小，第 1 小的数为 0；当其中一个数组没有 $k/2$ 个元素也没有关系，这样我们针对另外一个数组，下标取 `k-num1`，取的更大一些即可，当然在调用 `FindKthInTwoSortedArray` 时，要确保 k 的值是小于两个数组的长度之和。

【例 8.2.2.9-1】找出两个有序数组第 K 大的数

```
#include <stdio.h>
#include <stdlib.h>
/*在两个升序排序的数组中找到第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, l1, array2, l2, i);  
printf("第%d大: %d\n", i, ret);  
//}  
system("pause");  
}
```

下面我们再来看一道面试题，总计 25 匹马, 5 条赛道, 一匹马一个赛道, 比赛只能得到 5 匹马之间的快慢程度, 请问至少需要比几次, 才能得到前三名, 请写出分析思路(阿里面试题), 这个题目网易, 头条都面试过, 只是马的数目不同, 赛道不同, 分析思路如下:

给所有马标号, 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 $X_1 > X_2 > X_3 > X_4 > X_5$, ($X = A, B, C, \dots$)

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

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(哈哈 写到这想到守望先锋)为 A1, 第一名确定了, 接下来要确定第二名和第三名。(这时候最后两名的 D 组和 E 组都可以淘汰了, 同理, C 组的 C1 在第六次比赛拿第三, 则 C 组除了 C1, 其他的不可能进入前三。B 组 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 场比赛可以确定最快的前三匹马。

完成这种类型的题目，大家要注意通过画图分析，全面思考后，再跟面试官讲解，考的主要是分组思想与淘汰思想

8.2.3 二分查找

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看之外，更多的是做增删查改操作，针对有序数组，我们经常使用的是二分查找，比如有一个跟女生玩的一个游戏，让她心里想一个数，在 0-1000 之间，我们只需要 10 次就可以猜到，为什么呢，因为我们首先问比 500 大，还是比 500 小，如果比 500 小，我们接着问比 250 大，还是比 250 小，这样通过 10 次的询问，就可以得到那个数，原因是我们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，我们查到任意一个数所需的次数只需 10 次，1000 小于 1024，自然 10 次内肯定可以查到。对于有序数组，无论数的范围是如何的，查找的时间复杂度都是 $O(\log_2 n)$ ，因为 n 是数的个数，并不是数的范围。下面【例 8.2.3-1】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 8.2.3-1】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 我们传入的值为 9，能够访问到这个下标，所以这里我们是小于等于 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");
}

```

该代码实例为找到对应元素值后返回对应的数组下表，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，分解思想及其重要，我们通过分解降低问题的复杂度，从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，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：函数用折半查找法在从数组元素 `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，返回负值；如果等于返回零值；如果大于返回正值。数组 `buf` 中的元素应以升序排列。函数 `bsearch()` 的返回值是指向匹配项，如果没有发现匹配项，返回 `NULL`。

思考题：假如是一个有序的单向链表，我们如何使用二分查找呢？如果让你用 `bsearch`，具体如何使用才能够找到对应的元素呢？

8.2.4 哈希查找

二分查找的时间复杂度是 $O(\log n)$ ，但是在一些对查找性能要求极高的场景无法满足要求，比如淘宝的负载均衡服务器，如果我们需要 $O(1)$ 时间复杂度的查找，就必须使用哈希，**哈希又叫散列，散列表（Hash table，也叫哈希表），是根据键（Key）而直接访问在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一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做散列表。

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，显然比直接查找就要快得多。这里使用人名作为关键字，“取首字母”是这个例子中散列函数的函数法则{F(x)}，存放首字母的表对应散列表。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(key)`，对任意给定的关键字值 `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(Hash)表，函数 `f(key)`为哈希(Hash) 函数。所谓的地址就是一个整型数，我们的哈希表是一个数组，`f(key)`返回的值是一个整型数，这样我们就可以 $O(1)$ 复杂度找到数组对应下标位置。

散列函数能使对一个数据序列的访问过程更加迅速有效，通过散列函数，数据元素将

被更快地定位。

实际工作中需视不同的情况采用不同的哈希函数，通常考虑的因素有：

- 计算哈希函数所需时间
- 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

1. 直接寻址法：取关键字或关键字的某个线性函数值为散列地址。即 $H(\text{key})=\text{key}$ 或 $H(\text{key})=a\cdot\text{key}+b$ ，其中 a 和 b 为常数（这种散列函数叫做自身函数）。若其中 $H(\text{key})$ 中已经有值了，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，这样的话，出现冲突的几率就会很大，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散列地址，则冲突的几率会明显降低。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几率较低的散列地址。

3. 平方取中法：当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每一位都相关，故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，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。数位叠加可以有移位叠加和间界叠加两种方法。移位叠加是将分割后的每一部分的最低位对齐，然后相加；间界叠加是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：选择一随机函数，取关键字的随机值作为散列地址，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散列表表长 m 的数 p 除后所得的余数为散列地址。即 $H(\text{key}) = \text{key} \text{ MOD } p, p \leq m$ 。不仅可以对关键字直接取模，也可在折叠、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 选的不好，容易产生同义词。

下面【例 8.2.4-1】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`，这里没有写查找某个字符串的代码，大家可以自行增加一下，同时在存入 `hashTable` 时，我们也可以再次自行申请空间，工作时我们的 `hashTable` 是一个结构体指针数组，比如你查找某个人的名字后，得到 `key` 值后，在对应下标拿到的是一个结构体指针，里边有对应人的年龄，性别，联系方式等等信息。

【例 8.2.4-1】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，但是当数据量较大时，会发生哈希冲突，对不同的关键字可能得到同一散列地址，即 $k_1 \neq k_2$ ，而 $f(k_1)=f(k_2)$ ，这种现象称为哈希冲突（英语：Collision），解决哈希冲突有以下几种方法：

1. 开放寻址法： $H_i=(H(\text{key}) + d_i) \text{ MOD } m, i=1,2, \dots, k(k \leq m-1)$ ，其中 $H(\text{key})$ 为散列函数， m 为散列表长， d_i 为增量序列，可有下列三种取法：

1.1. $d_i=1,2,3, \dots, m-1$ ，称线性探测再散列；

1.2. $d_i=1^2,-1^2,2^2,-2^2, (3)^2, \dots, \pm (k)^2, (k \leq m/2)$ 称二次探测再散列；

1.3. d_i 为伪随机数序列，称伪随机探测再散列。

2. 再散列法： $H_i=RHi(\text{key}), i=1,2, \dots, k$ RHi 均是不同的散列函数，即在同义词产生地址冲突时计算另一个散列函数地址，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：将散列到同一个存储位置的所有元素保存在一个链表中。

除了二分查找，哈希查找，针对我们上面讲的二叉树，红黑树，可以进行二叉树，红黑树查找，红黑树的查找依然是 $O(\log_2 n)$ 的时间复杂度，相对于二分查找的好处是其新增和删除操作次数少，其实还有 B 树查找，B 树的增删查改在 Linux 系统编程讲解 MySQL 的索引实现原理时进行讲解，需要把编程作为职业的同学可以考虑报名王道训练营。

8.2.4 其他算法

计算机的算法还有很多，贪心，动态规划等，针对准备考研复试的同学，掌握了我们讲解的基本算法后，即可参考《王道机试指南》进行算法准备。如果数据结构薄弱，可以首先参考王道的考研数据结构，编写代码，完成对应数据结构的实现，然后写学习王道机试指南。