

第 3 部分

## 数据结构和设计模式

Data structure and design pattern

**本**部分主要介绍求职面试过程中出现的第二个重要的板块——数据结构，包括字符串的使用、堆、栈、排序方法等。此外，随着外企研发机构大量内迁我国，在外企的面试中，软件工程的知识，包括设计模式、UML、敏捷软件开发，以及.NET 技术和完全面向对象语言 C# 的面试题目将会有增无减。关于设计模式的题目在今后的面试中的比重会更加扩大。



# 第 13 章

## 数据结构基础

**面**试时间一般有两个小时，其中至少有 20~30 分钟时间是用来回答数据节构相关问题的。而由于链表是一种相对简单的数据节构，容易引起面试官多次反复发问。此外，数组的排序和逆置也是面试官必考的内容之一。事实上，单链表的复杂程度并不亚于树、图等复杂数据节构。面试官完全有可能构造出极富挑战性的试题。最后一个考点是堆栈，面试官会结合程序对你的思维能力进行考量。

### 13.1 单链表

**面试例题 1：**编程实现一个单链表的建立/测长/打印。[日本某著名家电/通信/IT 企业面试题]

**答案：**

完整代码如下：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>
using namespace std;

typedef struct student
{
    int data;
    struct student *next;
}node;
node *creat()
{
    node *head,*p,*s;
    int x,cycle=1;
    head=(node*)malloc(sizeof(node));
```

```
p=head;
while(cycle)
{
    printf("\nplease input the data:");
    scanf("%d",&x);
    if(x!=0)
    {
        s=(node *)malloc(sizeof(node));
        s->data=x;
        printf("\n%d",s->data);
        p->next=s;
        p=s;
    }
    else cycle=0;
}
head=head->next;
```

```

    p->next=NULL;
    printf("\n    yyy %d",head->data);
    return(head);
}

//单链表测长
int length(node *head)
{
    int n=0;
    node *p;
    p=head;
    while(p!=NULL)
    {
        p=p->next;
        n++;
    }
    return(n);
}

}
//单链表打印
void print(node *head)

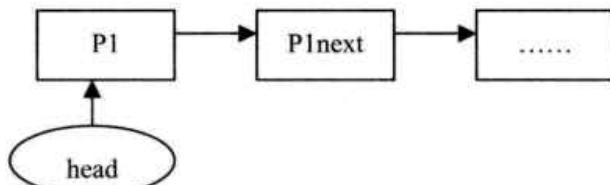
{
    node *p;int n;
    n=length(head);
    printf("\nNow, These %d records are :\n",n);
    p=head;
    if(head!=NULL)
    {

        while(p!=NULL)
        {
            printf("\n      uuu %d      ",p->data);
            p=p->next;
        }
    }
}

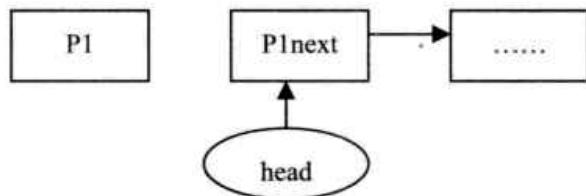
```

**面试例题 2：**编程实现单链表删除节点。[美国某著名分析软件公司面试题]

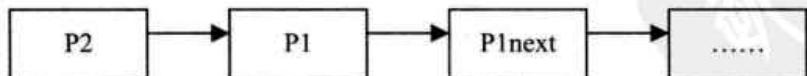
**解析：**如果删除的是头节点，如下图所示。



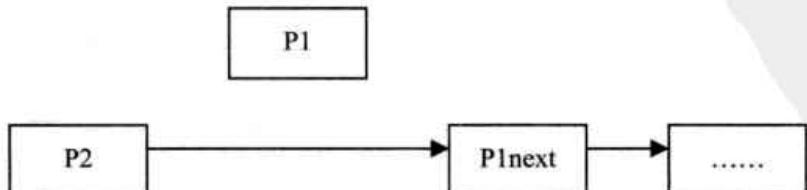
则把 head 指针指向头节点的下一个节点。同时 free p1，如下图所示。



如果删除的是中间节点，如下图所示。



则用 p2 的 next 指向 p1 的 next 同时，free p1，如下图所示。



**答案：**

完整代码如下：

```

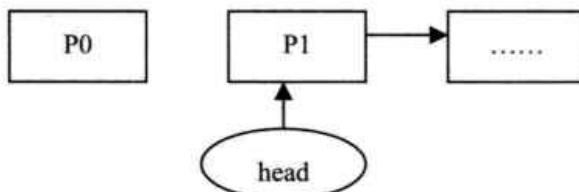
node *del(node *head, int num)
{
    node *p1, *p2;
    p1 = head;
    while (num != p1->data && p1->next != NULL)
        (p2 = p1; p1 = p1->next;)

    if (num == p1->data)
    {
        if (p1 == head)
            {head = p1->next;
             free(p1);}
        else
            p2->next = p1->next;
    }
    else
        printf("\n%d could not been found",
               num);
    return (head);
}

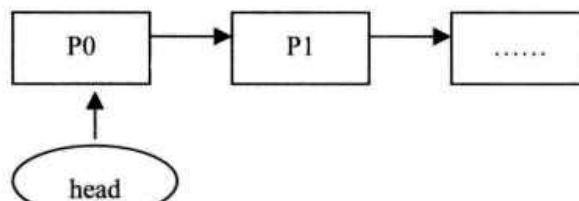
```

**面试例题3：**编写程序实现单链表的插入。[美国某著名计算机嵌入式公司2005年面试题]

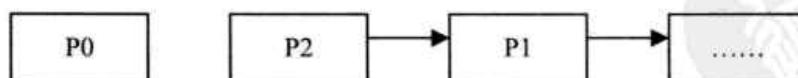
**解析：**单链表的插入，如下图所示。



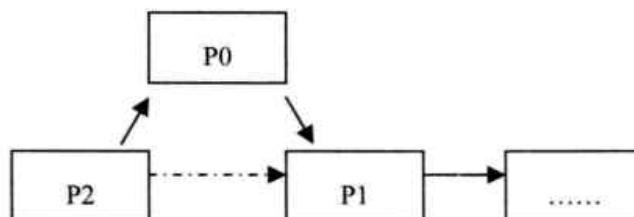
如果插人在头节点以前，则 p0 的 next 指向 p1，头节点指向 p0，如下图所示。



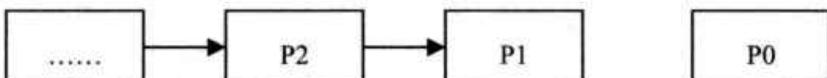
如果插入中间节点，如下图所示。



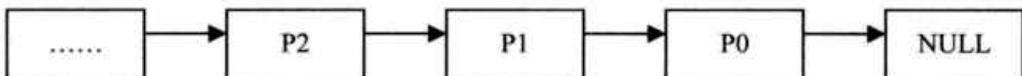
则先让 p2 的 next 指向 p0，再让 p0 指向 p1，如下图所示。



如果插入尾节点，如下图所示。



则先让 p1 的 next 指向 p0，再让 p0 指向空，如下图所示。



**答案：**完整代码如下：

```

node *insert(node *head, int num)
{
    node *p0, *p1, *p2;
    p1 = head;
    p0 = (node *)malloc(sizeof(node));
    p0->data = num;
    while (p0->data > p1->data && p1->next != NULL)
        {p2 = p1; p1 = p1->next;}
    if (p0->data <= p1->data)
    {
        if (head == p1)
            (p0->next = p1);
        else
            {
                p2->next = p0;
                p0->next = p1;
            }
        else
            {
                p1->next = p0; p0->next = NULL;
            }
    }
    return (head);
}
  
```

**面试例题 4：**编程实现单链表的排序。

**答案：**

完整代码如下：

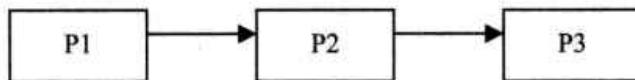
```

node *sort(node *head)
{
    node *p, *p2, *p3;
    int n; int temp;
    n = length(head);
    if (head == NULL || head->next == NULL)
        return head;
    p = head;
    for (int j = 1; j < n; ++j)
    {
        p = head;
        for (int i = 0; i < n - j; ++i)
        {
            if (p->data > p->next->data)
            {
                temp = p->data;
                p->data = p->next->data;
                p->next->data = temp;
                p = p->next;
            }
        }
    }
    return head;
}
  
```

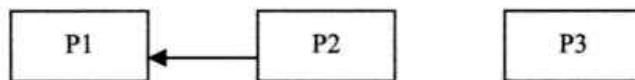
**面试例题 5：**编程实现单链表的逆置。[美国某著名分析软件公司 2005 年面试题]

**解析：**

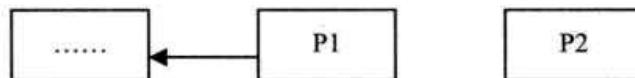
单链表模型如下图所示。



进行单链表逆置，首先要让 p2 的 next 指向 p1，如下图所示。



再由 p1 指向 p2, p2 指向 p3，如下图所示。



然后重复 p2 的 next 指向 p1, p1 指向 p2, p2 指向 p3。

**答案：**完整代码如下：

```

node *reverse(node *head)
{
    node *p1,*p2,*p3;

    if (head == NULL || head->next == NULL)
        return head;

    p1 = head, p2 = p1->next;
    while (p2)
    {
        p3=p2->next;
        p2->next=p1;
        p1=p2;
        p2=p3;
    }

    head->next = NULL;
    head = p1;
    return head;
}

int main()
{
    node *head,stud;
    int n,del num,insert num;
    head=creat();
    print(head);
    cout << "\nInt : ";
    cin >> del num;
    head=del(head,del num);
    print(head);
    cout<<"\nplease input the insert data:";
    cin >> insert num;
    head=insert(head,insert num);
    print(head);

    return 0;
}
  
```

**面试例题 6：**有一个 C 语言用来删除单链表的头元素的函数，请找出其中的问题并加以纠正。

[中国某著名综合软件公司 2005 年面试题]

```

void RemoveHead(node* head)
{
    free(head)
    head=head->next
}
  
```

**解析：**如果先做 free(head)之后，就找不到 head 了，下一句 head= head->next 就不能正确链接了。

**答案：**正确程序如下：

```

void RemoveHead(node* head)
{
    node *p;
    p=head->next;
    head->next=p->next;
    free(p);
}

```

**面试例题 7：**给出一个单链表，不知道节点 N 的值，怎样只遍历一次就可以求出中间节点，写出算法。[中国著名通信企业 H 公司 2007 年 11 月面试题]

```

void RemoveHead(node* head)
{
    free(head);
    head=head->next
}

```

**解析：**设立两个指针，比如\*p 和\*q。p 每次移动两个位置，即  $p=p->next->next$ ，q 每次移动一个位置，即  $q=q->next$ 。

当 p 到达最后一个节点时，q 就是中间节点了。

**答案：**

算法如下：

```

void searchmid(node* head,node*mid )
{
    node *temp=head;
    while(head->next->next!=NULL)
    {
        head=head->next->next;
        temp=temp->next;
        mid=temp;
    }
}

```

## 13.2 双链表

双链表的情况与单链表类似，只是增加了一个前置链而已。

**面试例题 1：**编程实现双链表的建立。

**答案：**完整代码如下：

```

#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>
using namespace std;

typedef struct student
{
    int data;
    struct student *next;
    struct student *pre;
}dnode;
//建立双链表
dnode *creat()
{
    dnode *head,*p,*s;
    int x,cycle=1;
    head=(dnode*)malloc(sizeof(dnode));
    p=head;
    while(cycle)
    {
        printf("\nplease input the data:");
        scanf("%d",&x);
        if(x!=0)
        {
            s=(dnode*)malloc(sizeof(dnode));
            s->data=x;
            printf("\n %d",s->data);
            p->next=s;
            s->pre=p;
            p=s;
        }
    }
}

```

```

        else cycle=0;
    }
    head=head->next;
    head->pre=NULL;
    p->next=NULL;
}

```

```

printf("\n    yyy %d",head->data);
return(head);
}

```

### 面试例题2：编程实现双链表删除/插入节点。

#### 答案：

完整代码如下：

```

//双链表删除节点
dnode *del(dnode *head, int num)
{
    dnode *p1,*p2;
    p1=head;
    while(num!=p1->data&&p1->next!=NULL)
    {p1=p1->next; }

    if(num==p1->data)
    {
        if(p1==head)
            {head=head->next;
             head->pre=NULL;
             free(p1); }
        else if(p1->next==NULL)
        {
            p1->pre->next=NULL;
            free(p1);
        }
        else
        {
            p1->next->pre=p1->pre;
            p1->pre->next=p1->next;
        }
    }
    else
        printf("\n%d could not been
               found",num);
    return(head);
}

//双链表插入节点
dnode *insert(dnode *head,int num)

{
    dnode *p0,*p1;
    p1=head;
    p0=(dnode *)malloc(sizeof(dnode));
    p0->data=num;

    while(p0->data>p1->data&&p1->next!=NULL)
        {p1=p1->next; }
}

```

```

if(p0->data<=p1->data)
{
    if(head==p1)
    {
        p0->next=p1;
        p1->pre=p0;
        head=p0;
    }
    else
    {
        p1->pre->next=p0;
        p0->next=p1;
        p0->pre=p1->pre;
        p1->pre=p0;
    }
}
else //比哪个都大的情况
{
    p1->next=p0;           p0->pre=p1;
p0->next=NULL;
}
return(head);
}

int main()
{
    dnode *head,stud;
    int n,del_num,insert_num;
    head=creat();
    print(head);
    cout << "\nInt : ";
    cin >> del_num;
    head=del(head,del_num);
    print(head);
    cout << "\nplease input the insert data:
";
    cin >> insert_num;
    head=insert(head,insert_num);
    print(head);
    return 0;
}

```

### 13.3 循环链表

**面试例题：**已知  $n$  个人（以编号 1, 2, 3, …,  $n$  分别表示）围坐在一张圆桌周围。从编号为  $k$  的人开始报数，数到  $m$  的那个人出列；他的下一个人又从  $K$  开始报数，数到  $m$  的那个人又出列；依此规律重复下去，直到圆桌周围的人全部出列。试用 C++ 编程实现。[中国著名门户网站 W 公司 2008 年面试题]

**解析：**本题就是约瑟夫环问题的实际场景。要通过输入  $n$ 、 $m$ 、 $k$  3 个正整数，求出列的序列。这个问题采用的是典型的循环链表的数据结构，就是将一个链表的尾元素指针指向队首元素：

```
p->link=head
```

解决问题的核心步骤如下：

- (1) 建立一个具有  $n$  个链节点、无头节点的循环链表。
- (2) 确定第 1 个报数人的位置。
- (3) 不断地从链表中删除链节点，直到链表为空。

**答案：**完整代码如下：

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#define ERROR 0

typedef struct LNode{
    int data;
    struct LNode *link;
}LNode,*LinkList;

void JOSEPHUS(int n,int k,int m)
//n 为总人数, k 为第一个开始报数的人,
//m 为出列者喊到的数
{
    /* p 为当前节点, r 为辅助节点,
       指向 p 的前驱节点, list 为头节点*/
    LinkList p,r,list, curr;

    /*建立循环链表*/
    p=(LinkList)malloc(sizeof(LNode));
    p->data = 0;
    p->link = p;
    curr = p;
    for(int i=1;i <n;i++)
    {

```

```
        LinkList t=(LinkList)malloc
            (sizeof(LNode));
        t->data = i;
        t->link = curr->link;
        curr->link = t;
        curr = t;
    }

    /*把当前指针移动到第一个报数的人*/
    r = curr;
    while (k--) r=p,p=p->link;
    while (n--)
    {
        for (int s=m-1;s--;r=p,p=p->link);
        r->link=p->link;
        printf("%d->", p->data);
        free(p);
        p = r->link;
    }
}

main()
{
    JOSEPHUS(13,4,1);
}
```

## 13.4 队列

**面试例题 1:** If the frequent-used operation done to a link list to access a random selected item with specified index and insert or delete item from the rest, then which of the following is the most suitable structure to save time? (如果按 index 访问 item 并就地插入或删除数据，这种操作比较频繁，那使用什么结构最节省时间？)

- A. Sequential list 顺序表（数组）。
- B. Double linked list 双向链表。
- C. Double linked list with header pointer (单独存储 head 指针的双向链表)。
- D. Linked list 链表。

**答案:A**

**面试例题 2:** 编程实现队列的入队/出队操作。[美国某著名计算机嵌入式公司 2005 年面试题]

**答案:**

完整代码如下：

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>
using namespace std;

typedef struct student
{
    int data;
    struct student *next;
}node;

typedef struct linkqueue
{
    node *first,*rear;
}queue;

//队列的入队
queue *insert(queue *HQ,int x)
{
    node *s;
    s=(node *)malloc(sizeof(node));
    s->data=x;
    s->next=NULL;
    if(HQ->rear==NULL)
    {
        HQ->first=s;
        HQ->rear=s;
    }
    else
    {
        HQ->rear->next=s;
        HQ->rear=s;
    }
    return(HQ);
}

//队列出队
queue *del(queue *HQ)
{
    node *p;int x;
    if(HQ->first==NULL)
    {
        printf("\n yichu");
    }
    else
```

```

{
    x=HQ->first->data;
    p=HQ->first;
    if(HQ->first==HQ->rear)
    {
        HQ->first=NULL;
        HQ->rear=NULL;
    }
    else
    {
        HQ->first=HQ->first->next;
        free(p);
    }
    return(HQ);
}

```

13.5 栈

**面试例题 1：**编程实现栈的入栈/出栈操作。[中国著名网络企业 XL 公司 2007 年 12 月面试题]

**答案：**完整代码如下：

```

#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>
using namespace std;

typedef struct student
{
    int data;
    struct student *next;
}node;

typedef struct stackqueue
{
    node *zhandi,*top;
}queue;

//入栈
queue *push(queue *HQ,int x)
{
    node *s,*p;
    s=(node *)malloc(sizeof(node));
    s->data=x;
    s->next=NULL;

    if(HQ->zhandi==NULL)
    {
        HQ->zhandi=s;
        HQ->top=s;
    }

    else
    {
        HQ->top->next=s;
        HQ->top=s;
    }
}

//出栈
queue *pop(queue *HQ)
{
    node *p;int x;

    if(HQ->zhandi ==NULL)
    {
        printf("\n yichu");
    }

    else
    {
        x=HQ->zhandi->data;
        p=HQ->zhandi;
        if(HQ->zhandi==HQ->top)
        {
            HQ->zhandi=NULL;
            HQ->top=NULL;
        }
        else
        {
            while(p->next!=HQ->top)
            {
                p=p->next;
            }
            HQ->top=p;
            HQ->top->next=NULL;
        }
    }
    return (HQ);
}

```

**面试例题2：**如下C++程序：

```

int i=0x22222222;
char szTest[] = "aaaa";           //a的ASCII码为0x61
func(i, szTest);                //函数原型为void func(int a, char *sz);

```

请问刚进入func函数时，参数在栈中的形式可为以下哪种？（左侧为地址，右侧为数据。）

[中国著名网络企业XL公司2007年11月面试题]

- |            |            |            |            |
|------------|------------|------------|------------|
| A.         | B.         |            |            |
| 0x0013FCF0 | 0x61616161 | 0x0013FCF0 | 0x22222222 |
| 0x0013FCF4 | 0x22222222 | 0x0013FCF4 | 0x0013FCF8 |
| 0x0013FCF8 | 0x0013FCF8 | 0x0013FCF8 | 0x61616161 |
| C.         | D.         |            |            |
| 0x0013FCF0 | 0x22222222 | 0x0013FCF0 | 0x0013FCF8 |
| 0x0013FCF4 | 0x61616161 | 0x0013FCF4 | 0x22222222 |
| 0x0013FCF8 | 0x00000000 | 0x0013FCF8 | 0x61616161 |

**解析：**本题考查的是函数调用时的参数传递和栈帧结构。

调用函数时首先进行参数压栈，一般情况下压栈顺序为从右到左，先压sz再压i……最后压函数地址，但是压sz的时候不是直接压0x61616161而压的是szTest的地址。不过除了压栈顺序，还要考虑栈的生长方向。事实上，在Windows平台上，栈都是从上向下生长的。

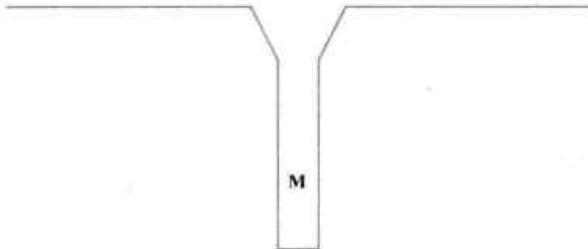
如果依题意压sz压的是0x61616161，压栈顺序为从右到左的话，答案应该是D。

- A: 栈的生长方向应该是从上到下。
- B: 压栈顺序不对。
- C: 压栈顺序不对。

**答案：**D

**面试例题3：**编号为123456789的火车经过如下轨道从左边入口处移到右边出口处（每车都必须且只能进临时轨道M一次，且不能再回左边入口处）

987654321



按照从左向右的顺序，下面的节果不可能的是哪项？[中国台湾某 CPU 硬件厂商 2009 年 11 月面试题]

- A. 123876549
- B. 321987654
- C. 321456798
- D. 987651234

**解析：**本题实际上考的是数据结构的栈。临时轨道 M 就是栈。

- A 是 123 挨个过去，45678 入栈再出栈变成 87654，9 再过去。
- B 是 123 入栈再出栈变成 321，456789 入栈再出栈变成 987654。
- C 是 123 入栈再出栈变成 321，4567 直接过去，89 入栈再出栈变成 98。
- D 是 98765 在前，则 1234 必须全部先进栈，98765 过去后，剩下 1234 必须先回到左边，再通过才满足 1234。但是题意要求不可再回到左边入口处，所以这个选项不可行。

**答案：**D

**扩展知识：**如果 M 只能容纳 4 列车。上面选项应该选哪个才可行？

**面试例题 4：**用两个栈实现一个队列的功能，请用 C++ 实现它。

**解析：**思路如下：

假设两个栈 A 和 B，且都为空。

可以认为栈 A 提供入队列的功能，栈 B 提供出队列的功能。

入队列：入栈 A。

出队列：

- 如果栈 B 不为空，直接弹出栈 B 的数据。
- 如果栈 B 为空，则依次弹出栈 A 的数据，放入栈 B 中，再弹出栈 B 的数据。

**答案：**

代码如下：

```
#include <iostream>
#include <stack>
using namespace std;
template<class T>
struct MyQueue
{
    void push(T &t)
    {
        s1.push(t);
    }
    T front()
    {
        if(s2.empty())
        {
            if(s1.size() == 0) throw;
            while(!s1.empty())
            {
                s2.push(s1.top());
                s1.pop();
            }
        }
        return s2.top();
    }
    void pop()
    {
        if(s2.empty())
        {
            while(!s1.empty())
            {

```

```
                s2.push(s1.top());
                s1.pop();
            }
        }
        if(!s2.empty())
            s2.pop();
    }
    stack<T> s1;
    stack<T> s2;
};

int main()
{
    MyQueue<int> mq;
    int i;
    for( i=0; i< 10; ++i)
    {
        mq.push(i);
    }

    for(i=0; i< 10; ++i)
    {
        cout<<mq.front()<<endl;
        mq.pop();
    }
    return 0;
}
```

## 13.6 堆

**面试例题 1：**请讲述 heap 与 stack 的差别。[美国著名数据库公司 S 2006 年、2008 年面试题]

**解析：**本题 2006 年、2008 年两次出现。

在进行 C/C++ 编程时，需要程序员对内存的了解比较精准。经常需要操作的内存可分为以下几个类别。

- 栈区（stack）：由编译器自动分配和释放，存放函数的参数值、局部变量的值等。其操作方式类似于数据结构中的栈。
- 堆区（heap）：一般由程序员分配和释放，若程序员不释放，程序结束时可能由操作系统回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
- 全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的全

全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。

- 文字常量区：常量字符串就是放在这里的。程序结束后由系统释放。
- 程序代码区：存放函数体的二进制代码。

以下是一段实际说明的程序代码：

```
//main.cpp
int a = 0;           //全局初始化区
char *p1;            //全局未初始化区
main()
{
    int b;           //栈
    char s[] = "abc"; //栈
    char *p2;          //栈
    char *p3 = "123456";
    //123456在常量区，p3在栈上
    static int c = 0
```

```
//全局（静态）初始化区
p1 = (char *)malloc(10);
p2 = (char *)malloc(20);
//分配得来的10和20字节的区域就在堆区
strcpy(p1, "123456");
//123456放在常量区，编译器可能会将它与p3
//所指向的“123456”优化成一个地方
}
```

堆和栈的理论知识如下。

### 1. 申请方式

栈：由系统自动分配。例如，声明在函数中的一个局部变量 int b，系统自动在栈中为 b 开辟空间。

堆：需要程序员自己申请，并指明大小，在 C 中用 malloc 函数。

如：

```
p1 = (char *)malloc(10);
```

在 C++ 中用 new 运算符，如：

```
p2 = (char *)malloc(10);
```

但是注意 p1、p2 本身是在栈中的。

### 2. 申请后系统的响应

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆节点，然后将该节点从空闲节点链表中删除，并将该节点的空间分配给程序。对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 delete 语句才能正确地释放本内存空间。另外，由于找到的堆节点的大小不一定正好等于申请的大小，系统会自动地将多余的那部分重新放入空闲链表中。

### 3. 申请大小的限制

栈：在 Windows 下，栈是向低地址扩展的数据节构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 Windows 下，栈的大小是 2MB（也有的说是 1MB，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间，将提示 overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据节构，是不连续的内存区域。这是由于系统是用链表存储空闲内存地址的，自然是不连续的。而链表的遍历方向是由低地址向高地址，堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

### 4. 申请效率的比较

栈：由系统自动分配，速度较快。但程序员无法控制。

堆：是由 new 分配的内存，一般速度比较慢，而且容易产生内存碎片，不过用起来最方便。

另外，在 Windows 下，最好的方式是用 VirtualAlloc 分配内存。不是在堆，也不是在栈，而是直接在进程的地址空间中保留一块内存，虽然用起来最不方便，但是速度最快，也最灵活。

### 5. 堆和栈中的存储内容

栈：在函数调用时，第一个进栈的是主函数中的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数。在大多数的 C 编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。

当本次函数调用节束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。

堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员安排。

### 6. 存取效率的比较

```
char s1[] = "aaaaaaaaaaaaaa";
char *s2 = "bbbbbbbbbbbbbbbb";
```

aaaaaaaaaa 是在运行时刻赋值的，而 bbbbbbbbbb 是在编译时就确定的。但是，在以后的存取中，在栈上的数组比指针所指向的字符串（例如堆）快。

比如：

```
#include
void main()
{
    char a = 1;
    char c[] = "1234567890";
```

```
char *p ="1234567890";
a = c[1];
a = p[1];
return;
```

对应的汇编代码如下：

```

10: a = c[1];
00401067 8A 4D F1 mov cl,byte ptr [ebp-0Fh]
0040106A 88 4D FC mov byte ptr [ebp-4],cl
11: a = p[1];

```

```

0040106D 8B 55 EC mov edx,dword ptr
[ebp-14h]
00401070 8A 42 01 mov al,byte ptr [edx+1]
00401073 88 45 FC mov byte ptr [ebp-4],al

```

第一种在读取时直接就把字符串中的元素读到寄存器 cl 中，而第二种则要先把 edx 指中，再根据 edx 读取字符，显然慢了。

## 7. 小节

堆和栈的区别可以用如下的比喻来描述。

使用栈就像我们去饭馆里吃饭，只管点菜（发出申请）、付钱和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作。好处是快捷，但是自由度小。使用堆就像是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

数据结构方面的堆和栈，这些都是不同的概念。这里的堆实际上指的就是（满足堆性质的）优先队列的一种数据结构，第一个元素有最高的优先权；栈实际上就是满足先进后出的性质的数据或数据结构。虽然“堆栈”的说法是连起来叫，但是它们还是有很大区别的。

### 答案：

heap 是堆， stack 是栈。

stack 的空间由操作系统自动分配/释放， heap 上的空间手动分配/释放。

stack 空间有限， heap 是很大的自由存储区。

C 中的 malloc 函数分配的内存空间即在堆上， C++ 中对应的是 new 操作符。

程序在编译期对变量和函数分配内存都在栈上进行，且程序运行过程中函数调用时参数的传递也在栈上进行。

**面试例题 2：**全部变量放在( )；函数内部变量 static int ncount 放在( )；函数内部变量 char \*p="AAA"， p 的位置在( )；指向空间的位置( )；函数内变量 char \*p=new char;， p 的位置( )；指向空间的位置( )。[中国某著名综合软件公司 2005 年面试题]

- A. 数据段      B. 代码段      C. 堆栈      D. 堆      E. 不一定，看情况

**解析：**堆栈是一种简单数据结构，是一种只允许在其一端进行插入或删除的线性表。允许插入或删除操作的一端称为栈顶，另一端称为栈底。对堆栈的插入和删除操作称为入栈和出栈。有一组 CPU 指令可以实现对进程的内存实现堆栈访问。其中， POP 指令实现出栈操作， PUSH 指令实现入栈操作。CPU 的 ESP 寄存器存放当前线程的栈顶指针， EBP 寄存器中保存当前线程的栈底指针。CPU 的 EIP 寄存器存放下一个 CPU 指令存放的内存地址。当 CPU 执行完当前的指令后，从 EIP 寄存器中读取下一条指令的内存地址，然后继续执行。

答案：A, A, C, A, C, D。

## 扩展知识（变量的内存分配情况）

接触过编程的人都知道，高级语言都能通过变量名访问内存中的数据。那么这些变量在内存中是如何存放的呢？程序又是如何使用这些变量的呢？

首先，来了解一下C语言的变量是如何在内存分布的。C语言有全局变量（Global）、本地变量（Local）、静态变量（Static）和寄存器变量（Register）。每种变量都有不同的分配方式。先来看下面这段代码：

```
#include <stdio.h>
int g1=0, g2=0, g3=0;
int main()
{
    static int s1=0, s2=0, s3=0;
    int v1=0, v2=0, v3=0;
    //打印出各个变量的内存地址
    printf("0x%08x ",&v1);
    //打印各本地变量的内存地址
    printf("0x%08x ",&v2);
    printf("0x%08x ",&v3);
}
```

```
printf("0x%08x ",&g1);
//打印各全局变量的内存地址
printf("0x%08x ",&g2);
printf("0x%08x ",&g3);
printf("0x%08x ",&s1);
//打印各静态变量的内存地址
printf("0x%08x ",&s2);
printf("0x%08x ",&s3);
return 0;
}
```

编译后的执行结果是：

```
0x0012ff78
0x0012ff7c
0x0012ff80
0x004068d0
0x004068d4
```

```
0x004068d8
0x004068dc
0x004068e0
0x004068e4
```

输出的结果就是变量的内存地址。其中v1、v2、v3是本地变量，g1、g2、g3是全局变量，s1、s2、s3是静态变量。你可以看到这些变量在内存中是连续分布的，但是本地变量和全局变量分配的内存地址差了十万八千里，而全局变量和静态变量分配的内存是连续的。这是因为本地变量和全局/静态变量是分配在不同类型的内存区域中的结果。对于一个进程的内存空间而言，可以在逻辑上分成3个部分：代码区、静态数据区和动态数据区。动态数据区一般就是“堆栈”。“栈(stack)”和“堆(heap)”是两种不同的动态数据区。栈是一种线性结构，堆是一种链式结构。进程的每个线程都有私有的“栈”，所以每个线程虽然代码一样，但本地变量的数据都是互不干扰的。

一个堆栈可以通过“基地址”和“栈顶”地址来描述。全局变量和静态变量分配在静态数据区，本地变量分配在动态数据区，即堆栈中。程序通过堆栈的基地址和偏移量来访问本地变量，如右图所示。



堆栈是一个先进后出的数据结构，栈顶地址总是小于等于栈的基址。我们可以先了解一下函数调用的过程，以便对堆栈在程序中的作用有更深入的了解。不同的语言有不同的函数调用规定，这些因素有参数的压入规则和堆栈的平衡。Windows API 的调用规则和 ANSI C 的函数调用规则是不一样的，前者由被调函数调整堆栈，后者由调用者调整堆栈。两者通过“`_stdcall`”和“`_cdecl`”前缀区分。先看下面这段代码：

```
void __stdcall func(int param1,int param2,int param3)
{
    int var1=param1;
    int var2=param2;
    int var3=param3;

    printf("&param1:0X%08X\n",&param1);
    printf("&param2:0X%08X\n",&param2);
    printf("&param3:0X%08X\n",&param3);
```

```
printf("&var1:0X%08X\n",&var1);

printf("&var2:0X%08X\n",&var2);

printf("&var3:0X%08X\n",&var3);

int main()
{
    func(1,2,3);
    return 0;
}
```

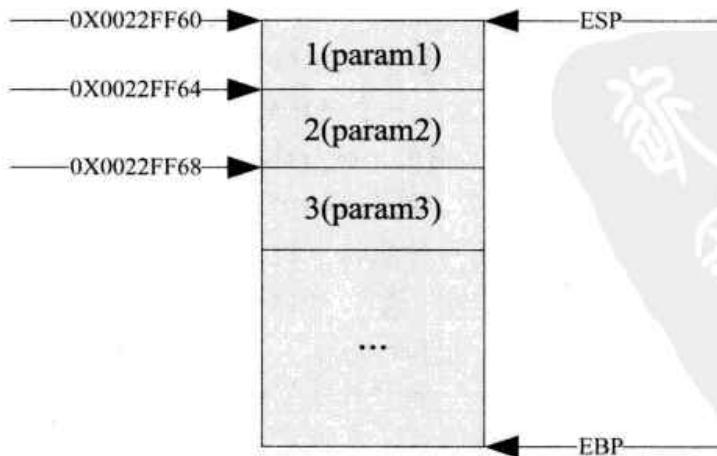
编译后的执行结果是：

&param1:0X0022FF60  
&param2:0X0022FF64  
&param3:0X0022FF68

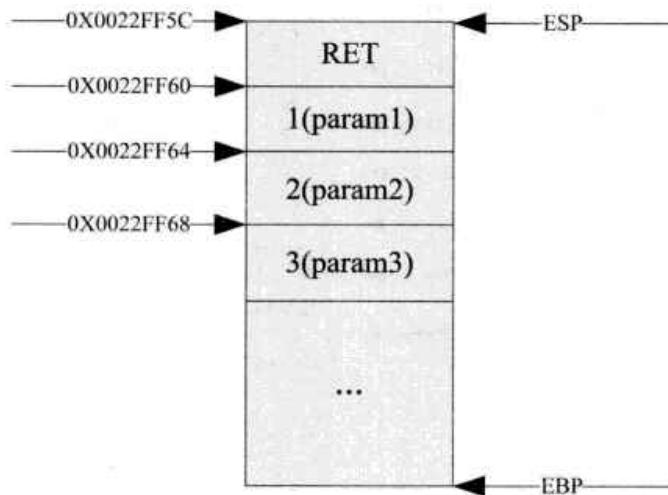
&var1:0X0022FF54  
&var2:0X0022FF50  
&var3:0X0022FF4C

下面详细解释函数调用的过程中堆栈的分布：

在堆栈中分布变量是从高地址向低地址分布，EBP 指向栈底指针，ESP 是指向栈顶的指针，根据 `_stdcall` 调用约定，参数从右向左入栈，所以首先，3 个参数以从右到左的次序压入堆栈，先压 “param3”，再压 “param2”，最后压入 “param1”。栈内分布如下图所示。



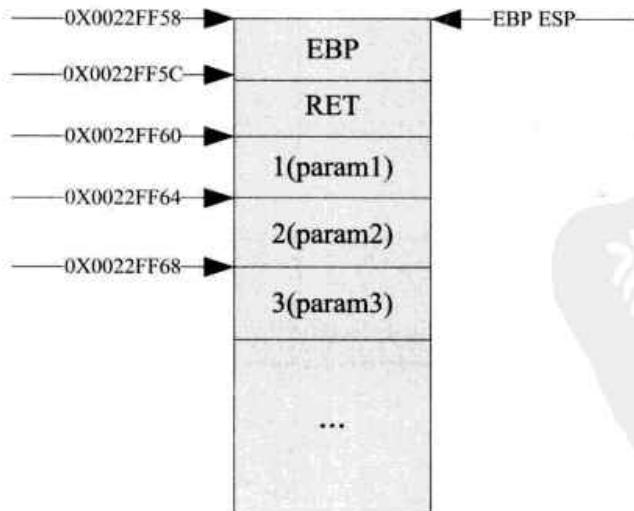
然后函数的返回地址入栈，栈内分布如下图所示。



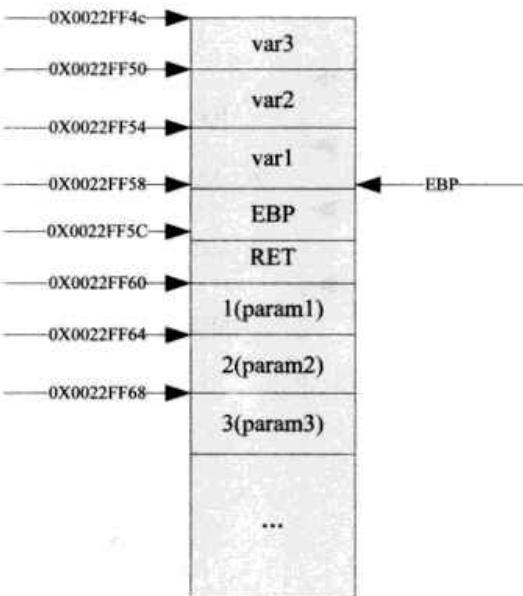
通过跳转指令进入函数。函数地址入栈后，EBP入栈，然后把当前的ESP的值给EBP，汇编下指令为：

```
push ebp  
mov ebp esp
```

此时栈底指针和栈顶指针指向同一位置，栈内分布为如下图所示。



然后是 int var1=param1;int var2=param2;int var3=param3;也就是变量var1,var2,var3的初始化(从左向右的顺序入栈)，按声明顺序依次存储到EBP-4,EBP-8,EBP-12位置，栈内分布如下图所示。



Windows 下的动态数据除了可存放在栈中，还可以存放在堆中。了解 C++ 的朋友都知道，C++ 可以使用 new 关键字来动态分配内存。来看下面的 C++ 代码：

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

void func()
{
    char *buffer=new char[128];
    char bufflocal[128];
    static char buffstatic[128];
    printf("0x%08x ",buffer);
    //打印堆中变量的内存地址
}

void main()
{
    func();
    return;
}
```

程序执行结果为：

0x004107d0  
0x0012ff04

0x004068c0

可以发现用 new 关键字分配的内存既不在栈中，也不在静态数据区。

### 面试例题 3：以下两种情况：

- (1) new 一个 10 个整型的数组。
- (2) 分 10 次 new 一个整型的变量。

哪个占的空间更大些？

- A. 1      B. 2      C. 一样多      D. 不确定

[美国著名操作系统、数据库软件公司 W 2008年4月面试题]

**解析：**如果所谓“占用空间”不包含操作系统 HeapAlloc 时的额外损耗，则两者占用空间是一样多的。如果包括额外损耗，自然是第二个占用空间多。

有人觉得 new 一个 10 个整型的数组会包含数组长度，以便 delete [] 时知道要释放多少空间，事实上，这种说法是不正确的。

在调用 HeapAlloc 的时候，操作系统已经记录了这块分配的内存的大小，因此 HeapFree 的时候直接给出指针就行，根本不必给出额外的大小信息。因此如果对于内建类型，完全不必另外开 4 字节去保存数组的长度，直接调用 HeapFree 就可以了。

可以编程做一个测试：

```
#include <stdio.h>
class CTest
{
public:
    ~CTest()
    { printf("~CTest dtor\n"); }
};

int main()
```

```
{
    int *p = new int[10];
    //所生成的汇编代码在下面
    printf( "%d\n", *((int*)p-1) );
    delete []p;
    return 0;
}
```

汇编代码如下：

```
int *p = new int[10];
0040101E push 28h;传参, 0x28h=40 即 10 个
int 的大小
00401020 call operator new[] (402F10h);调
用 operator new 分配内存, 里面实际调用了 malloc
的内核
```

```
00401025 add esp,4
00401028 mov dword ptr [ebp-0E0h],eax
0040102E mov eax,dword ptr [ebp-0E0h]
00401034 mov dword ptr [p],eax
```

但是到了带有析构函数的类就不行了，因为 delete []p 的时候需要知道调用析构函数的次数，此时便有额外的 4 字节被分配来记录数组大小。

如果上面的程序改成：

```
int main()
{
//所生成的汇编代码在下面
CTest *p = new CTest[10];
```

```
printf( "%d\n", *((int*)p-1) );
delete []p;
return 0;
}
```

汇编代码如下：

```
;CTest *p = new CTest[10];
0040101E push - 0Eh ;0x0e=14h 正好是 10*sizeof(CTest)+4
;其中 sizeof(CTest)=1
00401020 call operator new[] (403050h)
;调用 operator new 分配内存, 里面实际调用了 malloc 的内核
00401025 add esp,4 ;清理
00401028 mov dword ptr [ebp-0ECh],eax
0040102E cmp dword ptr [ebp-0ECh],0
```

```

00401035 je      main+54h (401054h)
00401037 mov     eax,dword ptr [ebp-0ECh]
0040103D mov     dword ptr [eax],0Ah
00401043 mov     ecx,dword ptr [ebp-0ECh]
00401049 add     ecx,4

0040104C mov     dword ptr [ebp-0F4h],ecx
00401052 jmp     main+5Eh (40105Eh)
00401054 mov     dword ptr [ebp-0F4h],0
0040105E mov     edx,dword ptr [ebp-0F4h]
00401064 mov     dword ptr [p],edx

```

在后面 `delete []p` 的时候，正是根据那个额外的信息来确定调用析构函数的次数的。实际是调用了 `CTest::vector` deleting destructor，这是为每个类默认生成的一个析构函数。

由题意来看，应该是考堆的分配的额外损耗的问题，所以笔者认为第二种做法占用空间多。

至于 `new` 是否会产生“内存碎片”的问题，因为内存碎片只是系统分给程序时会产生的：假如程序第一次分配了 10KB，而系统内存分配粒度为 32KB，剩余的 22KB 如果没有被程序用到则产生内存碎片。如果程序继续分配，则系统很可能优化，继续使用剩下的 22KB。而本例中没有大于分配粒度，因此产生的碎片应该是一样的。

但对于操作系统来说：`new` 一块内存，Windows 不仅分配给你内存，还用 4 字节在那段内存后作为内存分配边界，如果从这个角度来说的话第二次就分配多了。

**答案：**B

## 13.7 树、图、哈希表

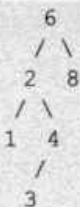
**面试例题 1：**There is binary search tree which is used to store characters 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', which of the following is post-order tree walk result? (有一个二叉搜索树用来存储字符'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'下面哪个结果是后序树遍历结果) [美国著名软件公司 M2009 年 11 月笔试题]

- A. ADBCEGFH
- B. BCAGEHFD
- C. BCAEFDHG
- D. BDACEFHG

**解析：**二叉搜索树（Binary Search Tree），或者是一棵空树，或者是具有下列性质的二叉树：对于树中的每个节点 X，它的左子树中所有关键字的值都小于 X 的关键字值，而它的右子树中的所有关键字值都大于 X 的关键字值。这意味着该树所有的元素都可以用某种统一的

方式排序。

例如下面就是一棵合法的二叉搜索树：



它的左、右子树也分别为二叉搜索树。

二叉搜索树的查找过程和次优二叉树类似，通常采取二叉链表作为二叉搜索树的存储结构。中序遍历二叉搜索树可得到一个关键字的有序序列，一个无序序列可以通过构造一棵二叉搜索树变成一个有序序列，构造树的过程即为对无序序列进行排序的过程。每次插入的新节点都是二叉搜索树上新的叶子节点，在进行插入操作时，不必移动其他节点，只需改动某个节点的指针，由空变为非空即可。搜索、插入、删除的复杂度等于树高，即  $O(\log(n))$ 。

二叉树的一个重要的应用是它们在查找中的使用。二叉搜索树的概念相当容易理解，二叉搜索树的性质决定了它在搜索方面有着非常出色的表现：要找到一棵树的最小节点，只需要从根节点开始，只要有左儿子就向左进行，终止节点就是最小的节点。找最大的节点则是往右进行。例如上面的例子中，最小的节点是 1，在最左边；最大的节点是 8，在最右边。

对于本题而言，二叉搜索树则必满足对树中任一非叶节点，其左子树都小于该节点值，右子树所有节点值都大于该节点值。结合二叉树后序遍历的特点，最后一个肯定是根节点

A. ADBCEGFH

->(H) 左子树(ADBCEGF)，右子树(空) (左子树必须都小于根 H，右子树都大于根 H)

-->(F) 左子树 (ADBCE)，右子树(G)

--->(E) 左子树 (ADBC)，右子树(空)

---->(C) 剩下(ADB)不能区别左子树，右子树，所以选项 A 不成立；

B. BCAGEHFD

->(D, (BCA), (GEHF))

--> GEHF, F 为根，剩下 GEH 不能根据 F 分成两个子段，所以 B 不成立；

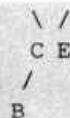
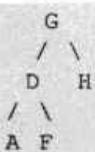
C. BCAEFDHG

->(G, (BCAEFD), (H))

-->(G, (D, (BCA), (EF)), (H))

--->(G, (D, (A, (), (BC)), (F, (E, ()))), (H))

---->(G, (D, (A, (), (C, (B), ()))), (F, (E, ()))), (H))



选项 C 成立；

D. BDACEFHG

-> (G, (BDACEF), (H))

--> (G, (F, (BDACE), (), (H)))

---> (G, (F, (E, (BDAC), (), 0), 0), (H))

----> BDAC 子树，C 为根，据 C 不能将序列 BDA 划分为两个子序列，使得左子序列全小于 C，右子序列全大于 C

所以选项 D 不成立。

**答案：**C

**面试例题 2：**如下数据节构：

```

typedef struct TreeNode {
    char c;
    TreeNode *leftchild;
    TreeNode *rightchild;
}
  
```

请实现两棵树是否相等的比较，相等返回 0 否则返回其他值。并说明你的算法复杂度。

[美国著名软件公司 M2009 年 11 月笔试题]

```
int CompTree(TreeNode* tree1, TreeNode* tree2);
```

注：A、B 两棵树相等当且仅当 RootA->c==RootB->c，而且 A 和 B 的左右子树对应相等或者左右互换后相等。

**答案：**这道题涉及二叉树，用递归方法比较方便，具体代码如下：

```

int CompTree(TreeNode *tree1, TreeNode *tree2)
{
    bool isTree1Null = (tree1 == NULL);
    bool isTree2Null = (tree2 == NULL);
    // 其中一个为 NULL，而另一个不为 NULL，肯定不相等
    if (isTree1Null != isTree2Null)
        return 1;
    // 两个都为 NULL，一定相等
    if (isTree1Null && isTree2Null)
        return 0;
    // 两个都不为 NULL，如果 c 不等，则一定不相等
    if (tree1->c != tree2->c)
        return 1;
  
```

```

// 两个都不为NULL, 且c相等, 则看两棵子树是否相等或者是否互换相等
return (CompTree(tree1->left, tree2->left) & CompTree(tree1->right, tree2->right)) ||
(CompTree(tree1->left, tree2->right) & CompTree(tree1->right, tree2->left));
}

```

**面试例题 3:** A quad-tree, starting from the root node, could consist of many nodes: leaf-node and non-leaf node. Each non-leaf node may have 1 to 4 child nodes; each node has an internal value V, if not null, which would refer to any node in the same quad-tree. Hierarchically, depth of node describes the distance between a node in a tree and the tree's root node, the farther the distance is, the deeper the node is at in the tree. The goal is to find all the nodes in the quad-tree which fulfills the condition: the value of node A refers to a node B in the same tree, where the depth of node A is larger than the depth of node B. The input would be a data structure representing the quad-tree; the output would be a data structure representing the list of nodes fulfilling the conditions. (四叉树由许多个节点组成，其起点是根节点。节点有两种：有叶节点，无叶节点，其中每个无叶节点又可分出1到4个子节点。每一节点都包含其内在价值V。如果这个价值是有效值，则可以表示同一四叉树上任一节点的价值。从等级上划分，节点的深度表明节点与根节点之间的距离：距离越远，节点的深度越深。目的是找出四叉树上所有符合以下条件的节点：在同一树上，节点A的价值说明节点B的价值但其深度要大于节点B的深度。四叉树以数据输入的节构呈现，而符合各条件的节点列以数据输出的节构呈现。)

Question: Describe how you will solve the problem and explain why you pick the solution. The best answer should consider multiple solutions and choose the optimal one in terms of time and space complexity, and explain why you choose this one. (说明你如何解决这个问题并解释你为什么采取那种解决方式。最好的回答应是在多种解决方案中根据时空的复杂程度选择最佳的一种并解释你为什么选择该项。) [英国某图形软件公司 2009 年 9 月笔试试题]

**解析：**用广度优先的方法遍历（先遍历离根近的节点），配合 hash（将所有节点信息保存至 hash 表，同时记录节点同根节点的距离），当某节点包含引用值时，判断 hash 里面是否存在该节点，如果不存在，则说明当前节点的级别  $<=$  引用节点的级别，如果 hash 里面存在该节点信息，再判断当前节点的级别是否  $=$  引用节点的级别， $!=$  说明引用节点的级别比当前节点高。复杂度为  $O(n)$ 。

**答案：**建立该树时按照完全四叉树的位置给每个节点编号。编号为 n 节点，其 4 个子节点为  $4n, 4n+1, 4n+2, 4n+3$ ，根节点编 0，编 1 都无所谓。要完成题目的工作，只需要把其引用编号（内涵值）与其编号一比就知道了。遍历一次即可，复杂度  $O(N)$ 。

**面试例题 4:** Please give a description of the algorithm to remove the dead code in a program (That is, “dead code elimination”),(An instruction and a variable are dead if it computes values which are not used on any executable path to the procedure’s exit.)For example, in below program, (请给出一个算法，实现移除死代码的功能。) [美国某数据库公司 2009 年 11 月笔试试题]

```
b=a+c
d=c+f
d=d*a
d=d*c
return d;
```

Instruction “ $b=a+c$ ” is dead since  $b$  has no effect to the final output  $d$  (比如说上面的代码中  $b=a+c$  就对最后结果  $d$  一点用处都没有。)

We need an algorithm to find out such dead code and remove them from the final program.(请写出算法和解题思路。)

**解析:** 本题解法涉及图论。

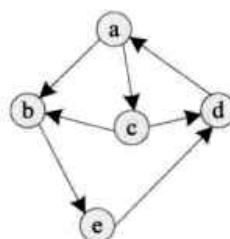
在数学上，一个图是表示物件与物件之间的关系的方法，是图论的基本研究对象。如果给图的每条边规定一个方向，那么得到的图称为有向图。在有向图中，与一个节点相关联的边有出边和入边之分。相反，边没有方向的图称为无向图。

度 (Degree) 是一个顶点的度是指与该边相关联的边的条数，顶点  $v$  的度记作  $d(v)$ 。有向图的顶点的度可分入度和出度。一个顶点的入度是指与该边相关联的入边的条数，出度则指与该边相关联的出边的条数。

图的存储结构表示法如下几种：

### 1. 邻接矩阵表示法

如下图所示。



有向图中顶点的度 = 顶点的入度 + 顶点的出度。邻接矩阵表示如下图所示。

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

无向图的邻接矩阵是对称的；有向图的邻接矩阵可能是不对称的。

在有向图中，统计第  $i$  行 1 的个数可得顶点  $i$  的出度，统计第  $j$  列 1 的个数可得顶点  $j$  的入度。

- 顶点 a；入度 1，出度 2，度 3。
- 顶点 b；入度 2，出度 1，度 3。
- 顶点 c；入度 1，出度 2，度 3。
- 顶点 d；入度 2，出度 1，度 3。
- 顶点 e；入度 1，出度 1，度 2。

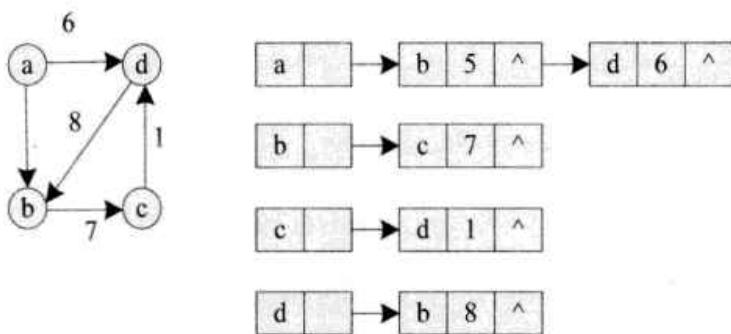
图的邻接矩阵存储表示：

```
#define INFINITY INT_MAX // 最大值
#define MAX VERTEX NUM 20 // 最大顶点个数
//(有向图, 有向网, 无向图, 无向网)
typedef enum {DG, DN, AG, AN} GraphKind;
typedef struct ArcCell {
    // VRTType 是顶点关系类型。对无权图, 用 1 或 0 表示
    // 相邻否; 对带权图, 则为权值类型
    VRTType adj;
    InfoType *info; // 该弧相关信息的指针
} ArcCell,
AdjMatrix[MAX VERTEX NUM][MAX VERTEX NUM];
typedef struct {
    // 顶点向量
    VertexType vexs[MAX VERTEX NUM];
    AdjMatrix arcs; // 邻接矩阵
    // 图的当前顶点数和弧(边)数
    int vexnum, arcnum;
    GraphKind kind; // 图的种类标志
} MGraph;
```

构造一个具有  $n$  个顶点和  $e$  条边的无向网的时间复杂度为  $O(n^2 + e * n)$ ，其中  $O(n^2)$  用于对邻接矩阵初始化。

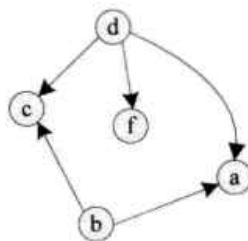
## 2. 邻接表存储表示法

邻接表是图的一种链式存储结构，它对图中每个顶点建立一个单链表，第  $i$  个单链表中的节点表示依附于顶点  $v_i$  的边（对有向图是以顶点  $v_i$  为尾的弧），每个节点由三个域组成：邻节点域（adjvex）指示与顶点  $v_i$  邻接的点在图中的位置，链域（nextarc）指示下一条边或弧的节点，数据域（info）存储和边或弧相关的信息（如权值）。每个链表上附设一个表头节点，包含数据域（data）和链域（firstarc）指向链表中的第一个节点，这些表头节点通常以顺序结构的形式存储，以便随机访问任一顶点的链表，如下图所示。



**答案：**本题可以使用图论解决。

根据每一个表达式逐渐建立一张有向图，如下图所示。



以邻接表方式存储：

<pre> struct adjacentNode{     char nodeName[16];     char *nodeNameList; } </pre>	<pre> int Listlength; // 出度数 }adjacentNode; </pre>
	<pre> adjacentNode map[MAX_NODE_NUM]; </pre>

比如读到  $d = c + f$ ，遍历一下  $d$  节点的出度表有没有  $c$ 、 $f$ ，如果没有则加入，并且出度+2。整个图建立完后，根据 return 哪一个节点开始，以该节点单向（出）遍历整个图，得到节点集合，没有包括的节点就是 dead expression。

以邻接矩阵表示，可以得到各点之间的邻接关系如下：

a b c d f
a: 0 0 0 0 0
b: 1 0 1 0 0

c: 0 0 0 0 0
d: 1 0 1 0 1
f: 0 0 0 0 0

可以发现  $d$  与  $a$ 、 $c$ 、 $f$  直接相关，然后会再递归求  $a$ 、 $c$ 、 $f$  的出度，直到所有相关的出度点都求出来。然后和节点总集合相比，没有包括的节点就是 dead expression。

**面试例题 5：**如何设计一个魔方（六面）的程序，说说方法。[中国某著名搜索引擎公司 2010 年 6 月笔试题]

**答案：**把魔方展开，得到 6 个正方形，定义 6 个节构体，内容为一个 9 个点和一个编号，

每个点包括一个颜色标识；在魔方展开图中根据正方形的相邻关系编号，每个正方形都有 4 个函数：左翻、右翻、上翻、下翻。

根据相邻关系，每个操作会引起相邻面的相关操作；比如一个面的左翻会调用右边相邻面的左翻；也就意味着左相邻面的 0、1、2 三个元素与当前面互换；递归下去，直到所有面都交换完毕。

**面试例题 6：**有 1 千万条短信，有重复，以文本文件的形式保存，一行一条，有重复。请用 5 分钟时间，找出重复出现最多的前 10 条。[中国某著名互联网公司 2010 年 5 月笔试题]

**解析：**对于本题来说，某些面试者想用数据库的办法实现：首先将文本导入数据库，再利用 select 语句某些方法得出前 10 条短信。但是实际上用数据库是绝对满足不了 5 分钟解决这个条件的。这是因为 1 千万条短信即使 1 秒钟录入 1 万条（这已经是很快的数据导入了）5 分钟才 3 百万条。即便真的能在 5 分钟内录完 1 千万条，也必须先建索引，不然 sql 语句 5 分钟内肯定得不出结果。但对 1 千万条记录建索引即使在 5 分钟内都不能完成的。所以用数据库的办法不行。

这种类型的题之所以会出现，这是因为互联网公司无时无刻都在需要处理由用户产生的海量数据/日志，所以海量数据的题现在很热，基本上互联网公司都会考。重点考察的是你的数据结构设计与算法基本功。类似题目是如何根据关键词搜索访问最多的前 10 个网站。

**答案：方法 1：**可以用哈希表的方法对 1 千万条分成若干组进行边扫描边建散列表。第一次扫描，取首字节，尾字节，中间随便两字节作为 Hash Code，插入到 hash table 中。并记录其地址和信息长度和重复次数，1 千万条信息，记录这几个信息还放得下。同 hash code 且等长就是疑似相同，比较一下。相同记录只加 1 次进 hash table，但将重复次数加 1。一次扫描以后，已经记录各自的重复次数，进行第二次 hash table 的处理。用线性时间选择可在  $O(n)$  的级别上完成前 10 条的寻找。分组后每份中的 top 10 必须保证各不相同，可 hash 来保证。也可直接按 hash 值的大小来分类。

**方法 2：**可以采用从小到大排序的办法，根据经验，除非是群发的过节短信，否则字数越少的短信出现重复的几率越高。建议从字数少的短信开始找起，比如一开始搜一个字的短信，找出重复出现的 top10 并分别记录出现次数，然后搜两个字的，依次类推。对于对相同字数的比较长的短信的搜索，除了 hash 之类的算法外，可以选择只抽取头、中和尾等几个位置的字符进行粗判，因为此种判断方式是为了加快查找速度但未必能得到真正期望的 top10，因此需要做标记；如此搜索一遍后，可以从各次 top10 节果中找到备选的 top10，如果这 top10 中有刚才做过标记的，则对其对应字数的所有短信进行

精确搜索以找到真正的 top10 并再次比较。

**方法 3：**可以采用内存映射办法，首先，1 千万条短信按现在的短信长度将不会超过 1G 空间，使用内存映射文件比较合适。可以一次映射（当然如果更大的数据量的话，可以采用分段映射），由于不需要频繁使用文件 I/O 和频繁分配小内存，这将大大提高数据的加载速度。其次，对每条短信的第 i (i 从 0 到 70) 个字母按 ASCII 码进行分组，其实也就是创建树。i 是树的深度，也是短信第 i 个字母。

该问题主要是解决两方面的内容，一是内容加载，二是短信内容比较。采用文件内存映射技术可以解决内容加载的性能问题（不仅仅不需要调用文件 I/O 函数，而且也不需要每读出一条短信都分配一小块内存），而使用树技术可以有效减少比较的次数。代码如下：

```
struct TNode
{
    BYTE* pText;
    //直接指向文件映射的内存地址
    DWORD dwCount;
    //计算器，记录此节点的相同短信数
    TNode* ChildNodes[256];
    //子节点数据，由于一个字母的 ASCII 值不可能超过
    256，所以子节点也不可能超过 256
    TNode()
    {
        //初始化成员
    }
    ~TNode()
    {
        //释放资源
    }
};

//int nIndex 是字母下标
void CreateChildNode(TNode* pNode, const
BYTE* pText, int nIndex)
{
    if (pNode->ChildNodes[pText[nIndex]] ==
NULL)
        //如果不存在此子节点，就创建 TNode 构造函数
        //应该有初始化代码
        //为了处理方便，这里也可以在创建的同时把此节
        //点加到一个数组中
}
```

```
pNode->ChildNodes[pText[nIndex]] =
new TNode();
}
if (pText[nIndex+1] == '\0')
    //此短信已完成，记数器加 1，并保存此短信内容
pNode->ChildNodes[pText[nIndex]]->dwCount++;

pNode->ChildNodes[pText[nIndex]]->pText=pText;
}
else //if(pText[nIndex] != '\0')
    //如果还未结束，就创建下一级节点
CreateNode(pNode->ChildNodes[pText[nIndex]], pText,
nIndex+1);
}

//创建根节点，pTexts 是短信数组，dwCount 是短信
//数量（这里是 1 千万）
void CreateRootNode(const BYTE** pTexts,
DWORD dwCount)
{
    TNode RootNode;
    for (DWORD dwIndex=0; dwIndex<dwCount; dwIndex++)
    {
        CreateNode(&RootN, pTexts[dwIndex],
0);
    }
    //所有节点按 dwCount 的值进行排序
    //取前 10 个节点，显示结果
}
```

## 扩展知识：

有 1 亿个浮点数，请找出其中最大的 10000 个。提示：假设每个浮点数占 4 个字节，1 亿个浮点数就要占到相当大的空间，因此不能一次将全部读入内存进行排序。

既然不可以一次读入内存，那可以使用如下方法：

**方法1：**读出100万个数据，找出最大的1万个，如果这100万数据选择够理想，那么最小的这1万个数据里面最小的为基准，可以过滤掉1亿数据里面99%的数据，最后就再一次在剩下的100万(1%)里面找出最大的1万个。

**方法2：**分块查找，比如100万一个块，找出最大1万个，一次下来就剩下100万数据需要找出1万个。

找出100万个数据里面最大的1万个，可以采用快速排序的方法，分2堆，如果大的那堆个数N大于1万个，继续对大堆快速排序一次分成2堆，如果大堆个数N小于1万，就在小的那堆里面快速排序一次，找第 $10000-N$ 大的数字；递归以上过程，就可以找到相关结果。

## 13.8 排序

排序问题是各大IT公司必考的题目。

所谓排序，就是整理文件中的记录，使之按关键字递增（或递减）的顺序排列起来。其确切定义如下：

输入：n个记录 $R_1, R_2, \dots, R_n$ ，其相应的关键字分别为 $K_1, K_2, \dots, K_n$ 。

输出： $R_{i1}, R_{i2}, \dots, R_{in}$ ，使得 $K_{i1} \leq K_{i2} \leq \dots \leq K_{in}$ （或 $K_{i1} \geq K_{i2} \geq \dots \geq K_{in}$ ）。

### 1. 被排序对象——文件

被排序的对象——文件由一组记录组成。

记录则由若干个数据项（或域）组成。其中有一项可用来标识一个记录，称为关键字项。该数据项的值称为关键字（Key）。

### 2. 排序运算的依据——关键字

用来做排序运算依据的关键字，可以是数字类型，也可以是字符类型。

关键字的选取应根据问题的要求而定。

在高考成绩统计中将每个考生作为一个记录。每条记录包含准考证号、姓名、各科的分數和总分数等项内容。若要唯一地标识一个考生的记录，则必须用“准考证号”作为关键字。若要按照考生的总分数排名次，则需用“总分数”作为关键字。

### 3. 排序的稳定性

当待排序记录的关键字均不相同时，排序结果是唯一的，否则排序结果不唯一。

在待排序的文件中，若存在多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，该排序方法是稳定的；若具有相同关键字的记录之间的相对次序发生变化，则称这种排序方法是不稳定的。排序算法的稳定性是针对所有输入实例而言的。即在所有可能的输入实例中，只要有一个实例使得算法不满足稳定性要求，则该排序算法就是不稳定的。稳定的排序如下表所示。

稳定的排序	时间复杂度	空间复杂度
气泡排序（bubble sort）	最差、平均都是 $O(n^2)$ ; 最好是 $O(n)$	1
鸡尾酒排序（Cocktail sort, 双向的冒泡排序）	最差、平均都是 $O(n^2)$ ; 最好是 $O(n)$	1
插入排序（insertion sort）	最差、平均都是 $O(n^2)$ ; 最好是 $O(n)$	1
归并排序（merge sort）	最差、平均、最好都是 $O(n \log n)$	$O(n)$
桶排序（bucket sort）	$O(n)$	$O(k)$
基数排序（Radix Sort）	$O(dn)$ ( $d$ 是常数)	$O(n)$
二叉树排序（Binary tree sort）	$O(n \log n)$	$O(n)$
图书馆排序（Library sort）	$O(n \log n)$	$(1+\epsilon)n$

不稳定的排序如下表所示：

不稳定的排序	时间复杂度	空间复杂度
选择排序（selection sort）	最差，平均都是 $O(n^2)$	1
希尔排序（shell sort）	$O(n \log n)$	1
堆排序（heapsort）	最差、平均、最好都是 $O(n \log n)$	1
快速排序（quicksort）	平均 $O(n \log n)$ ；最坏情况下 $O(n^2)$	$O(\log n)$

#### 4. 排序方法的分类

##### 1) 按是否涉及数据的内、外存交换分

在排序过程中，若整个文件都是放在内存中处理，排序时不涉及数据的内、外存交换，则称为内部排序（简称内排序）。反之，若排序过程中要进行数据的内、外存交换，则称为外部排序。

注意：

- 内排序适用于记录个数不很多的小文件。
- 外排序则适用于记录个数太多，不能一次将其全部记录放入内存的大文件。

##### 2) 按策略划分内部排序方法

可以分为5类：插入排序、选择排序、交换排序、归并排序和分配排序。

#### 5. 排序算法的基本操作

大多数排序算法都有两个基本的操作：

- 比较两个关键字的大小。
- 改变指向记录的指针或移动记录本身。

注意：第二种基本操作的实现依赖于待排序记录的存储方式。

#### 6. 待排文件的常用存储方式

##### 1) 以顺序表（或直接用向量）作为存储节构

排序过程：对记录本身进行物理重排，即通过关键字之间的比较判定，将记录移到合适的位置。

##### 2) 以链表作为存储节构

排序过程：无须移动记录，仅需修改指针。通常将这类排序称为链表（或链式）排序。

3) 用顺序的方式存储待排序的记录，但同时建立一个辅助表（如包括关键字和指向记录位置的指针组成的索引表）

排序过程：只需对辅助表的表目进行物理重排（即只移动辅助表的表目，而不移动记录本身）。适用于难于在链表上实现，且仍需避免排序过程中移动记录的排序方法。

#### 7. 排序算法性能评价

##### 1) 评价排序算法好坏的标准

评价排序算法好坏的标准主要有两条：

- 执行时间和所需的辅助空间。
- 算法本身的复杂程度。

## 2) 排序算法的空间复杂度

若排序算法所需的辅助空间并不依赖于问题的规模  $n$ ，即辅助空间是  $O(1)$ ，则称为就地排序（In-PlaceSort）。

非就地排序一般要求的辅助空间为  $O(n)$ 。

## 3) 排序算法的时间开销

大多数排序算法的时间开销主要是关键字之间的比较和记录的移动。有的排序算法其执行时间不仅依赖于问题的规模，还取决于输入实例中数据的状态。

**面试例题 1：**The following is an improved quick sort algorithm. Please fill in the blank. (下面的程序是一个快速排序问题，请填空。) [美国某著名计算机嵌入式公司 2005 年面试题]

```
# include<iostream>
# include<stdio.h>
void improveqsort(int *list,int m,int n)
{
    int k,t,i,j;
    /* 
    for(i=0;i<10;i++)
        printf("%3d",list[i]);*/
    if(m<n)
    {
        i=m;j=n+1;k=list[m];
        while(i<j)
        {
            for(i=i+1;i<n;i++)
                if(list[i]>=k)
                    break;
            for(j=j-1;j>m;j--)
                if(list[j]<=k)
                    break;
            if(i<j)
                {t=list[i];list[i]=
                 list[j];list[j]=t;}
        }
        t=list[m];list[m]=list[j];
        list[j]=t;
        improveqsort( , , );
        improveqsort( , , );
    }
}
main()
{
    int list[10];
    int n=9,m=0,i;
    printf(" input 10 number: ");
    for(i=0;i<10;i++)
        scanf("%d",&list[i]);
    printf("\n ");
    improveqsort(list,m,n);
    for(i=0;i<10;i++)
        printf("%5d",list[i]);
    printf("\n");
}
```

**解析：**数据结构的快速排序问题。

**答案：**improveqsort(list,m,j-1);

improveqsort(list,i,n);

## 扩展知识（快速排序的算法思想）

快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它采用了一种分治的策略，通常称为分治法（Divide-and-Conquer Method）。

### 1. 分治法的基本思想

分治法的基本思想是将原问题分解为若干个规模更小但结构与原问题相似的子问题。递归地解决这些子问题，然后将这些子问题的解组合为原问题的解。

## 2. 快速排序的基本思想

设当前待排序的无序区为  $R[\text{low}..\text{high}]$ , 利用分治法可将快速排序的基本思想描述为:

### 1) 分解

在  $R[\text{low}..\text{high}]$  中任选一个记录作为基准 (Pivot), 以此基准将当前无序区划分为左、右两个较小的子区间  $R[\text{low}..\text{pivotpos}-1]$  和  $R[\text{pivotpos}+1..\text{high}]$ , 并使左边子区间中所有记录的关键字均小于等于基准记录 (不妨记为 pivot) 的关键字 pivot.key, 右边的子区间中所有记录的关键字均大于等于 pivot.key, 而基准记录 pivot 则位于正确的位位置 (pivotpos) 上, 它无须参加后续的排序。

注意:

划分的关键是要求出基准记录所在的位置 pivotpos。划分的节果可以简单地表示为 (注意  $\text{pivot}=R[\text{pivotpos}]$ ):

$R[\text{low}..\text{pivotpos}-1].\text{keys} \leq R[\text{pivotpos}].\text{key} \leq R[\text{pivotpos}+1..$

. $\text{high}]$ . $\text{keys}$

其中  $\text{low} \leq \text{pivotpos} \leq \text{high}$ .

### 2) 求解

通过递归调用快速排序对左、右子区间  $R[\text{low}..\text{pivotpos}-1]$  和  $R[\text{pivotpos}+1..\text{high}]$  快速排序。

### 3) 组合

因为当“求解”步骤中的两个递归调用节束时, 其左、右两个子区间已有序。对快速排序而言, “组合”步骤无须做什么, 可看做是空操作。

## 3. 快速排序算法 QuickSort

代码如下:

```
void QuickSort(SeqList R, int low, int high)
{
    // 对 R[low..high] 快速排序
    int pivotpos;
    // 划分后的基准记录的位置
    if (low < high) {
        // 仅当区间长度大于 1 时才须排序
        pivotpos=Partition(R, low, high);
        // 对 R[low..high] 做划分
        QuickSort(R, low, pivotpos-1);
        // 对左区间递归排序
        QuickSort(R, pivotpos+1, high);
        // 对右区间递归排序
    }
} //QuickSort
```

注意:

为排序整个文件, 只须调用  $\text{QuickSort}(R, 1, n)$  即可完成对  $R[1..n]$  的排序。

## 4. 划分算法 Partition

### 1) 简单的划分方法

#### ① 具体做法

第1步：（初始化）设置两个指针  $i$  和  $j$ ，它们的初值分别为区间的下界和上界，即  $i=low$ ,  $i=high$ ；选取无序区的第一个记录  $R[i]$ （即  $R[low]$ ）作为基准记录，并将它保存在变量  $pivot$  中。

第2步：令  $j$  自  $high$  起向左扫描，直到找到第1个关键字小于  $pivot.key$  的记录  $R[j]$ ，将  $R[j]$  移至  $i$  所指的位置上，这相当于  $R[j]$  和基准  $R[i]$ （即  $pivot$ ）进行了交换，使关键字小于基准关键字  $pivot.key$  的记录移到了基准的左边，交换后  $R[j]$  中是  $pivot$ ；然后，令  $i$  指针自  $i+1$  位置开始向右扫描，直至找到第1个关键字大于  $pivot.key$  的记录  $R[i]$ ，将  $R[i]$  移到  $i$  所指的位置上，这相当于交换了  $R[i]$  和基准  $R[j]$ ，使关键字大于基准关键字的记录移到了基准的右边，交换后  $R[i]$  中又相当于存放了  $pivot$ ；接着令指针  $j$  自位置  $j-1$  开始向左扫描，如此交替改变扫描方向，从两端各自往中间靠拢，直至  $i=j$  时， $i$  便是基准  $pivot$  最终的位置，将  $pivot$  放在此位置上就完成了一次划分。

## ② 划分算法

代码如下：

```

int Partition(SeqList R, int I, int j)
{
    // 调用 Partition(R, low, high) 时, 对 R[low..high]
    // 做划分并返回基准记录的位置
    RecType pivot=R[i];
    // 用区间的第 1 个记录作为基准
    while(i<j){
        // 从区间两端交替向中间扫描, 直至 i=j 为止
        while(i<j&&R[j].key>=pivot.key)
            // pivot 相当于在位置 i 上
            j--;
            // 从右向左扫描, 查找第 1 个关键字小于 pivot.key
            // 的记录 R[j]
        if(i<j)
            // 表示找到的 R[j] 的关键字 < pivot.key
            R[i++]=R[j];
            // 相当于交换 R[i] 和 R[j],
            // 交换后 i 指针加 1
        while(i<j&&R[i].key<=pivot.key)
            // pivot 相当于在位置 j 上
            i++;
            // 从左向右扫描, 查找第 1 个关键字大于 pivot.
            // key 的记录 R[i]
        if(i<j)
            // 表示找到了 R[i], 使 R[i].key > pivot.key
            R[j--]=R[i];
            // 相当于交换 R[i] 和 R[j],
            // 交换后 j 指针减 1
    }
    R[i]=pivot;
    // 基准记录已被最后定位
    return i;
} //partition

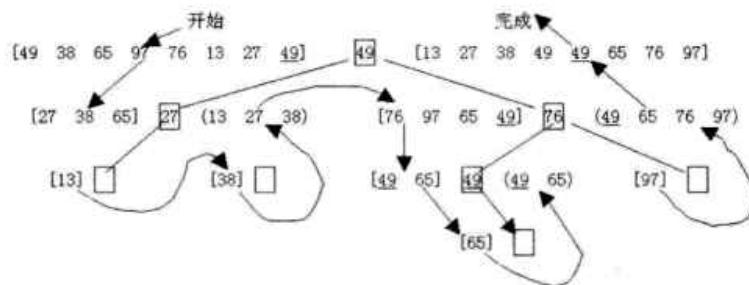
```

## 5. 快速排序执行过程

快速排序执行的全过程可用递归树来描述，如下图所示。

初始关键字:	[49 38 65 97 76 13 27 49]
j 向左扫描	[49 38 65 97 76 13 27 49] j
第一次交换后	[27 38 65 97 76 13 □ 49] j
i 向右扫描	[27 38 65 97 76 13 □ 49] j
第二次交换后	[27 38 □ 97 76 13 65 49] j
j 向左扫描, 位置不变, 第三次交换后	[27 38 13 97 76 □ 65 49] j
i 向左扫描, 位置不变, 第四次交换后	[27 38 13 □ 76 97 65 49] j
j 向左扫描后分区过程结束	[27 38 13 49 76 97 65 49] j

一次划分过程如下图所示。



QuickSort 执行时的递归树:

```

[49 38 65 97 76 13 27 49]           // 初始关键字
[27 38 13] 49 [76 97 65 49]         // 第一次划分完成后, 对应递归树第 2 层
[13] 27 [38] 49 [49 65] 76 [97]      // 对上一层一个无序区划分完成后, 对应递归树第 3 层
13 27 38 49 49 [65] 76 97          // 对上一层一个无序区划分完成后, 对应递归树第 4 层
13 27 38 49 49 65 76 97          // 最后的排序结果

```

对递归树的每层上一个无序区划分之后的状态。

## 6. 时间复杂度

快速排序法是一种不稳定的排序方法, 平均时间复杂度  $O(n \times \lg n / \lg 2)$ , 最差情况时间复杂度为  $O(n^2)$ 。

**面试例题 2:** 请用 C 或 C++写出一个冒泡排序程序, 要求输入 10 个整数, 输出排序结果。[中国著名通信企业 H 公司面试题]

**解析:** 交换排序中的冒泡排序问题。

**答案:**

程序如下：

```
# include<iostream>
# include<stdio.h>
void maopao(int *list)
{
    int i,j,temp;
    for (i=0;i<9;i++)
        for(j=0;j<9-i;j++)
        {
            if (list[j]>list[j+1])
            { temp=list[j];list[j]=list[j+1];list[j+1]=temp; }
        }
}
```

```
main()
{
    int list[10];
    int n=9,m=0,i;
    printf(" input 10 number: ");
    for(i=0;i<10;i++)
        scanf("%d",&list[i]);
    printf("\n ");
    maopao(list);
    for(i=0;i<10;i++)
        printf("%5d",list[i]);
    printf("\n");
}
```

## 扩展知识（交换排序的算法思想）

交换排序的基本思想是两两比较待排序记录的关键字，发现两个记录的次序相反时即进行交换，直到没有反序的记录为止。

应用交换排序基本思想的主要排序方法有冒泡排序和快速排序。

### 1. 冒泡排序方法

将被排序的记录数组  $R[1..n]$  垂直排列，每个记录  $R[i]$  看做是重量为  $R[i].key$  的气泡。根据轻气泡不能在重气泡之下的原则，从下往上扫描数组  $R$ 。凡扫描到违反本原则的轻气泡，就使其向上“飘浮”。如此反复进行，直到最后任何两个气泡都是轻者在上，重者在下为止。

#### 1) 初始

$R[1..n]$  为无序区。

#### 2) 第一趟扫描

从无序区底部向上依次比较相邻的两个气泡的重量，若发现轻者在下、重者在上，则交换二者的位置。即依次比较  $(R[n], R[n-1])$ 、 $(R[n-1], R[n-2])$ 、 $\dots$ 、 $(R[2], R[1])$ ；对于每对气泡  $(R[j+1], R[j])$ ，若  $R[j+1].key < R[j].key$ ，则交换  $R[j+1]$  和  $R[j]$  的内容。

第一趟扫描完毕时，“最轻”的气泡就飘浮到该区间的顶部，即关键字最小的记录被放在最高位置  $R[1]$  上。

#### 3) 第二趟扫描

扫描  $R[2..n]$ 。扫描完毕时，“次轻”的气泡飘浮到  $R[2]$  的位置上。

最后，经过  $n - 1$  趟扫描可得到有序区  $R[1..n]$ 。

注意：

第  $i$  趟扫描时， $R[1..i-1]$  和  $R[i..n]$  分别为当前的有序区和无序区。扫描仍是从无序区

底部向上直至该区顶部。扫描完毕时，该区中最轻气泡飘浮到顶部位置  $R[i]$  上，结果是  $R[1..i]$  变为新的有序区。

## 2. 排序算法

### 1) 分析

因为每一趟排序都使有序区增加了一个气泡，在经过  $n-1$  趟排序之后，有序区中就有  $n-1$  个气泡，而无序区中气泡的重量总是大于等于有序区中气泡的重量，所以整个冒泡排序过程至多需要进行  $n-1$  趟排序。

若在某一趟排序中未发现气泡位置的交换，则说明待排序的无序区中所有气泡均满足轻者在上、重者在下的原则，因此，冒泡排序过程可在此趟排序后终止。为此，在下面给出的算法中，引入一个布尔量  $exchange$ ，在每趟排序开始前，先将其置为 FALSE。若排序过程中发生了交换，则将其置为 TRUE。各趟排序结束时检查  $exchange$ ，若未曾发生过交换则终止算法，不再进行下一趟排序。

### 2) 具体算法

代码如下：

```
void BubbleSort(SeqList R)
{ //R(1..n)是待排序的文件，采用自下向上扫描，对R做冒泡排序
    int i, j;
    Boolean exchange; //交换标志
    for(i=1;i<n;i++) { //最多做n-1趟排序
        exchange=FALSE; //本趟排序开始前，交换标志应为假
        for(j=n-1;j>=i; j--) //对当前无序区R[i..n]自下向上扫描
            if(R[j+1].key<R[j].key){ //交换记录
                R[0]=R[j+1]; //R[0]不是哨兵，仅做暂存单元
                R[j+1]=R[j];
                R[j]=R[0];
                exchange=TRUE; //发生了交换，故将交换标志置为真
            }
        if(!exchange) //本趟排序未发生交换，提前终止算法
            return;
    } //endfor(外循环)
} //BubbleSort
```

## 3. 算法分析

### 1) 算法的最好时间复杂度

若文件的初始状态是正序的，一趟扫描即可完成排序。所需的关键字比较次数 C 和记录移动次数 M 均达到最小值：

$$C_{\min}=n-1$$

$$M_{\min}=0$$

冒泡排序最好的时间复杂度为  $O(n)$ 。

### (2) 算法的最坏时间复杂度

若初始文件是反序的，需要进行  $n - 1$  趟排序。每趟排序要进行  $n - i$  次关键字的比较 ( $1 \leq i \leq n - 1$ )，且每次比较都必须移动记录 3 次来交换记录位置。在这种情况下，比较和移动次数均达到最大值：

$$C_{\max} = n(n - 1)/2 = O(n^2)$$

$$M_{\max} = 3n(n - 1)/2 = O(n^2)$$

冒泡排序的最坏时间复杂度为  $O(n^2)$ 。

3) 算法的平均时间复杂度为  $O(n^2)$

虽然冒泡排序不一定要进行  $n - 1$  趟，但由于它的记录移动次数较多，故平均时间性能比直接插入排序要差得多。

4) 算法稳定性

冒泡排序是就地排序，且它是稳定的。

**面试例题 3：**请用 C 或 C++写出一个 Shell 排序程序，要求输入 10 个整数，输出排序结果。

[中国某著名通信企业 H 公司面试题]

**解析：**希尔排序（Shell Sort）是插入排序的一种，因 D.L.Shell 于 1959 年提出而得名。

**答案：**

完整代码如下：

```
# include<iostream>
# include<stdio.h>
void ShellSort(int* data,int left,
    int right)
{
    int len = right - left + 1;
    int d = len;
    while (d > 1)
    {
        d = (d + 1) / 2;
        for (int i = left; i < right + 1
            - d;i++)
        {
            if (data[i + d] < data[i])
            {
                int tmp = data[i + d];
                data[i + d] = data[i];
                data[i] = tmp;
            }
        }
    }
}

void ShellSort2(int* data,int len)
{
    int d = len;
    while (d > 1)
    {
        d = (d + 1)/2;
        for (int i = 0; i < len - d;i++)
        {
            if (data[i + d] < data[i])
            {
                int tmp = data[i + d];
                data[i + d] = data[i];
                data[i] = tmp;
            }
        }
    }
    for (int i=0;i<10;i++)
}
```

```

        printf("%5d", data[i]);
        printf("\n");
    }
}

main()
{
    int list[10];
    int n=9, m=0, i;
    printf(" input 10 number: ");
    for(i=0; i<10; i++)
        scanf("%d", &list[i]);
}
    printf("\n ");
    ShellSort2(list, 10);
    //ShellSort(list, 0, 9);
    printf("\n");
    for(i=0; i<10; i++)
        printf("%5d", list[i]);
    printf("\n");
}

```

## 扩展知识（希尔（Shell）排序基本思想）

先取一个小于  $n$  的整数  $d_1$  作为第一个增量，把文件的全部记录分成  $d_1$  个组。所有距离为  $d_1$  的倍数的记录放在同一个组中。先在各组内进行直接插入排序，然后，取第二个增量  $d_2 < d_1$  重复上述的分组和排序，直至所取的增量  $d_t = l$  ( $d_t < d_{t-1} < \dots < d_2 < d_1$ )，即所有记录放在同一组中进行直接插入排序为止。

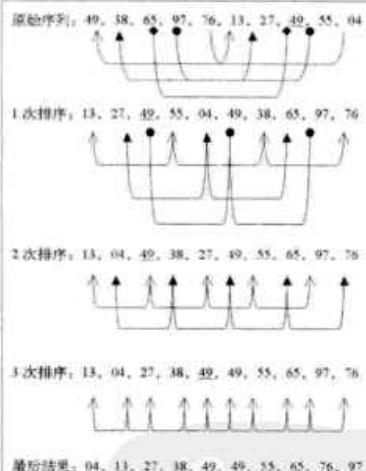
该方法实质上是一种分组插入方法。

给定实例的 Shell 排序的排序过程如下。

假设待排序文件有 10 个记录，其关键字分别是 49, 38, 65, 97, 76, 13, 27, 49, 55, 04。

增量序列的取值依次为 5, 3, 2, 1。排序过程如右图所示。

Shell 排序的算法实现如下：



```

void ShellPass(SeqList R, int d)           // 希尔排序中的一趟排序, d 为当前增量
{
    for(i=d+1; i<=n; i++)                  // 将 R[d+1..n] 分别插入各组当前的
                                            // 有序区
        if(R[i].key<R[i-d].key){           // R[0] 只是暂存单元, 不是哨兵
            R[0]=R[i]; j=i-d;               // 查找 R[i] 的插入位置
            do {                           // 后移记录
                R[j+d]=R[j];               // 查找前一记录
                j=j-d;                     // 插入 R[i] 到正确的位置上
            } while(j>0&&R[0].key<R[j].key);
            R[j+d]=R[0];                 // endif
        }
}

```

```

void ShellSort(SeqList R)
{
    int increment=n;           //增量初值，不妨设 n>0
    do {
        increment=increment/3+1; //求下一增量
        ShellPass(R, increment); //一趟增量为 increment 的 Shell 插入排序
    }while(increment>1)
}
//ShellSort

```

注意：当增量  $d=1$  时，ShellPass 和 InsertSort 基本一致，只是由于没有哨兵而在内循环中增加了一个循环判定条件 “ $j>0$ ”，以防下标越界。

算法分析：

### 1) 增量序列的选择

Shell 排序的执行时间依赖于增量序列。

好的增量序列的共同特征如下：

- 最后一个增量必须为 1。
- 应该尽量避免序列中的值（尤其是相邻的值）互为倍数的情况。

有人通过大量的实验，给出了目前较好的结果：当  $n$  较大时，比较和移动的次数约在  $n^{1.25} \sim 1.6n^{1.25}$  之间。

### 2) Shell 排序的时间性能优于直接插入排序

希尔排序的时间性能优于直接插入排序的原因如下：

- 当文件初态基本有序时，直接插入排序所需的比较和移动次数均较少。
- 当  $n$  值较小时， $n$  和  $n^2$  的差别也较小，即直接插入排序的最好时间复杂度  $O(n)$  和最坏时间复杂度  $O(n^2)$  差别不大。

在希尔排序开始时增量较大，分组较多，每组的记录数目少，故各组内直接插入较快。后来增量  $di$  逐渐缩小，分组数逐渐减少，而各组的记录数目逐渐增多。但由于已经按  $di - 1$  作为距离排过序，使文件较接近于有序状态，所以新的一趟排序过程也较快。

因此，希尔排序在效率上较直接插入排序有较大的改进。

### 3) 稳定性

希尔排序是不稳定的。参见上述实例，该例中两个相同关键字 49 在排序前后的相对次序发生了变化。

**面试例题 4：**以下哪种排序属于稳定排序？[美国某著名分析软件公司 2005 年面试题]

- A. 归并排序    B. 快速排序    C. 希尔排序    D. 堆排序

**解析:** 只有归并排序是稳定排序, 其他3个都是不稳定的。

**答案:** D

## 扩展知识(各种排序方法比较)

按平均时间将排序分为以下4类。

- 平方阶( $O(n^2)$ )排序: 一般称为简单排序, 例如直接插入、直接选择和冒泡排序。
- 线性对数阶( $O(nlgn)$ )排序: 如快速、堆和归并排序。
- $O(n^{1+\varepsilon})$ 阶排序:  $\varepsilon$ 是介于0和1之间的常数, 即 $0 < \varepsilon < 1$ , 如希尔排序。
- 线性阶( $O(n)$ )排序: 如桶、箱和基数排序。

简单排序中直接插入排序最好, 快速排序最快。当文件为正序时, 直接插入排序和冒泡排序均最佳。

1. 影响排序效果的因素因为不同的排序方法适应不同的应用环境和要求, 所以选择合适的排序方法应综合考虑下列因素:

- 待排序的记录数目n。
- 记录的大小(规模)。
- 关键字的结构及其初始状态。
- 对稳定性的要求。
- 语言工具的条件。
- 存储结构。
- 时间和辅助空间复杂度等。

### 2. 不同条件下排序方法的选择

(1) 若n较小(如 $n \leq 50$ ), 可采用直接插入或直接选择排序。

当记录规模较小时, 直接插入排序较好。否则因为直接选择移动的记录数少于直接插入, 应选直接选择排序为宜。

(2) 若文件初始状态基本有序(指正序), 则应选用直接插入排序、冒泡排序或随机的快速排序为宜。

(3) 若n较大, 则应采用时间复杂度为 $O(nlgn)$ 的排序方法(快速排序、堆排序或归并排序)。

快速排序被认为是目前基于比较的内部排序中最好的方法。当待排序的关键字随机分布时, 快速排序的平均时间最短。

堆排序所需的辅助空间少于快速排序, 并且不会出现快速排序可能出现的最坏情

况。这两种排序都是不稳定的。

若要求排序稳定，则可选用归并排序。但本章介绍的从单个记录起进行两两归并的排序算法并不值得提倡，通常可以将它和直接插入排序结合在一起使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并之。因为直接插入排序是稳定的，所以改进后的归并排序仍是稳定的。

## 13.9 时间复杂度

**面试例题 1：**定义了如下类和有序表关键字序列为 b c d e f g q r s t，则在二分法查找关键字 b 的过程中，先后进行比较的关键字依次是多少？[中国某互联网公司 2009 年 11 月面试题]

- A. f c b
- B. f d b
- C. g c b
- D. g d b

**解析：**二分法查找是指已知有序队列中找出与给定关键字相同的数的具体位置。原理是分别定义三个指针 low、high、mid，分别指向待查元素所在范围的下界和上界及区间的中间位置，即  $mid = (\text{low} + \text{high}) / 2$ ，让关键字与 mid 所指的数比较，若相等则查找成功并返回 mid，若关键字小于 mid 所指的数则 high = mid - 1，否则 low = mid + 1，然后继续循环直到找到或找不到为止。下面代码是二分法的 C++ 实现：

```
#include <stdio.h>
#include <iostream>
using namespace std;

#define MAXSIZE 10

typedef struct{
    int list[MAXSIZE];
    int length;
}List;

int dichotomy_search(List s,int k)
{
    int low,mid,high;
    low=0;
    high=s.length-1;
    mid=(low+high)/2;
    while(high>=low)
    {
        cout << mid << endl;
        if(s.list[mid]>k){ //turn to the
left part
            high=mid-1;
            mid=(low+high)/2;
        }
        else if(s.list[mid]<k){ //turn
to the right part
            low=mid+1;
            mid=(low+high)/2;
        }
        else
            return(mid+1); //The key has
been searched
    }
    return 0; //no such key
}
```

```

    }

int main(int argc, char **argv)
{
    List s;
    int i, k, rst;
    int
a[MAXSIZE]={1, 3, 6, 12, 15, 19, 25, 32, 38, 87
};;
    for(i=0;i<MAXSIZE;i++){
        s.list[i]=a[i];
    }
    s.length=MAXSIZE;
}

```

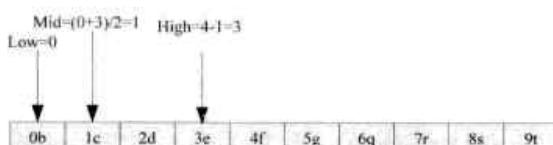
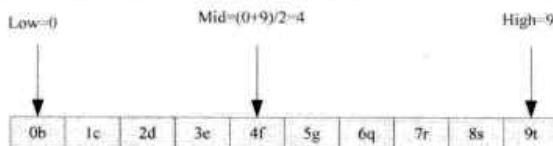
```

printf("Input key number:");
scanf("%d", &k);

rst=dichotomy search(s,k);
if(rst==0){
    printf("Key:%d is not in the
list!\n", k);
}
else{
    printf("The key is in the
list,position is:%d \n",rst);
}
return 0;
}

```

对于本题而言，要比较三个关键字，分别是 f、e、b。具体情况如下图所示。



答案：A

**面试例题2：** Which of the choices below correctly describes the amount of time used by the following code? (下面哪个选项正确地描述了代码运行的调度次数?) [美国著名软件公司 M2009 年 10 月面试题]

```

n=10;
for(i=1; i<n; i++)
    for(j=1; j<n; j+=n/2)
        for(k=1; k<n; k=2*k)
            x = x +1;

```

A  $\Theta(n^3)$

B  $\Theta(n^2 \log n)$

C  $\Theta(n(\log n)^2)$

D  $\Theta(n \log n)$

**解析：**本题考量面试者对时间复杂度的理解。本题涉及如下概念：

### 1) 时间频度

一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为  $T(n)$ 。

### 2) 时间复杂度

在刚才提到的时间频度中， $n$  称为问题的规模，当  $n$  不断变化时，时间频度  $T(n)$  也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。

一般情况下，算法中基本操作重复执行的次数是问题规模  $n$  的某个函数，用  $T(n)$  表示，若有某个辅助函数  $f(n)$ ，使得当  $n$  趋近于无穷大时， $T(n)/f(n)$  的极限值为不等于零的常数，则称  $f(n)$  是  $T(n)$  的同数量级函数。记作  $T(n)=O(f(n))$ ，称  $O(f(n))$  为算法的渐进时间复杂度，简称时间复杂度。

在各种不同算法中，若算法中语句执行次数为一个常数，则时间复杂度为  $O(1)$ ，另外，在时间频度不相同时，时间复杂度有可能相同，如  $T(n)=n^2+3n+4$  与  $T(n)=4n^2+2n+1$ ，它们的频度不同，但时间复杂度相同，都为  $O(n^2)$ 。

按数量级递增排列，常见的时间复杂度有：

常数阶  $O(1)$

对数阶  $O(\log_2 n)$

线性阶  $O(n)$

线性对数阶  $O(n \log_2 n)$

平方阶  $O(n^2)$

立方阶  $O(n^3)$

...

$k$  次方阶  $O(n^k)$

指数阶  $O(2^n)$

随着问题规模  $n$  的不断增大，上述时间复杂度不断增大，而算法的执行效率不断降低。

### 3) 算法的时间复杂度

若要比较不同的算法的时间效率，受限要确定一个度量标准，最直接的办法就是将计算法转化为程序，在计算机上运行，通过计算机内部的计时功能获得精确的时间，然后进行比较。但该方法受计算机的硬件、软件等因素的影响，会掩盖算法本身的优劣，所以一般采用

事先分析估算的算法，即撇开计算机软硬件等因素，只考虑问题的规模（一般用自然数  $n$  表示），认为一个特定的算法的时间复杂度，只取决于问题的规模，或者说它是问题的规模的函数。

为了方便比较，通常的做法是，从算法选取一种对于所研究的问题（或算法模型）来说是基本运算的操作，以其重复执行的次数作为评价算法时间复杂度的标准。该基本操作多数情况下是由算法最深层环内的语句表示的，基本操作的执行次数实际上就是相应语句的执行次数。

一般说来：

$T(n) = O(f(n))$   
 $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

所以要选择时间复杂度量级低的算法。

至于本题，在这里观看代码可知， $x=x+1$ ，是循环最内侧代码，其时间复杂度最高，所以只求这句代码的复杂度即可。从内到外看， $k$  循环从  $1=2^0$  开始每次变成原来的 2 倍，一直到大于  $n-1$ ，所以应该是循环体运行次数是  $\lfloor \log(n) \rfloor$ ，时间复杂度为  $O(\log(n))$ （计算机中  $\log$  默认底数是 2）； $j$  循环从 1 开始每次递增  $n/2$ ，一直到  $n-1$ ，第一次递增之后  $j$  变成  $(n+2)/2$ ，第二次递增  $j$  则是  $n+1$  所以应该是循环了 2 次，但是时间复杂度还是  $O(1)$ ，因为常数次数的时间复杂度都是  $O(1)$  的， $i$  循环从 1 开始，每次增 1 一直到  $n-1$ ，所以循环体运行  $n-1$  次，时间复杂度为  $O(n)$ 。最后相乘得到总的时间复杂度就是  $O(n * 1 * \log(n)) = O(n * \log(n))$ ；这里要强调一下：时间复杂度都不带常数项或者常数系数的，所以不存在所谓  $O(2n)$  这样的时间复杂度。

答案：D

**面试例题 3：**以下哪种节构，平均来讲获取任意一个指定值最快，为什么？[中国某互联网公司 2009 年 11 月面试题]

- A. 二叉排序树 B. 哈希表 C. 栈 D. 队列

**解析：**一般来说，哪个需要的额外空间越多，哪个越快。

哈希表和哈希函数是大学数据节构中的课程，实际开发中我们经常用到 `Hashtable` 这种节构，当遇到键-值对存储，采用 `Hashtable` 比 `ArrayList` 查找的性能高。为什么呢？我们在享受高性能的同时，需要付出高额外空间的代价。那么使用 `Hashtable` 是否就是很好的选择呢？就此疑问，做分析如下：

### 1. 于键-值查找性能高

数据节构描述线性表和树时，记录在节构中的相对位置是随机的，记录和关键字之间不存在明确的关系，因此在查找记录的时候，需要进行一系列的关键字比较，这种查找方式建

立在比较的基础之上，在 Java 中（Array,ArrayList,List）这些集合节构采用了上面的存储方式。比如，现在我们有一个班同学的数据，包括姓名、性别、年龄、学号等。假如数据如下：

姓名	性别	年龄	学号
张三	男	15	1
李四	女	14	2
王五	男	14	3

假如，我们按照姓名来查找，查找函数 `FindBy Name(string name)`：

(1) 查找“张三”：

只需在第一行匹配一次。

(2) 查找“王五”

在第一行匹配，失败；

在第二行匹配，失败；

在第三行匹配，成功。

上面两种情况，分析了最好的情况和最坏的情况，那么平均查找次数应该为  $(1+3)/2=2$  次，即平均查找次数为（记录总数+1）的  $1/2$ 。尽管有一些优化的算法，可以使查找排序效率增高，但是复杂度会保持在  $\log_2^n$  的范围之内。

如何更更快的进行查找呢？我们所期望的效果是一下子就定位到要找记录的位置之上，这时候时间复杂度为 1，查找最快。如果我们事先为每条记录编一个序号，然后让它们按号入位，我们又知道按照什么规则对这些记录进行编号的话，如果我们再次查找某个记录的时候，只需要先通过规则计算出该记录的编号，然后根据编号，在记录的线性队列中，就可以轻易地找到记录了。

注意，上述的描述包含了两个概念，一个是对学生进行编号的规则，在数据节构中，称之为哈希函数，另外一个是按照规则为学生排列的顺序节构，称之为哈希表。

仍以上面的学生为例，假设学号就是规则，老师手上有一个规则表，在排座位的时候也按照这个规则来排序，查找李四，首先该教师会根据规则判断出，李四的编号为 2，就是在座位中的 2 号位置，直接走过去，就可以找到李四了。

流程如下：

从上面的图中，可以看出哈希表可以描述为两个表，一个表用来装记录的位置编号，另一个表用来装记录；此外存在一套规则，用来表述记录与编号之间的联系。这个规则通常是如何制定的呢？

1) 直接定址法

对于整型的数据 `GetHashCode()` 函数返回的就是整型本身，其实就是基于直接定址的方法，比

如有一组0~100的数据，用来表示人的年龄。那么，采用直接定址的方法构成的哈希表为：

0	1	2	3	4	5
0岁	1岁	2岁	3岁	4岁	5岁
.....					

这样的定址方式简单方便，适用于原数据能够用数字表述或者原数据具有鲜明顺序关系的情形。

### 2) 数字分析法：

有这样一组数据，用于表述一些人的出生日期：

年	月	日
75	10	1
75	12	10
75	02	14

分析一下，年和月的第一位数字基本相同，造成冲突的几率非常大，而后面三位差别比较大，所以采用后三位：

### 3) 平方取中法

取关键字平方后的中间几位作为哈希地址。

### 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相同，然后取这几部分的叠加和（取出进位）作为哈希地址，比如有这样的数据：

20144545473

可以：

$$\begin{array}{r}
 5473 \\
 + 4454 \\
 + 201 \\
 = 10128
 \end{array}$$

取出进位1，取0128为哈希地址。

### 5) 取余法

取关键字被某个不大于哈希表表长m的数p除后所得余数为哈希地址。 $H(key)=key \bmod p$  ( $p \leq m$ )。

### 6) 随机数法

选择一个随机函数，取关键字的随机函数值为它的哈希地址，即  $H(key)=\text{random}(key)$ ，其中 random 为随机函数。通常关键字长度不等时采用此法。

总之，哈希函数的规则是通过某种转换关系，使关键字适度的分散到指定大小的顺序结构中。越分散，则以后查找的时间复杂度越小，空间复杂度越高。

## 2. 使用 hash 付出的代价

hash 是一种典型以空间换时间的算法，比如原来一个长度为 100 的数组，对其查找，只需要遍历且匹配相应记录即可，从空间复杂度上来看，假如数组存储的是 Byte 类型数据，那么该数组占用 100Byte 空间。现在我们采用 hash 算法，我们前面说的 hash 必须有一个规则，约束键与存储位置的关系，那么就需要一个固定长度的 hash 表，此时，仍然是 100Byte 的数组，假设我们需要的 100Byte 用来记录键与位置的关系，那么总的空间为 200Byte，而且用于记录规则的表大小会根据规则，大小可能是不定的。

hash 表最突出的问题在于冲突，就是两个键值经过哈希函数计算出来的索引位置很可能相同。

答案：B

**面试例题 4：** Which of the following operation performs NOT faster on an ordered data over a disordered data? (有序队列数据相对于无序队列数据，下列哪种操作并不快？) [中国某杀毒软件公司 2009 年 11 月面试题]

- A. Find the minimum (找出最小值)
- B. Calculate the average value (估算平均值)
- C. Find the median (找出中间值)
- D. Find the one with maximal occurrence (找出最大出现可能性)

**解析：**对于这 4 种情况分别分析如下：

- 对于寻找最小值：有序队列数据的时间复杂度是  $O(1)$ ，无序队列数据的时间复杂度是  $O(n)$ 。
- 对于估算平均值：有序队列数据的时间复杂度是  $O(n)$ ，无序队列数据的时间复杂度是  $O(n)$ 。
- 对于找出中间值：有序队列数据的时间复杂度是  $O(1)$ ，无序队列数据的时间复杂度是  $O(n)(O(n))$  的算法类似于快排)。
- 对于找出最大出现可能性：有序队列数据的时间复杂度是  $O(n)$ ，无序队列数据的时间复杂度是  $O(nlgn)$  (使用平衡查找节构而不是哈希表)。

答案：B

# 第 14 章

## 字 符 串

**基**本上求职者进行笔试时没有不考字符串的。字符串也是一种相对简单的数据结构，容易多次引起面试官反复发问。我曾不止一次在面试时被考官要求当场写出 strcpy 函数的表达方式。事实上，字符串也是一个考验程序员编程规范和编程习惯的重要考点。不要忽视这些细节，因为这些细节会体现你在操作系统、软件工程、边界内存处理等方面的知识掌控能力，也会成为企业是否录用你的参考因素。

### 14.1 整数字符串转化

**面试例题 1：**怎样将整数转化成字符串数，并且不用函数 itoa？

**解析：**整数转化成字符串，可以采用加'0'，再逆序的办法，整数加'0'就会隐性转化成 char 类型的数。

**答案：**程序代码如下：

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

int main(void)
{
    intnum=12345,j=0,i=0;
    char temp[7],str[7];

    //itoa(number, string, 10);

    while(num)
    {
        temp[i]=num%10+'0';
        i++;
        num=num/10;
    }

    temp[i]=0;
    printf(" temp=%s\n", temp);
    i=i-1;
    printf(" temp=%d\n", i);
    //刚刚转化的字符串是逆序的，必须把它反转过来
    while(i>=0)
    {
        str[j]=temp[i];
        j++;
        i--;
    }
    str[j]=0;
    printf(" string=%s\n", str);
    return 0;
}
```

## 扩展知识

如果可以使用 `itoa` 函数的话，则十分简单，答案如下：

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

int main(void)
{
    int number = 12345;
    char string[7];

    itoa(number, string, 10);
    printf("integer = %d string = %c\n", number, string[1]);
    return 0;
}
```

**面试例题 2：**编程实现字符串数转化成整数的办法。[中国某著名 IT 培训企业公司 2005 年面试题]

**解析：**字符串转化成整数，可以采用减'0'再乘 10 累加的办法，字符串减'0'就会隐性转化成 `int` 类型的数。

**答案：**程序代码如下：

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

int main(void)
{
    int num = 12345, j=0, i=0, sum=0;
    char temp[7]={'1','2','3','4','5',
                  '\0'}, str[7];
    while(temp[i])
    {
        sum=sum*10+(temp[i]-'0');
        i++;
    }
    printf(" sum=%d\n", sum);
    return 0;
}
```

## 14.2 字符数组和 `strcpy`

**面试例题 1：**Write a function about string copy, the `strcpy` prototype is "char\* `strcpy( char* strDest, const char* strSrc);`". Here `strDest` is destination string, `strSrc` is source string. (已知 `strcpy` 函数的原型是 `char *strcpy(char *strDest, const char *strSrc);`，其中 `strDest` 是目的字符串，`strSrc` 是源字符串。)

(1) Write the function `strcpy`, don't call C/C++ string library. (不调用 C++/C 的字符串库函数，请编写函数 `strcpy`。)

(2) Here `strcpy` can copy `strSrc` to `strDest`, but why we use `char*` as the return value of `strcpy`? (`strcpy` 能把 `strSrc` 的内容复制到 `strDest`，为什么还要 `char *` 类型的返回值？) [中国台湾某

著名CPU生产公司2005年面试题]

**解析:** 字符串拷贝函数问题。

**答案:** (1) 代码如下:

```
char *strcpy(char *strDest, const char *strSrc);
{
    assert((strDest!=NULL)&&(strSrc!=NULL));
    char *address=strDest;
    while( (*strDest++=*strSrc++)!='\0')
        NULL ;
    return address ;
}
```

(2) 为了实现链式表达式, 返回具体值。

例如:

```
int length = strlen( strcpy( strDest, "hello world" ) );
```

**面试例题2:** 下面的程序会出现何种问题? [美国某著名计算机软件公司面试题]

<pre>#include &lt;iostream&gt; #include &lt;stdio.h&gt; int main(void) {     char s[]="123456789";</pre>	<pre>char d[]="123"; strcpy(d,s); printf("%s,\n%s",d,s); return 0;</pre>
--	--

**解析:** 以上程序输出结果是 123456789,56789。

没经验的程序员一定会在此大跌眼镜的, 源字串竟然被截掉了一部分 (截掉的长度恰是目标字串原来的长度。至于原因, 应该是当初分配的内存地址是连续内存的问题, 原来是 1234\0123456789\0, strcpy 后变成了 123456789\06789\0), 所以在分配空间的时候要给源字符串和目标字符串留足够的空间。

把目标字串定义在前, 源字串定义在后, 虽然可以看到正确的输出结果 123456789, 123456789。但会产生一个运行期错误, 原因估计是越过了目标字串的实际空间, 访问到了不可预知的地址。

微软在这里是写得非常简单的, 代码如下:

```
char * cdecl strcpy(char * dst, const char * src)
{
    char * cp = dst;
    while( *cp++ = *src++ )
        /* Copy src over dst */
    return( dst );
}
```

微软为什么这么写? 它这样安全漏洞太多了, 所以必须预先为目标字串分配足够的空间, 并且使用这个函数的时候得小心翼翼才行。

为了提高性能，减去那些罗嗦的安全检查是必要的。况且程序员在使用时应该知道哪些条件下会发生访问违例，这种做法就是把责任推给了程序员，让他来决定安全与性能的取舍。

**答案：**123456789,56789。

拷贝函数的一个完整的标准写法如下：

```
#include<stdio.h>
#include<malloc.h>
#include<assert.h>
#include<string.h>
void stringcpy(char *to,
               const char *form)
{
    assert(to!=NULL && form!=NULL);
    while(*form]!='\0')
    {
        *to++=*form++;
    }
    *to='\0';
}
```

```
}
int main(void)
{
    char *f;
    char *t;
    f=(char *)malloc(15);
    t=(char *)malloc(15);
    stringcpy(f,"asdfghjkl");
    stringcpy(t,f);
    printf("%s\n",f);
    printf("%s\n",t);
    return 0;
}
```

## 扩展知识（数组大小分配）

在使用数组的时候，总有一个问题困扰着我们：数组应该有多大？

在很多的情况下，你不能确定要使用多大的数组。你可能并不知道该班级的学生的人数，那么你就要把数组定义得足够大。这样，你的程序在运行时就申请了固定大小的、你认为足够大的内存空间。即使你知道该班级的学生数，但是如果因为某种特殊原因人数有增加或者减少，你又必须重新修改程序，扩大数组的存储范围。这种分配固定大小的内存分配方法称为静态内存分配。但是这种内存分配的方法存在比较严重的缺陷，特别是处理某些问题时，在大多数情况下会浪费大量的内存空间；在少数情况下，当你定义的数组不够大时，还可能引起下标越界错误，甚至导致严重后果。

那么有没有其他的方法来解决这样的问题呢？有，那就是动态内存分配。

所谓动态内存分配就是指在程序执行的过程中动态地分配或者回收存储空间的内存分配方法。动态内存分配不像数组等静态内存分配方法那样需要预先分配存储空间，而是由系统根据程序的需要即时分配，且分配的大小就是程序要求的大小。从以上动、静态内存分配比较可以知道动态内存分配相对于静态内存分配的特点：

- 不需要预先分配存储空间。
  - 分配的空间可以根据程序的需要扩大或缩小。
1. 如何实现动态内存分配及其管理

要实现根据程序的需要动态分配存储空间，就必须用到以下几个函数。

### 1) malloc 函数

malloc 函数的原型为：

```
void *malloc (unsigned int size)
```

其作用是在内存的动态存储区中分配一个长度为 size 的连续空间。其参数是一个无符号整型数，返回值是一个指向所分配的连续存储域的起始地址的指针。还有一点必须注意的是，若函数未能成功分配存储空间（如内存不足）就会返回一个 NULL 指针，所以在调用该函数时应该检测返回值是否为 NULL 并执行相应的操作。

下例是一个动态分配的程序：

```
main()
{
    int count,*array;
    /*count 是一个计数器，array 是一个整型指针，也可以理解为指向一个整型数组的首地址*/
    if((array(int *) malloc(10*sizeof(int)))==NULL)
    {
        printf("不能成功分配存储空间。");
        exit(1);
    }
    for (count=0;count < 10;count++) /*给数组赋值*/
        array[count]=count;
    for(count=0;count < 10;count++) /*打印数组元素*/
        printf("%2d",array[count]);
}
```

上例中动态分配了 10 个整型存储区域，然后进行赋值并打印。例中 if((array(int \*) malloc(10\*sizeof(int)))==NULL)语句可以分为以下几步：

- (1) 分配 10 个整型的连续存储空间，并返回一个指向其起始地址的整型指针。
- (2) 把此整型指针地址赋给 array。
- (3) 检测返回值是否为 NULL。

## 2) free 函数

由于内存区域总是有限的，不能无限制地分配下去，而且一个程序要尽量节省资源，所以当所分配的内存区域不用时，就要释放它，以便其他的变量或者程序使用。这时我们就要用到 free 函数。其函数原型是：

```
void free(void *p)
```

作用是释放指针 p 所指向的内存区域。

其参数 p 必须是先前调用 malloc 函数或 calloc 函数（另一个动态分配存储区域的函数）时返回的指针。给 free 函数传递其他的值很可能造成死机或其他灾难性的后果。

注意：这里重要的是指针的值，而不是用来申请动态内存的指针本身。例如：

```
int *p1,*p2;
p1=malloc(10*sizeof(int));
p2=p1;
```

```
.....  
free(p2) /*或者 free(p1)*/
```

malloc 返回值赋给 p1，又把 p1 的值赋给 p2，所以此时 p1、p2 都可作为 free 函数的参数。

数的参数。malloc 函数对存储区域进行分配。free 函数释放已经不用的内存区域。所以有这两个函数就可以实现对内存区域进行动态分配并进行简单的管理了。

**面试例题 3：**编写一个函数，作用是把一个 char 组成的字符串循环右移 n 个。比如原来是“abcdefghi”，如果 n=2，移位后应该是“hiabcdefg”。

函数头是这样的：

```
//pStr 是指向以'\0'结尾的字符串的指针  
//steps 是要求移动的n  
  
void LoopMove ( char * pStr, int steps )  
{  
    //请填充  
}
```

**解析：**这个试题主要考查面试者对标准库函数的熟练程度，在需要的时候引用库函数可以很大程度上简少程序编写的工作量。

最频繁被使用的库函数包括 Strcpy、memcpy、memset。

**答案：**

解答 1：

```
void LoopMove ( char *pStr, int steps )  
{  
    int n = strlen( pStr ) - steps;  
    char tmp[MAX_LEN];  
    strcpy ( tmp, pStr + n );  
    strcpy ( tmp + steps, pStr );  
    *( tmp + strlen ( pStr ) ) = '\0';  
    strcpy( pStr, tmp );  
}
```

解答 2：

```
void LoopMove ( char *pStr, int steps )  
{  
    int n = strlen( pStr ) - steps;  
    char tmp[MAX_LEN];  
    memcpy( tmp, pStr + n, steps );  
    memcpy(pStr + steps, pStr, n );  
    memcpy(pStr, tmp, steps );  
}
```

### 14.3 数组初始化和数组越界

**面试例题 1：**下面关于数组的初始化正确的是哪项？[中国著名网络企业 XL 公司面试题]

- A. char str[2]={"a","b"};
- B. char str[2][3]={"a","b"};
- C. char str[2][3]={{'a','b'},{'e','d'},{'e','f'}};
- D. char str[]={"a","b"};

**解析：**数组初始化问题。

**答案：**B

**面试例题 2：**Find the defects in each of the following programs, and explain why it is incorrect.(找

出下面程序的错误，并解释它为什么是错的。) [中国台湾某著名杀毒软件公司2005年面试题]

```

void test1() {
    char string[10];
    char* str1="0123456789";
    strcpy(string,str1);
    std::cout<<string<<'\n';
}

void test2() {
    char string[10],str1[10];
    for(int i=0;i<10;i++) {
        str1[i]='a';
    }
}

void test3(char* str1) {
    char string[10];
    if(strlen(str1)<=10) {
        strcpy(string,str1);
    }
    std::cout<<string<<'\n';
}

```

**解析：**字符数组和 strcpy 问题。

对于函数 test，这里 string 数组越界。因为字符串长度为 10，还有一个结束符'0'，所以总共有 11 个字符长度。string 数组大小为 10，这里越界了。但是虽然越界但并不报错，整个程序无论编译还是运行都可以正常通过。字符数组并不要求最后一个字符为'0'。是否需要加入'0'，完全由系统决定。

使用 strcpy 函数的时候一定要注意前面上目的数组的大小必须大于后面字符串的大小，否则便是访问越界。

对于函数 test2，这里最大的问题还是 str1 没有结束符，因为 strcpy 的第二个参数应该是一个字符串常量。该函数就是利用第二个参数的结束符来判断是否复制完毕，所以在 for 循环后面应加上 str1[9] = '\0'。

字符数组和字符串的最明显的区别就是字符串会被默认地加上结束符'0'。

对于函数 test3，这里的问题仍是越界问题。strlen 函数得到字符串除结束符外的长度。如果这里是大于等于 10 的话，就很明显是越界了。

**小结：**上面的 3 个找错的函数，主要是考查对字符串和字符数组概念的掌握，以及对 strcpy 函数和 strlen 函数的理解。

字符数组并不要求最后一个字符为'0'。是否需要加入'0'，完全由系统决定。但是字符数组的初始化要求最后一个字符必须为'0'，所以 test2 虽然能够编译通过，但是会出现运行时错误。类似于 char c[5]={'C','h','i','n','a'} 这样的定义是错误的。

**答案：**

可以编译通过的程序如下所示：

```

#include <iostream>

void test1() {
    char string[10];
    char* str1="0123456789";
    strcpy(string,str1);
    std::cout<<string<<'\n';
}

void test2() {
    char string[10],str1[10];
    for(int i=0;i<9;i++) {
        // 错误 1
        str1[i]='a';
    }
}

```

```

    }
    str1[9]='\0';
    strcpy(string,str1);
    std::cout<<string<<'\n';
}

void test3(char* str1) {
    char string[10];
    if(strlen(str1)<=10) {
        strcpy(string,str1);
    }
}

```

```

    std::cout<<string<<'\n';

}

int main()
{
    test1();
    test2();
    char* str="0123456789";
    test3(str);
    return 0;
}

```

**面试例题 3：**本段代码有什么问题，如何修改？[中国台湾著名杀毒软件公司 Q 2007 年 9 月面试题]

```

#include <iostream>
using namespace std;

#define MAX 255

main()
{
    char p[MAX+1];

```

```

    char ch;
/* 将 <= 变成 < */
for (ch=0; ch <=MAX; ch++)
    { p[ch]=ch; cout << ch << " " ; }

cout << ch << " " ;
}

```

**解析：**这是 char 数值问题。char 值范围为 -128~127，由于 char ch; 的执行，程序会在栈中开辟一个大小为 256 的 char 空间，而第 256 个 char 就是我们声明的 ch。

当执行 ch++，而使 ch = 128 时，这会改变第 256 个空间的值，也就是 ch 的值，改变的结果是 ch 重新变成 -128，那么就持续小于 255，继续进行循环，从而陷入死循环。

如果把 char 修改成 unsigned char 还是会有问题，因为 ch <=MAX; 这个等号的存在，所以在等于 255 的时候一样执行循环，然后 255 加 1，一样溢出，ch 的值变为 0，然后不停地循环。唯一的解决方法是将 “<=” 变成 “<”，循环结束后单独给数组最后一个元素 p[255] 赋值。

**答案：**程序陷入死循环。

正确代码如下：

```

#include <iostream>
using namespace std;

#define MAX 255

main()
{
    char p[MAX+1];
    // 修改 ch 为 unsigned char
    unsigned char ch;

```

```

/* 将 <= 变成 < */
for (ch=0; ch <MAX; ch++)
{
    p[ch]=ch; cout << ch << " " ;
}
/* 在此添加一句：给数组最后一个元素 p[255] 赋值 */
p[ch] = ch;
cout << ch << " " ;
}

```

## 14.4 数字流和数组声明

**面试例题 1：**Which is not the standard I/O channel? (下面哪一个不是标准输入 / 输出通道？)

[中国某著名综合软件公司 2005 年面试题]

- A. std::cin      B. std::cout      C. std::cerr      D. stream

**解析:** I/O stream 问题。头文件 iostream 中含有 cin、cout、cerr 几个对象，对于标准输入流、标准输出流和标准错误流。

**答案:** D

**面试例题 2:** Which definition is correct? (下面哪个数组的声明是正确的?) [中国台湾某著名杀毒软件公司 2005 年 9 月面试题]

- |                     |                        |
|---------------------|------------------------|
| A. int a[];         | B. int n=10,a[n];      |
| C. int a[10+1]={0}; | D. int a[3]={1,2,3,4}; |

**解析:** 数组定义问题。

int a[] 是错误的，不允许建立空数组。

int n=10,a[n]; 是可以的。

int a[10+1]={0}; 也是允许的。

int a[3]={1,2,3,4}; 会造成越界问题，因此不允许。

**答案:** B, C。

## 14.5 字符串其他问题

**面试例题 1:** 求一个字符串中连续出现次数最多的子串，请给出分析和代码。[中国著名 IT 培训企业 2008 年 3 月面试题]

**解析:** 这里首先要搞清楚子串的概念，1 个字符当然也算字串，注意看题目，是求连续出现次数最多的子串。如果字符串是 abcbcbcabc，这个连续出现次数最多的子串是 bc，连续出现次数为 3 次。如果类似于 abcccabc，则连续出现次数最多的子串为 c，次数也是 3 次。这个题目可以首先逐个子串扫描来记录每个子串出现的次数。比如：abc 这个字串，对应子串为 a/b/c/ab/bc/abc，各出现过一次，然后再逐渐缩小字符子串来得出正确的结果。

**答案:**

完整代码如下：

```
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;
pair<int, string> fun(const string& str)
{
```

```

vector<string> substrs;
int maxcount = 1, count = 1;
string substr;
int i, len = str.length();
for (i = 0; i < len; ++i)
    substrs.push_back(str.substr(i, len - i));

for (i = 0; i < len; ++i)
{
    for (int j = i + 1; j < len; ++j)
    {
        count = 1;
        if (substrs[i].substr(0, j - i) ==
            substrs[j].substr(0, j - i))
        {
            ++count;
            for (int k = j + (j - i); k < len; k += j - i)
            {
                if (substrs[i].substr(0, j - i) ==
                    substrs[k])
                    substrs[0, j - i])++count;
                else
                    break;
            }
        }
    }
}

```

```

if (count > maxcount)
{
    maxcount = count;
    substr=substrs[i].substr(0, j - i);
}

return make_pair(maxcount, substr);
}

int main()
{
    string str;
    pair<int, string> rs;
    while(cin>>str)
    {
        rs = fun(str);
        cout<<rs.second<<':'<<rs.first<<'\n';
    }

    return 0;
}

```

**面试例题 2：**编程：输入一行字符串，找出其中出现的相同且长度最长的字符串，输出它及其首字符的位置。例如“yyabcdabjcabcdeg”，输出结果应该为 abc 和 3。[中国著名 IT 培训企业 2008 年 3 月面试题]

### 解析：

可以将字符串 yyabcdabjcabcdeg 分解成：

```

yyabcdabjcabcdeg
yabcdabjcabcdeg
abcdabjcabcdeg
bcdabjcabcdeg
cdabjcabcdeg
.....
ceg
eg
g

```

对这几个字符串排序，然后比较相邻字符串的前驱就可以了，很容易求出最长的公共前驱。

**答案：**完整代码如下：

```
#include <iostream>
#include<string>
using namespace std;
int main()
{
    string str,tep;
    cout<<"请输入字符串"=>str;
    for(int i=str.length()-1;i>1;i--)
    {
        for(int j=0;j<str.length();j++)
        {
            if(j+i<=str.length())
            {
                size_t t=0;
                size_t num=0;
                tep=str.substr(j,i); //从大到小取
            }
        }
    }
}
```

```
t=str.find(tep); //正序查找
num=str.rfind(tep); //逆序查找
if(t!=num) //如果两次查找位置不一致
//说明存在重复子串
{
    cout<<tep<<" "<<t+1<<endl; //输出子串及位置
    return 0;
}
}

}

return 0;
}
```

**面试例题3：**Please implement the function strstr() (Find a substring, returns a pointer to the first occurrence of strCharSet in string), DO NOT use any C run-time functions. const char\* strstr(const char\* string, const char\* strCharSet); (请写一个函数来模拟C++中的strstr()函数：该函数的返回值是主串中字符子串的位置以后的所有字符。请不要使用任何C程序已有的函数来完成。)

[中国台湾某著名杀毒软件公司 2005 年面试题]

**解析：**string字符串问题。做一个程序模拟C++中的strstr()函数。strstr()函数是把主串中子串及以后的字符全部返回。比如主串是“12345678”，子串是“234”，那么函数的返回值就是“2345678”。

**答案：**正确程序如下：

```
#include <iostream>
using namespace std;
const char* strsr1(const char* string,
const char* strCharSet)
{
    for(int i=0;string[i]!='\0';i++)
    {
        int j=0;
        int temp=i;
        if(string[i]==strCharSet[j])
        {

while(string[i]==strCharSet[j++])
        {
            if((strCharSet[j]=='\0'))
                return &string[i-j];
        }
        i=temp;
    }
}
```

```

    }

    return NULL;
}

int main()
{
    char* string="12345554555123";
    cout<<string<<endl;
    char strCharSet[10]={};
    cin>>strCharSet;
    cout<<strsr1(string,strCharSet)
<<endl;
    //char*string2=strstr("123455545
//55123","234");
    // cout<<string2<<endl;
    return 0;
}
```

**面试例题 4：**将一句话里的单词进行倒置，标点符号不倒换。比如一句话“i come from tianjin.”倒换后变成“tianjin. from come i”。

**解析：**解决该问题可以分为两步：第一步全盘置换将该句变成“.nijnait morf emoc i”，第二步进行部分翻转，如果不是空格，则开始翻转单词。

### 答案：

具体代码如下：

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

int main(void)
{
    intnum=-12345,j=0,i=0,flag=0,begin,
end;
    charstr[]="icomfromtianjin.",temp;
    j=strlen(str)-1;

    printf(" string = %s\n", str);
    //第一步是进行全盘翻转，将单词变成
    // ".nijnait morf emoc i"
    while(j>i)
    {
        //str[j]=temp[i];
        temp=str[i];
        str[i]=str[j];
        str[j]=temp;
        j--;
        i++;
    }
    printf(" string = %s\n", str);
    i=0;
    //第二步进行部分翻转，如果不是空格 则开始翻
//转单词
    while(str[i])
    {
        //str[j]=temp[i];
        if(str[i]!=' ')
        {
            begin = i;
            while(str[i]&&str[i]!=' ')
                {i++;}
            i=i-1;
            end=i;
        }
        while(end>begin)
        {
            //str[j]=temp[i];
            temp=str[begin];
            str[begin]=str[end];
            str[end]=temp;
            end--;
            begin++;
        }
        i++;
    }
    printf(" string = %s\n", str);
    return 0;
}
```

**面试例题 5：**Consider a function which, for a given whole number  $n$ , returns the number of ones required when writing out all numbers between 0 and  $n$ . For example,  $f(1)=1$ ,  $f(13)=6$ . Notice that  $f(1)=1$ . What is the next largest  $n$  such that  $f(n)=n$ ?  $n < 4\,000\,000\,000$ .

e.g.  $f(13)=6$

because the number of “1” in 1,2,3,4,5,6,7,8,9,10,11,12,13 is 6. (1, 11, 12, 13)

(现在要我们写一个函数，计算 4 000 000 000 以内的最大的那个  $f(n)=n$  的值，函数  $f$  的功能是统计所有 0 到  $n$  之间所有含有数字 1 的数字和。

比如:  $f(13)=6$

因为“1”在“1,2,3,4,5,6,7,8,9,10,11,12,13”中的总数是 6 (1, 11, 12, 13)) [美国著名搜索引擎公司 G 2008 年 4 月面试题]

**解析:** 字符串数字统计问题。

**答案:** 完整代码如下:

```
#include <windows.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
int f(int n);
int count1(int n);
int cal(unsigned int number,int nwei,int count1,unsigned int ncount);

int gTable[10];
const unsigned int gMAX = 40000000000L;

int main(int argc, char* argv[])
{
    int i;
    unsigned int n=1;
    unsigned int ncount = 0;
    int nwei = 0;
    int ncount1;

    /*if(argc>1)
    {
        n = atoi(argv[1]);
        ncount = f(n);
        printf("f(%d) = %d\n",n,ncount);
    }*/

    int beginTime=GetTickCount();
    int endTime;
    //init gTable
    for(i=0;i <10;++i)
    {
        n *= 10;
        gTable[i] = f(n-1);
    }

    n=0;
    nwei = 0;
    ncount1 = 0;
    while(n <gMAX)
    {
        unsigned int temp;

        temp = 1;

        ncount =cal(n,nwei,ncount1,ncount);
        for(i=0;i <nwei;++i)
            temp *= 10;
        n += temp;
    }
}
```

```
if( (n/temp)/10 == 1)
    ++nwei;
ncount1 = count1(n);
}

endTime=GetTickCount();
endTime-=beginTime;

printf("time: %d ms\n",endTime);
return 0;
}

int f(int n)
{
    int ret = 0;
    int ntemp=n;
    int ntemp2=1;
    int i=1;
    while(ntemp)
    {
        ret += (((ntemp-1)/10)+1) * i;
        if( (ntemp%10) == 1 )
        {
            ret -= i;
            ret += ntemp2;
        }
        ntemp = ntemp/10;
        i*=10;
        ntemp2 = n%i+1;
    }
    return ret;
}

int count1(int n)
{
    int count = 0;
    while(n)
    {
        if( (n%10) == 1)
            ++count;
        n /= 10;
    }
    return count;
}

int cal(unsigned int number,int nwei,int count1,unsigned int ncount)
{
}
```

```

int i, n=1;
unsigned int maxcount;
if(nwei==0)
{
    ncount += count1;
    if(number == ncount)
    {
        printf("f(%d)=%d \n", number, number);
    }
    return ncount;
}
for(i=0;i <nwei;++i)
    n *= 10;
maxcount = ncount + gTable[nwei-1];
maxcount += count1*n;
if(ncount > (number + (n-1)))
{
    return maxcount;
}
    }
    if(maxcount < number)
    {
        return maxcount;
    }
    n /= 10;
    for(i=0;i <10;++i)
    {
        if(i==1)
            ncount=cal(number+i*n,nwei-1,count1+1,ncount);
        else
            ncount=cal(number+i*n,nwei-1,count1,ncount);
    }
    return ncount;
}

```

## 14.6 字符子串问题

**面试例题：**转换字符串格式为原来字符串里的字符+该字符连续出现的个数，例如字符串 1233422222，转化为 1121324125（1 出现 1 次，2 出现 1 次，3 出现 2 次……）。

怎么实现比较简便？[美国著名搜索引擎公司 G 2007 年面试题]

**解析：**可以通过 sprintf 语句来实现算法。

sprintf 跟 printf 在用法上几乎一样，只是打印的目的地不同而已，前者打印到字符串中，后者则直接在命令行上输出。

### 1) 打印字符串

sprintf 最常见的应用之一莫过于把整数打印到字符串中，所以， sprintf 在大多数场合可以替代 itoa。如：

```
//把整数123 打印成一个字符串保存在s 中
sprintf(s, "%d", 123); //产生"123"
```

可以指定宽度，不足的左边补空格：

```
 sprintf(s, "%8d%8d", 123, 4567); //产生: " 123 4567"
```

当然也可以左对齐：

```
 sprintf(s, "%-8d%-8d", 123, 4567); //产生: "123 4567"
```

也可以按照十六进制打印：

```
 sprintf(s, "%8x", 4567); //小写十六进制，宽度占 8 个位置，右对齐
 sprintf(s, "%-8X", 4568); //大写十六进制，宽度占 8 个位置，左对齐
```

## 2) 连接字符串

`sprintf` 的格式控制串中既然可以插入各种东西，并最终把它们“连成一串”，自然也就能够连接字符串，从而在许多场合可以替代 `strcat`。`sprintf` 能够一次连接多个字符串，可以同时在它们中间插入别的内容，非常灵活。比如：

```
char* who = "I";
char* whom = "Duantao";
sprintf(s, "%s love %s.", who, whom); //产生: "I love Duantao."
```

**答案：**程序源代码如下：

```
# include <iostream>
# include <string>
using namespace std;
int main()
{
    cout << "Enter the numbers " << endl;
    string str;
    char reschar[50];
    reschar[0] = '\0';
    getline(cin, str);
    int len = str.length();
    int count = 1;
    int k;
    for (k = 0; k <= len - 1; k++)
    {
        if (str[k + 1] == str[k])
        {
            count++;
        }
        else
        {
            sprintf(reschar + strlen(reschar), "%c%d ", str[k], count);
            count = 1;
        }
    }
    if (str[k] == str[k - 1])
        count++;
    else
        count = 1;
    sprintf(reschar + strlen(reschar), "%c%d ", str[k], count);
    cout << reschar << "gg" << endl;
    cout << endl;
    return 0;
}
```