

## 二、线性表

### 1 线性表的定义和基本操作

#### 1.1 线性表的定义

线性表是具有相同数据类型的 $n$  ( $n \geq 0$ ) 个数据元素的有限序列，其中 $n$ 为表长，当 $n=0$ 时线性表是一个空表。

若用 $L$ 命名线性表，则其一般表示为  $L = (a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$ ，式中 $a_1$ 是唯一的“第一个”数据元素，又称表头元素；

$a_n$ 是唯一一个“最后一个”数据元素，又称表尾元素。

除第一个元素外，每个数据元素有且仅有一个直接前驱；除最后一个元素外，每个数据元素有且仅有一个直接后继。

#### 1.2 线性表的特点

- 表中元素的个数有限。
- 表中元素的都具有逻辑上的顺序性，表中元素有其先后次序。
- 表中元素都是数据元素，每个元素都是单个元素。
- 表中元素的数据类型都相同，这意味着每个元素占有相同大小的存储空间。
- 表中元素具有抽象性，即仅讨论元素间的逻辑关系，而不考虑元素究竟表示什么内容。

#### 1.3 线性表的抽象数据类型ADT

```
1 ADT List{
2   数据对象:  $D = \{a_i | a_i \in \text{ElemSet}, i = 1, 2, \dots, n, n \geq 0\}$ 
3   数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i = 2, \dots, n \}$ 
4   基本操作:
5   InitList(&L)
6       操作结果: 构造一个空的线性表L。
7   DestroyList(&L)
8       初始条件: 线性表L已存在
9       操作结果: 销毁线性表L
10  ClearList(&L)
11      初始条件: 线性表L已存在
12      操作结果: 将L重置为空表
13  ListEmpty(L)
14      初始条件: 线性表L已存在
15      操作结果: 若L为空表，则返回true，否则返回false
16  ListLength(L)
17      初始条件: 线性表L已存在
18      操作结果: 返回L中数据元素个数
19  GetElem(L, i, &e)
20      初始条件: 线性表L已存在,  $1 \leq i \leq \text{ListLength}(L)$ 
21      操作结果: 用e返回L中第i个数据元素的值
22  LocateElem(L, e, compare())
23      初始条件: 线性表L已存在
```

```

24      操作结果：返回L中第一个值与e相同的数据元素在L中的位置。若这样的数据元素不存在，则返回
      值为0。
25      PriorElem(L,cur_e,&pre_e)
26      初始条件：线性表L已存在
27      操作结果：若cur_e是L的数据元素，且不是第一个，则用pre_e返回它的前驱，否则操作失败，
      pre_e无定义
28      NextElem(L,cur_e,&next_e)
29      初始条件：线性表L已存在
30      操作结果：若cur_e是L的数据元素，且不是最后一个，则用next_e返回它的后继，否则操作失
      败，next_e无定义
31      ListInsert(&L,i,e)
32      初始条件：线性表已存在， $1 \leq i \leq \text{ListLength}(L)+1$ 
33      操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1
34      ListDelete(&L,i,&e)
35      初始条件：线性表存在且非空， $1 \leq i \leq \text{ListLength}(L)$ 
36      操作结果：删除L的第i个数据元素，并用e返回其值，L的长度减1
37      TraversalList(L)
38      初始条件：线性表已存在
39      操作结果：依次对线性表L进行遍历，在遍历过程中对L的每个结点访问一次
40  }ADT List

```

## 1.4 线性表基本操作

```

1  InitList(&L): 初始化表。构造一个空表L，分配内存空间。
2  DestroyList(&L): 销毁操作。销毁线性表，并释放线性表L所占用的内存空间。
3  ListInsert(&L,i,e): 插入操作。在表L中第i个位置上插入指定元素e。
4  ListDelete(&L,i,&e): 删除操作。删除表L中第i个位置的元素，并用e返回删除元素的值。
5  LocateElem(L,e): 按值查找操作。在表L中查找具有给定关键字值的元素。
6  GetElem(L,i): 按位查找操作。获取表L中第i个位置的元素的值。
7  Length(L): 求表长。返回线性表L的长度，即L中数据元素的个数。
8  PrintList(L): 输出操作。按前后顺序输出线性表L的所有元素的值。
9  Empty(L): 判空操作。若L为空表，则返回true，否则返回false。

```

## 2 顺序表

### 2.1 顺序表的定义

**顺序表的存储结构：**顺序存储，即逻辑上相邻的数据元素在物理上也相邻

**顺序表的特点：**

1. **随机存取**，即可以在 $O(1)$ 时间内找到第i个元素
2. 存储密度高，每个节点只存储数据元素
3. 拓展容量不方便（即便采用动态分配的方式实现，拓展长度的时间复杂度也比较高）
4. 插入、删除操作不方便，需要移动大量元素

**优点：**可随机存取，存储密度高

**缺点：**需要大片连续的空间，改变容量不方便

#### 2.1.1 顺序表的实现方式

**顺序表有两种实现方式：**

1. 静态分配
  - 使用"静态数组"实现，大小一旦确定就无法改变
2. 动态分配

- 使用"动态数组"实现
- 顺序表满时，可以再用new关键字重新申请更大的内存空间来拓展顺序表的最大长度
- 需要将数据元素复制到新的存储空间，并用delete关键字释放原来的空间

```

1 //静态分配的顺序表（大小一旦确定就无法改变）
2 #define MAXSIZE 10//线性表的最大长度
3 typedef struct{
4     ElemType data[MAXSIZE];
5     int length;
6 }SqlList;
7 //动态分配的顺序表（使用动态数组实现）
8 #define INITSIZE 10//顺序表的初始长度
9 typedef struct{
10     ElemType *data;//指示动态分配数组的指针
11     int MaxSize;//顺序表的最大容量
12     int length;//顺序表当前的长度
13 }SeqList;

```

## 2.1.2 初始化顺序表 InitList

### 1) 静态分配顺序表

```

1 //初始化顺序表（固定长度，不设置默认值）
2 void InitList(SqlList &L){
3     L.length = 0;
4 }
5 //初始化顺序表（固定长度，设置默认值）
6 void InitList(SqlList &L){
7     for(int i = 0; i < L.MaxSize; i++){
8         L.data[i] = 0;
9     }
10    L.length = 0;
11 }

```

### 2) 动态分配顺序表的初始化

```

1 //初始化顺序表
2 void InitList(SeqList &L){
3     L.data = new ElemType[INITSIZE]; //申请初始空间
4     if(!L.data) //如果申请失败，结束程序
5         exit(OVERFLOW);
6     L.MaxSize = INITSIZE;
7     L.length = 0;
8 }

```

### 3) 动态分配长度顺序表增加表长

```

1 void IncreaseSize(SeqList &L,int len){
2     ElemType *p = L.data;
3     L.data = new ElemType[L.MaxSize+len]; //重新申请更大的空间
4     for(int i = 0; i<L.length;i++){
5         L.data[i] = p[i]; //将数据复制到新的空间
6     }
7     L.MaxSize += len; //顺序表长度增加 len
8     delete(p); //释放原来的内存空间
9 }

```

## 2.2 顺序表的插入和删除

### 2.2.1 顺序表的插入 ListInsert

```

1 bool ListInsert(SeqList &L,int i,Elemtype e){
2     if(i<1 || i>L.length+1) //判断i的范围是否有效
3         return false;
4     if(L.length>=L.MaxSize) //如果内存空间不够，重新申请空间
5         IncreaseSize(L,10);
6     for(int j=L.length;j>=i;j--) //将第i个及之后的元素依次后移
7         L.data[j] = L.data[j-1];
8     L.data[i-1] = e; //在第i个位置插入元素e
9     L.length++; //顺序表长度加1
10    return true;
11 }

```

**时间复杂度：**

最好情况：新元素插到表尾，不需要移动元素，循环0次，最好时间复杂度=O(1)

最坏情况：新元素插到表头，移动n个元素，循环n次，最坏时间复杂度=O(n)

平均情况：插入每个位置的概率相等，为 $p=\frac{1}{1+n}$ ，平均循环次数= $\frac{0+1+2+\dots+n}{1+n} = \frac{n}{2}$ ，平均时间复杂度=O(n)

### 2.2.2 顺序表的删除操作 ListDelete

```

1 bool ListDelete(SeqList &L,int i,Elemtype &e){
2     if(i<1 || i>L.length) //判断i的范围是否合格
3         return false;
4     e = L.data[i-1]; //返回位置i元素
5     for(int j = i;j<L.length;j++) //第i个及之后的元素以此前移
6         L.data[j-1] = L.data[j];
7     L.length--; //顺序表长度减1
8     return true;
9 }

```

**时间复杂度：**

最好情况：删除表尾元素，不需要移动其他元素，循环0次，最好时间复杂度=O(1)

最坏情况：删除表头元素，移动后续n-1个元素，循环n-1次，最坏时间复杂度=O(n)

平均情况：删除每个元素的概率相等，为 $p=\frac{1}{n}$ ，平均循环次数= $\frac{0+1+\dots+(n-1)}{n} = \frac{n-1}{2}$ ，平均时间复杂度=O(n)

## 2.3 顺序表的查找

### 2.3.1 按位查找(GetElem)

```
1 //获取表L中第i个位置的元素的值
2 //通过数组下标直接获取
3 ElemType getElem(SeqList L,int i){
4
5     return L.data[i-1];
6 }
```

时间复杂度= $O(1)$

### 2.3.2 按值查找(LocateElem)

```
1 //在顺序表L中查找第一个元素值等于e的元素，并返回其位序
2 //从第一个元素一次往后检索（从头遍历）
3 int LocateElem(SeqList L,ElemType e){
4     for(int i = 0;i<L.length;i++)
5         if(L.data[i]==e)
6             return i+1;//数组下标为i的元素值等于e，返回其位序i+1
7     return 0;//退出循环，说明查找失败
8 }
```

时间复杂度：

最好情况：目标元素在表头，循环1次，最好时间复杂度= $O(1)$

最坏情况：目标元素在表尾，循环n次，最坏时间复杂度= $O(n)$

平均情况：目标元素出现的概率相同，为 $p=\frac{1}{n}$ ，平均循环次数= $\frac{1+2+\dots+n}{n} = \frac{1+n}{2}$ ，平均时间复杂度= $O(n)$

## 2.4 顺序表基本操作全部实现

```
1 #define OK 1
2 #define ERROR 0
3 #define OVERFLOW -2
4 #define INITSIZE 10
5 typedef int Status;
6 typedef int ElemType;
7 //结构体
8 typedef struct {
9     ElemType *data;
10     int MaxSize;
11     int length;
12 } SeqList;
13 //初始化顺序表
14 Status InitList(SeqList &L) {
15     L.data = new ElemType[INITSIZE];
16     if (!L.data)
17         exit(OVERFLOW);
18     L.MaxSize = INITSIZE;
19     L.length = 0;
20     return OK;
21 }
```

```

22 //销毁顺序表
23 Status DestroyList(SeqList &L) {
24     delete (L.data);
25     L.MaxSize = 0;
26     L.length = 0;
27     return OK;
28 }
29 //增加表长
30 Status IncreaseList(SeqList &L, int len) {
31     int *p = L.data;
32     L.data = new ElemType[L.MaxSize + len];
33     if (!L.data)
34         exit(OVERFLOW);
35     for (int i = 0; i < L.length; i++)
36         L.data[i] = p[i];
37     L.MaxSize += len;
38     return OK;
39 }
40 //判空
41 bool ListEmpty(SeqList L) {
42     if (L.length == 0)
43         return true;
44     else
45         return false;
46 }
47 //清空顺序表
48 Status ClearList(SeqList &L) {
49     L.length = 0;
50     return OK;
51 }
52 //按位插入
53 Status ListInsert(SeqList &L, int i, ElemType e) {
54     if (i < 1 || i > L.length + 1)
55         return ERROR;
56     if (L.length == L.MaxSize)
57         return ERROR;
58     for (int j = L.length; j >= i; j--)
59         L.data[j] = L.data[j - 1];
60     L.data[i - 1] = e;
61     L.length++;
62     return OK;
63 }
64 //按位删除
65 Status ListDelete(SeqList &L, int i, ElemType &e) {
66     if (i < 1 || i > L.length)
67         return 0;
68     e = L.data[i - 1];
69     for (int j = i; j < L.length; j++)
70         L.data[j - 1] = L.data[j];
71     L.length--;
72     return OK;
73 }
74 //按位查找
75 ElemType GetElem(SeqList L, int i) {
76     if (i < 1 || i > L.length)
77         exit(OVERFLOW);
78     return L.data[i - 1];
79 }

```

```

80 //按值查找，返回其位序
81 int LocateElem(SeqList L, ElemType e) {
82     for (int i = 0; i < L.length; i++)
83         if (L.data[i] == e)
84             return i;
85     return 0;
86 }

```

## 2 单链表

### 2.1 单链表的定义

**单链表**：线性表的链式存储，它是指通过一组任意的存储单元来存储线性表中的数据元素。

**优点**：不要求大片连续空间，改变容量方便

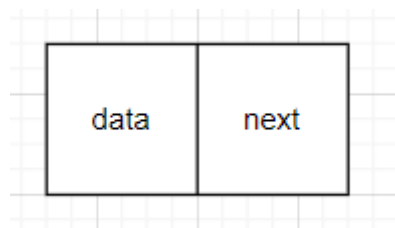
**缺点**：不可随机存取，要耗费一定空间存放指针

**结点**：

```

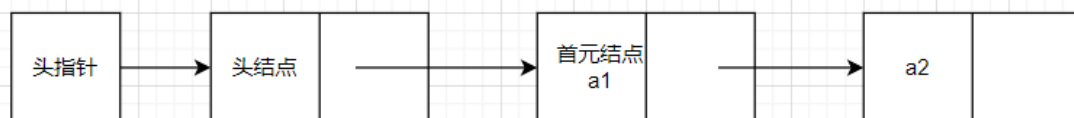
1 typedef struct LNode {
2     ElemType data; //数据域,存放数据元素
3     struct LNode *next; //指针域,指向下一个结点
4 } LNode, *LinkList; //LNode表示结点, LinkList表示链表

```



**定义辨析**：

1. 头指针：指向链表中第一个结点的指针
2. 头结点：在首元结点之前的一个结点，不存放数据，其指针域指向首元结点
3. 首元结点：链表中存储第一个数据元素的结点



#### 2.1.1 不带头结点的单链表

**初始化单链表**：

```

1 //初始化单链表
2 bool InitList(LinkList &L) {
3     L = nullptr; //表空，暂时没有任何结点，防止头指针的地址原来有脏数据
4     return true;
5 }

```

**判断表空**：

```

1 //判断单链表是否为空
2 bool Empty(LinkList L) {
3     if (!L)
4         return true;
5     else
6         return false;
7 }

```

## 2.1.2 带头结点的单链表

```

1 //初始化单链表
2 bool InitList(LinkList &L) {
3     L = new LNode;//申请头结点
4     if (!L)//空间申请失败
5         return false;
6     L->next = nullptr;
7     return true;
8 }

```

判断表空：

```

1 //判断单链表是否为空
2 bool Empty(LinkList L) {
3     return (L->next == nullptr);
4 }

```

## 2.2 单链表的插入

### 2.2.1 单链表的按位插入

带头结点：

```

1 //按位序插入
2 bool ListInsert(LinkList &L, int i, ElemType e) {
3     if (i < 1)
4         return false;
5     LNode *p = L;//指向头结点，头结点是第0个结点
6     int j = 0;//计数器,p指向的第几个结点
7     while (p && j < i - 1) { //循环找到第i-1个结点或到表尾
8         p = p->next;
9         ++j;
10    }
11    if (!p)//p指针为空，也就是i值不合法，i>单链表长度
12        return false;
13    LNode *s = new LNode;
14    s->data = e;
15    s->next = p->next;
16    p->next = s;//s插到p之后
17    return true;
18 }

```

时间复杂度：O(n)

不带头结点：



```

1 //按位插入
2 bool ListInsert(LinkList &L, int i, ElemType e) {
3     if (i < 1)
4         return false;
5     if (i == 1) { //插入第1个结点
6         LNode *s = new LNode;
7         s->data = e;
8         s->next = nullptr;
9         L = s;
10        return true;
11    }
12    //插入第2个及之后的结点
13    int j = 1;
14    LNode *p = L;
15    while (p && j < i - 1) { //i>n
16        p = p->next;
17        ++j;
18    }
19    if (!p)
20        return false;
21    LNode *s = new LNode;
22    s->data = e;
23    s->next = p->next;
24    p->next = s;
25    return true;
26 }

```

### 2.2.2 指定结点后插入结点

方法:

1. 判断给定结点p不为空
2. 创建新结点s, 并将给定值e赋给结点s
3. s结点插入到结点p之后

```

1 //指定结点后插入结点
2 bool InsertNextNode(LNode *p, ElemType e) {
3     if (p == nullptr)
4         return false;
5     LNode *s = new LNode;
6     if (!s)
7         return false;
8     s->data = e;
9     s->next = p->next;
10    p->next = s;
11    return true;
12 }

```

时间复杂度:  $O(1)$

### 2.2.3 指定结点前插入结点

方法:

1. 判断给定结点p不为空
2. 创建新结点s, 并将结点p的数据元素复制到结点s
3. 将结点s插入到结点p之前

#### 4. 将要插入的元素e赋给结点p

```
1 //指定结点前插入结点
2 bool InsertPriorNode(LNode *p, ElemType e) {
3     if (p == nullptr)
4         return false;
5     LNode *s = new LNode;
6     if (!s)
7         return false;
8     s->data = p->data;
9     s->next = p->next;
10    p->data = e;
11    p->next = s;
12    return true;
13 }
```

时间复杂度:  $O(1)$

## 2.3 单链表的删除

### 2.3.1 按位序删除

带头结点:

方法:

1. 扫描到第*i*-1个结点
2. 保存第*i*个结点
3. 删除第*i*个结点
4. 释放第*i*个结点内存空间

```
1 //按位序删除
2 bool ListDelete(LinkList &L, int i, ElemType &e) {
3     if (i < 1)
4         return false;
5     LNode *p = L; //p指向头结点, 也就是第0个结点
6     int j = 0; //计数器, p指向第几个结点
7     while (p && j < i - 1) { //扫描到第i-1个结点, 或者扫描到表尾
8         p = p->next;
9         ++j;
10    }
11    if (!p) //i值不合格, i>表长, 扫描全表不存在p
12        return false;
13    LNode *q = p->next; //保存要删除的结点
14    e = q->data; //e存放要删除结点的元素值
15    p->next = q->next; //删除结点
16    delete (q); //释放删除结点的内存空间
17    return true;
18 }
```

时间复杂度:

最好时间复杂度 =  $O(1)$

最坏、平均时间复杂度 =  $O(n)$

按位序删除操作, 带头结点和不带头结点的区别就在于带头结点*j*=0, 不带头结点时*j*=1。

### 2.3.2 指定结点的删除

```
1 //指定结点的删除(给定结点不为表尾结点的情况)
2 bool DeleteNode(LNode *p) {
3     if (!p)
4         return false;
5     LNode *q = p->next;
6     p->data = q->data;
7     p->next = q->next;
8     delete (q);
9 }
```

## 2.4 单链表的查找

### 2.4.1 按位查找

```
1 //按位查找，返回第i个结点
2 LNode *GetElem(LinkList L, int i) {
3     if (i < 0)
4         return nullptr;
5     LNode *p = L;
6     int j = 0;
7     while (p && j < i) {
8         p = p->next;
9         ++j;
10    }
11    return p;
12 }
```

按位查找操作，带头结点和不带头结点的区别就在于带头结点 $j=0$ ，不带头结点时 $j=1$ 。

### 2.4.2 按值查找

带头结点：

```
1 //按值查找，在表中查找具有给定关键字值的元素
2 LNode *LocateElem(LinkList L, ElemType e) {
3     LNode *p = L->next; //从首元结点开始查找
4     while (p && p->data != e) {
5         p = p->next;
6     }
7     return p; //如果找到了就返回结点指针，没找到就返回空指针
8 }
```

时间复杂度= $O(n)$

不带头结点：

```

1 //按值查找，在表中查找具有给定关键字值的元素
2 LNode *LocateElem(LinkList L, ElemType e) {
3     LNode *p = L;
4     while (p && p->data != e) {
5         p = p->next;
6     }
7     return p;
8 }

```

## 2.5 求表长

```

1 //求表长
2 int Length(LinkList L) {
3     int len = 0;
4     LNode *p = L;
5     while (p->next) {
6         p = p->next;
7         len++;
8     }
9     return len;
10 }

```

时间复杂度= $O(n)$

求表长操作，带头结点和不带头结点的区别就在于带头结点 $j=0$ ，不带头结点时 $j=1$ 。

## 2.6 单链表的建立

### 2.6.1 尾插法

步骤：

1. 初始化单链表
2. 保存表尾结点
3. while 循环插入结点

#### 1) 带头结点

```

1 //尾插法建立单链表
2 LinkList list_tail_insert(LinkList &L) {
3     ElemType x;
4     L = new LNode;
5     LNode *s, *r = L; //r为表尾结点
6     cin >> x; //输入第一个结点的值
7     while (x != 9999) {
8         s = new LNode;
9         s->data = x;
10        r->next = s;
11        r = s; //r移到新的表尾结点
12        cin >> x;
13    }
14    r->next = nullptr; //表尾指针置空
15    return L;
16 }

```

## 2) 不带头结点

```
1  LinkList list_tail_insert(LinkList &L) {
2      ElemType x;
3      cin >> x;
4      L = new LNode;
5      L->data = x;
6      LNode *r, *s;
7      r = L; //尾指针
8      cin >> x;
9      while (x != 9999) {
10         s = new LNode;
11         s->data = x;
12         r->next = s;
13         r = s; //指向表尾结点
14         cin >> x;
15     }
16     r->next = nullptr;
17     return L;
18 }
```

不带头结点的是我自己写的，带头结点和不带头结点的区别是：不带头结点需要先插入首元结点，在进行循环插入；而带头结点可以直接进行循环插入。

### 2.6.2 头插法

#### 1) 带头结点：

```
1  //头插法
2  LinkList list_head_insert(LinkList &L) {
3      ElemType x;
4      L = new LNode; //建立头结点
5      LNode *s;
6      L->next = nullptr; //表尾指针置空
7      cin >> x;
8      while (x != 9999) {
9          s = new LNode;
10         s->data = x;
11         s->next = L->next; //s插入到表头
12         L->next = s; //头结点指向首元结点
13         cin >> x;
14     }
15     return L;
16 }
```

#### 重要应用：链表的逆置

#### 2) 不带头结点：

```
1  //头插法
2  LinkList list_head_insert(LinkList &L) {
3      ElemType x;
4      LNode *s;
5      L = nullptr;
6      cin >> x;
7      while (x != 9999) {
```

```

8         s = new LNode;
9         s->data = x;
10        s->next = L;
11        L = s;
12        cin >> x;
13    }
14    return L;
15 }

```

## 2.7 单链表的逆置（头插法思想）

### 1) 带头结点

```

1 //逆置
2 void reverse(LinkList &L) {
3     LNode *p, *q;
4     p = L->next;
5     L->next = nullptr;
6     while (p) {
7         q = p;
8         p = p->next;
9         q->next = L->next;
10        L->next = q;
11    }
12 }

```

### 2) 不带头结点

```

1 //逆置
2 void reverse(LinkList &L) {
3     LNode *p, *q;
4     p = L;
5     L = nullptr;
6     while (p) {
7         q = p;
8         p = p->next;
9         q->next = L;
10        L = q;
11    }
12 }

```

## 2.8 全部操作的实现

### 1) 带头结点

```

1 typedef int ElemType;
2 //定义结点
3 typedef struct LNode {
4     ElemType data; //数据域, 存放数据元素
5     struct LNode *next; //指针域, 指向下一个结点
6 } LNode, *LinkList; //LNode表示结点, LinkList表示链表
7 //初始化单链表
8 bool InitList(LinkList &L) {
9     L = new LNode; //申请头结点
10    if (!L) //空间申请失败

```

```

11         return false;
12         L->next = nullptr;
13         return true;
14     }
15     //判断表空
16     bool Empty(LinkList L) {
17         return (L->next == nullptr);
18     }
19     //按位序插入
20     bool ListInsert(LinkList &L, int i, ElemType e) {
21         if (i < 1)
22             return false;
23         LNode *p = L; //指向头结点, 头结点是第0个结点
24         int j = 0; //计数器, p指向的第几个结点
25         while (p && j < i - 1) { //循环找到第i-1个结点或到表尾
26             p = p->next;
27             ++j;
28         }
29         if (!p) //p指针为空, 也就是i值不合法, i>单链表长度
30             return false;
31         LNode *s = new LNode;
32         s->data = e;
33         s->next = p->next;
34         p->next = s; //s插到p之后
35         return true;
36     }
37     //指定结点后插入结点
38     bool InsertNextNode(LNode *p, ElemType e) {
39         if (p == nullptr)
40             return false;
41         LNode *s = new LNode;
42         if (!s)
43             return false;
44         s->data = e;
45         s->next = p->next;
46         p->next = s;
47         return true;
48     }
49     //指定结点前插入结点
50     bool InsertPriorNode(LNode *p, ElemType e) {
51         if (p == nullptr)
52             return false;
53         LNode *s = new LNode;
54         if (!s)
55             return false;
56         s->data = p->data;
57         s->next = p->next;
58         p->data = e;
59         p->next = s;
60         return true;
61     }
62     //按位序删除
63     bool ListDelete(LinkList &L, int i, ElemType &e) {
64         if (i < 1)
65             return false;
66         LNode *p = L; //p指向头结点, 也就是第0个结点
67         int j = 0; //计数器, p指向第几个结点
68         while (p && j < i - 1) { //扫描到第i-1个结点, 或者扫描到表尾

```

```

69         p = p->next;
70         ++j;
71     }
72     if (!p)//i值不合格,i>表长, 扫描全表不存在p
73         return false;
74     LNode *q = p->next;//保存要删除的结点
75     e = q->data;//e存放要删除结点的元素值
76     p->next = q->next;//删除结点
77     delete q;//释放删除结点的内存空间
78     return true;
79 }
80 //指定节点的删除(给定结点不为最后结点的情况)
81 bool DeleteNode(LNode *p) {
82     if (!p)
83         return false;
84     LNode *q = p->next;
85     p->data = q->data;
86     p->next = q->next;
87     delete q;
88     return true;
89 }
90 //按位查找, 获取表中第i个位置的元素的值
91 bool GetElem(LinkList L, int i, ElemType &e) {
92     if (i < 1)
93         return false;
94     LNode *p = L;
95     int j = 0;
96     while (p && j < i) {
97         p = p->next;
98         ++j;
99     }
100     if (!p)
101         return false;
102     e = p->data;
103     return true;
104 }
105 //按位查找, 返回第i个结点
106 LNode *GetElem(LinkList L, int i) {
107     if (i < 0)
108         return nullptr;
109     LNode *p = L;
110     int j = 0;
111     while (p && j < i) {
112         p = p->next;
113         ++j;
114     }
115     return p;
116 }
117 //按值查找, 在表中查找具有给定关键字值的元素
118 LNode *LocateElem(LinkList L, ElemType e) {
119     LNode *p = L->next;//从首元结点开始查找
120     while (p && p->data != e) {
121         p = p->next;
122     }
123     return p;//如果找到了就返回结点指针, 没找到就返回空指针
124 }
125 //求表长
126 int Length(LinkList L) {

```



```

127     int len = 0;
128     LNode *p = L;
129     while (p->next) {
130         p = p->next;
131         len++;
132     }
133     return len;
134 }
135 //尾插法建立单链表
136 LinkList list_tail_insert(LinkList &L) {
137     ElemType x;
138     L = new LNode;
139     LNode *s, *r = L; //r为表尾结点
140     cin >> x; //输入第一个结点的值
141     while (x != 9999) {
142         s = new LNode;
143         s->data = x;
144         r->next = s;
145         r = s; //r移到新的表尾结点
146         cin >> x;
147     }
148     delete s;
149     r->next = nullptr; //表尾指针置空
150     return L;
151 }
152 //头插法
153 LinkList list_head_insert(LinkList &L) {
154     ElemType x;
155     L = new LNode; //建立头结点
156     LNode *s;
157     L->next = nullptr; //表尾指针置空
158     cin >> x;
159     while (x != 9999) {
160         s = new LNode;
161         s->data = x;
162         s->next = L->next; //s插入到表头
163         L->next = s; //头结点指向首元结点
164         cin >> x;
165     }
166     return L;
167 }
168 //逆置
169 void reverse(LinkList &L) {
170     LNode *p, *q;
171     p = L->next;
172     L->next = nullptr;
173     while (p) {
174         q = p;
175         p = p->next;
176         q->next = L->next;
177         L->next = q;
178     }
179 }
180 }

```

## 2) 不带头结点

```
1  typedef int ElemType;
2  //定义结点
3  typedef struct LNode {
4      ElemType data; //数据域,存放数据元素
5      struct LNode *next; //指针域,指向下一个结点
6  } LNode, *LinkList; //LNode表示结点, LinkList表示链表
7  //初始化链表
8  bool InitList(LinkList &L) {
9      L = nullptr; //表空,暂时没有任何结点,防止头指针的地址原来有脏数据
10     return true;
11 }
12 //判断表空(无头结点)
13 bool Empty(LinkList L) {
14     if (!L)
15         return true;
16     else
17         return false;
18 }
19 //按位插入
20 bool ListInsert(LinkList &L, int i, ElemType e) {
21     if (i < 1)
22         return false;
23     if (i == 1) { //插入第1个结点
24         LNode *s = new LNode;
25         s->data = e;
26         s->next = nullptr;
27         L = s;
28         return true;
29     }
30     //插入第2个及之后的结点
31     int j = 1;
32     LNode *p = L;
33     while (p && j < i - 1) { //i>n
34         p = p->next;
35         ++j;
36     }
37     if (!p)
38         return false;
39     LNode *s = new LNode;
40     s->data = e;
41     s->next = p->next;
42     p->next = s;
43     return true;
44 }
45 //指定结点后插入结点
46 bool InsertNextNode(LNode *p, ElemType e) {
47     if (!p)
48         return false;
49     LNode *s = new LNode;
50     if (!s)
51         return false;
52     s->data = e;
53     s->next = p->next;
54     p->next = s;
55     return true;
```

```

56 }
57 //指定结点前插入结点
58 bool InsertPriorNode(LNode *p, ElemType e) {
59     if (!p)
60         return false;
61     LNode *s = new LNode;
62     if (!s)
63         return false;
64     s->data = p->data;
65     s->next = p->next;
66     p->data = e;
67     p->next = s;
68     return true;
69 }
70 }
71 //按位序删除
72 bool ListDelete(LinkList &L, int i, ElemType &e) {
73     if (i < 1)
74         return false;
75     LNode *p = L; //首元结点
76     int j = 1;
77     while (p && j < i - 1) {
78         p = p->next;
79         j++;
80     }
81     if (!p)
82         return false;
83     LNode *q = p->next;
84     e = q->data;
85     p->next = q->next;
86     delete q;
87     return true;
88 }
89 //指定节点的删除(给定结点不为最后结点的情况)
90 bool DeleteNode(LNode *p) {
91     if (!p)
92         return false;
93     LNode *q = p->next;
94     p->data = q->data;
95     p->next = q->next;
96     delete q;
97     return true;
98 }
99 }
100 //按位查找, 获取表中第i个位置的元素的值
101 bool GetElem(LinkList L, int i, ElemType &e) {
102     if (i < 1)
103         return false;
104     int j = 1;
105     LNode *p = L;
106     while (p && j < i) {
107         p = p->next;
108         j++;
109     }
110     if (!p)
111         return false;
112     e = p->data;
113     return true;

```

```

114 }
115 //按位查找，返回第i个结点
116 LNode *GetElem(LinkList L, int i) {
117     if (i < 0)
118         return nullptr;
119     LNode *p = L;
120     int j = 1;
121     while (p && j < i) {
122         p = p->next;
123         j++;
124     }
125     return p;
126 }
127 //按值查找，在表中查找具有给定关键字值的元素
128 LNode *LocateElem(LinkList L, ElemType e) {
129     LNode *p = L;
130     while (p && p->data != e) {
131         p = p->next;
132     }
133     return p;
134 }
135 //求表长
136 int Length(LinkList L) {
137     LNode *p = L;
138     int len = 1;
139     while (!p->next) {
140         p = p->next;
141         len++;
142     }
143     return len;
144 }
145 //尾插法建立单链表
146 LinkList list_tail_insert(LinkList &L) {
147     ElemType x;
148     cin >> x;
149     L = new LNode;
150     L->data = x;
151     LNode *r, *s;
152     r = L; //尾指针
153     cin >> x;
154     while (x != 9999) {
155         s = new LNode;
156         s->data = x;
157         r->next = s;
158         r = s; //指向表尾结点
159         cin >> x;
160     }
161     delete s;
162     r->next = nullptr;
163     return L;
164 }
165 //头插法
166 LinkList list_head_insert(LinkList &L) {
167     ElemType x;
168     LNode *s;
169     L = nullptr;
170     cin >> x;
171     while (x != 9999) {

```

```

172         s = new LNode;
173         s->data = x;
174         s->next = L;
175         L = s;
176         cin >> x;
177     }
178     return L;
179 }
180 //逆置
181 void reverse(LinkList &L) {
182     LNode *p, *q;
183     p = L;
184     L = nullptr;
185     while (p) {
186         q = p;
187         p = p->next;
188         q->next = L;
189         L = q;
190     }
191 }

```

## 3 双向链表

### 3.1 结点



```

1 typedef struct DNode {
2     ElemType data;
3     struct DNode *prior; //前指针
4     struct DNode *next;  //后指针
5 } DNode, *DLinkedList;

```

### 3.2 初始化

```

1 bool initList(DLinkedList &L) {
2     L = new DNode; //头结点
3     if (!L)
4         return false;
5     L->prior = nullptr; //头结点prior指向空
6     L->next = nullptr;
7     return true;
8 }

```

### 3.3 判空

```

1 bool empty(DLinkedList L) {
2     if (!L->next)
3         return false;
4     else
5         return true;
6 }

```

### 3.4 插入结点

```
1 //在p结点后插入s结点
2 bool insert_next_node(DNode *p, DNode *s) {
3     if (!p || !s)
4         return false;
5     s->next = p->next;
6     if (p->next)//如果p结点有后继结点
7         p->next->prior = s;
8     s->prior = p;
9     p->next = s;
10    return true;
11 }
```

### 3.5 删除结点

```
1 //删除结点p的后继结点
2 bool delete_next_node(DNode *p) {
3     if (!p || !p->next)
4         return false;
5     DNode *q = p->next;
6     p->next = q->next;
7     if (q->next)
8         q->next->prior = p;
9     delete q;
10    return true;
11 }
```

### 3.6 按位查找

```
1 //按位查找，返回第i个结点
2 DNode *get_elem(DLinkedList &L, int i) {
3     if (!L)
4         return nullptr;
5     int j = 1;
6     DNode *p = L->next;
7     while (p && j < i) {
8         p = p->next;
9         j++;
10    }
11    return p;
12 }
```

### 3.7 后插法

```
1 bool insert_tail_list(DLinkedList &L) {
2     if (!L)
3         return false;
4     L = new DNode;
5     L->prior= nullptr;
6     ElemType x;
7     DNode *p, *s;
8     p = L;
9     if (!p)
```

```

10         return false;
11     cin >> x;
12     while (x != 9999) {
13         s = new DNode;
14         s->data = x;
15         s->prior = p;
16         p->next = s;
17         p = s;
18         cin >> x;
19     }
20     p->next = nullptr;
21     return true;
22 }

```

## 3.8 遍历

### 1) 后向遍历

```

1 //后向遍历打印双链表
2 void next_to_string(DLinkedList L) {
3     if (!L)
4         exit(-1);
5     cout << "打印双链表: ";
6     DNode *p = L->next; //首元结点
7     while (p) {
8         cout << p->data << " ";
9         p = p->next;
10    }
11    cout << endl;
12 }

```

### 2) 前向遍历

```

1 //前向遍历打印链表
2 void prior_to_string(DNode *p){ //p为表中最后一个结点
3     cout << "前项遍历链表: ";
4     while(p->prior){ //p->prior为空表示当前p已经指向头指针
5         cout << p->data << " ";
6         p=p->prior;
7     }
8     cout << endl;
9 }

```

时间复杂度=O(n)

## 4 循环链表

### 4.1 循环单链表

特点:

1. 表尾指针的next指向头结点
2. 从一个节点出发，可以找到其他任何一个结点。

## 1) 初始化

```
1  bool init_list(LinkList &L) {
2      L = new LNode;
3      if (!L)
4          return false;
5      L->next = L;
6      return true;
7  }
```

## 2) 判空

```
1  bool empty(LinkList L) {
2      if (L->next == L)
3          return true;
4      else
5          return false;
6  }
```

## 3) 判断结点是否是表尾结点

```
1  bool is_tail(LinkList L, LNode *p) {
2      if (p->next == L)
3          return true;
4      else
5          return false;
6  }
```

## 4.2 循环双链表

特点:

1. 表头指针prior指向表尾结点
2. 表尾指针的next指向头结点

### 1) 初始化

```
1  bool init_list(DLinkList &L) {
2      L = new DNode;
3      if (!L)
4          return false;
5      L->prior = L;
6      L->next = L;
7      return true;
8  }
```

### 2) 判空

```
1  bool empty(DLinkList L) {
2      if (L->next == L)
3          return true;
4      else
5          return false;
6  }
```



### 3) 判断结点是否为表尾结点

```
1 bool is_tail(DLinkedList L, DNode *p) {
2     if (p->next == L)
3         return true;
4     else
5         return false;
6 }
```

### 4) 插入

```
1 //结点p后插入元素值为e的结点
2 bool insert_next_node(DNode *p, ElemType e) {
3     if (!p)
4         return false;
5     DNode *s = new DNode;
6     s->data = e;
7     s->next = p->next;
8     if (p->next)
9         p->next->prior = s;
10    s->prior = p;
11    p->next = s;
12    return true;
13 }
```

### 5) 删除

```
1 //删除结点p后的结点
2 bool delete_next_node(DNode *p) {
3     if (!p || !p->next)
4         return false;
5     DNode *q = p->next;
6     p->next = q->next;
7     if (q->next)
8         q->next->prior = p;
9     delete q;
10    return true;
11 }
```

### 6) 获取结点

```
1 //获取第i个结点
2 DNode *get_elem(DLinkedList L, int i) {
3     if (i < 1 || !L)
4         return nullptr;
5     DNode *p = L->next;
6     int j = 1;
7     while (p != L && j < i) {
8         p = p->next;
9         j++;
10    }
11    if (p == L)
12        return nullptr;
13    return p;
14 }
```

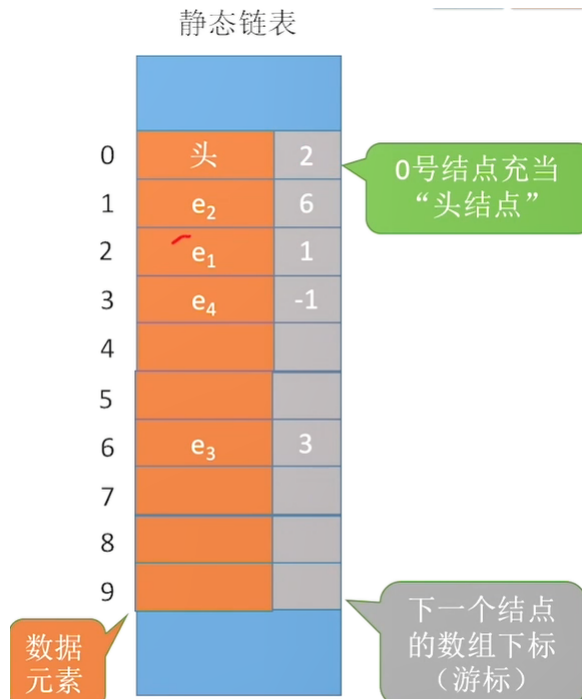
## 5 静态链表（不一定考）

**特点：**分配一整片连续的内存空间，各个结点集中安置

**优点：**增删操作不需要大量移动元素

**缺点：**不能随机存取，只能从头结点开始依次往后查找；容量固定不可变

**使用场景：**1) 不支持指针的低级语言；2) 数据元素数量固定不变的场景(如操作系统的文件分配表FAT)



**结点：**

```
1 struct Node{
2     ElemType data;
3     int next; //游标，充当指针，指向下一个元素的数组下标
4 };
5 struct Node node[MAX_SIZE]; //初始化
6 //或者
7 typedef struct{
8     ElemType data;
9     int next;
10 }SLinkList[MAX_SIZE];
11 SLinkList sLinkList; //初始化
```

### 1) 查找

从头结点出发挨个往后遍历

时间复杂度=O(n)

## 2) 插入位序为i的结点

1. 找到一个空的结点，存入数据
2. 从头结点出发找到位序为i-1的结点
3. 修改新结点的next
4. 修改i-1号结点的next

## 3) 删除某个结点

1. 从头结点出发找到前驱结点
2. 修改前驱节点的游标
3. 被删除结点next设置为-2(设置为-2只是为了表示next为空)

# 6 顺序表和链表的对比

---

## 1) 逻辑结构

都属于线性表，都是线性结构

## 2) 存储结构

**顺序表**：采用**顺序存储**，逻辑上相邻的元素，对应的物理存储位置也相邻

1. 优点：支持随机存取，存储密度高
2. 缺点：大片连续空间分配不方便，改变容量不方便，会出现空间闲置或溢出现象

**链表**：采用**链式存储**，逻辑上相邻的元素，物理存储位置不一定相邻

1. 优点：理想的小空间分配方便，改变容量方便，不会出现空间闲置或溢出现象
2. 缺点：不可随机存取，存储密度低

## 3) 基本操作

**创建操作：**

- 顺序表
  - 静态分配，需要与分配一大块连续的内存空间，会造成浪费或溢出，容量不可改变。
  - 也可采用动态分配，容量可以改变，但是需要移动大量的元素，时间代价高
- 链表：动态分配，只需要先申请一个头结点，容量可改变

**销毁操作：**

- 顺序表：系统自动回收内存空间
- 链表：需要手动申请和释放内存

**插入、删除操作：**

- 顺序表：
  - 插入/删除元素需要将后续元素都后移/前移
  - 时间复杂度 $O(n)$ ，时间开销主要来自移动元素
- 链表：
  - 插入/删除元素只需要修改指针即可
  - 时间复杂度 $O(n)$ ，时间开销主要来自查找目标元素

**查找操作：**

- 顺序表：

- 按位查找:  $O(1)$
- 按值查找:  $O(n)$ , 若表内有序可采用折半查找, 时间复杂度 $O(\log_2 n)$
- 链表:
  - 按位查找:  $O(n)$
  - 按值查找:  $O(n)$

## 4) 顺序表和链表的选择

	顺序表	链表
弹性 (可扩容)	×	√
增、删	×	√
查	√	×

## 7 线性表的应用

### 7.1 线性表的合并

步骤:

1. 分别获取LA表长m、LB表长n
2. 遍历LB表, 如果表中元素不在A中, 插入到LA表后

```

1 //顺序表为例
2 void merge_list(SeqList &s1, SeqList &s2){
3     int m=Length(s1);
4     int n=Length(s2);
5     for (int i = 0; i < n; i++) {
6         ElemType x = s2.GetElem(s2,i+1);
7         if(!LocateElem(s1,x))
8             ListInsert(s1,++m,x);
9     }
10 }
```

时间复杂度= $O(m*n)$ , 顺序表和链表都是。

### 7.2 有序表的合并

#### 1) 有序顺序表的合并

```

1 void merge_list_seq(SeqList LA, SeqList LB, SeqList &LC) {
2     if ((LA.length + LB.length) > LC.MaxSize)
3         LC.IncreaseList(LA.length + LB.length - LC.length); //扩展LC的容量
4     int a = 0, b = 0, c = 0;
5     while (a < LA.length && b < LB.length) { //遍历LA和LB
6         if (LA.data[a] <= LB.data[b]) //两两相比, 把小的那个元素插入到LC表
7             LC.data[c++] = LA.data[a++];
8         else
9             LC.data[c++] = LB.data[b++];
10    }
11    while (a < LA.length) //如果LA表还有剩余元素, 插入到LC表中
12        LC.data[c++] = LA.data[a++];
13    while (b < LB.length) //如果LB表还有剩余元素, 插入到LC表中
```

```
14     LC.data[c++] = LB.data[b++];
15     LC.length = c;
16 }
```

## 三、栈、队列、数组

### 1 栈

#### 1.1 栈的定义和特点

##### 1.1.1 栈的基本概念

**定义：**栈是只允许在一端进行插入或删除操作的线性表

**术语：**

1. 栈顶：表尾，允许插入和删除的一端
2. 栈底：表头，不允许插入和删除的一端
3. 空栈：不含元素的空表。

**特点：**后进先出（Last In First Out, LIFO）

##### 1.1.2 栈的抽象数据类型定义

```
1  ADT Stack {
2      数据对象:  $D = \{a_i | a_i \in \text{ElemSet}, i=1,2,3,\dots,n, n \geq 0\}$  // ElemSet 表示元素的集合
3      数据关系:  $R1 = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,\dots,n \}$  //  $a_{i-1}$  为前驱,  $a_i$  为后继
4          约定  $a_n$  端为栈顶,  $a_1$  端为栈底
5      基本操作:
6          InitStack (&S) 初始化操作
7              操作结果: 构造一个空栈 S
8          DestroyStack (&S) 销毁栈操作
9              初始条件: 栈S已存在
10             操作结果: 栈S被销毁
11          ClearStack (&S) 栈指控操作
12              初始条件: 栈S已存在
13              操作结果: 将S清为空栈
14          StackEmpty (S) 判定栈是否为空栈
15              初始条件: 栈S已存在
16              操作结果: 若栈S为空栈, 则返回true, 若栈S为非空, 则返回false
17          StackLength (S) 求栈的长度
18              初始条件: 栈S已存在
19              操作结果: 返回栈S的元素个数, 即栈的长度
20          GetTop (S) 取栈顶元素
21              初始条件: 栈S已存在且非空
22              操作结果: 返回S的栈顶元素, 不修改栈顶指针
23          Push (&S, e) 入栈操作
24              初始条件: 栈S已存在
25              操作结果: 插入元素e为新的栈顶元素
26          Pop (&S, &e) 出栈操作
27              初始条件: 栈S存在且非空
28              操作结果: 删除栈S的栈顶元素, 并用e返回其值
29          StackTraverse(S)
30              初始条件: 栈S已存在且非空
31              操作结果: 从栈底到栈顶依次对S的每个数据元素进行访问
32 }ADT Stack
33
```

### 1.1.3 基本操作

- 1 `InitStack(&S)`: 初始化栈。构造一个空栈S, 分配内存空间。
- 2 `DestroyStack(&L)`: 销毁栈。销毁并释放栈S所占用的内存空间。
- 3 `Push(&S,x)`: 进栈。若栈S未满, 则将x加入使之成为新的栈顶。
- 4 `Pop(&S,&x)`: 出栈。若栈S非空, 则弹出栈顶元素, 并用x返回。
- 5 `GetTop(S,&x)`: 读栈顶元素。若栈S非空, 则用x返回栈顶元素。
- 6 `StackEmpty(S)`: 判空操作。若S为空栈, 则返回true, 否则返回false。

### 1.1.4 卡特兰数

n个不同元素进栈, 出栈元素不同排列的个数  $\frac{1}{n+1} C_{2n}^n$

## 1.2 栈的顺序存储的实现

### 1.2.1 顺序栈的定义

顺序栈: 利用顺序存储结构实现的栈, 用静态数组实现, 需要记录栈顶指针

```
1 #define MAX_SIZE 10
2 typedef int ElemType;
3 typedef struct{
4     ElemType data[MAX_SIZE];
5     int top;
6 }SqStack;
```

### 1.2.2 顺序栈的基本操作的实现

```
1 #define MAX_SIZE 10
2 typedef int ElemType;
3 typedef struct {
4     ElemType data[MAX_SIZE];
5     int top;
6 } SqStack;
7 //初始化栈
8 void InitStack(SqStack &S) {
9     S.top = 0;
10 }
11 //判空
12 bool Empty(SqStack S) {
13     if (S.top == 0)
14         return true;
15     else
16         return false;
17 }
18 //入栈
19 bool Push(SqStack &S, ElemType x) {
20     if (S.top == MAX_SIZE)//栈满
21         return false;
22     S.data[S.top++] = x;//新元素入栈
23     return true;
24 }
25 //出栈
26 bool Pop(SqStack &S, ElemType &x) {
```

```

27     if (S.top == 0) //栈空
28         return false;
29     x = S.data[--S.top];
30     return true;
31 }
32 //读栈顶
33 bool GetTop(SqStack S, ElemType &x) {
34     if (S.top == 0)
35         return false;
36     x = S.data[S.top-1];
37     return true;
38 }

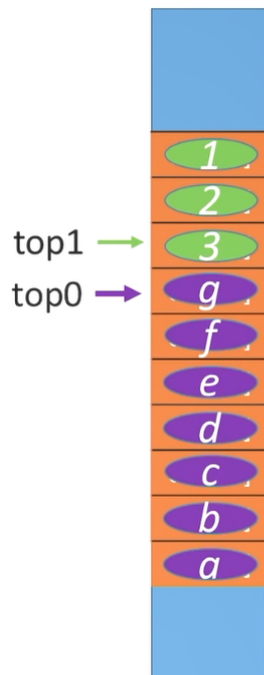
```

创、增、删、查 时间复杂度为 $O(1)$

## 1.3 共享栈

### 1.3.1 共享栈的定义

共享栈：两个栈共享一片内存空间



### 1.3.2 共享栈的实现

```

1  #define MAX_SIZE 10
2  typedef int ElemType;
3  typedef struct {
4      ElemType data[MAX_SIZE];
5      int top0;
6      int top1;
7  } ShareStack;
8  //初始化
9  void InitStacki(ShareStack &s) {
10     s.top0 == -1;
11     s.top1 == MAX_SIZE;
12 }
13 //判空
14 bool Empty(ShareStack s) {

```

```

15     if (S.top0 == -1 && S.top1 == MAX_SIZE)
16         return true;
17     else
18         return false;
19 }
20 //判断栈满
21 S.top0+1==S.top1;

```

## 1.3 栈的链式存储的实现

### 1.3.1 链式栈的实现

```

1  typedef int ElemType;
2  typedef struct StackNode {
3      ElemType data;
4      struct StackNode *next;
5  } StackNode, *LinkStack;
6  //初始化
7  void InitStack(LinkStack &S) {
8      S = nullptr;
9  }
10 //入栈
11 bool Push(LinkStack &S, ElemType x) {
12     StackNode *s = new StackNode;
13     s->data = x;
14     s->next = S;
15     S = s;
16 }
17 //出栈
18 bool Pop(LinkStack &S, ElemType &e) {
19     if (!S)
20         return false;
21     e = S->data;
22     StackNode *p = S;
23     S = S->next;
24     delete p;
25     return true;
26 }
27 //取栈顶元素
28 bool GetTop(LinkStack S, ElemType &e) {
29     if (!S)
30         return false;
31     e = S->data;
32     return true;
33 }

```

### 1.3.2 汉诺塔问题的递归算法

算法步骤:

1. 如果 $n=1$ ，则直接将编号为1的圆盘从A移到C，递归结束
2. 否则：
  - 递归，将A上编号为1至 $n-1$ 的圆盘移动到B，C做辅助塔
  - 直接将编号为 $n$ 的圆盘从A移到C
  - 递归，将B上编号为1至 $n-1$ 的圆盘移动到C，A做辅助塔



```

1 void move(LinkStack &from, LinkStack &to) {
2     ElemType data;
3     Pop(from, data);
4     Push(to, data);
5 }
6 void hanoi(int n, LinkStack &A, LinkStack &B, LinkStack &C) {
7     if (n == 1)
8         move(A, C);
9     else {
10        hanoi(n - 1, A, C, B);
11        move(A, C);
12        hanoi(n - 1, B, A, C);
13    }
14 }

```

时间复杂度= $O(2^n)$

空间复杂度= $O(n)$

## 2 队列

### 2.1 队列的定义和特点

#### 2.1.1 队列的概念

**定义：**队列是只允许在一端进行插入，在另一端进行删除的线性表。

**特点：**先进先出 (First In First Out, FIFO)

**术语：**

- 队头：允许删除的一端，又称队首。
- 队尾：允许插入的一端。
- 空队列：不含任何元素的空表。

#### 2.1.2 队列的抽象数据类型定义

```

1 ADT Queue{
2     数据对象:  $D = \{a_i | a_i \in \text{ElemSet}, i=1, 2, 3, \dots, n, n \geq 0\}$ 
3     数据关系:  $R1 = \{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2, \dots, n \}$ 
4         约定  $a_1$  端为队列头,  $a_n$  端为队列尾
5     基本操作:
6         InitQueue(&Q)
7             操作结果: 构造一个空队列Q
8         DestroyQueue(&Q)
9             初始条件: 队列Q已存在
10            操作结果: 队列Q被销毁, 不再存在
11        ClearQueue(&Q)
12            初始条件: 队列Q已存在
13            操作结果: 将Q清为空队列
14        QueueEmpty(Q)
15            初始条件: 队列Q已存在
16            操作结果: 若Q为空队列, 则返回true, 否则返回false
17        GetHead(Q)
18            初始条件: Q为非空队列
19            操作结果: 返回Q的队头元素
20        EnQueue(&Q, e)

```

```

21         初始条件: 队列Q已存在
22         操作结果: 插入元素e为Q的新的队尾元素
23         DeQueue(&Q,&e)
24         初始条件: Q为非空队列
25         操作结果: 删除Q的队头元素, 并用e返回其值
26         QueueTraverse(&Q)
27         初始条件: Q已存在且非空
28         操作结果: 从队头到队尾, 依次对Q的每个数据元素访问
29     }ADT Queue

```

### 2.1.3 基本操作

```

1  InitQueue(&Q): 初始化队列, 构造一个空队列Q。
2  DestroyQueue(&Q): 销毁队列。销毁并释放队列Q所占用的内存空间。
3  EnQueue(&Q,x): 入队, 若Q未满, 将x加入, 使之成为新的队尾。
4  DeQueue(&Q,&x): 出队, 若Q非空, 删除队头元素, 并用x返回。
5  GetHead(Q,&x): 度队头元素, 若队列Q非空, 则将队头元素赋值给x。
6  QueueEmpty(Q): 判断队空, 若队列Q为空返回true, 否则返回false。

```

## 2.2 队列的顺序实现

### 2.2.1 循环队列

```

1  #define MAX_SIZE 10//队列初始容量
2  typedef int ElemType;
3  //队列的顺序存储结构
4  typedef struct {
5      ElemType *data;
6      int front;//队头指针
7      int rear;//队尾指针
8  } SqQueue;
9  //初始化
10 bool InitQueue(SqQueue &Q) {
11     Q.data = new ElemType[MAX_SIZE];
12     if (!Q.data)
13         return false;
14     Q.front = Q.rear = 0;
15     return true;
16 }
17 //判空
18 bool QueueEmpty(SqQueue Q) {
19     if (Q.front == Q.rear)//队空条件
20         return true;
21     else
22         return false;
23 }
24 //入队
25 bool EnQueue(SqQueue &Q, ElemType e) {
26     if ((Q.rear + 1) % MAX_SIZE == Q.front)//队满
27         return false;
28     Q.data[Q.rear] = e;//添加数据
29     Q.rear = (Q.rear + 1) % MAX_SIZE;//循环队列, 队尾指针+1
30     return true;
31 }
32 //出队
33 bool DeQueue(SqQueue &Q, ElemType &e) {

```

```

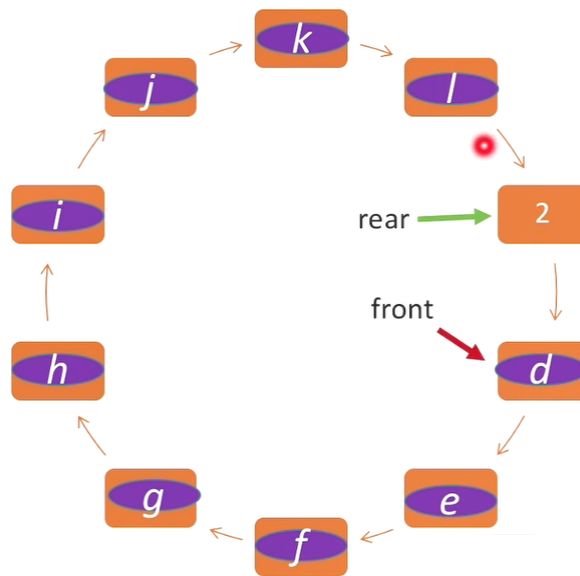
34     if (Q.front == Q.rear) //队空
35         return false;
36     e = Q.data[Q.front]; //保存队头元素
37     Q.front = (Q.front + 1) % MAX_SIZE; //队头指针加1
38     return true;
39 }
40 //取队头元素
41 bool GetHead(SqQueue Q, ElemType &e) {
42     if (Q.front == Q.rear)
43         return false;
44     e = Q.data[Q.front];
45     return true;
46 }
47 //求循环队列长度
48 int Length(SqQueue Q) {
49     return (Q.rear - Q.front + MAX_SIZE) % MAX_SIZE;
50 }

```

## 2.2.2 判断队满\队空的三种方式

**第一种方式：** 上述循环队列操作中的方式

- 队满:  $(Q.rear + 1) \% MAX\_SIZE == Q.front$
- 队空:  $Q.front == Q.rear$



**注意：** 这种方式下，数组空间中会有一个位置的浪费。

**第二种方式：**

如果想要不浪费这一个空间，需要在结构体中添加一个size属性，用来记录队列的长度，在进行入队出队操作的时候size++或者size--，判断队满size==MAXSIZE

```

1 //队列的顺序存储结构
2 typedef struct {
3     ElemType *data;
4     int front; //队头指针
5     int rear; //队尾指针
6     int size;
7 } SqQueue;
8 //初始化
9 bool InitQueue(SqQueue &Q) {

```

```

10     Q.data = new ElemType[MAX_SIZE];
11     if (!Q.data)
12         return false;
13     Q.front = Q.rear = 0;
14     Q.size = 0;
15     return true;
16 }

```

- 队满:  $Q.size == MAX\_SIZE$
- 队空:  $Q.size == 0$

### 第三种方式:

结构体中添加一个tag属性, 用来记录最近一次的插入或者删除操作。删除操作完成时设置tag=0, 插入操作完成时设置tag=1。只有删除操作, 才可能导致队空; 只有插入操作, 才可能导致队满。

```

1  #define MAX_SIZE 10//队列初始容量
2  typedef int ElemType;
3  typedef struct {
4      ElemType *data;
5      int front;//队头指针
6      int rear;//队尾指针
7      int tag;//入队操作后设为1, 出队操作后设为0
8  } SqQueue;
9
10 //初始化
11 bool InitQueue(SqQueue &Q) {
12     Q.data = new ElemType[MAX_SIZE];
13     if (!Q.data)
14         return false;
15     Q.front = Q.rear = 0;
16     Q.tag = 0;
17     return true;
18 }

```

- 队空:  $Q.rear == Q.front \ \&\& \ Q.tag == 0$
- 队满:  $Q.rear == Q.front \ \&\& \ Q.tag == 1$

## 2.3 队列的链式实现

### 2.3.1 带头结点 (方便操作)

```

1  typedef int ElemType;
2  typedef struct LinkNode {
3      ElemType data;
4      struct LinkNode *next;
5  } LinkNode;
6  typedef struct {
7      LinkNode *front;
8      LinkNode *rear;
9  } LinkQueue;
10 //初始化
11 void InitQueue(LinkQueue &Q) {
12     Q.front = Q.rear = new LinkNode;//头结点
13     Q.front->next = nullptr;//头结点的指针域置空
14 }

```

```

15 //判空
16 bool Empty(LinkQueue Q) {
17     if (Q.front == Q.rear)
18         return true;
19     else
20         return false;
21 }
22 //入队
23 bool EnQueue(LinkQueue &Q, ElemType e) {
24     LinkNode *s = new LinkNode;
25     s->data = e;
26     s->next = nullptr;
27     Q.rear->next = s;
28     Q.rear = s;
29     return true;
30 }
31 //出队
32 bool DeQueue(LinkQueue &Q, ElemType &e) {
33     if (Q.rear == Q.front)
34         return false;
35     LinkNode *p = Q.front->next;
36     e = p->data;
37     Q.front->next = p->next;
38     if (Q.rear == p)
39         Q.rear = Q.front;
40     delete p;
41     return true;
42 }
43 //取队头元素
44 bool GetHead(LinkQueue Q, ElemType &e) {
45     if (Q.rear == Q.front)
46         return false;
47     e = Q.front->next->data;
48     return true;
49 }
50 //获取队列长度
51 int Length(LinkQueue Q) {
52     if (Q.front == Q.rear)
53         return 0;
54     int len = 0;
55     LinkNode *p = Q.front;
56     while (p->next) {
57         p = p->next;
58         len++;
59     }
60     return len;
61 }
62 //遍历
63 bool traverse(LinkQueue Q) {
64     if (Q.rear == Q.front)
65         return false;
66     std::cout << "打印链式队列: ";
67     LinkNode *p = Q.front->next;
68     while (p) {
69         std::cout << p->data << "\t";
70         p = p->next;
71     }
72     std::cout << std::endl;

```

```
73     return true;
74 }
```

### 2.3.2 不带头结点

```
1  typedef int ElemType;
2  typedef struct LinkNode {
3      ElemType data;
4      struct LinkNode *next;
5  } LinkNode;
6  typedef struct {
7      LinkNode *front;
8      LinkNode *rear;
9  } LinkQueue;
10 //初始化
11 void InitQueue(LinkQueue &Q) {
12     Q.front = nullptr;
13     Q.rear = nullptr;
14 }
15 //判空
16 bool Empty(LinkQueue &Q) {
17     if (Q.front == nullptr)
18         return true;
19     else
20         return false;
21 }
22 //入队
23 bool EnQueue(LinkQueue &Q, ElemType e) {
24     LinkNode *s = new LinkNode;
25     s->data = e;
26     s->next = nullptr;
27     if (Q.front == nullptr) { //在队列中插入第一个结点
28         Q.front = s;
29         Q.rear = s;
30     } else {
31         Q.rear->next = s; //新结点插入到Q.rear之后
32         Q.rear = s; //修改Q.rear
33     }
34     return true;
35 }
36 //出队
37 bool DeQueue(LinkQueue &Q, ElemType &e) {
38     if (Q.front == nullptr)
39         return false;
40     LinkNode *p = Q.front;
41     e = p->data;
42     Q.front = p->next;
43     if (p == Q.rear) { //如果删除的p是最后一个结点
44         Q.rear = nullptr;
45         Q.front = nullptr;
46     }
47     delete p;
48     return true;
49 }
50 //获取队列长度
51 int Length(LinkQueue Q) {
52     if (Q.front == nullptr)
```

```

53         return 0;
54     int len = 0;
55     LinkNode *p = Q.front;
56     while (p) {
57         p = p->next;
58         len++;
59     }
60     return len;
61 }
62 //遍历
63 bool traverse(LinkQueue Q) {
64     if (Q.front == nullptr)
65         return false;
66     LinkNode *p = Q.front;
67     while (p) {
68         std::cout << p->data << "\t";
69         p = p->next;
70     }
71     std::cout << std::endl;
72     return true;
73 }

```

## 2.4 双端队列

分类：

- 双端队列：只允许从两端插入、两端删除的线性表。
- 输入受限的双端队列：只允许从一端输入，两端删除的线性表
- 输出受限的双端队列：只允许从两端插入，一端删除的线性表

考点：判断输出序列合法性

PS：在栈中合法的输出序列，在双端队列中必定合法

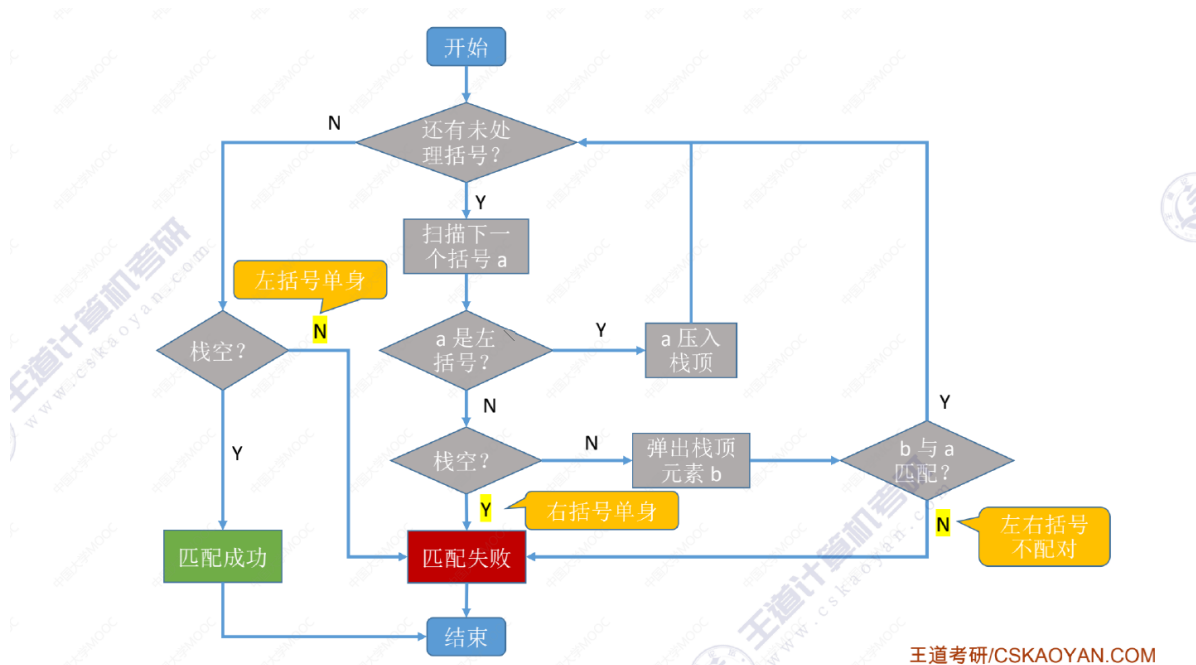
## 3 栈的应用

### 3.1 括号匹配

实现思路：依次扫描所有字符，遇到左括号入栈，遇到有括号则弹出栈顶元素检查是否匹配

匹配失败情况：

1. 右括号和左括号不匹配
2. 右括号消耗完，左括号还有剩余（左括号单身）
3. 左括号消耗完，还有有括号（右括号单身）



代码实现:

```

1  bool bracket(char str[], int length) {
2      LinkStack S;
3      InitStack(S);
4      for (int i = 0; i < length; i++) {
5          if (str[i] == '(' || str[i] == '[' || str[i] == '{') {
6              Push(S, str[i]); // 扫描到左括号, 出栈
7          } else {
8              if (Empty(S)) // 匹配到右括号, 且当前栈空
9                  return false; // 匹配失败
10             char topElem;
11             Pop(S, topElem); // 在栈顶元素出栈
12             if (str[i] == ')' && topElem != '(')
13                 return false;
14             if (str[i] == ']' && topElem != '[')
15                 return false;
16             if (str[i] == '}' && topElem != '{')
17                 return false;
18         }
19     }
20     return Empty(S);
21 }
  
```

## 3.2 表达式求值

三种算术表达式:

1. 中缀表达式: 运算符在两个操作数中间, 例:  $a+b-c*d$
2. 后缀表达式 (逆波兰表达式): 运算符在两个操作数后面, 例:  $ab+cd*-$
3. 前缀表达式 (波兰表达式): 运算符在两个操作数前面, 例:  $-+ad*cd$

表达式的三种成分: 操作数、运算符、界限符(括号)



### 3.2.1 中缀表达式转后缀表达式

#### 1) 手算方法:

1. 确定中缀表达式中各个运算符的运算顺序
2. 选择下一个运算符, 按照[左操作数 右操作数 运算符]的方式组合成一个新的操作数
3. 如果还有运算符没被处理, 就继续步骤2

注: 由于运算顺序不唯一, 因此对应的后缀表达式也不唯一

例:  $A+B*(C-D)-E/F$

转后缀表达式:  $ABCD-*+EF/-$  (计算机应该得到的结果)

$ABCD-*EF/-+$

注: 要保证手算和机算结果相同 (为了确保算法的“正确性”), 应采用“左优先”原则

“左优先”原则: 只要左边的运算符能先计算, 就优先算左边的

例:  $A+B-C*D/E+F \Rightarrow AB+CD*F/-F+$

#### 2) 机算方法:

1. 初始化一个栈, 用于保存暂时还不能确定运算顺序的运算符
2. 从左到右处理各个元素, 直到末尾。可能遇到三种情况:
  1. 遇到操作数。直接加入后缀表达式。
  2. 遇到界限符。遇到 '(' 直接入栈; 遇到 ')' 则依次弹出栈内运算符并加入后缀表达式, 知道弹出 '(' 为止。注意: '(' 不加入后缀表达式。
  3. 遇到运算符。依次弹出栈内优先级高于或等于当前运算符的所有运算符, 并加入后缀表达式, 若碰到 '(' 或栈空则停止。之后再把当前运算符入栈。
3. 按上述方法处理完所有字符后, 将栈中剩余运算符依次弹出, 并加入后缀表达式

### 3.2.2 后缀表达式的计算

#### 后缀表达式的手算方法:

从左往右扫描, 每遇到一个运算符, 就让运算符前面最近的两个操作数执行对应运算, 合为一个操作数

#### 用栈实现后缀表达式的计算:

1. 从左到右扫描下一个元素, 直到处理完所有元素
2. 若扫描到操作数则压入栈, 并返回步骤1; 否则执行步骤3
3. 若扫描到运算符, 则弹出两个栈顶元素, 执行相应运算, 运算结果压入栈顶, 回到步骤1

注意: 先出栈的是右操作数

### 3.2.3 中缀表达式转前缀表达式

#### 手算方法:

1. 确定中缀表达式中各个运算符的运算顺序
2. 选择下一个运算符, 按照[运算符 左操作数 右操作数]的方式组合成一个新的操作数
3. 如果还有运算符没被处理, 就继续步骤2

“右优先”原则: 只要右边的运算符能先计算, 就优先算右边的

注: 一个中缀表达式可以对应多个后缀、前缀表达式

### 3.2.4 前缀表达式的计算

用栈实现前缀表达式的计算：

1. 从右往左扫描下一个元素，知道处理完所有元素
2. 若扫描到操作数则压入栈，并回到步骤1；否则执行步骤3
3. 若扫描到运算符，则弹出两个栈顶元素，执行相应运算，运算结果压入栈顶，回到步骤1

注意：先出栈的是左操作数

### 3.2.5 中缀表达式的计算

用栈实现中缀表达式的计算：

1. 初始化两个栈，操作数栈和运算符栈
2. 若扫描到操作数，压入操作数栈
3. 若扫描到运算符或界限符，则按照“中缀转后缀”相同逻辑压入运算符栈（期间也会弹出运算符，每当弹出一个运算符时，就需要再弹出两个操作数栈的栈顶元素并执行相应的运算，运算结果在压回操作数栈）

## 3.3 递归中的应用

### 3.2.1 递归

递归调用其实就是特殊的函数调用，只不过它调用的函数是其本身而已。

函数调用的特点：最后被调用的函数最先执行结束（LIFO）

函数调用时需要用一个栈存储：

1. 调用返回地址
2. 实参
3. 局部变量

递归算法的两部分：

1. 递归表达式（递归体）
2. 边界条件（递归出口）

缺点：递归层数太多会导致栈溢出。

缺点：可能包含很多重复计算。

### 3.3.2 阶乘

```
1  int factorial(int n) {
2      if (n == 0 || n == 1)
3          return 1;
4      else
5          return n * factorial(n - 1);
6  }
```

### 3.3.3 斐波那契数列

```
1 int fib(int n) {
2     if (n == 0)
3         return 0;
4     else if (n == 1)
5         return 1;
6     else
7         return fib(n - 1) + fib(n - 2);
8 }
```

### 3.3.4 利用栈将递归转换为非递归的方法

利用栈消除递归的步骤：

1. 设置一个工作栈存放递归工作记录（包括实参、返回地址、及局部变量等）
2. 进入非递归调用入口（即被调用程序开始处）将调用程序传来的实在参数和返回地址入栈（递归程序不可以作为主程序，因而可认为初始是被某个调用程序调用）。
3. 进入递归调用入口：当不满足递归结束条件时，逐层递归，将实参、返回地址及局部变量入栈，这一过程可用循环语句来实现——模拟递归分解的过程。
4. 递归结束条件满足，将到达递归出口的给定常数作为当前的函数值。
5. 返回处理：在栈不空的情况下，反复退出栈顶记录，根据记录中的返回地址进行题意规定的操作，即逐层计算当前函数值，直至占空为止——模拟递归求值过程。

### 3.3.5 用栈实现的非递归阶乘

```
1 int non_recursion_factorial_1(int n) {
2     stack<int> stack; //新建一个栈，存储每层递归的计算结果
3     int result = 1;
4     if (n == 0 || n == 1) //边界条件
5         return 1;
6     while (n != 1 && n != 0) //while循环，将每层的计算结果入栈
7         stack.push(n--);
8     while (!stack.empty()) { //栈非空，逐层计算函数值，依次退栈，直到栈空
9         result = result * stack.top();
10        stack.pop();
11    }
12    return result;
13 }
```

## 4 队列的应用

树的遍历、图的广度优先遍历

- 1.
- 1.

## 四、串

### 1 串的定义

## 1.1 串的定义

**定义：**串(string)是由0个或多个字符组成的有限序列。一般记为  $S = 'a_1a_2 \cdots a_n' (n \geq 0)$

其中S是串名，单引号中的内容是串的值； $a_i$ 可以是字母、数字或其他字符；串中字符的数目n称为串的长度。n=0时的串称为空串（用  $\emptyset$  表示）。

**子串：**串中任意个连续的字符组成的子序列。（任意的意思是可以为0）

**主串：**包含子串的串相应的称为主串。

**字符在主串中的位置：**字符在串中的序号。

**子串在主串中的位置：**子串的第一个字符在主串中的位置。

**两个串相等：**当且仅当两个串的值相等。

**空格串：**由一个或多个空格组成的串“ ”称为空格串。

注意：空格串不是空串。

## 1.2 串的特点

字符串一般简称为串。串是一种特殊的线性表，数据元素之间呈线性关系，其特殊性体现在数据元素是一个字符，也就是说，串是一种内容受限的线性表。

## 1.3 串的抽象数据类型

```
1  ADT String{
2      数据对象:  $D=\{a_i | a_i \in \text{CharacterSet}, i=1, 2, \dots, n, n \geq 0\}$ 
3      数据关系:  $R_1=\{<a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=1, 2, \dots, n\}$ 
4      基本操作:
5          StrAssign(%T, chars) //赋值操作
6              初始条件: chars是字符串常量
7              操作结果: 生成一个其值等于chars的串T
8          StrCopy(&T, S) //复制
9              初始条件: 串S存在
10             操作结果: 有串S复制得串T
11          StrEmpty(S) //判空
12              初始条件: 串S存在
13              操作结果: 若S为空串, 则返回true, 否则返回false
14          StrCompare(S, T) //比较
15              初始条件: 串S和T存在
16              操作结果: 若S>T, 则返回值>0; 若S=T, 则返回值=0, 若S<T, 则返回值<0
17          StrLength(S) //求串长
18              初始条件: 串S存在
19              操作结果: 返回串的元素个数, 称为串的长度
20          ClearString(&S) //清空串
21              初始条件: 串S存在
22              操作结果: 将S清为空串
23          Concat($T, S1, S2) //串联接
24              初始条件: 串S1和串S2存在
25              操作结果: 用T返回由S1和S2连接而成的新串
26          SubString(&Sub, S, pos, len) //获取子串
27              初始条件: 串S存在,  $1 \leq \text{pos} \leq \text{StrLength}(S)$  且  $0 \leq \text{len} \leq \text{StrLength}(S) - \text{pos} + 1$ 
28              操作结果: 用Sub返回串S的第pos个字符起长度为len的子串
29          Index(S, T) //定位子串
30              初始条件: 串S和T存在, T是非空串
```

```

31      操作结果：若主串S中存在和串T值相同的子串，则返回它在主串S中第一次出现的位置；
      否则函数值为0
32      Replace(&S,T,V)//替换子串
33      初始条件：串S,T,V存在,T是非空串
34      操作结果：用V替换主串S中出现的所有与T相等的不重叠的字串
35      StrInsert(&S,pos,T)//插入子串
36      初始条件：串S和T存在，1≤pos≤StrLength(S)+1
37      操作结果：在串S的第pos个字符之前插入串T
38      StrDelete(&S,pos,len)//删除子串
39      初始条件：串S存在，1≤pos≤StrLength(S)-len+1
40      操作结果：从串S中删除第pos个字符起长度为len的子串
41      DestroyString(&S)//销毁串
42      初始条件：串S存在
43      操作结果：串S被销毁
44  }

```

### 字符串比较：

1. 两个串逐字符对比，先出现较大字符那个串就大
2. 长串的前缀与短串相同时，长的那个串大

## 1.4 字符集

1. ASCII字符集——英文字符——八个比特位表示
2. Unicode字符集——中英文

基于相同的字符集可以有多个不同的编码方式，比如：UTF-8、UTF-16等。

采用不同的编码方式，每个字符所占的空间不同，考研默认每个字符1B即可。

## 2 串的实现

### 2.1 串的顺序存储

串的定长顺序存储结构：

```

1  #define MAXLEN 255//串的最大长度
2  typedef struct{
3      char ch[MAXLEN];
4      int length;
5  }SString;

```

串的堆式顺序存储结构：

```

1  typedef struct{
2      char *ch;//动态数组（堆分配存储）
3      int length;
4  }HString;

```

不同的实现方式：（选择方法4）

## 串的顺序存储



## 2.2 串的链式存储

结构1:

```
1 typedef struct StringNode{
2     char ch;
3     struct StringNode *next;
4 }StringNode,*String;
```

缺点: 存储密度低, 每个字符1B, 每个指针4B

结构2: (提高存储密度)

```
1 #define CHUNKSIZE 80//由用户定义的块大小
2 typedef struct Chunk{
3     char ch[CHUNKSIZE];
4     struct Chunk *next;
5 }Chunk;
6 typedef struct{
7     Chunk *head,*tail;//串的头、尾指针
8     int length;//串的当前长度
9 }LString;
```

注: 如果某个结点的数组存不满, 可以用特殊字符填充。

## 2.3 基本操作的实现

### 2.3.1 求子串 SubString(&Sub,S,pos,len)

```

1  bool SubString(SString &Sub,SString S,int pos,int len){
2      if((pos+len-1)>S.length)
3          return false;
4      for(int i=pos;i<pos+len;i++)
5          Sub.ch[i-pos+1]=S.ch[i];
6      Sub.length=len;
7      return true;
8  }

```

### 2.3.2 比较 StrCompare(S,T)

```

1  //比较操作。若S>T,则返回值>0;若S=T,则返回值=0,若S<T,则返回值<0
2  int StrCompare(SString S,SString T){
3      for(int i=1;i<=S.length&&i<T.length;i++){
4          if(S.ch[i]!=T.ch[i])
5              return S.ch[i]-T.ch[i];
6      }
7      return S.length-T.length;//扫描过的所有字符都相同，则长度大的串更大
8  }

```

### 2.3.3 定位 Index(S,T)

```

1  //定位子串。若主串S中存在和串T值相同的子串，则返回它的主串S中第一次出现的位置;否则函数值为0
2  int Index(SString S,SString T){
3      int i=1;
4      SString Sub;
5      while(i<S.length-T.length+1){
6          SubString(Sub,S,i,T.length);
7          if(StrCompare(Sub,T)!=0)
8              ++i;
9          else
10             return i;//返回子串在主串中的位置
11     }
12     return 0;//没有相同子串
13 }

```

## 3 串的模式匹配算法

子串的**定位操作**通常称为串的**模式匹配**或**串匹配**。

模式匹配中有两个字符串S和T，S称为**主串**，也称为**正文串**；T为**子串**，也称为**模式**。

### 3.1 BF(Brute-Force)算法

BF算法为暴力算法，又称为朴素模式匹配算法。

**算法步骤：**

1. 分别利用计数指针i和j指示主串S和模式T中当前正待比较的字符位置，i初值为pos，j初值为1。
2. 如果两个串均未比较到串尾，即i和j均分别小于等于S和T的长度时，则循环执行以下操作：
  1. S.ch[i]和T.ch[j]比较，若相等，则i和j分别指示串中下一个位置，继续比较后续字符。

2. 若不相等，指针后退重新开始匹配，从主串的下一个字符( $i=i-j+2$ )起再重新和模式的第一个字符( $j=1$ )比较。
3. 如果 $j>T.length$ ，说明模式T中的每个字符依次和主串S中的一个连续的字符序列相等，则匹配成功，返回模式T中的第一个字符相等的字符在主串S中的序号( $i-T.length$ )；否则称匹配不成功，返回0。

```
1 //指定主串查找的起始位置
2 int Index_BF(SString S,SString T,int pos){
3     int i=pos,j=1;
4     while(i<S.length&& j<T.length){
5         if(S.ch[i]==T.ch[j]){
6             i++;
7             j++;
8         }
9         else{
10            i=i-j+2;
11            j=1;
12        }
13    }
14    if(j>T.length)
15        return i-T.length;
16    else
17        return 0;
18 }
```

从头开始扫描：

```
1 int Index_BF(SString S,SString T){
2     int i=1,j=1;
3     while(i<S.length&& j<T.length){
4         if(S.ch[i]==T.ch[j]){
5             ++i;
6             ++j;
7         }
8         else{
9             i=i-j+2;
10            j=1;
11        }
12    }
13    if(j>T.length)
14        return i-T.length;
15    else
16        return 0;
17 }
```

**时间复杂度：**主串长度为 $n$ ，模式长度为 $m$ ，最坏时间复杂度 $O(nm)$

## 3.2 KMP 算法

### 3.2.1 KMP算法原理

Knuth-Morris-Pratt 字符串查找算法。

**前缀：**除最后一个字符外，字符串的所有头部子串。（所有头部子串的集合）

**后缀：**除第一个字符外，字符串的所有尾部子串。（所有尾部子串的集合）



**部分匹配值：**字符串的前缀和后缀的最长相等前后缀长度。（前缀和后缀交集中最长的那个的长度）

KMP算法需要根据模式串T，计算出next数组，next数组中存放的是每个长度的部分匹配值，KMP算法利用next数对模式串组进行回溯，但是主串不回溯。

KMP算法，**最坏时间复杂度：** $O(m+n)$ ，求next数组时间复杂度 $O(m)$ ，模式匹配最坏时间复杂度 $O(n)$ 。

**代码实现：**

```
1  int Index_KMP(HString S, HString T) {
2      int i = 1, j = 1;
3      int next[T.length + 1];
4      get_next(T, next);
5      while (i <= S.length && j <= T.length) {
6          if (j == 0 || S.ch[i] == T.ch[j]) {
7              ++i;
8              ++j;
9          } else {
10             j = next[j];
11         }
12     }
13     if (j > T.length)
14         return i - T.length;
15     else
16         return 0;
17 }
```

### 3.2.2 next数组

#### 1) 手算

先填写：next[1]=0,next[2]=1

后续：在不匹配的位置前上画一条线，模式串一步一步后退，知道分界线之前能“对上”，或模式串完全退到分界线后，也就是不匹配的位置。此时j指向哪儿，next数组值就是多少。

#### 2) 实现

```
1  void get_next(HString T, int next[]) {
2      if (T.length == 0)
3          exit(-2);
4      int i = 1, j = 0;
5      next[1] = 0;
6      while (i < T.length) {
7          if (j == 0 || T.ch[i] == T.ch[j]) {
8              ++i;
9              ++j;
10             next[i] = j;
11         } else {
12             j = next[j];
13         }
14     }
15 }
```

### 3.2.3 nextval数组代码实现

```
1 void get_next_val(HString T, int nextval[]) {
2     if (T.length < 0)
3         return;
4     int i = 1, j = 0;
5     nextval[1] = 0;
6     while (i <= T.length) {
7         if (j == 0 || T.ch[i] == T.ch[j]) {
8             ++i;
9             ++j;
10            if (T.ch[i] != T.ch[j])
11                nextval[i] = j;
12            else
13                nextval[i] = nextval[j];
14        } else {
15            j = nextval[j];
16        }
17    }
18 }
```

## 五、数组

### 5.1 数组的定义

**数组：**数组是由类型相同的数据元素组成的有序集合，每个元素称为数组元素。

### 5.2 数组的存储结构

#### 5.2.1 一维数组

```
1 ElemType arr[10];
```

内存：各数组元素大小相同，且物理上连续存放。已知起始地址，数组各元素的物理位置可以直接计算出来。

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]	arr[9]
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

起始地址LOC=arr[0]地址，则a[i]存放地址=LOC+i\*sizeof(ElemType);

注：除非特别说明，下表默认从0开始。

```
1 例： int arr[10]={0,1,2,3,4,5,6,7,8,9};
2      int *p = arr;// *p = 0;
3      int *q=arr+3;// *q = 4;
```

#### 5.2.2 二维数组

```
1 ElemType arr[2][3]={0,1,2,3,4,5}; //两行三列的二维数组
```

逻辑视角：

arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]

内存：（行优先存储）

arr[0][0]	arr[0][1]	arr[0][2]	arr[1][0]	arr[1][1]	arr[1][2]
-----------	-----------	-----------	-----------	-----------	-----------

M行N列二维数组arr[M][N]

arr[i][j]的存储地址=LOC+(i\*N+j)\*sizeof(ElemType);

```
1 int arr[2][3]={0,1,2,3,4,5};
2 int i=j=1;
3 int *p=&arr[0][0]+i*N+j; // *p=4
```

内存：（列优先存储）

arr[0][0]	arr[1][0]	arr[0][1]	arr[1][1]	arr[0][2]	arr[1][2]
-----------	-----------	-----------	-----------	-----------	-----------

M行N列二维数组arr[M][N]

arr[i][j]的存储地址=LOC+(j\*M+i)\*sizeof(ElemType);

```
1 int arr[2][3]={0,1,2,3,4,5};
2 int i=j=1;
3 int *p=&arr[0][0]+j*M+i; // *p=4
```

## 5.3 特殊矩阵的压缩存储

普通矩阵：

可用二维数组存储。

注：矩阵的下表从1开始，数组的下标从0开始

特殊矩阵：

下列压缩策略中计算矩阵到数组的映射函数，可能需要用到等差数列的求和公式

$$S_n = n * a_1 + \frac{n*(n-1)}{2} * d \text{ 或者 } S_n = \frac{n*(a_1+an)}{2}$$

### 5.3.1 对称矩阵

若n阶方阵中任意一个元素 $a_{ij}$ ，都有 $a_{ij}=a_{ji}$ ，责成该局真伪对称矩阵。

普通存储：n\*n二维数组。

压缩存储策略：

- 只存储主对角线+下三角区(i>j)
  - 按行优先原则将各元素存入一维数组中。

$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{3,1}$	.....	$a_{n,n-1}$	$a_{n,n}$
-----------	-----------	-----------	-----------	-------	-------------	-----------

- 数组大小为：1+2+3+.....+n= $\frac{n*(n+1)}{2}$

- **使用方法：**构造一个映射函数，通过矩阵下标计算出数组下标。

$a_{i,j} \rightarrow \text{arr}[k]$

按照行优先原则， $a_{i,j}$ 是第 $1+2+3+\dots+(i-1)+j = \frac{i*(i-1)}{2} + j$ 个元素

$$k = \frac{i*(i-1)}{2} + j - 1$$

- 主对角线+上三角区( $i < j$ )

方法和上面类似。

### 5.3.2 三角矩阵

- 下三角矩阵：除了主对角线和下三角区，其余元素均为常量c。
  - 压缩存储策略：按行优先原则将下三角矩阵存入一维数组，并在最后一个位置存储常量c。

○	$a_{1,1}$	$a_{2,1}$	$a_{2,2}$	$a_{3,1}$	.....	$a_{n,n-1}$	$a_{n,n}$	c
---	-----------	-----------	-----------	-----------	-------	-------------	-----------	---

- **数组大小：** $1 + 2 + 3 + \dots + n + 1 = \frac{n*(n+1)}{2} + 1$
- **使用方法：**构造映射函数

- 上三角矩阵：与上面类似

### 5.3.3 三对角矩阵

**三对角矩阵：**又称为带状矩阵，当 $|i - j| > 1$ 时，有 $a_{i,j}=0(1 \leq i, j < n)$

- **压缩策略：**按行优先原则（或列优先原则），只存储带状部分
- **数组大小：** $2 + 3 * (n - 2) + 2 = 3 * n - 2$
- 已知矩阵元素下标，计算 $a_{i,j}$ 在数组中的位置，数组下标从0开始

前 $i - 1$ 行共有 $3 * (i - 1) - 1$ 个元素

$a_{i,j}$ 是第 $i$ 行第 $j - i + 2$ 个元素

$a_{i,j}$ 是第 $2 * i + j - 2$ 个元素

$$k = 2 * i + j - 3$$

- 已知数组下标k，计算矩阵元素下标 $a_{i,j}$

明显可知 $3 * (i - 1) - 1 < k + 1 \leq 3 * i - 1$

$i \geq \frac{k+2}{3}$  可以理解为“刚好”大于等于

向上取整  $i = \lceil \frac{k+2}{3} \rceil$  ,  $j = k - 2 * i + 3$

### 5.3.4 稀疏矩阵

**稀疏矩阵：**非零元素个数远远小于矩阵元素的个数

**压缩策略：**

- 顺序存储——三元组<行，列，值>

```

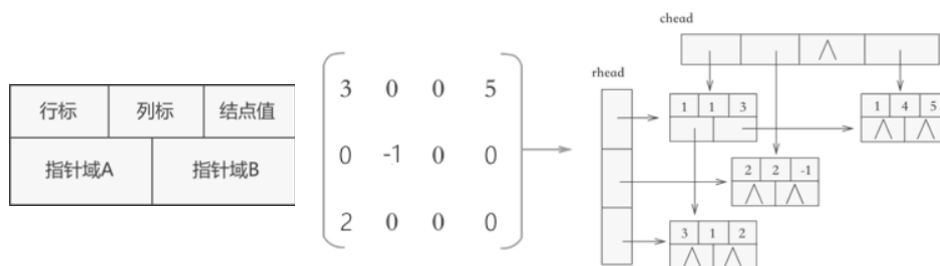
1 struct node{
2     int i;
3     int j;
4     int value;
5 };
6 struct node arr[n];
  
```

- 这种方式是访问稀疏矩阵只能顺序存取，失去了随机存取的特性
- 十字链表法——链式存储

```

1 //结点
2 typedef struct OLNode{
3     int i,j;//行标和列标
4     int data;//数据
5     struct OLNode *right,*down;//右指针和下指针
6 }OLNode,*OLink;
7 //十字链表结构体
8 typedef struct{
9     LONode *rhead,*chead;//行和列表头指针
10    int m,n,count;//行数、列数、非零元素个数
11 }

```



## 5.4 矩阵考点易错点

1. 矩阵的压缩存储需要的数组大小
2. 由矩阵元素的行标和列标*<i,j>*推导出对应的数组下标*k*（数列求和）
3. 由数组下标*k*，推导出*<i,j>*
  1. 如何处理不等式中的“刚好大于等于/小于等于”
  2. 向上取整/向下取整
4. 易错点：
  1. 存储上三角？下三角
  2. 行优先存储？列优先存储？
  3. 矩阵下标从0？1？开始
  4. 数组下标从0？1？开始

# 六、树

## 6.1 树的定义及相关概念

### 6.1.1 定义

**树：**树是 $n(n \geq 0)$ 个结点的有限集。

**空树：**结点数为0的数，即 $n = 0$ ；

**非空树应满足：**

1. 有且仅有一个称之为根节点。
2. 除根节点以外的其余结点可分为 $m(m > 0)$ 个互不相交的有限集 $T_1, T_2, \dots, T_m$ ，其中每一个集合本身又是一棵树，并且称为根的子树。

除了根结点外，每个结点有且仅有一个前驱。

## 6.1.2 树的基本术语

**结点：**树中的一个独立单元。包含一个数据元素及若干指向其子树的分支。

**结点的度：**结点拥有的子树称为结点的度。

**结点的高度：**从下往上数。

**树的度：**树的度是树内各结点度的最大值。

**树的深度：**树中结点的最大层次称为树的深度或高度。

**叶子：**度为0的结点称为叶子或终端结点。

**非终端节点：**度不为0的结点称为非终端结点或分支结点。除根结点外，非终端结点也称为内部结点。

**双亲和孩子：**结点的子树称为该结点的孩子，相应的该结点称为孩子的双亲。

**兄弟：**同一个双亲的孩子之间互称兄弟。

**祖先：**从根到该结点所经历分支上的所有结点。

**子孙：**以某结点为根的子树中的任一结点都称为该结点的子孙。

**层次：**结点的层次从根开始定义起，根为第一层，根的孩子为第二层。树中任意结点的层次等于其双亲结点的层次加1。

**堂兄弟：**双亲在同一层的，不互为兄弟的结点互为堂兄弟。

**有序树和无序树：**如果将树中结点的各子树看成从左至右是有次序的(即不能互换)，则称该树为有序树，否则成为无序树。有序树中最左边的子树的根称为第一个孩子，最右边的称为最后一个孩子。

**森林：** $m(m \geq 0)$ 棵互不相交的树的集合。对树中的每个结点而言，其子树的集合即为森林。

**结点之间的路径：**只能从上往下

**路径长度：**经过了几条边

## 6.1.3 树和森林的关系

$$RF = \langle root, r_i \mid i = 1, 2, 4, m, m > 0 \rangle$$

## 6.1.4 常考性质

1. 结点数=总度数+1
2. 度为m的树，和m叉树的区别

度为m的树	m叉树
任意结点的度 $\leq m$ (最多m个孩子)	任意结点的度 $\leq m$ (最多m个孩子)
至少有一个结点的度 = m (有m个孩子)	允许所有结点的度都 $< m$
一定是非空树，至少有m+1个结点	可以是空树

3. 度为m的树第i层至多有 $m^{(i-1)}$ 个结点( $i \geq 1$ )
4. 高度为h的m叉树至多有 $\frac{m^h-1}{m-1}$ 个结点
5. 高度为h的m叉树至少有 $h + m - 1$ 个结点
6. 具有n个结点的m叉树的最小高度为 $\log_m(n(m-1) + 1)$

## 6.2 二叉树

### 6.2.1 二叉树的定义

**二叉树**：是 $n(n \geq 0)$ 个结点所构成的集合。空树 $n=0$ ；非空树应满足：

1. 有且仅有一个根结点。
2. 除根结点外，其余结点分为两个互不相交的子集 $T_1$ 和 $T_2$ ，分别称为 $T$ 的左子树和右子树，且 $T_1$ 和 $T_2$ 本身又是二叉树。

**二叉树与树的区别**：

1. 二叉树每个结点最多只有两棵子树。
2. 二叉树的子树有左右之分，不能颠倒。

### 6.2.2 两种特殊的二叉树

**满二叉树**：深度为 $k$ 且含有 $2^k - 1$ 个结点的二叉树。

**特点**：

1. 每一层上的结点数都是最大结点数。
2. 只有最后一层有叶子结点。
3. 不存在度为1的结点。
4. 按层序从1开始编号，结点 $i$ 的做孩子为 $2i$ ，有孩子为 $2i + 1$ ；结点 $i$ 的父节点为 $\lfloor \frac{i}{2} \rfloor$

**完全二叉树**：深度为 $k$ 的，有 $n$ 个结点的二叉树，当且仅当其每一个结点都与深度为 $k$ 的满二叉树中编号为1至 $n$ 的结点一一对应时，称之为完全二叉树。

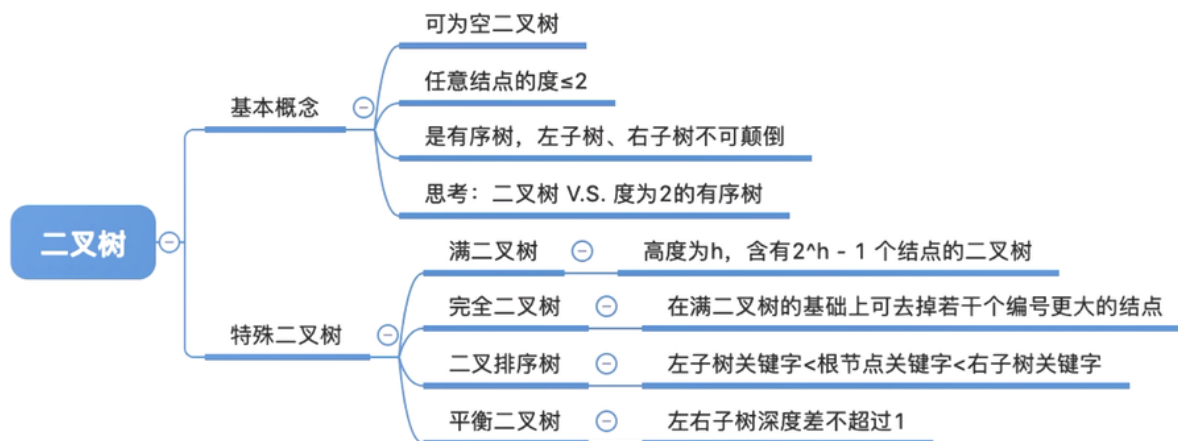
**特点**：

1. 叶子结点只可能在层次最大的两层上出现。
2. 对任意结点，若其右分支下的子孙的最大层次为 $l$ ，则其左分支下的子孙最大层次必为 $l$ 或 $l+1$ 。
3. 最多只有一个度为1的结点。
4. 按层序从1开始编号，结点 $i$ 的做孩子为 $2i$ ，有孩子为 $2i + 1$ ；结点 $i$ 的父节点为 $\lfloor \frac{i}{2} \rfloor$

**二叉排序树**：一棵二叉树或是空二叉树，或是具有以下性质的二叉树：

1. 左子树上所有结点的关键字均小于根结点的关键字。
2. 右子树上所有结点的关键字均大于根结点的关键字。
3. 左子树和右子树又各是一棵二叉排序树。

**平衡二叉树**：树上任一结点的左子树和右子树的深度只差不超过1。



### 6.2.3 二叉树的性质

**性质1:** 在二叉树的第 $i$ 层上至多有 $2^{(i-1)}$ 个结点( $i \geq 1$ )

**性质2:** 深度为 $k$ 的二叉树至多有 $2^k - 1$ 个结点( $k \geq 1$ )

**性质3:** 对任何一棵二叉树 $T$ , 如果其终端结点数为 $n_0$ , 度为2的结点数为 $n_2$ , 则 $n_0 = n_2 + 1$

**性质4:** 具有 $n$ 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$

**性质5:**