数据结构核心考点

0. 绪论

0.1 数据结构基本概念

0.1 基本概念和术语

术语	定义
数据	数据是信息的载体，是描述客观事物属性的数、 字符及所有能输入到计算机中并被计算机程序识别和处理的符号的集合
数据元素	数据元素是数据的基本单位，通常作为一个整体进行考虑和处理，含有多个数据项
数据项	是构成数据元素的不可分割的最小单位
数据对象	具有相同性质的数据元素的集合，是数据的一个子集
数据类型	一个值的集合和定义在此集合上的一组操作的总称 {原子类型、结构类型、抽象数据类型}
数据结构	相互之间存在一种或多种特定关系的数据元素的集合

数据类型

数据类型	定义
原子类型	其值不可再分的数据类型
结构类型	其值可以再分解为若干成分的数据类型
抽象数据类型 ADT	抽象数据组织及与之相关的操作

抽象数据类型的定义格式

```
ADT 抽象数据类型名{//抽象数据类型定义格式
        数据对象:<数据对象的定义>          //自然语言
        数据关系:<数据关系的定义>          //自然语言
        基本操作:<基本操作的定义>
}ADT 抽象数据类型名;

//基本操作的定义格式
基本操作名(参数表)        //参数表中赋值参数只提供输入值，引用参数以&打头，可提供输入值和返回操作结果
    初始条件:<初始条件描述>
    操作结果:<操作结果描述>
```

0.1 数据结构三要素

逻辑结构

定义：逻辑结构是指数据元素之间的逻辑关系，即从逻辑关系上描述数据。

分类：

- 线性结构：一般线性表、受限线性表（栈和队列）、线性表推广（数组）
- 非线性结构：集合结构、树结构、图结构

存储结构

定义：存储结构是指数据结构在计算机中的表示，也称物理结构

分类：

存储结构	定义	优点	缺点
顺序存储	把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来体现	随机存取， 占用空间少	使用一整块相邻的存储单元， 产生较多碎片
链式存储	不要求逻辑上相邻的元素在物理位置上也相邻， 借助指示元素存储地址的指针来表示元素之间的逻辑关系	不会出现碎片， 充分利用所有存储单元	需要额外空间，只能顺序存取
索引存储	在存储元素信息的同时，还建立附加的索引表。	检索速度快	附加的索引表需要额外空间。 增删数据修改索引表时花费时间
散列存储	根据元素的关键字直接计算出该元素的存储地址，又称哈希(Hash)存储。	检索、 增加和删除结点的操作很快	可能出现元素存储单元的冲突， 解决冲突会增加时间和空间开销

数据的运算

定义：施加在数据上的运算包括运算的定义和实现。

- 定义是针对逻辑结构的，指出运算的功能；
- 运算的实现是针对存储结构的，指出运算的具体操作步骤。

0.2 算法和算法评价

算法的定义、特性和评价标准

定义：算法是针对特定问题求解步骤的一种描述，它是指令的有限序列，其中的每条指令表示一个或多个操作。

特性：

- 输入，零个或多个
- 输出，一个或多个
- 确定性，每条指令含义确定，相同的输入得出相同的结果
- 有穷性
- 可行性，所有操作可以通过已经实现的基础运算操作有限次来实现

评价标准:

- 正确性：正确结果
- 可读性
- 健壮性：输入数据非法时，能够适当的作出反应或相应处理，不会产生莫名其妙的输出结果
- 高效性：时间和空间

算法效率的度量

- 算法效率分为时间效率和空间效率

- 时间复杂度定义: $T(n) = O(f(n)$
- 空间复杂度定义: $S(n) = O(g(n))$
- 函数渐近的界$O(g(n))$,存在正数c和n_0使得对于一切$n \geq n_0, 0 \leq f(n) \leq cg(n)$

注意:

- 原地工作的算法的空间复杂度为O(1);
- 算法的时间复杂度不仅仅依赖于数据的规模，也取决于待输入数据的性质，如数据的初始状态;

算法复杂度分析步骤

- 确定表示输入规模的参数
- 找出算法的基本操作
- 检查基本操作的执行次数是否只依赖于输入规模。这决定是否需要考虑最差、平均以及最优情况下的复杂性
- 对于非递归算法，建立算法基本操作执行次数的求和表达式;
- 对于递归算法，建立算法基本操作执行次数的递推关系及其初始条件，利用求和公式和法则建立一个操作次数的闭合公式，或者求解递推公式，确定增长的阶

加法法则:

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(max(f(n), g(n)))$$

乘法法则:

$$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n)) \times O(g(n))$$

常见的复杂度:

$$O(1) \leq O(log_2 n) \leq O(n) \leq O(nlog_2 n) \leq O(n^2)$$
$$\leq O(n^3) \leq O(2^n) \leq O(n!) \leq O(n^n)$$

两类递归算法问题的复杂度求解:

- 线性分解

$$T(n) = \begin{cases} O(1) & n = 1 \\ aT(n-1) + f(n) & n > 1 \end{cases}$$

$$T(n) = a^{n-1}T(1) + \sum_{i=2}^{n} a^{n-i} f(i)$$

- 指数分解

$$T(n) = \begin{cases} O(1) & n = 1 \\ aT(\frac{n}{b}) + f(n) & n > 1 \end{cases}$$

$$T(n) = n^{log_b a}T(1) + \sum_{j=0}^{log_b n - 1} a^j f(\frac{n}{b^j})$$

1. 线性表

1.1 线性表的定义和基本操作

定义

线性表是具有**相同数据类型**的 n 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 $L=(a_1,a_2,...,a_i,a_{i+1},...,a_n)$。

特点

- a_1是唯一的 “第一个” 数据元素，又称表头元素
- a_n是唯一的 “最后一个” 数据元素，又称表尾元素
- 除第一个元素外，每个元素有且仅有一个直接前驱。除最后一个元素外，每个元素有且仅有一个直接后继。

基本操作

```
InitList(&L);      //初始化表：构造一个空的线性表L，分配内存空间
DestoryList(&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

ListInsert(&L,i,e);  //插入操作：在表L中第i个位置上查入指定元素e
ListDelete(&L,i,&e); //删除操作：删除表L中第i个位置的元素，/*并用e反回删除元素的值*/

LocateElem(L,e); //按值查找操作
GetElem(L,i);    //按位查找操作

//其它常用操作
Length(L);  //求表长
Print(L);   //输出操作
Empty(L);   //判空操作
```

1.2 线性表的顺序表示

顺序表的定义

线性表的顺序存储又称顺序表。它是用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。顺序表的特点是表中元素的逻辑顺序与物理顺序相同。

- 线性表 A 中第 i 个元素的内存地址： `&(A[0])+i*sizeof(ElemType)`
- 一维数组可以是静态分配，也可以动态分配
- 静态分配时，数组的大小和空间事先已经固定，一旦空间占满，再加入新的数据就会产生溢出，进而导致程序崩溃
- 动态分配时，存储数组的空间是在程序执行过程中通过动态存储分配语句分配的，一旦数据空间占满，就另外开辟一块更大的存储空间，用以替换原来的存储空间。

静态分配的实现

```
#define MaxSize 50                //定义线性表的最大长度
typedef struct{
    ElemType data[MaxSize];        //顺序表的元素
    int length;                    //顺序表的当前长度
}SqList;                          //顺序表的类型定义
```

动态分配的实现

```
#define InitSize 100            //表长度的初始定义
typedef struct{
    ElemType *data;                 //指示动态分配数组的指针
    int MaxSize,length;             //数组的最大容量和当前个数
}SqList;                        //动态分配数组顺序表的类型定义
//C的初始动态分配语句
L.data = (ElemType*)malloc(sizeof(ElemType)*InitSize);
free(L);

//C++的初始动态分配语句
L.data = new ElemType[InitSize];
delete L;
```

特点

- 随机访问
- 存储密度高
- 插入删除需要移动大量元素

顺序表的实现

注意算法对i的描述是第i个元素，它是以1为起点的

插入

```
//插入操作：在顺序表L的第i个(位序)上插入x
bool ListInsert(SqList &L, int i,int e){
    if(i<1||i>L.length+1)               //判断i的范围是否有效
        return false;
    if(L.length>=MaxSize)               //当存储空间已满时，不能插入
        return false;
    for(int j=L.length; j>=i; j--)
        L.data[j]=L.data[j-1];  //将第i个及后面的元素后移
    L.data[i-1]=e;                              //将e放到第i个位置
    L.length++;                 //长度+1
}
```

插入的时间复杂度：

最好情况：插到表尾，不需移动元素，循环0次， 最好时间复杂度 =O(1)

最坏情况：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：设插入概率为p=1/n+1，则循环np+(n-1)p+…+1p=n/2， 平均时间复杂度 =O(n)

删除

```
//删除操作：删除顺序表L中第i个元素并返回其元素值
bool ListDelete(SqList &L, int i,int &e){
    if(i<1||i>L.length+1){                  //判断i的范围是否有效
        return false;
    }else{
        e = L.data[i-1];                //将被删除的元素赋值给e
        for(int j=i; j<L.length; j++){
            L.data[j]=L.data[j-1];      //将第i个后面的元素前移
        }
        L.length--;                     //长度-1
        return ture;
    }
}
```

删除的时间复杂度：

最好情况：删除表尾，不需移动元素，循环0次， 最好时间复杂度 =O(1)

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

平均情况：设删除概率为p=1/n，则循环(n-1)p+(n-2)p+…+1p=(n-1)/2， 平均时间复杂度 =O(n)

查找

按位查找

```
//按位查找：返回顺序表中第i个元素的元素值
int GetElem(Sqlist L, int i){
    return L.data[i-1];
}
```

按位查找的时间复杂度

时间复杂度=O(1)

按值查找

```
//按值查找：返回顺序表L中第一个值为x的元素的位置
int LocateElem(Sqlist L, int e){
    for(int i=0; i<L.length; i++){
        if(L.data[i] == e)
            return i+1;  //返回元素位置
    }
    return -1;  //查找失败，返回-1
}
```

按值查找的时间复杂度

最好情况：目标在表头，循环1次， 最好时间复杂度 =O(1)

最坏情况：目标在表尾，循环n次， 最坏时间复杂度 =O(n)

平均情况：设删除概率为p=1/n，则循环(n-1)p+(n-2)p+…+1p=(n+1)/2， 平均时间复杂度 =O(n)

1.3 线性表的链式表示

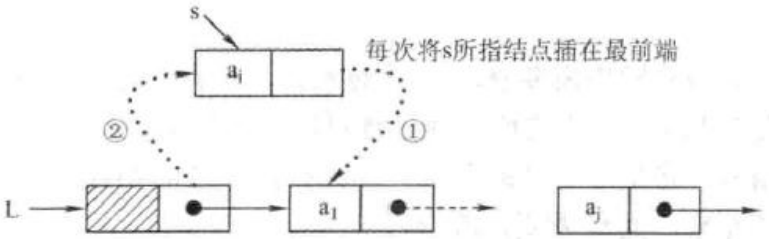
单链表

- 结点描述：

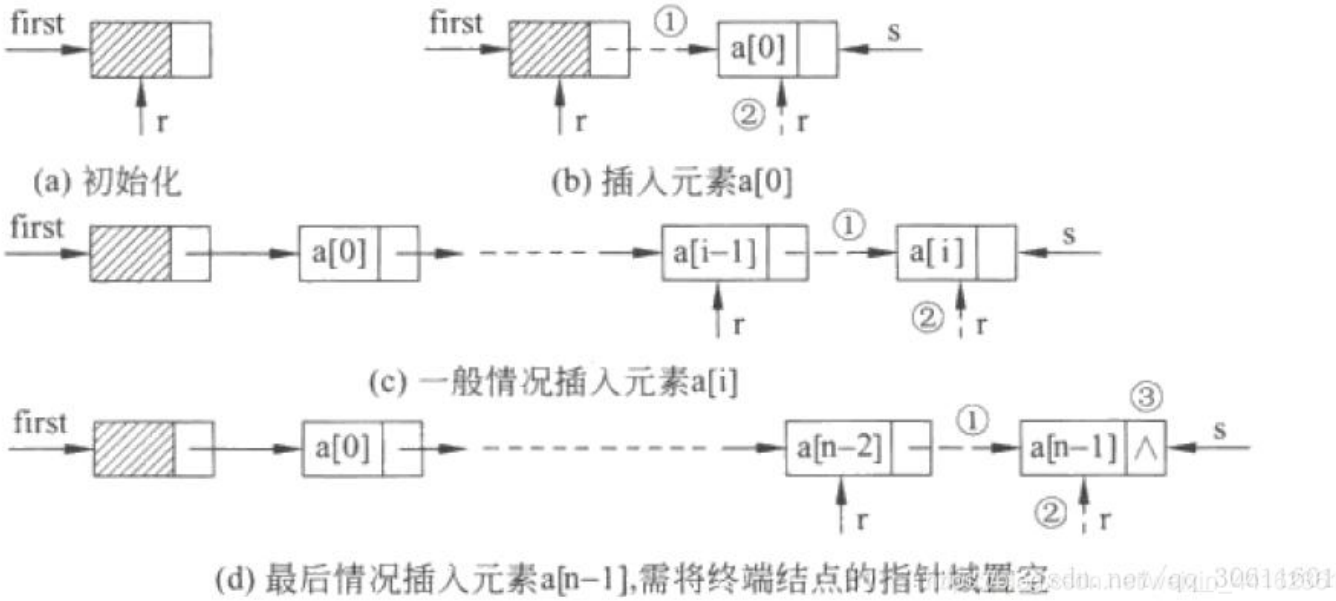
```
typedef struct LNode{        //定义单链表结点类型
    ElemType data;            //数据域
    struct LNode *next;        //指针域
}LNode, *LinkList;        //LinkList为指向结构体LNODE的指针类型
```

- 通常用头指针来标示一个单链表。
- 有头结点或者没头结点之分
- 头结点的作用
  - 便于首元结点的处理，对链表的第一个数据元素的操作与其他数据元素相同，无需特殊处理
  - 便于空表与非空表的统一处理：头指针永远不为空

单链表的实现




```
LinkList List_HeadInsert(LinkList &L){
    LNode *s;int x;
    L=(LinkList)malloc(sizeof(LNode));//创建头结点
    L->next = NULL;                    //初始为空链表
    scanf("%d",&x);
    while(x!=9999){
        s = (LNode*)malloc(sizeof(LNode));
        s->data = x;
        s->next = L->next;
        L->next = s;
        scanf("%d",&x);
    }
    return L;
}
```




```
LinkList List_TailInsert(LinkList &L){
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    LNode *s,*r=L;            //r为表尾指针
    scanf("%d",&x);
    while(x!=9999){
        s = (LNode *)malloc(sizeof(LNode));
        s->data = x;
        r->next = s;
        r = s;
        scanf("%d",&x);
    }
    r->next = NULL;                //尾结点指针置空
    return L;
}
```

```
LNode *LocateElem(LinkList L,ElemType e){//按值查找
    LNode *p = L->next;
    while(p!=NULL&&p->data!=e)
        p = p->next;
    return p;
}
```

删除

按位序删除

```
//删除操作:将单链表中的第i个结点删除
bool Delete(LinkList &L, int i int &e){
    if(i<1 || i>Length(L))
        return false;
    LNode *p = GetElem(L,i-1);  //查找第i个位置
    LNode *q = p->next;
    e = q->data;
    p->next = q->next;
    free(q);
    return true;
}
```

按位序删除的时间复杂度：

最好情况：删除第一个，不需查找位置，循环0次， 最好时间复杂度 =O(1)

最坏情况：删除最后一个，需查找第n位，循环n次， 最坏时间复杂度 =O(n)

平均情况：删除任意一个， 平均时间复杂度 =O(n)

指定结点的删除

时间复杂度 =O(n)

方法：p的后一个为q，p指向q的下一个，把q的值给p，最后释放q

```
//删除指定结点p
bool Delete(LNode *p){
    if(p==NULL) return false;
    LNode *q = p->next;
    p->data = q->data
    p->next = q->next;
    free(q);
    return true;
}
```

查找

按位查找

平均时间复杂度 =O(n)

```
//按位查找：查找在单链表L中第i个位置的结点
LNode *GetElem(LinkList L, int i){
    int j=0;
    LNode *p = L;
    if(i<0) return NULL;
    while(p && j<i){
        p = p->next;
        j++;
    }
    return p; //如果i大于表长，p=NULL,直接返回p即可
}
```

按值查找

平均时间复杂度 =O(n)

```
//按值查找：查找e在L中的位置
LNode *LocateElem(LinkList L, int e){
    LNode *p = L->next;
    while(p && p->data != e){
        p = p->next;
    }
    return p;
}
```

求表长

平均时间复杂度 =O(n)

```
//求表的长度
int Length(LinkList L){
    int len = 0;
    LNode *p = L;
    while(P->next){
        p = p->next;
        len++;
    }
    return len;
}
```

遍历

```
//遍历操作
void PrintList(LinkList L){
    LNode *p = L->next;
    while(p){
        printf("%d ", p->data);
        p = p->next;
    }
}
```

双链表

```
typedef struct DNode{
    ElemType data;
    struct DNode *prior,*next;    //前驱和后继指针
}DNode,*DLinkList;
```

循环链表

- 循环单链表

初始化和判空(与单链表不一样)

`L->next = NULL改为L->next = L`

- 循环双链表

初始化和判空(与双链表不一样)

```
L->next = NULL改为L->next = L
L->prior = NULL改为L->prior = L
```

静态链表

借助数组来描述线性表的链式存储结构，结点也有数据域 `data` 和指针域 `next` ，这里的指针是节点的相对地址（数组下标），又称**游标**

```
#define MaxSize 50
typedef struct{
    ElemType data;
    int next;
}SLinkList[MaxSize];
```

2. 栈和队列

2.1 栈

2.1.1 栈的基本概念

栈的定义

- 栈是只允许在一端进行插入或删除操作的线性表。后进先出 LIFO
- 栈顶（Top）：线性表允许进行插入删除的那一端。
- 栈底（Bottom）：固定的，不允许进行插入和删除的另一端
- 空栈：不包含任何元素的空表

栈的基本操作

```
InitStack(&S):初始化一个空栈S
StackEmpty(S):判断一个栈是否为空，若栈S为空则返回true，否则返回false
Push(&S,x):进栈，若栈S未满，则将x加入使之成为新栈顶
Pop(&S,&x):出栈，若栈S非空，则弹出栈顶元素，并用x返回
GetTop(S,&x):读取栈顶元素，若栈S非空，则用x返回栈顶元素
DestroyStack(&S):销毁栈，并释放栈S占用的存储空间
```

2.1.2 栈的顺序存储结构

顺序栈的实现

利用一组地址连续的存储单元存放自栈底到栈顶的数据元素，并附设一个指针 `top` 指示当前栈顶元素的位置

```
#define MaxSize 50            //定义栈中元素最大个数
typedef struct{
    Elemtype data[MaxSize];    //存放栈中元素
    int top;//栈顶指针
}
```

- 栈顶指针： `S.top` ，初始时设置 `S.top=-1` ； 栈顶元素： `S.data[S.top]`
- 进栈操作：栈不满时，栈顶指针先加 1，再送值到栈顶元素
- 出栈操作：栈非空时，先取栈顶元素值，再将栈顶指针减 1
- 栈空条件： `S.top==-1` ； 栈满条件： `S.top==MaxSize-1` ； 栈长： `S.top+1`

共享栈

利用栈底位置相对不变的特性，可让两个顺序栈共享一个一维数组空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶共享空间的中间延伸。

- 两个栈的栈顶指针都指向栈顶元素
- `top0=-1` 时 0 号栈为空， `top1=MaxSize` 时 `1` 号栈为空
- `top1-top0==1` 为栈满
- 当 `0` 号栈进栈时 `top0` 先加 1 再赋值， `1` 号栈进栈时 `top1` 先减 1 再赋值；出栈是刚好相反

2.1.3 栈的链式存储结构

采用链式存储的栈称为**链栈**，链栈的优点是便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况。这里规定链栈没有头结点， `Lhead` 指向栈顶元素

```
typedef struct Linknode{
    ElemType data;//数据域
    struct Linknode *next;//指针域
} *LiStack;//栈类型定义
```

2.2 队列

2.2.1 队列的基本概念

队列的定义

- 队列简称队，也是一种操作受限的线性表，只允许在表的一端进行插入，而在表的另一端进行删除。
- 向队列中插入元素称为**入队**或**进队**
- 删除元素称为**出队**或**离队**
- 操作的特性是先进先出

队列常见的基本操作

```
InitQueue(&Q):初始化队列，构造一个空队列Q
QueueEmpty(Q):判队列空
EnQueue(&Q,x):入队，若队列Q非满，将x加入，使之成为新的队尾
DeQueue(&Q,&x):出队，若队列Q非空，删除队头元素，并用x返回
GetHead(Q,&x):读队头元素，若队列Q非空，则将队头元素赋值给x
```

2.2.2 队列的顺序存储结构

队列的顺序存储

队列的顺序实现是指分配一块连续的存储单元存放队列中的元素，并附设两个指针：队头指针 `front` 指向队头元素，队尾指针 `rear` 指向队尾元素的下一个位置

```
#define MaxSize 50//定义队列中元素的最大个数
typedef struct{
    ElemType data[MaxSize];//存放队列元素
    int front,rear;//队头指针和队尾指针
} SqQueue;
```

- 初始状态： `Q.front==Q.rear==0`
- 进队操作：队不满时，先送值到队尾元素，再将队尾指针加 1
- 出队操作：队不空时，先取队头元素值，再将队头指针加 1

循环队列

将顺序队列臆造为一个环状的空间，即把存储队列元素的表从逻辑上视为一个环，称为循环队列。当队首指针 `Q.front=MaxSize-1` 后，再前进一个位置就自动到 0，这可以利用除法取余运算 `%` 来实现

- 初始状态： `Q.front=Q.rear=0`
- 队首指针进 1： `Q.front=(Q.front+1)%MaxSize`
- 队尾指针进 1： `Q.rear=(Q.rear+1)%MaxSize`
- 队列长度： `(Q.rear+MaxSize-Q.front)%MaxSize`
- 出队入队时：指针都按顺时针方向进 1

判断循环队列队空或队满的三种方式

1. 牺牲一个单元来区分队空和队满，入队时少用一个队列单元，约定以 “队头指针在队尾指针的下一位置作为队满的标志”
    - 队满条件： `(Q.rear+1)%MaxSize==Q.front`
    - 队空条件： `Q.front=Q.rear`
    - 队列中元素的个数： `(Q.rear-Q.front+MaxSize)%MaxSize`
2. 类型中增设表示元素个数的数据成员。
    - 队空条件： `Q.size==0`
    - 队满条件： `Q.size==MaxSize`
3. 类型中增设 `tag` 数据成员，以区分是队满还是队空。
    - `tag=0` 时，若因删除导致 `Q.front==Q.rear`，则为队空
    - `tag=1` 时，若因插入导致 `Q.front==Q.rear`，则为队满

2.2.3 队列的链式存储结构

队列的链式存储

队列的链式表示称为链队列，它实际是一个同时带有队头指针和队尾指针的单链表。头指针指向队头结点，尾指针指向队尾结点。

```
typedef struct{//链式队列结点
    ElemType data;
    struct LinkNdoe *next;
}LinkNode;
typedef struct{//链式队列
    LinkNode *front,*rear;//队列的队头和队尾指针
}LinkQueue;
```

通常将链式队列设计成一个带头结点对的单链表，这样插入和删除就统一了

2.2.4 双端队列

双端队列是指允许两端都可进行入队和出队操作的队列，其元素的逻辑结构仍是线性结构。将队列的两端分别称为前端和后端。

- 输出受限的双端队列：允许在一端进行插入和删除，另一端只允许插入的双端队列

- 输入受限的双端队列：允许在一端进行插入和删除，另一端只允许删除的双端队列

2.3 栈和队列的应用

栈在括号匹配中的应用

- 初始设置一个空栈，顺序读入括号
- 若是右括号，则或者置于栈顶的最急迫期待得以消解，或者是不合法的情况
- 若是左括号，则作为一个新的更急迫的期待压入栈中
- 算法结束时，栈为空，否则括号序列不匹配

栈在表达式求值中的应用

后续表达式计算方式

顺序扫描表达式的每一项，然后根据它的类型作出如下相应操作：若该项是操作数，则将其压入栈中；若该项是操作符 `<op>` ，则连续从栈中退出两个操作数 `Y` 和 `X` ，形成运算指令 `X<op>Y` ，并将计算结果重新压入栈中。当表达式的所有项扫描并处理完毕后，栈顶存放的就是最后的结果

中缀表达式转换为前缀或后缀表达式的手工做法

- 按照运算符的优先级对所有的运算单位加括号
- 转换为前缀或后缀表达式。前缀把运算符移动到对应的括号前面，后缀把运算符移动到对应的括号后面
- 把括号去掉

中缀表达式转换为后缀表达式的算法思路

- 从左向右开始扫描中缀表达式
- 遇到数字时，加入后缀表达式
- 遇到运算符时
  - 若为 `(` , 入栈
  - 若为 `)` ，则依次把栈中的运算符加入后缀表达式，直到出现 `(` , 从栈中删除 `(`
  - 若为除括号外的其他运算符，当其优先级高于除 `(` 外的栈顶运算符时，直接入栈。否则从栈顶开始，依次弹出比当前处理的运算符优先级高和优先级相等的运算符，直到一个比它优先级低的或遇到一个左括号为止。

栈在递归中的应用

可以将递归算法转换为非递归算法。通常需要借助栈来实现这种转换

队列在层次遍历中的应用

1. 根节点入队
2. 若队空，则结束遍历；否则重复 3 操作
3. 队列中第一个结点出队，并访问之。若其没有左孩子，则将左孩子入队，若其有左孩子，则将其右孩子入队，返回 2

队列在计算机系统中的应用

- 解决主机与外部设备之间速度不匹配的问题
- 解决由多用户引起的资源竞争问题

2.4 特殊矩阵的压缩存储

数组的定义

数组是由 n 个相同类型的数据元素构成的有限序列，每个数据元素成为一个**数据元素**，每个元素在 n 个线性关系中的序号称为该元素的**下标**，下标的取值范围称为数组的**维界**

数组是线性表的推广。一维数组可视为一个线性表；二维数组可视为其元素也是定长线性表的线性表。

数组的存储结构

多维数组的映方法：按行优先和按列优先

矩阵的压缩存储

指为多个值相同的元素只分配一个存储空间，对零元素不分配存储空间。其目的是节省存储空间

稀疏矩阵

矩阵中非零元素的个数远远小于矩阵元素的个数

使用**三元组**（行、列、值）或**十字链表法**存储，失去了**随机存取特性**