小结

第一章 绪论

一、基本概念

数据、数据元素、数据项（数据元素的某一个属性，结构体中某一处成员）、数据对象（特性相同）、数据结构（关系）。

数据结构：逻辑结构和存储结构。逻辑结构是数据元素间内在的、本质的关系，与计算机无关（线性表、队列）。存储结构强调的是数据元素之间的关系在计算机中的存储与表示（顺序表、线索二叉树、哈希表）。

存储结构：顺序存储和链式存储。顺序存储借助存储数据元素的位置来表示数据元素之间的逻辑关系（逻辑相邻，物理相邻；连续的存储单元；随机的存取）；链式存储借助存储数据元素地址的指针来表示数据元素之间的关系（存储单元不一定是连续的，逻辑相邻，物理上不一定相邻，随机的存储单元、顺序存取）。

逻辑结构：线性结构和非线性结构；线性结构、树形结构、图状（网状）结构、集合。

二、抽象数据类型的定义

数据对象、数据关系、基本操作

三、算法及算法分析

1、算法的特性：输入、输出、有穷性、确定性、可行性。

2、算法评价：正确性、可读性、键状性、高效性（时间复杂度和空间复杂度）。

时间复杂度：算法中基本语句重复执行的次数是问题规模 n 的某个函数 f(n).

O(n) = O(f(n))

例 1、分析以下算法的时间复杂度

```
Void fun(int n)
{
    int y=0;
    while(y*y <= n)
        y++;
}
```

设 while 语句的执行频度为 T(n)，每循环一次 y 值增大 1，即 y=T(n)，有以下关系：

$T(n)*T(n) <= n$ 即 $T(n)^2<=n$ $T(n)=O(n^{1/2})$

例 2、int Find(ElemType a[],int s,int t ,ElemType x)

```
{
    int m= (s+t)/2;
    if( s<=t)
    {
        if (a[m]== x)          return m;
        else if (x <a[m])      return Find(a,s,m-1,x);
        else     return Find(a,m+1,t,x);
    }
    return -1;
}
```

T(n)=O($\log_2 n$)

第二章 线性表

一、线性表的逻辑结构特征

L=(a_1,a_2,.....a_n) ,<ai,ai+1>

(1) 存在唯一的称之为第一个的数据元素；

(2) 存在　　最后一个.....

(3) 除第一个元素之外，每个元素有唯一的直接前驱。

(4) 除　后　　后继

二、线性表的顺序存储表示与操作

1、顺序表的类型定义

```
typedef int INTGER;
typedef struct {
    ElemType *elem;   //连续存储单元的起始地址或基地址
    int length;
}SqList;
SqList L;
```

2、基本操作

(1) Init(SqList &L);

(2) insert(SqList &L,int i, ElemType e);

(3)del(SqList &L,int i, ElemType &e);

引用传递与值传递，引用是已经存在的一个变量的别名

SqList a;

例1、 设计算法删除顺序表中的[x,y]元素。(并不是指有序，而是采用顺序存储结构的线性表)

```
void Del(Sqlist &L, ElemType x,ElemType y)
{
    for(i=0;i<L.length;i++)   //扫描顺序表
    {
        if(L.elem[i] >= x && L.elem[i] <=y )
        {
            for(j=i; j<L.length;j++)
                L.elem[j]=L.elem[j+1];
            L.length--;
        }
    }
}
```

算法的时间复杂度 $O(n^2)$

思考题：如何改进该算法，使其时间复杂度为 O(n)？

```
Void Del(Sqlist &L, ElemType x,ElemType y)
{
    int k=0 ;
    for(int i=0 ;i<L.length ;i++)
        if( !(L.elem[i] >=x && L.elem[i] <= y) )
        {
            L.elem[k] = L.elem[i];
            k++;
        }
    L.length = k;
}
```

算法的时间复杂度为 O(n)

例2、 试分析如下算法的功能

```
void Fun(SqList L, ElemType x)
{
    int i,j=0;
    for( i=1;i<L.length;i++)
        if(L.elem[i] <= L.elem[j])   j=i;
    for (i=L.length;i>j; i--)
        L.elem[i] = L.elem[i-1];
    L.elem[j]=x;
    L.length ++;
}
```

L=(30,3,4,8,3,6)

三、线性表的链式存储表示与操作

与顺序存储结构不同，链式存储采用随机的存储单元。

1、基本概念

结点：构成链表的基本单位，它是由数据域(data)和指针域(next)构成。

头指针：访问链表的入口，指针变量(int *p；p 的基类型为 int)，头指针的基类型是什么类型？基类型应该是节类型（LNode）。LNode *L; L= new LNode;

头结点：通常是在首元结点的前面，附加的一个节点。其数据域存储的并不是线性表中的第一个数据元素的值，可以为空。但是头结点的指针域，它存储的是线性表中首元结点的地址。头结点本身也是有地址的。

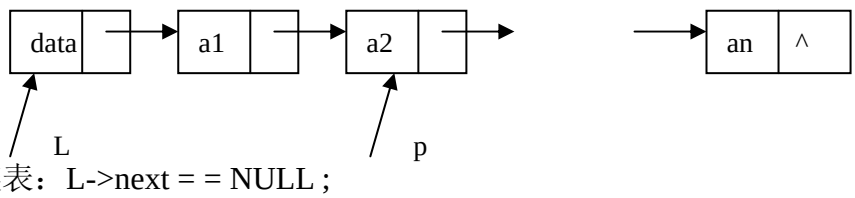
为什么要增加头结点？ L== NULL

首元结点：存储线性表中第一个数据元素的结点。

2、单链表（链表中的结点只有一个指针域，通常这个指针域指向其后继结点）

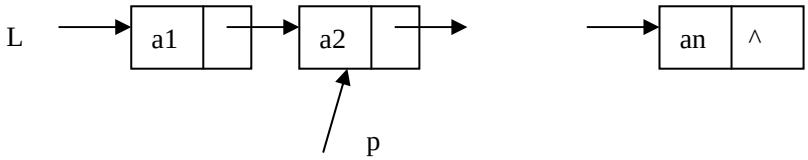
对于一个单链表来说，头结点不一定是必需的，但是头指针是必需要的。

1）带有头结点的单链表：LinkList L1; LNode *L1,*p; L=new LNode(); L->data



空链表：L->next = = NULL ;

2）不带有头结点的单链表：LinkList L1;



空链表: L= =NULL
L1= new LNode(); 且 L1->data = a_1;

3、单链表的类型定义

```
struct LNode     //结点类型
{
      ElemType data;
      struct LNode    *next;
};
```

struct LNode n1;极少使用
主要采用的是 struct LNode *p;
为此，我们将链表定义为一个 struct LNode *的别名，采用 typedef (typedef int ElemType).

```
typedef    struct LNode     //结点类型
           {
               ElemType data;
               struct LNode    *next;
           }
LNode;   //结点类型的别名
```

如：struct LNode n1 可写为：LNode n1，但实际应中，多采用的是 LNode *L;

```
typedef    struct LNode     //结点类型
           {
               ElemType data;
               struct LNode    *next;
           }
LNode,*LinkList;
```

如：LinkList L; //L 是指针类型，等价于 LNode * L，显然，L 的基类型是 LNode

4、单链表基本操作的实现
以带头结点的单链表为例
1）初始化操作

```
void init(LinkList &L)
{
      L=new Lnode;    L->next=NULL;
}
```

2)插入操作

void insert(LinkList &L, int i, ElemType e)

3)删除操作

void insert(LinkList &L, int i, ElemType &e)

4)构造一个链表

(1)头插法：构造的链表与输入的次序是相反的。

(2)尾插法：构造的链表与输入的次序是相同的。

四、两种存储结构的比较

1、顺序存储的优缺点

1）优点：简单，易使用；访问元素速度快（常数时间）

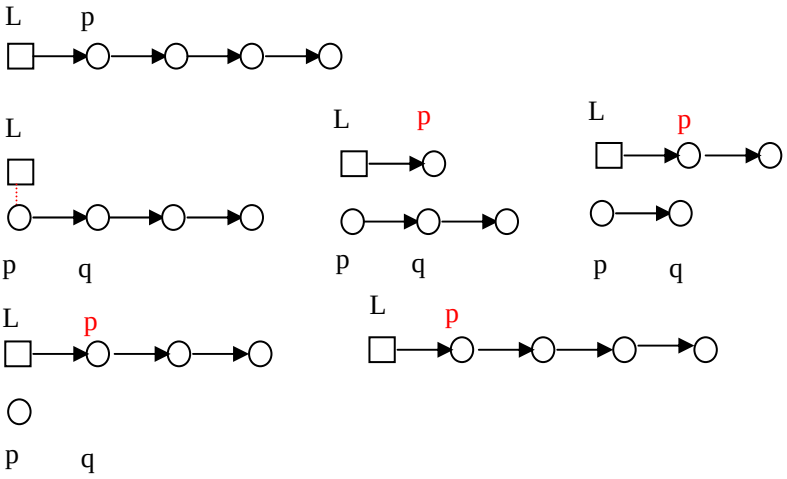
2）不足：固定大小；占据一个连续的存储空间；复杂的定位插入运算；

2、链式存储的优缺点：

1）优点：可以在常数时间内实现扩张；插入删除运算不需要移动数据元素，只需要调整逻辑关系。随机的存储单元。

2）不足：顺序存取；复杂；存储密度

问题 1、将单链表就地逆置。




```
void Reverse(LinkList &L)
{
    LNode *p,*q;
    p=L->next;
    L->next = NULL;
    while(p!=NULL)
    {
        q=p->next;
        p->next = L->next;    //p 的后继变为头结点的后继
        L->nxet = p;          //头结点的后继变为 p
        p=q;                  //更新 p,继续插入下一个结点
    }
}
```

问题 2、如何从表尾开始依次输出链表的所有结点的值。

```
void Reverse(LinkList &L)
{
    LNode *p,*q;
    p=L->next;
    L->next = NULL;
    while(p!=NULL)
    {
        q=p->next;
        p->next = L->next;
        L->nxet = p;
        p=q;
    }
}

void Print(LinkLkist L)
{
    LNode *p;
    p=L->next;
    while(p!=NULL)
    {
        cout << p->data;
        p=p->next;
    }
}

void PrintFromEnd(LinkList L)
{
    Reverse(L);
    Print(L);
}
```

算法的时间复杂度：O(2n)

问题：能否使时间复杂度由 O(2n)变为 O(n)？

采用递归的方法进行求解。递归遍历链表直到到达表尾为止。每次递归返回后再打印输出当前结点的值。

```
void PrintFromEnd(LinkList L)
{
 if(!L)
     return;
 PrindFormEnd(L->next);
 cout << L->data;
}
```

算法的时间复杂度：O(n)，空间复杂度 O(n)。

问题 3、判断给定链表的长度是偶数还是奇数。

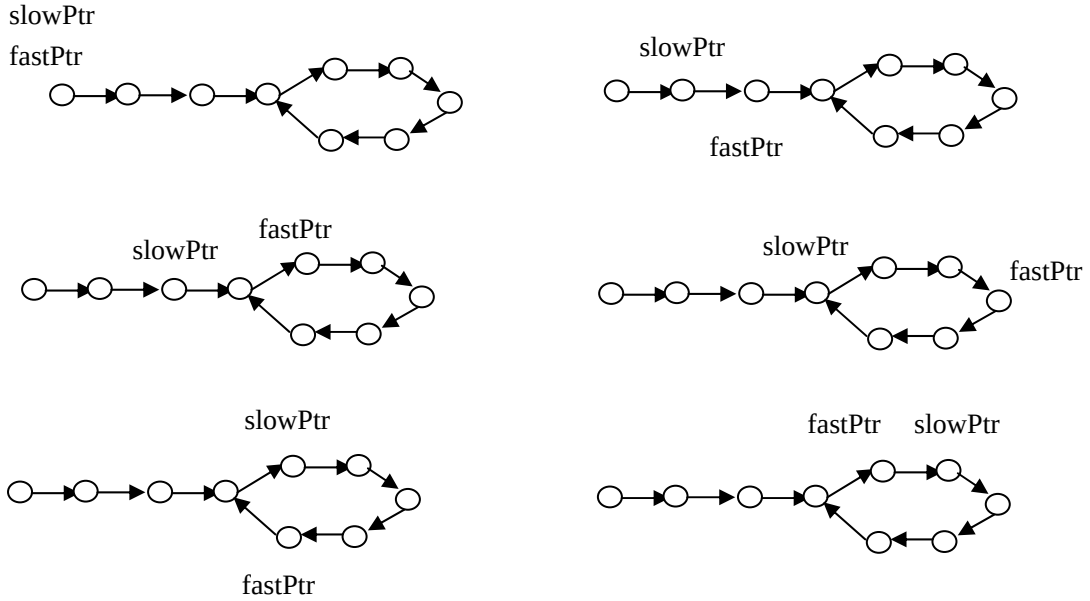
```
int GetLength(LinkList L)
{
    int len=0;
    LNode *p=L;
    while(p->next !=NULL)
    {
        len++; p=p->next;
    }
    return len;
}

int IsLengthEven(LinkList L)
{
    int count;
    count = GetLength(L);
    if(count % 2== 0) return 0;
    else return 1;
}
```

使用一个 2x 指针。所谓 2x 指针是指一次移动两个结点的指针。指针移动结束时，如表长是偶数，那么指针将是一个空指针（NULL），否则指针指向最后一个结点。

```
int IsLinkListLengthEven(LinkList L)
{
    while(L && L->next)
        L= L->next->next;
    if(!L)
        return 0;
    else
        return 1;
}
```

问题 4、判断给定链表是以指针域为 NULL 的结点结束还是以一个循环链表结束。



Floyd-cycle-finding 算法，高效的算法 O(n)，空间复杂度 O(1)

```
int IsLinkListHasLoop(LinkList L)
{
    LNode *slowPtr=L,*fastPtr=L;
    while(slowPtr && fastPtr && fastPtr->next)
    {
        slowPtr= slowPtr->next;
        fastPtr =fastPtr->next->next;
        if(slowPtr == fastPtr) return 1;
    }
    return 0;
}
```

问题 5-问题 7：关于有序单链表

问题 5：在一个递增有序单链表中插入一个结点，使该链表仍然有序。

```
void InsertInSortedList(LinkList &L,LNode *newNode)
{
    LNode *pre=L; //L 是待插入（处理）结点的前趋结点
    while((pre->next)    && (pre->next->data < newNode->data) )
        pre=pre->next;
    newNode->next=pre->next;
    pre->next = newNode;
}
```

时间复杂度 O(n)，空间复杂度为 O(1)

思考：L 为不带头结点的单链表的情况。

求解：遍历链表（根据结点值域的大小）找出结点的插入位置，然后插入结点。

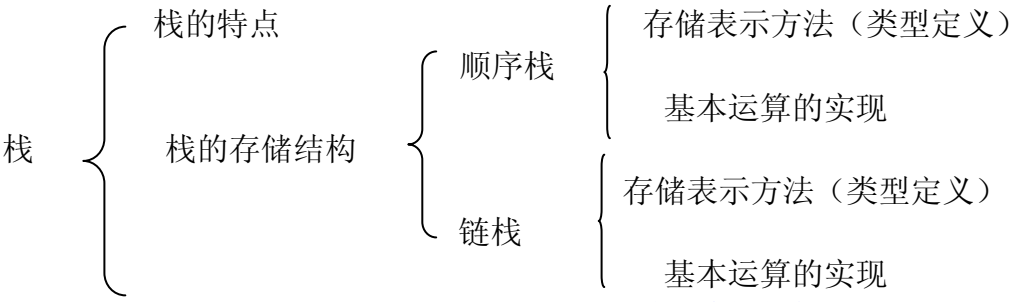
问题 6：已知单链表 L 是一个递增有序表，试写一个高效算法删除表中值大于 min 且小于 max 的结点，同时释放被删除结点的空间，请分析算法的时间复杂度。

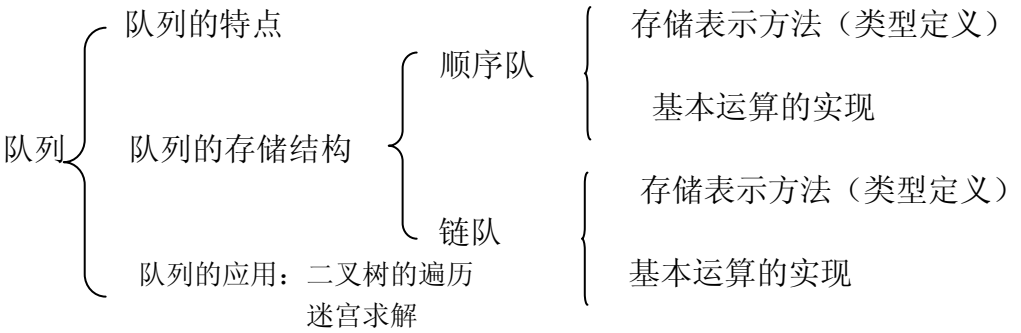
```
void DeltInSortedList(LinkList &head,ElemType min, ElemType max)
{
    LNode *p=head->next, *pre=head;     //pre 指向 p 的前趋
    while(p!=NULL && p->data <= min) //查找刚好大小 min 的结点 p
    {
        pre= p;
        p=p->next;
    }
    while( p!=null && p->data < max)    //删除所有小于 max 的结点
    {
        pre->next = p->next;
        free(p);
        p=pre->next;
    }
}
```

问题 7：有一个递增单链表 L（允许出现值域相同的结点），设计算法删除值域重复的结点。

第三章：栈和队列

一、知识结构





二、基本知识点

（1）栈、队列和线性表之间的异同；

（2）顺序栈的基本运算算法设计；

（3）链栈的基本运算算法设计；

（4）循环队列的基本运算算法设计；

（5）链队列的基本运算算法设计；

（6）利用栈和队列求解复杂的应用问题。

三、要点归纳

（1）栈和队列的共同点是它们的数据元素都是呈线性关系，且只允许在端点处插入和删除。

（2）栈是一种后进先出的数据结构，只能在同一端进行插入和删除。

（3）栈既可采用顺序存储，也可采用链式存储；

（4）n 个不同元素的进栈顺序和出栈顺序不一定相同。

（5）在顺序栈中通常用栈顶指针指向当前的栈顶元素。

（6）栈顶与栈底的位置。

（7）顺序栈的四个要素（栈空、栈满、入栈、出栈）。

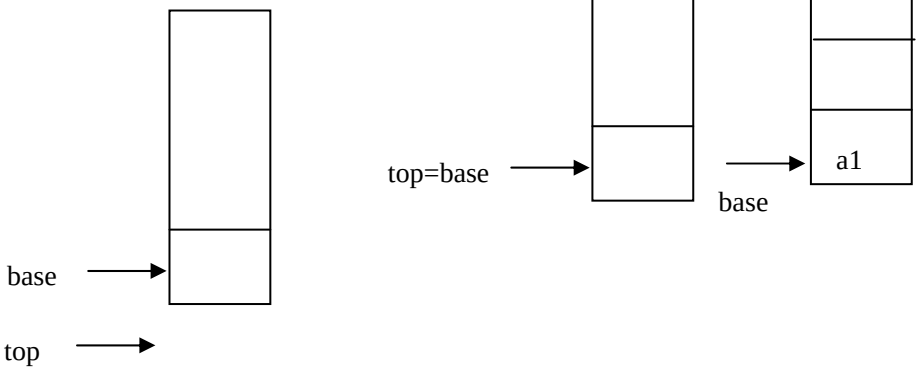
如：顺序栈的类型定义（本教材）

```
#define MAXSIZE 100    //顺序栈存储空间的初始分配大小
typedef struct{
    SElemType *base;    //栈底指针
    SElemType *top;    //栈顶指针
```

```
    int stacksize;   //栈可用的最大容量
}SqStack;
```

1) 栈空：S.top == S.base;

2) 入栈： *S.top++=e; *S.top=e; S.top++;



3) 出栈：--S.top

4) 栈满：S.top – S.base == MAXSIZE

（8）在顺序栈或链栈中，进栈和出栈操作不涉及栈中元素的移动，且运算的时间复杂度均为 O(1)。

（9）在链栈中，由于每个结点是单独分配的，通常不考虑上溢出问题。

（10）队列是一种先进先出的数据结构，只能从一端插入元素，从另一端删除元素。

（11）队列可采用顺序队和链队两类存储结构。

（12）环形队列也是一种顺序队，是通过逻辑方法使其首尾相连的，解决非环形队列的假溢出现象。

（13）在环形队列中，队头指针指向队头元素，队尾指针指向队尾元素的后一个位置。

（14）环形队列的定义及四个要素。

```
#define MAXQSIZE 100
typedef struct {
    QElemType *base;   //连续存储单元的起始地址
    int front;   //队头
    int rear;    //队尾
}SeQueue;
```

(1) 队空：Q.rear = = Q.front;

(2) 队满：(Q.rear + 1) % MAXQSIZE = = Q.front;

(3) 入队：Q.base[Q.rear]= e, Q.rear = (Q.rear + 1) % MAXQSIZE

(4) 出队：e=Q.base[Q.front], Q. front = (Q. front + 1) % MAXQSIZE

(5) 队长：length= (Q.rear-Q.front + MAXQSIZE) % MAXQSIZE

（15）无论环队还是链队，进队和出队的时间复杂度均为 O(1)。

（16）在实际应用中，一般栈和队列都是用来存放临时数据，如果先保存的元素先处理，应采用队列；如果后保存的元素先处理，应采用栈。

问题一、利用栈实现中缀表达式转为后缀表达式的算法。

1、重要特性：

（1） 中缀表达式和后缀表达式中操作数（或数字）之间的相对位置没有改变，但是运算符的相对位置在两个表达式中可能不同。

（2） 仅使用一个栈就能实现中缀表达式到后缀表达式的转换。在算法中所使用的栈被用来实现将运算符在中缀表达式中的位置顺序转换为运算符在后缀表达式中的相对位置顺序。这个栈包含运算符和开括号“（”。

2、算法：

（1） 创建一个栈。

（2） 对输入流中的每个字符{

if(c 是一个操作数)

输出 c

else if(c 是右括号’)’){

从栈中弹出并输出运算符直到弹出的是左括号为止（左括号不输出）

}

else // c 是运算符或是左括号

从栈中弹出并输出运算符直到栈顶运算符的优先级小于 c 的优先级或栈顶元素是左括号或栈为空为止

将 c 压入栈

（3）弹出并输出栈中的运算符直到栈为空为止。

3、实例：A*B-（C+D）+E

输入字符	栈上的操作	栈的内容	后缀表达式
A		空	A
*	Push	*	A
B		*	AB
-	Check and Push	-	AB*
(	Push	-(	AB*
C		-(	AB*C
+	Check and Push	-(+	AB*C
D		-(+	AB*CD
)	一直做 pop 操作并将弹出的符号追加到后缀表达式中，直到遇到’(‘为止	-	AB*CD+
+		+	AB*CD+-
E		+	AB*CD+-E
输入结束	一直做 pop 直到栈空为止		AB*CD+-E+

习题：1、中缀表达式为 a-b*(c+d)，求后缀表达式。

2、中缀表达式为 1+6/(8-5)*3，求后缀表达式。

问题二、利用栈直接计算中缀表达式的算法。

1、求解：利用两个栈可实现对中缀表达式的直接计算（即不需要将其转换为后缀表达式之后再进行求解的计算方法）。
2、算法：
1）创建一个空的操作符栈；
2）创建一个空的操作数栈；
3）对于输入字符串中的每个字符
a)从输入串中读入下一个字符
b)如果输入的字符是操作数，则将其压入操作数栈中
c) 如果输入的字符是操作符，则评估该操作符
(1) 若小于，则操作符压入操作符栈中
(2)等于，弹出操作符栈栈顶’(‘
(3)若大于，弹出栈顶运算符，并从操作数栈中弹出两个操作数，运算后再压入操作数栈。
4）弹出操作数栈的栈顶元素（栈中仅有一个元素）即为计算结果。
3、算例：#3*(7-2)#

步骤	OPTR 栈	OPND 栈	读入字符	主要操作
1	#	空	3	Push(OPND,’3’)
2	#	3	*	Push(OPTR,’*’)
3	#*	3	(	Push(OPTR,’(’)
4	#*(	3	7	Push(OPND,’7’)
5	#*(	3 7	-	Push(OPTR,’-’)
6	#*(-	3 7	2	Push(OPND,’2’)
7	#*(-	3 7 2	)	Pop(OPTR,’-’) Pop(OPND,t1),Pop(OPND,t2) Push(OPND ,Oper(t2,- ,t1))
8	#*(	3 5		Pop(OPTR,’(’)
9	#*	3 5	#	Pop(OPTR,’*’) Pop(OPND,t1),Pop(OPND,t2) Push(OPND ,Oper(t2,* ,t1))
10	#	15		return GetTop(OPND)

问题三、利用栈求解后缀表达式的算法。

算法：

（1） 从左向右扫描后缀表达式；

（2） 初始化一个空栈；

（3） 重复步骤 4 和 5，直到后缀表达式扫描结束；

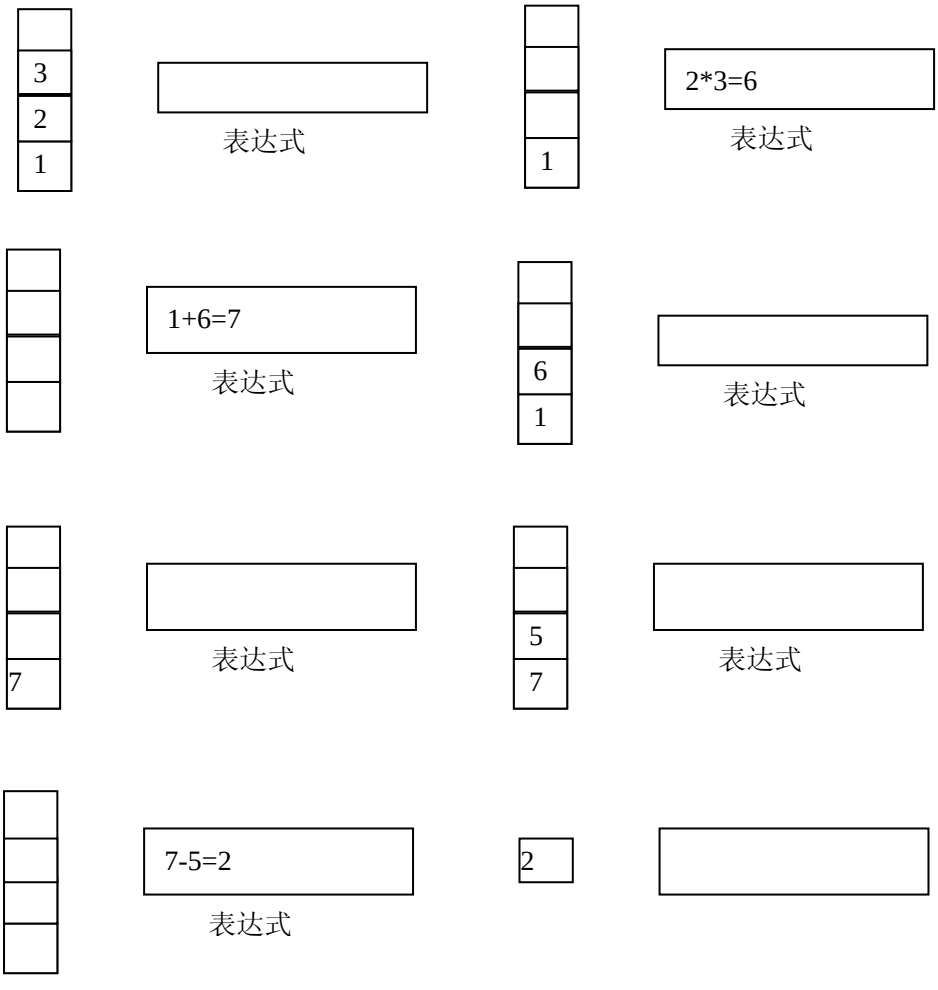
（4） 如果扫描到的字符是操作数，那么将其压入栈中。

（5） 如果扫描到的字符是操作符，如是二元操作符，那么从栈中弹出栈顶元素和次栈顶元素。然后对弹出的操作数实施扫描到的操作符，并将结果压入栈入。

（6） 表达式扫描完，栈中仅有一个元素；

（7） 返回栈顶元素作为表达式的计算结果。

算例：1 2 3 * + 5 -



问题四、利用两个栈实现一个队列。

```
typedef struct{
      Stack *s1; //用来实现 EnQueue
     Stack *s2; //用来实现 DeQueue
}Queue;
```

(1)入队算法：仅仅将元素压入栈 s1 中

```
 void EnQueue(Queue &Q, ElemType data)
{
     Push(Q.s1,data);
}
```

时间复杂度 O(1);

出队算法：

(1) 如栈 s2 不为空，则从 s2 中弹出并返回栈顶元素

(2) 如栈 s2 为空，那么将 s1 中的所有元素移至 s2 中，然后从 s2 中弹出并返回栈顶元素。

```
void DeQueue(Queue &Q, ElemType &data)
{
     if(!IsEmptyStack(Q.s2))
          Pop(Q.s2,data);
     else{
          while(!IsEmptyStack(Q.s1)){
               Pop(Q.s1,data);
               Push(Q.s2,data);
     }
     Pop(Q.s2,data);
}
```

第四章、串、数组和广义表

1、串的 KMP 算法（next 函数值的计算、nextval 计算）

2、数据中元素地址的计算方法。

3、能够利用广义表的两个重要运算（Head(),Tail）来求解问题。

Gethead(),GetTail(), H(),T()

Head(L)：取出的表头为非空广义表的第一个元素，它可以是一个单原子，也可以是一个子表；

Tail(L)：取出的表尾为除去表头之外，由其余元素构成的表。它一定是一个广义表。

1、已知模式串 T=’abcaabbcabcaabdab’，计算 next 和 nextval 值。

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
模式串 T	a	b	c	a	a	b	b	c	a	b	c	a	a	b	d	a	b
next[j]	0	1	1	1	2	2	3	1	1	2	3	4	5	6	7	1	2
nextval[j]	0	1	1	0	2	1	3	1	0	1	1	0	2	1	7	0	1

显然，next[1]=0,next[2]=1
next[3]=? ,利用 next[2]求得。T[2]与 T[next[2]]=T[1]进行比较，
(1) 若 T[2]=T[1],则 next[3]=next[2]+1,T[2]=’b’　T[1]=’a’,不相等。
(2) 若 T[2]!=T[1],
Next[4]=?　利用 next[3]求得。T[3]与 T[next[3]]=T[1]进行比较
next[5]=? ,利用 next[4]求得。T[4]与 T[next[4]]=T[1]进行比较，显然相等，所以 next[5]=next[4]+1=1+1=2
next[6]=? ,利用 next[5]求得。T[5]与 T[next[5]]=T[2]进行比较，显然不相等，T[5]应继续与 T[next[2]]=T[1],若相等，则 next[6]=next[2]+1=2
next[7]=? ,利用 next[6]求得。T[6]与 T[next[6]]=T[2]进行比较，显然相等，则 next[7]=next[6]+1=3
next[8]=? ,利用 next[7]求得。T[7]与 T[next[7]]=T[3]进行比较，显然不相等，则继续 T[7]与 T[next[3]]=T[1]比较，显然不相等，所以 next[8]=1
next[9]=? ,利用 next[8]求得。T[8]与 T[next[8]]=T[1]进行比较，显然不相等，所以 next[9]=1
next[10]=? ,利用 next[9]求得。T[9]与 T[next[9]]=T[1]进行比较，显然相等，所以 next[10]=next[9]+1=2
next[11]=? ,利用 next[10]求得。T[10]与 T[next[10]]=T[2]进行比较，显然相等，所以 next[11]=next[10]+1=3
next[12]=? ,利用 next[11]求得。T[11]与 T[next[11]]=T[3]进行比较，显然相等，所以 next[12]=next[11]+1=4
next[13]=? ,利用 next[12]求得。T[12]与 T[next[12]]=T[4]进行比较，显然相等，所以 next[13]=next[12]+1=5
next[14]=? ,利用 next[13]求得。T[13]与 T[next[13]]=T[5]进行比较，显然相等，所以 next[14]=next[13]+1=6
next[15]=? ,利用 next[14]求得。T[14]与 T[next[14]]=T[6]进行比较，显然相等，所以 next[15]=next[14]+1=7
next[16]=? ,利用 next[15]求得。T[15]与 T[next[15]]=T[7]进行比较，显然不相等，所以应继续比较，按照 KMP 算法思想，

T[15]与 T[next[7]]比较，即 T[15]与 T[3]比较，若相等，则 next[16]=next[7]+1。

显然 T[15]与 T[3]不等，按照 KMP 算法思想，T[15]应该继续与 T[next[3]]比较。即 T[15]与 T[1]比较，若相等，则 next[16]=next[3]+1；显然 T[15]与 T[1]不

相等，所以 next[16]=1

next[17]=? ,利用 next[16]求得。T[16]与 T[next[16]]=T[1]进行比较，显然相等，所以 next[17]= next[16]+1=2

2、已知模式串 T=’bacbababababacaca’，计算 next 和 nextval 值。

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
模式串 T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
next[j]	0	1	1	1	2	3	2	3	2	3	2	3	4	1	1
nextval[j]	0	1	1	0	1	3	1	3	1	3	1	1	4	1	1

3、数组 a[0..9] 的每个元素占 2 个单元, 将其按行优先次序存储在起始地址为 1000 的连续内存单元中，问数组 a 中共有多少个元素，a[5]元素的地址为多少？

&a[5]=&a[0]+偏移量=1000+5*2=1010

4、数组 a[0..9][0..10] 的每个元素占 2 个单元, 将其按行优先次序存储在起始地址为 1000 的连续内存单元中，问数组 a 中共有多少个元素，a[3][6]元素的地址为多少？

10*11=110

&a[3][6]= &a[0][0]+偏移量=1000 +(3*11+6)*2

5、数组 a[0..4][0..5][0..6]的每个元素占 8 个单元，将其按行优先次序存储在起始地址为 1000 的连续内存单元中，问数组 a 中共有多少个元素，a[2][3][4]元素的地址为多少？

=5*6*7

& a[2][3][4] = & a[0][0][0] + (2*6*7+3*7+4)*8

6、数组 a[1..4][1..5][1..6][1..9]的每个元素占 2 个单元，将其按行优先次序存储在起始地址为 1000 的连续内存单元中，问数组 a 中共有多少个元素，a[2][5][4][6]元素的地址为多少？

& a[2][5][4][6] = & a[1][1][1][1] + (1*5*6*9+4*6*9+3*9+5)*2

7、已知广义表 L=((x,y,z), (u,t,w))，利用 head 和 tail 取出原子 t.

长度为 2 的广义表，其中每个元素都是表元素。（x,y,z）、(u,t,w)

tail(L)=((u,t,w))

head(tail(L)) =(u,t,w)

tail(head(tail(L)))=(t,w)

head(tail(head(tail(L)))) =t

8、已知广义表 L=((x,y,z), (u,t,w),(m,(n,p)))，利用 head 和 tail 取出原子 p.

Tail(L)=((u,t,w),(m,(n,p)))

Tail(Tail(L)) =((m,(n,p)))

Head(Tail(Tail(L))) =(m,(n,p))

Tail(Head(Tail(Tail(L)))) = ((n,p))

Head(Tail(Head(Tail(Tail(L))))) = (n,p)

Tail(Head(Tail(Head(Tail(Tail(L)))))) =(p)

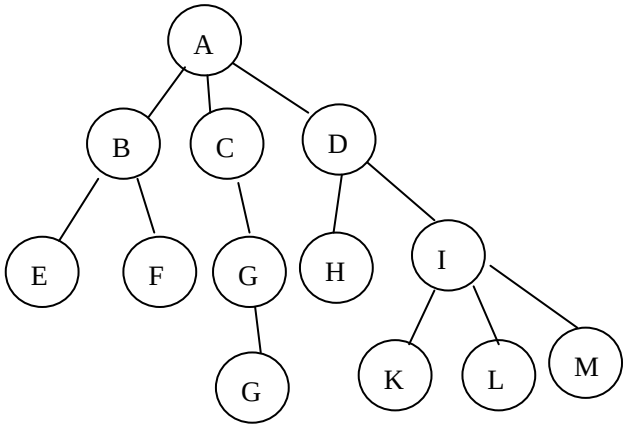
Head(Tail(Head(Tail(Head(Tail(Tail(L))))))) =p

第五章、树和二叉树

学习要点：

一、 树的相关概念，包括树、结点的度、树的度、分支结点、叶子结点、孩子结点、双亲结点、子孙结点、祖先结点、结点层次、树的高度、森林。

二、 树的表示。包括树形表示法、文氏图表示法、凹入表示法和括号表示法。例：括号表示法的一棵树。A(B(E,F),C(G(J)),D(H,I(K,L,M)))，对应的树形表示法如下：



三、树的存储结构

（1）双亲表示法；（2）孩子表示法；（3）孩子兄弟表示法。

四、树的遍历

（1）先根遍历；（2）后根遍历

五、二叉树的概念，包括二叉树、满二叉树、完全二叉树

六、二叉树的性质

1、第 i 层最多可有 2^i 个结点。

2、深度为 K 的二叉树最多可以有 2^K-1 个结点

3、$n_0=n_2+1$（掌握该性质的证明方法）

二叉树中的结点可分为三类：度为 0 的结点（叶子结点）、度为 1 的结点、度为 2 的结点；

n_0：通常用来表示度为 0 的结点（叶子结点）的数量

n_1：通常用来表示度为 1 的结点（叶子结点）的数量

n_2：通常用来表示度为 2 的结点（叶子结点）的数量

n：通常用来表示结点的数量

B: 通常用来表示分支的数量

显然：$n= n_0+ n_1+ n_2$

$n=B+1$

$B=n_0*0+ *1+ *2= n_1+2 n_2$

4、 n 个结点的完全二叉树，K=log2n+1

5、针对 n 个结点的完全二叉树，包括 3 点：

（1）parent(i)=i/2;

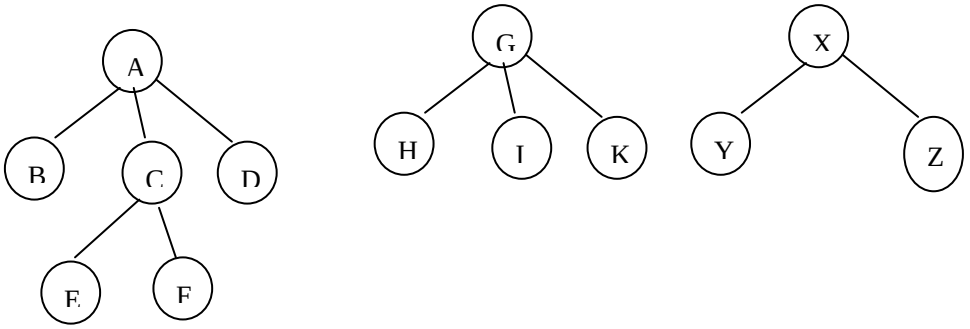
（2）lchild(i)=2i，2i<=n。

2*i>n，编号为 i 的结点无左孩子，也就是说编号为 i 的结点一定是叶子结点；

（3）rchild(i)=2i+1，2i+1<=n。

该性质的证明了解。

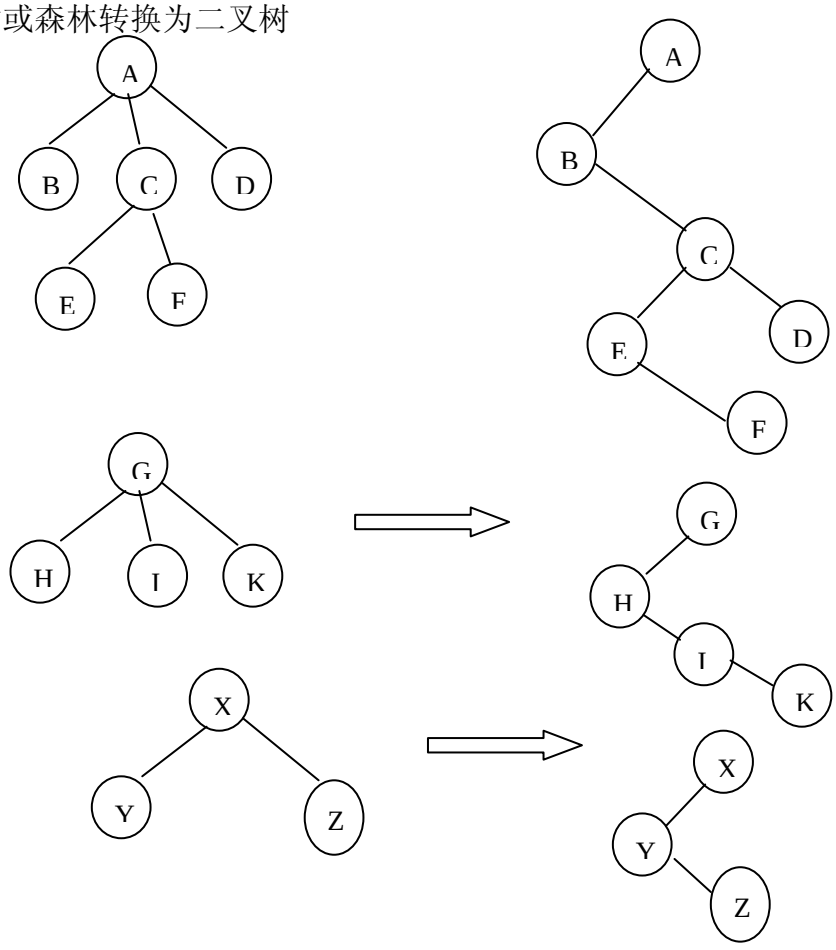
七、树、森林与二叉树的转换

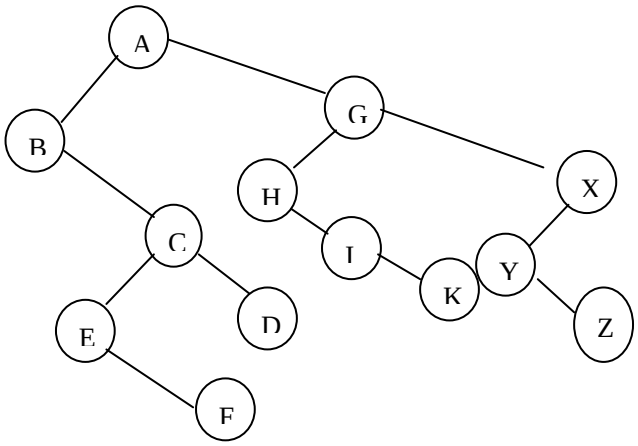


方法：二叉树采用二叉链表（左子树，右子树），树采用孩子兄弟表示法（第一个孩子，下一个兄弟）。森林只要把每棵树的根结点视为兄弟结点。

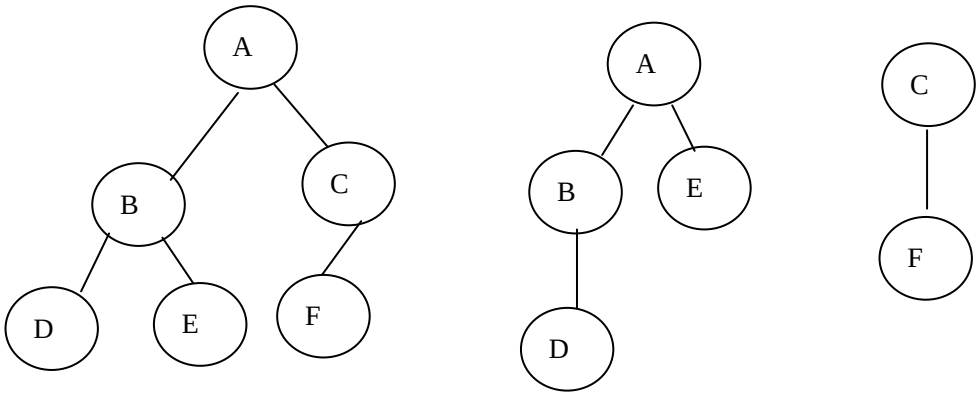
转换后的二叉树如下：

（1）树或森林转换为二叉树





（3） 二叉树转换为对应的树或森林。



八、二叉树的存储结构

二叉链表，顺序存储（适合于完全二叉树）

九、二叉树的基本运算

十、二叉树的遍历算法及其运用

（1）先序、中序、后序、层次

（2）先序+中序，构造二叉树，得到后序

（3）后序+中序，构造二叉树，得到先序

（4）层次遍历算法（借助队列）

（5）基于遍历的应用（求叶子结点数量，度为 2 的结点的个数，度为 1 的结点个数，二叉树的高度，销毁二叉树，二叉树是否相等，叶子结点到根结点的路径、复制二叉树，左右交换….）

十一、线索二叉树

能够画出线索二叉树。

十二、Huffman 树及 Huffman 编码

1、什么是 Huffman 树？主要特征？

2、Huffman 树的构造算法。

3、求 Huffman 树的带权路径长度。

4、Huffman 编码

第六章：图（算法设计考题相对较少）

1、图的相关概念

有向图和无向图、度（入度或出度）、完全图、子图、连通图与强连通图、连通分量与强连通分量

2、图的二种存储结构

（1）邻接矩阵

（2）邻接表

3、图的遍历

（1）深度优先

（2）广度优先

4、图的典型应用

1、求解最小生成树问题

（1）普里姆算法求解最小生成树；

（2）克鲁斯卡尔算法求解最小生成树。

2、最短路径问题

利用迪杰斯特拉算法求解最短路径问题。

3、拓扑排序

4、关键路径

第七章、查找

1、相关概念

平均查找长度、性能分析

2、静态查找

（1）顺序查找：算法的基本思路、算法的具体实现、查找效率分析。

（2）二分查找：算法的基本思路、适用条件、算法的具体实现、查找效率分析（二分查找的判定树）。

（3）分块查找

3、树表的查找

（1）二叉排序树：概念与特征、构造二叉排序树（逐点输入法）、基本思路、查找效率分析。

（2）AVL 树：概念、基本思路、查找效率分析、构造 AVL 树的过程（给定一个关键字序列来构造 AVL 树。

4、哈希查找（散列查找）

（1）哈希函数（散列函数）与哈希地址：除留余数法

H(key) = key % p, p 通常取小于哈希表长度 m 的一个质数。

（2）哈希冲突及解决方法。

1）开放定址法：线性探测法，二次探测法，再哈希法。

2）链地址法（拉链法）

（3）哈希表（散列表）的构造：装填因子、哈希表的长度

（4）查找效率分析

1）在等概率，查找成功情况下，ASL 计算。

2）在等概率，查找不成功情况下，ASL 计算。

线性探测法解决冲突：

$H_i=(H(key)+d_i)\%m$, $d_i=1,2,3,\ldots,m-1$
H_i 为散列地址，H(key)为散列函数，m 是哈希表或散列表的长度。

H(key)=key % 11;　哈希表长 m=16
如 key=40,H(40)=7
H1=(H(40)+1) % 16
第八章、内排序
1、相关概念：排序的稳定性
2、插入排序：直接插入排序、折半插入排序、希尔排序三种排序的过程和算法的实现。
2、交换排序：冒泡排序和快速排序的过程和算法实现
3、选择排序：简单选择排序和堆排序（堆的概念）的过程和算法实现
4、归并排序：二路归并排序的过程与算法实现
5、基数排序的过程与算法实现
6、排序方法的比较和选择（时间复杂度和空间复杂度）