绪论

算法，算法分析，时间复杂度，空间复杂度

- 时间复杂度

```
int i = 1;
while(i<n)
{
    i = i * 2;
}
// 时间复杂度：log2n
```

```
for(m=1; m<n; m++)
{
    i = 1;
    while(i<n)
    {
        i = i * 2;
    }
}
// 时间复杂度：nlog2n
```

双重循环的时间复杂度：n^2，其他以此类推。

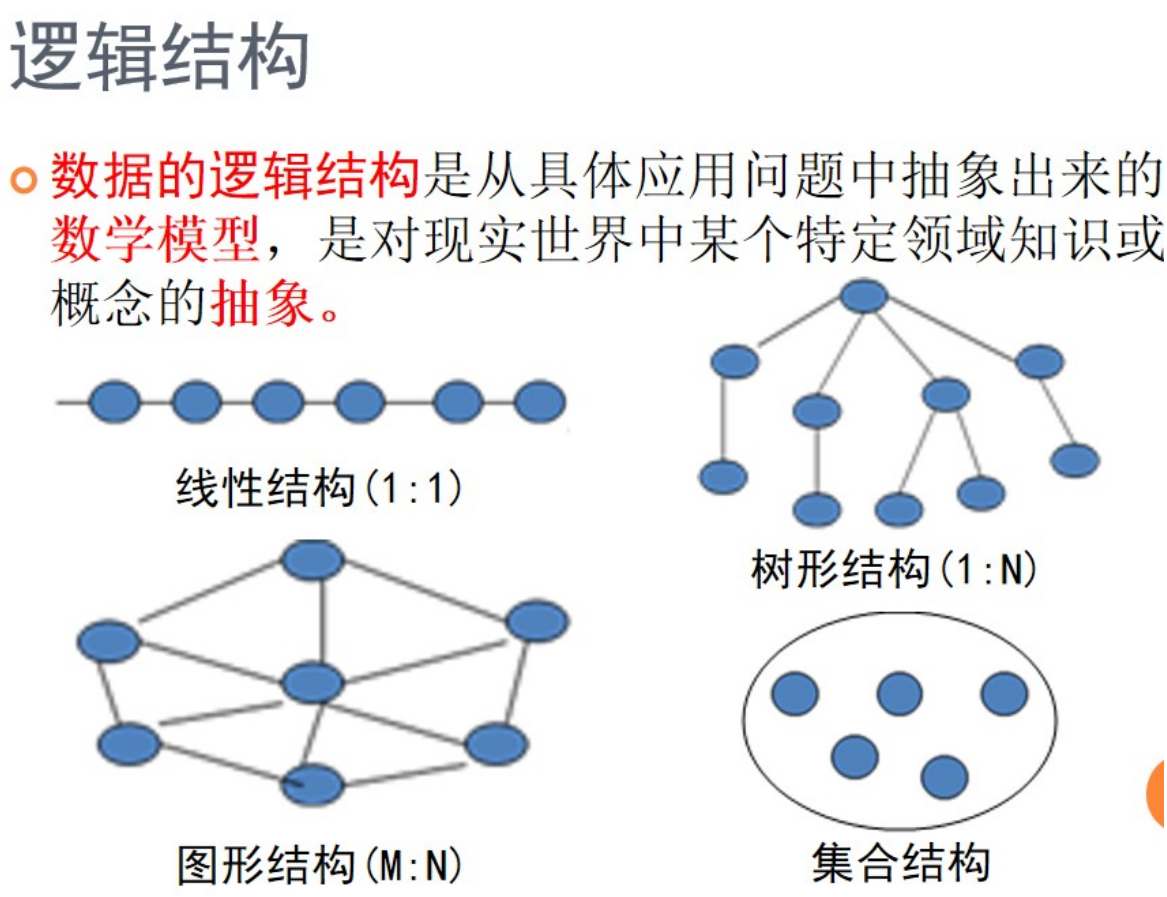
> 时间复杂度一般表示运行的时间长短

- 空间复杂度

```
int[] m = new int[n]
for(i=1; i<=n; ++i)
{
   j = i;
   j++;
}
```

这段代码中，第一行new了一个数组出来，这个数据占用的大小为n，这段代码的2-6行，虽然有循环，但没有再分配新的空间，因此，这段代码的空间复杂度主要看第一行即可，即 S(n) = O(n)

- 一些逻辑结构：



> 空间复杂度一般表示占用的内存/空间

第一单元

1.1 链表，链表的合并、循环、分解，单链表、双链表

(1)线性表的操作

a1-a2-a3-a4-a5-...-an

在某一元素前面的叫前驱元素，在后面的叫后继元素。

线性表中第i个元素的存储位置和第一个元素a1的存储位置满足以下关系

```
LOC(ai) = LOC(a1)+(i-1)m   // m表示第一个元素占用的存储单元数量
```

线性表的定义：

```
void InitList (SeqList *L)
/* 初始化线性表 */
{
    L->length=0; /*把线性表的长度设置为0*/
}
```

插入运算与删除运算

如果使用C++较为简单

用C语言实现：原理与插入排序类似

```
int InsertList (SeqList *L,int i,DataType e)
*/在顺序表的第i个位置插入元素*/
{
    int j;
    if(i<1||i>L->length-1)   /*在插入前，判断插入位置是否合法 */
    {
        printf("插入位置i不合法！");
```

```
        return -1;
    }
    else if(L->length>=ListSize) /*在插入前，判断顺序表是否已满 */
    {
        printf("顺序表已满，不能插入元素。");
        return 0;
    }
    else
    {
        for (j=L->length;j>=i;j--)
        {
            L->list[j]=L->list[j-1]; /*将第i个位置以后的元素依次后移 */
        }
        L->list[i-1]=e;              /*把元素插入第i个位置*/
        L->length=L->length+1;       /*将顺序表的表长增1*/
        return 1;
    }
}
```

删除元素的代码类似：

```
int InsertList (SeqList *L,int i,DataType *e)
*/在顺序表的第i个位置插入元素*/
{
    int j;
    if(L->length<=0)    /*判断顺序表有无元素 */
    {
        printf("顺序表已空，不能进行删除");
        return -1;
    }
    else if(i<1||i>L->length) /*在插入前，判断顺序表是否已满 */
    {
```

```
        printf("删除位置不合适");
        return -1;
    }
    else
    {
        *e=L->list[i-1];            /*删除元素 */
        for (j=;j<=L->length-1;j++)
        {
            L->list[j-1]=L->list[j]; /*将第i个位置以后的
元素依次后移 */
        }
        L->length=L->length-1;      /*将顺序表的表长减1
*/
        return 1;
    }
}
```

元素的查找：分为按序号查找与按内容查找

```
int GetElem (SeqList L,int i,DataType *e)
*/查找线性表的第i个元素*/
{
    if(i<1||i>L->length-1)   /*在插入前，判断插入位置是否
合法 */
    return -1;
    *e=L.list[i-1]; /*将第i个元素的值赋给e，e就是查找到的
元素*/
    return 1;
}
/* 分割线 */
int LocateElem (SeqList L,int i,DataType *e)
/* 查找线性表值为e的元素*/
{
    int i;
    for (i=0;i<L.length;i++)
    {
        if(L.list[i]==e)
```

```
        {
            return i+1;
        }
    return 0;
    }
}
```

将两个有序的链表合并(难，一般不考)

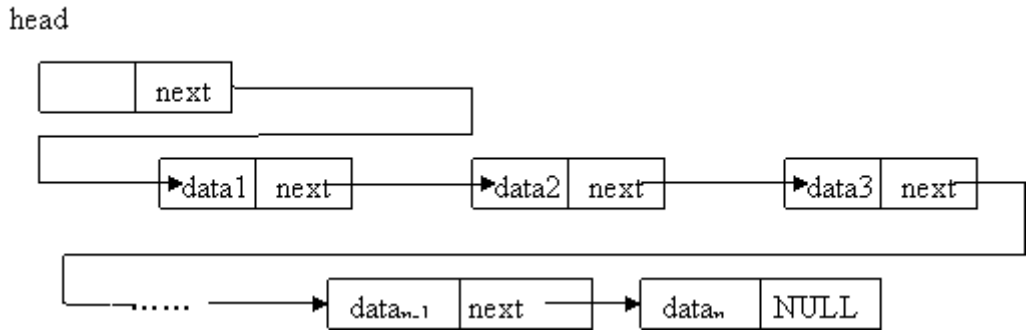
```
**********************************
*实例说明:合并两个有序的线性表为一个有序的线性表
********************************************** /
#include<stdio.h>  /*包含输入/输出头文件*/
#define ListSize 200
typedef int DataType;   /*元素类型定义为整型*/
#include"SeqList.h"     /*包含顺序表的基本运算*/
void MergeList (SeqList A,SeqList B,SeqList *C) ;
/*合并顺序表A和B中元素的函数声明*/
void main ()
{
    int i, flag;
    DataType a[]={8,17,17,25,29};
    DataType b[]={3, 9,21,21,26,57};
    DataType e;
    SeqList A,B,C;
    InitList(&A) ;
    InitList(&B) ;
    InitList (&C) ;
    for(i=l;i<=sizeof(a);i++) /*将数组a中的元素插入顺序表
A中*/
    {
    if (InsertList(&A, i,a[i-1])==0)
    {
        printf ("位置不合法") ;
        return;
    }
    }
```

```
    for(i=1;i<=sizeof(b);i++) /*将数组b中的元素插入顺序表
B中*/
    {
        if (InsertList(&B,i,b[i-1])==0)
        {
            printf("位置不合法") ;
            return;
        }
    }
    printf("顺序表A中的元素: \n"); /*输出顺序表 A中的每个
元素*/
    for(i=1;i<=A. length;i++)
    {
        flag=GetElem(A,i,&e); /*返回顺序表A中的每个元素到
e中*/
        if (flag==1)
        {
            printf ("%4d",e);
        }
    }
    printf ("\n") ;
    printf("顺序表B中的元素: \n"); /*输出顺序表B中的每个元
素*/
    for(i=l;i<=B.length;i++)
    {
        flag=GetElem(B,i,&e);
        if (flag==1)
        {
            printf("%4d",e);
        }
    }
    printf("\n");
    printf("将顺序表A和B中的元素合并得到C: \n");
    MergeList(A,B,&C);          /*将顺序表A和B中的元素合并
*/
    for (i=1;i<=C. length;i++) /*显示合并后顺序表C中的所
有元素*/
```

```
    {
        flag=GetElem(C,i,&e);
        if (flag==1)
        {
            printf("%4d",e);
        }
        }
    }
    printf("\n");
}
void MergeList(SeqList A, SeqList B, SeqList *C)
/*合并顺序表A和B的元素到顺序表C中，并保持元素非递减排序*/
{
    int i,j,k;
    DataType el,e2;
    i=1;j=1;k=l;
    while (i<=A. length&&j<=B. length)
    {
        GetElem(A,i, &e1) ;
        GetElem(B,j, &e2) ;
        if (el<=e2)
        {
            InsertList(C,k,e1) ; /*将较小的一个元素插入顺
序表C中*/
            i++;            /*往后移动一个位置，准备比较下
一个元素*/
            k++;
        }
        else
        InsertList(C,k,e2) ;
        j++;
        k++;
    }
    while (i<=A. length) /*如果顺序表A中元素还有剩余，顺序
表B中已经没有元素。
    {
        GetElem(A,i,&e1) ;
```

```
            InsertList(C,k,e1) ;
            i++;
            k++;
        }
        while (j<=B. length) /*如果顺序表B中元素还有剩余，顺序表A中已经没有元素
        {
            GetElem(B,j,&e2);
            InsertList(C,k,e2);
            j++;
            k++;
        }
        C->length=A.length+B.length; /*顺序表c的长度等于顺序表A和顺序表B的长度的和*/
    }
```

(2) 单链表



初始化单链表

带头节点：head-> p1->p2->p3 ->p1->p2->p3-> p1.....

不带头节点： p1->p2->p3 ->p1->p2->p3-> p1.....

带头节点与不带头节点的差距一目了然。

```
void InitList (LinkList *head)   /* 初始化单链表 */
{
    if((*head=
(LinkList)malloc(sizeof(ListNode)))==NULL)
    {
        exit(-1);
    }
    (*head)->next=NULL;      /*"head->next的意思就是指向
下一个节点"*/
}
```

链表元素的查找

按序号查找

```
ListNode *Get (LinkList head, int i)
/* 按序号查找单链表的第i个节点，成功返回指针，失败返回NULL
*/
{
    ListNode *p;
    int j;
    if(ListEmpty(head))
        return NULL;
    if(i<1)
        return NULL;
    j=0;
    p=head;
    while(p->next!=NULL&&j<i)
    {
        p=p->next;
        j++
    }
    if(j==i)
        return p;
    else
        return NULL;
}
```

按照元素值查找

```
ListNode *LocateElem (LinkList head, DataType e)
/* 按内容查找，成功返回指针，失败返回NULL */
{
    ListNode *p;
    p=head->next;
    while(p)
    {
        if(p->data!=e)
            p=p->next;
        else
            break;
    }
    return p;
}
```

难点：在单链表中插入元素

原理

①：在单链表中找到需要插入位置的前驱节点，即i-1，并由指针pre指向该节点。

②：申请一个新节点空间，由p指向该节点，将e赋值给p所指向节点的数据域。

③：修改`*p`和`*pre`节点的指针域。

如图所示

表的第 i 个位置插入一个新元素 e 的步骤如下。

单链表中找到其直接前驱节点，即第 $i-1$ 个节点，并由指针 pre 指向该节点，如图 1.16

请一个新节点空间，由 p 指向该节点，将 e 赋值给 p 所指向节点的数据域。

*p 和*pre 节点的指针域，如图 1.17 所示。

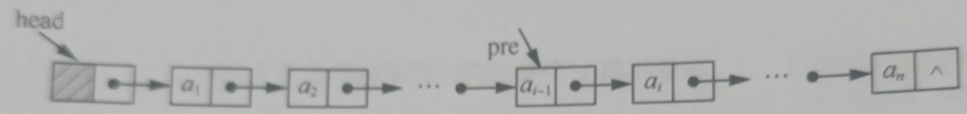


图 1.16　找到第 i 个节点的直接前驱节点

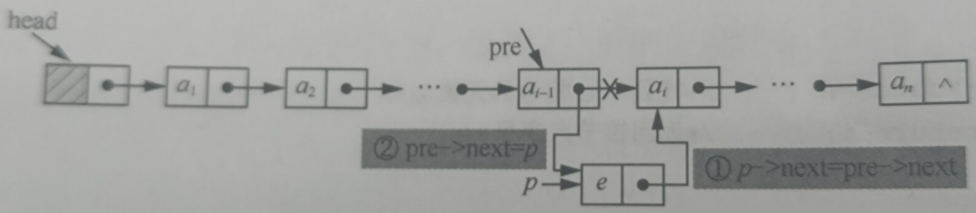


图 1.17　将新节点插入第 i 个位置

中插入元素 e 的算法实现如下。

```
List(LinkList head,int i,DataType e)
第 i 个位置插入一个节点，节点存放元素 e*/
```

线性表和链表的对比

存储类别	顺序存储结构	单链表
存储分配方式	用一段连续的存储单元依次存储线性表的数据元素	采用链式存储结构，用一组任意的存储单元存放线性表的元素
时间性能	查找O（1）、插入和删除O（n）	查找O（n）、插入和删除O（1）
空间性能	需要预分配存储空间，分大了浪费，小了容易发生上溢	不需要分配存储空间，只要有就可以分配，元素个数不受限制

通过上面的对比，可以得出一些经验性的结论：

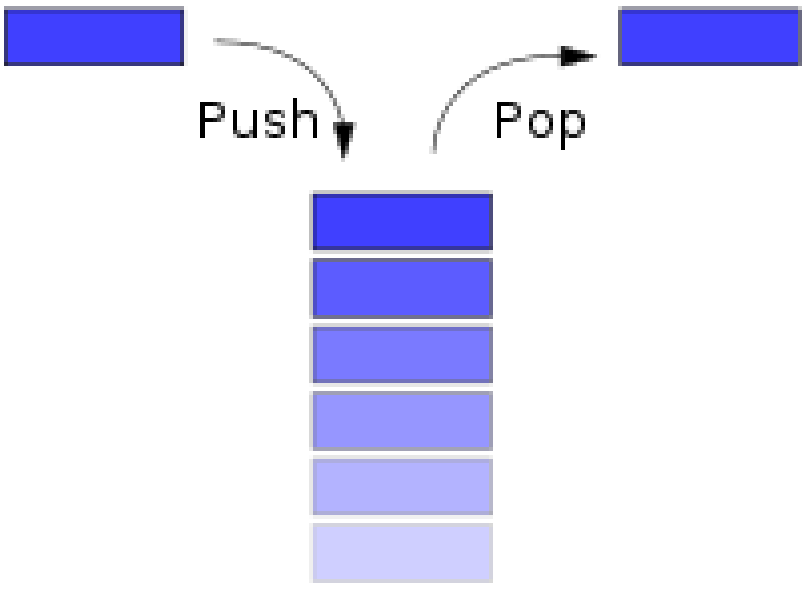
- 若线性表需要频繁查找，很少进行插入和删除操作时，宜采用顺序存储结构。若需要频繁插入和删除时，宜采用单链表结构。
- 当线性表中的元素个数变化较大或者根本不知道有多大时，最好用单链表结构，这样可以不需要考虑存储空间的大小问题。而如果事先知道线性表的大致长度，用顺序存储结构效率会高很多。

顺序表里面元素的地址是连续的，插入删除需要移动元素。
链表里面节点的地址不是连续的，是通过指针连起来的。插入删除不需要移动元素，不可以随机访问任一元素，所需空间与线性表长度成正比。

第二单元

- 栈，队列

2.1 栈



在栈S=(a1,a2,...,an)中，a1称为栈底元素，an称为栈顶元素。

- 上溢：太多；下溢：太少

初始化顺序结构栈

```
int StackEmpty(SeqStack *S)
/*初始化栈*/
{
    S->top=0;       /*把栈顶指针置为0*/
}
```

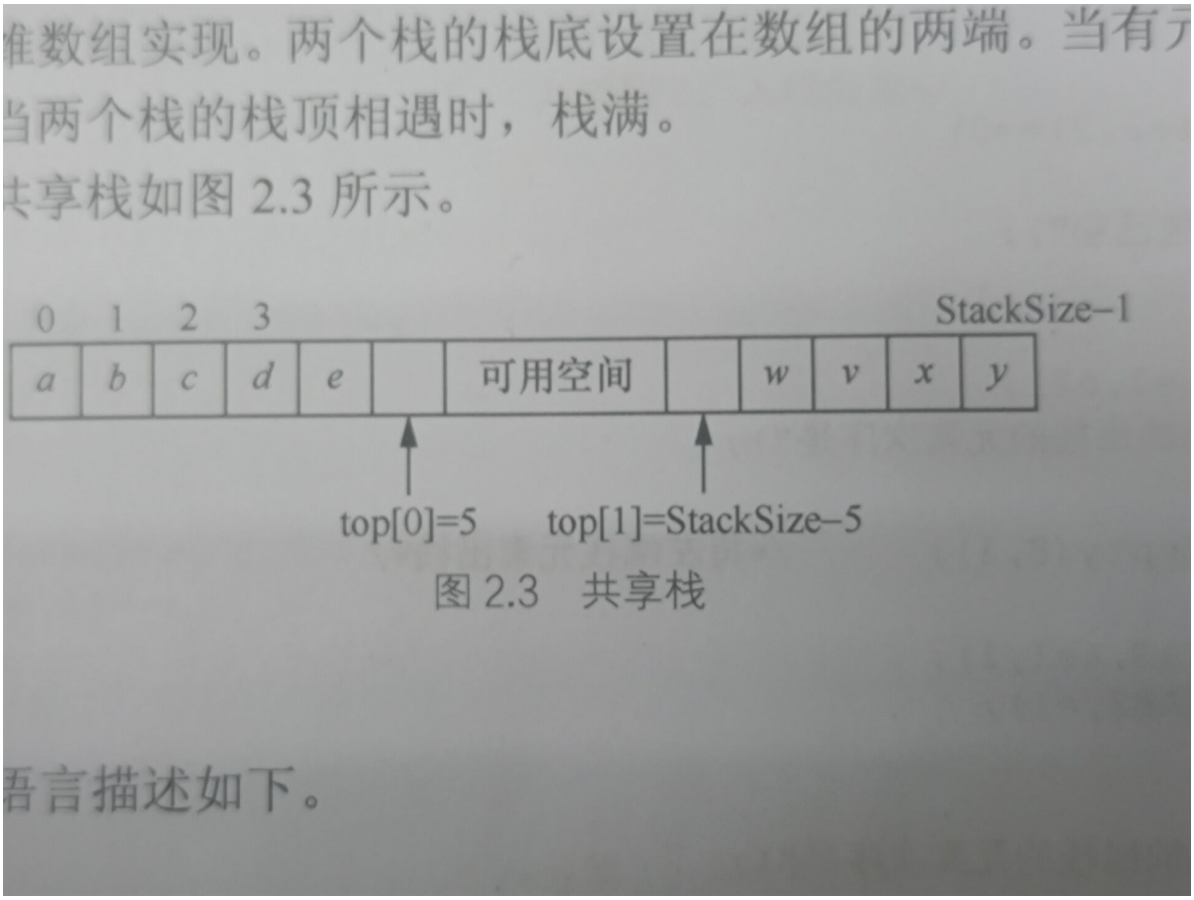
入栈出栈

```
int PushStack(SeqStack *S,DataType e)
/*将元素e入栈*/
{
    if(S->top>=StackSize)   /*若栈已满*/
    {
        printf("栈已满，不能入栈");
        return 0;
    }
    else
    {
        S->stack[S->top]=e;     /*入栈*/
        S->top++;               /*修改栈顶指针*/
        return 1;
    }
}
/*分割线*/
int PopStack (SeqStack *S,DataType *e)
/*将栈顶元素出栈，并将其赋给e */
{
    if(S->top==0)       /*栈为空*/
    {
        printf("栈中已经没有元素，不能进行出栈操作");
        return 0;
    }
    else
    {
        S->top--;           /*修改栈指针，即出栈*/
```

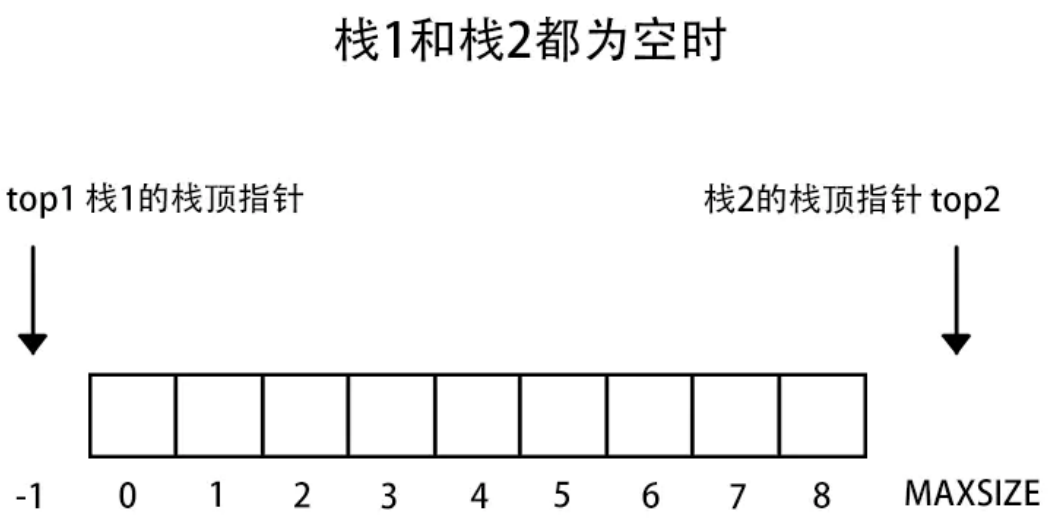
```
        *e=S->stack[S->top];        /*将出栈的元素赋给e */
        return 1;
    }
}
```

栈的共享空间

两栈共享空间结构：使用数组同时实现两个栈，即栈1和栈2；栈1为空时，栈1的栈顶指针指向-1；栈2为空时，栈2 的栈顶指针指向MAXSIZE；栈1和栈2添加元素时，都会向数据中间靠拢，当栈1的指针+1等于栈2的指针的时候，栈满。



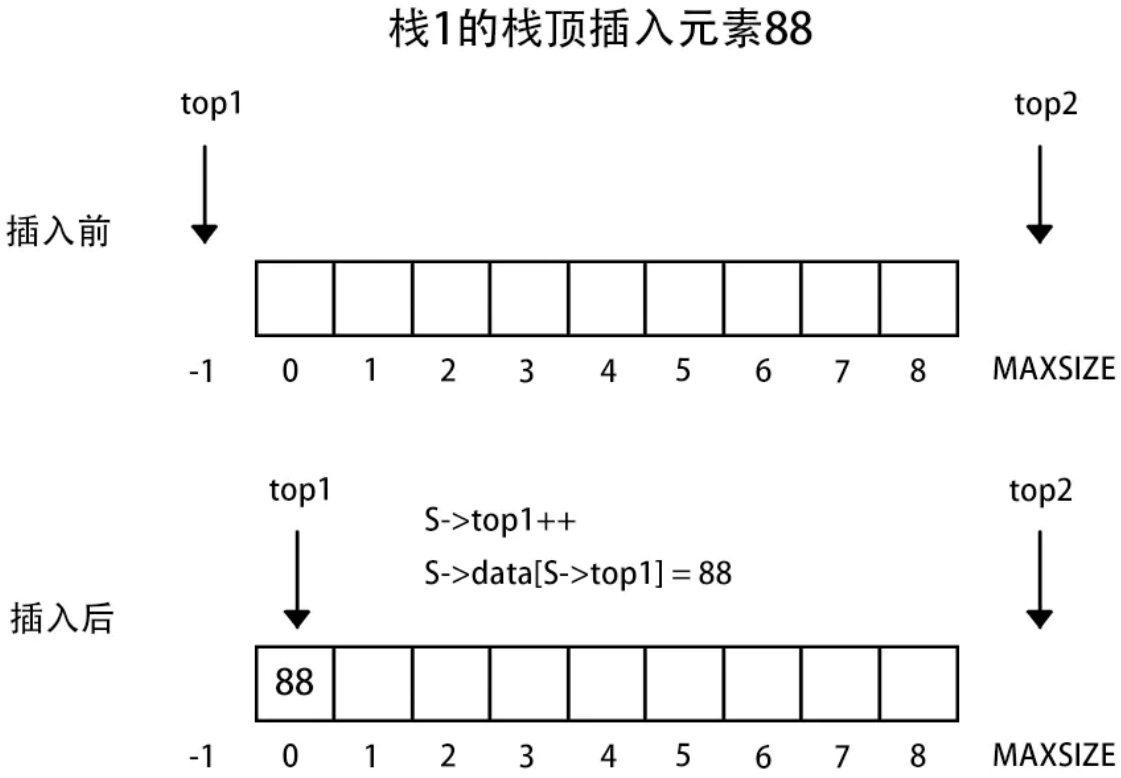
- 分别只能在对应的栈顶进行Push和Pop操作

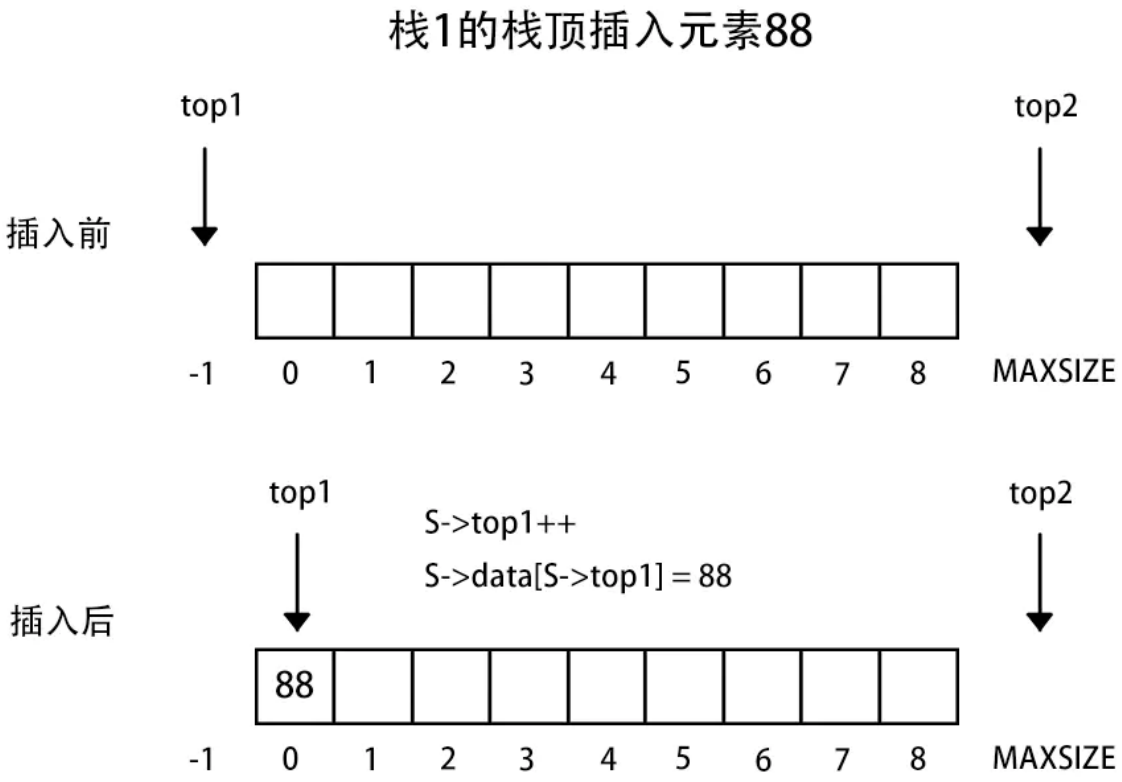


栈的时间复杂度

全部都为O(1)

其中包含读取栈，插入，删除时的时间复杂度都为O(1)。





优点：

- 具有记忆功能，可用于表达式求值等操作。
- 添加和删除元素不需要移动大量元素，只需要移动栈顶指针。
- 可有效利用剩下的栈空间。

缺点：

- 只适用于类型相同的两个栈。
- 两个栈是此消彼长的关系，导致两个栈分别的栈长是动态变化的，无法确定。

共享空间栈的入栈出栈

```
// 两栈共享空间
#include <stdio.h>
#include <malloc.h>
#include <time.h>

#define OK 1        // 执行成功
#define ERROR 0     // 执行失败
#define TRUE 1      // 返回值为真
```

```c
#define FALSE 0   // 返回值为假
#define MAXSIZE 20 // 存储空间初始分配大小

typedef int Status; // 函数返回结果类型
typedef int ElemType; // 元素类型

// 两栈共享结构
typedef struct {
    ElemType data[MAXSIZE]; // 用于存储元素值
    int top1; // 用于指示栈1的栈顶指针
    int top2; // 用于指示栈2的栈顶指针
}SqStack;

/**
 * 初始化栈
 * @param S 栈
 * @return 执行状态
 */
Status InitStack(SqStack *S) {
    S->top1 = -1; // 栈1的栈顶指针指向-1，此时栈1为空
    S->top2 = MAXSIZE; // 栈2的栈顶指针指向MAXSIZE，此时
栈2为空
    return OK;
}

/**
 * 清空栈中元素
 * @param S 栈
 * @return 执行状态
 */
Status ClearStack(SqStack *S) {
    S->top1 = -1; // 栈1的栈顶指针指向-1，此时栈1为空
    S->top2 = MAXSIZE; // 栈2的栈顶指针指向MAXSIZE，此时
栈2为空
    return OK;
}
```

```c
/**
 * 判断栈是否为空
 * @param S 栈
 * @return 执行状态
 */
Status StackEmpty(SqStack *S) {
    // 栈1和栈2都为空时，栈才为空
    if (S->top1 == -1 && S->top2 == MAXSIZE) {
        return TRUE;
    } else {
        return FALSE;
    }
}

/**
 * 获取栈中元素个数（栈1和栈2的元素总数）
 * @param S 栈
 * @return 执行状态
 */
int StackLength(SqStack *S) {
    return (S->top1 + 1) + (MAXSIZE - S->top2);
}

/**
 * 获取栈顶元素的值，存到元素e中
 * @param S 栈
 * @param e 用于存储栈顶元素的值
 * @param stackNumber 栈号
 * @return 执行状态
 */
Status GetTop(SqStack *S, ElemType *e, int stackNumber)
{
    // 获取栈1的栈顶元素值
    if (stackNumber == 1) {
        // 栈1为空时，获取栈顶元素失败
        if (S->top1 == -1) {
            return ERROR;
```

```
        }

        // 将栈1的栈顶元素的值赋值给e元素
        *e = S->data[S->top1];
    } else { // 获取栈2的栈顶元素值
        // 栈2为空，获取栈顶元素失败
        if (S->top2 == MAXSIZE) {
            return ERROR;
        }

        // 将栈2的栈顶元素的值赋值给e元素
        *e = S->data[S->top2];
    }

    return OK;
}

/**
 * 添加新元素e到栈顶
 * @param S 栈
 * @param e 新元素
 * @param stackNumber 栈号
 * @return 执行状态
 */
Status Push(SqStack *S, ElemType e, int stackNumber) {
    // 栈满时，添加失败
    if (S->top1 + 1 == S->top2) {
        return ERROR;
    }

    // 给栈1添加新元素
    if (stackNumber == 1) {
        S->data[++S->top1] = e; // 将新元素添加到栈1的栈
顶
    } else { // 给栈2添加新元素
        S->data[--S->top2] = e; // 将新元素添加到栈2的栈
顶
```

```
    }

    return OK;
}

/**
 * 弹出栈顶元素
 * @param S 栈
 * @param e 弹出元素
 * @param stackNumber 栈号
 * @return 执行状态
 */
Status Pop(SqStack *S, ElemType *e, int stackNumber) {
    // 栈1弹出元素
    if (stackNumber == 1) {
        // 栈为空时，弹出元素失败
        if (S->top1 == -1) {
            return ERROR;
        }
        *e = S->data[S->top1--]; // 将栈顶元素的值赋给e元素，栈1的栈顶指针减1
    } else { // 栈2弹出元素
        if (S->top2 == MAXSIZE) {
            return ERROR;
        }
        *e = S->data[S->top2++]; // 将栈顶元素的值赋给e元素，栈2的栈顶指针加1
    }

    return OK;
}

/**
 * 打印单个元素的值
 * @param e 元素
 * @return 执行状态
 */
```

```
Status visit(ElemType e) {
    printf("%d ", e);
    return OK;
}

/**
 * 从栈底开始遍历栈中元素
 * @param S 栈
 * @return 执行状态
 */
Status StackTraverse(SqStack S) {
    int i = 0; // 指示器，用于指示栈顶指针的位置

    printf("栈1中的元素为: [ ");
    // 指示器位置小于栈1的栈顶指针
    while (i <= S.top1) {
        visit(S.data[i++]); // 打印i位置元素，i向下一个元素移动
    }
    printf("]\n");

    i = MAXSIZE - 1;
    printf("栈2中的元素为: [ ");
    while (i >= S.top2) {
        visit(S.data[i--]);
    }
    printf("]\n");
    return OK;
}

int main() {
    int j; // 用于遍历
    SqStack s; // 栈
    ElemType e; // 元素

    // 如果初始化成功
    if (InitStack(&s) == OK) {
```

```
        // 向栈1中插入5个元素
        for (j = 1; j <= 5; j++) {
            Push(&s, j, 1); // 向栈1的栈顶插入元素j
        }

        // 向栈2中插入5个元素
        for (j = MAXSIZE; j > MAXSIZE - 5; j--) {
            Push(&s, j, 2); // 向栈2的栈顶插入元素j
        }
    }

    printf("栈中的元素如下：\n");
    StackTraverse(s); // 遍历栈中元素

    Pop(&s, &e, 1); // 弹出栈1的栈顶元素
    printf("弹出的栈顶元素为：e = %d\n", e);
    printf("弹出一个元素之后，栈是否为空：%s\n",
StackEmpty(&s) == TRUE ? "是" : "否");

    GetTop(&s, &e, 1); // 获取栈1的栈顶元素的值
    printf("栈1的栈顶元素的值为：e = %d\n", e);

    GetTop(&s, &e, 2); // 获取栈1的栈顶元素的值
    printf("栈2的栈顶元素的值为：e = %d\n", e);

    printf("栈的长度为：%d\n", StackLength(&s)); // 获取
栈的长度

    ClearStack(&s); // 清空栈中元素
    printf("清空栈后，栈是否为空：%s\n", StackEmpty(&s)
== TRUE ? "是" : "否");

    return 0;
}
```

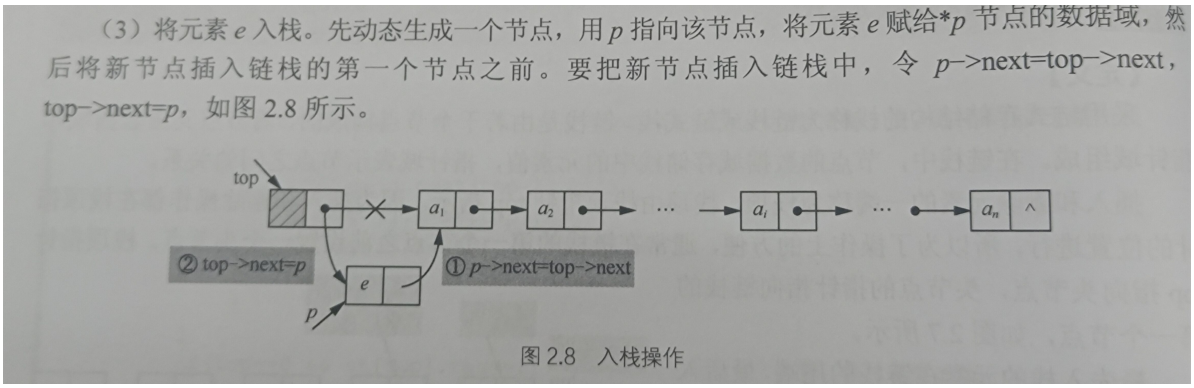
链式结构栈

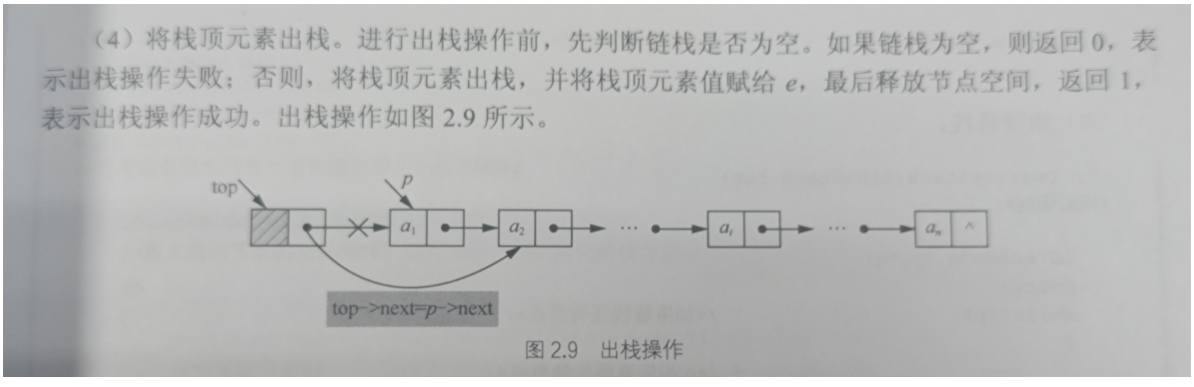
也叫链栈，与链表类似，运用指针指向next

初始化链栈

```
void InitStack(Linktack *top)
/*链栈的初始化*/
{
    if((*top=
(LinkStack)malloc(sizeof(LStackNode)))==NULL)
                    /*为头节点分配空间*/
        exit(-1);
    (*top)->next=NULL;      /*将链栈的头节点指针域置为空*/
}
```

元素的入栈与出栈



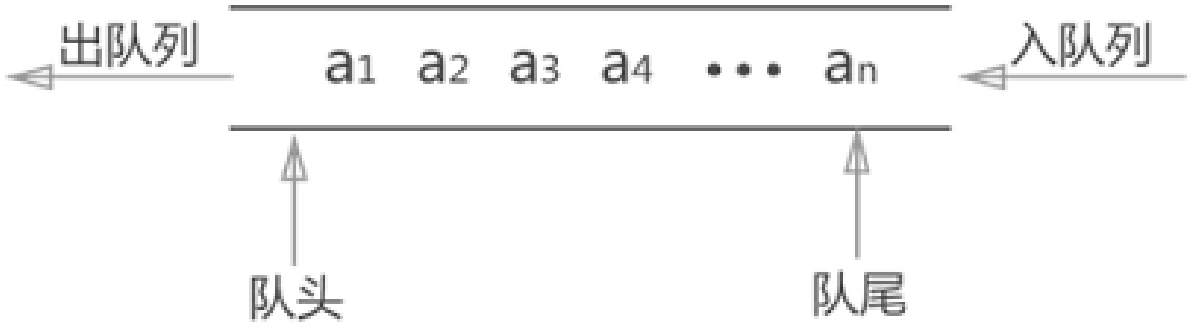



```
int PushStack(LinkStack top,DataType e)
/*将元素e入栈，入栈返回1*/
{
    LStackNode *p;    /*定义指针p，指向新生成的节点*/
```

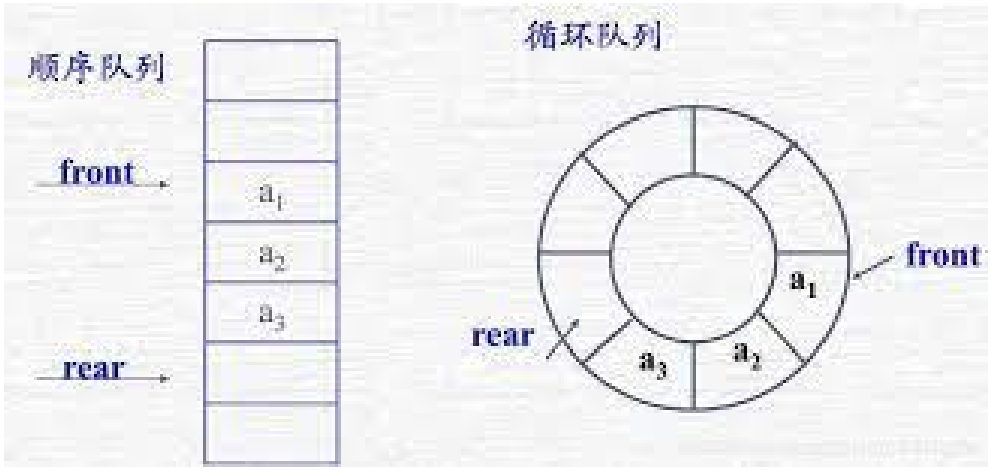
```
    if((p=
(LStackNode*)malloc(sizeof(LStackNode)))==NULL) /*生成新
节点*/
    {
        printf("内存分配失败");
        exit(-1);
    }
    p->data=e;
    p->next=top->next;
    top->next=p;
    return 1;
}
/*分割线，下面是出栈操作*/
int PopStack(LinkStack top,DataType *e)
/*将栈顶元素出栈*/
{
    LStackNode *p;
    p=top->next;
    if(!p)            /*判断栈是否为空*/
    {
        printf("栈已空");
        return 0;
    }
    top->next=p->next;      /*出栈*/
    *e=p->next;             /*将出栈元素值赋给e*/
    free(p);                 /*释放p指向的节点的空间*/
    return 1;
}
```

2.2队列

- 如图所示，队列和栈不同，采用先进先出的形式，有列头与列尾。

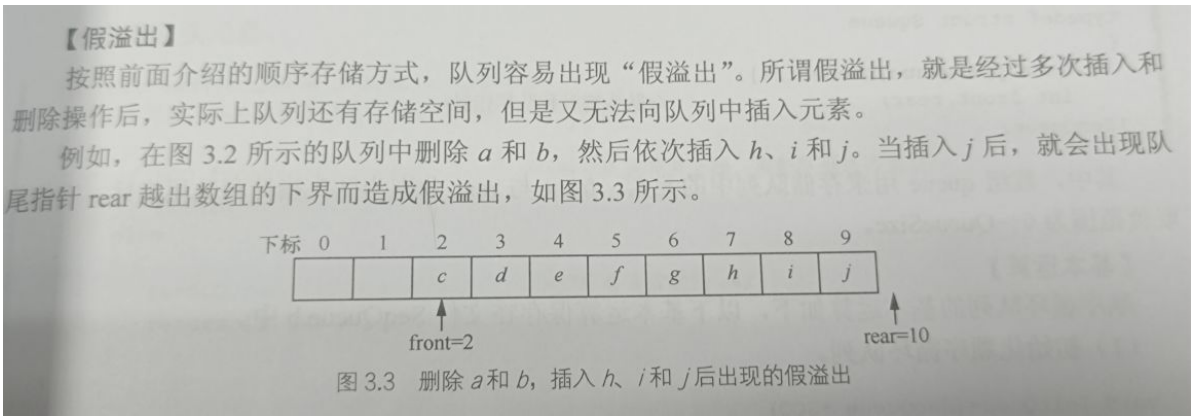


顺序队列与顺序循环队列



顺序队列容易理解，队尾入列，队头出列，头尾两个指针-------'front'和'rear'

> 这种队列方式会引起假溢出的问题


【假溢出】

按照前面介绍的顺序存储方式，队列容易出现“假溢出”。所谓假溢出，就是经过多次插入和删除操作后，实际上队列还有存储空间，但是又无法向队列中插入元素。

例如，在图 3.2 所示的队列中删除 *a* 和 *b*，然后依次插入 *h*、*i* 和 *j*。当插入 *j* 后，就会出现队尾指针 rear 越出数组的下界而造成假溢出，如图 3.3 所示。

图 3.3　删除 *a* 和 *b*，插入 *h*、*i* 和 *j* 后出现的假溢出

- 于是有了顺序循环队列

当队尾指针rear或队头指针front到达存储空间的最大值时(假定队列的存储空间为QueueSize)，让队尾指针或队头指针转化为0，就可以将元素插入队列的空闲存储单元中，从而有效地利用存储空间，消除假溢出。

顺序循环队列的初始化与实现

```
void InitQueue(SeqQueue *SCQ)
/*顺序循环队列的初始化*/
{
    SCQ->front=SCQ->rear=0;  /*把队头指针和队尾指针同时置为0*/
}
```

元素的入队与出队

入队的核心语句`rear=(rear+1)%QueueSize`

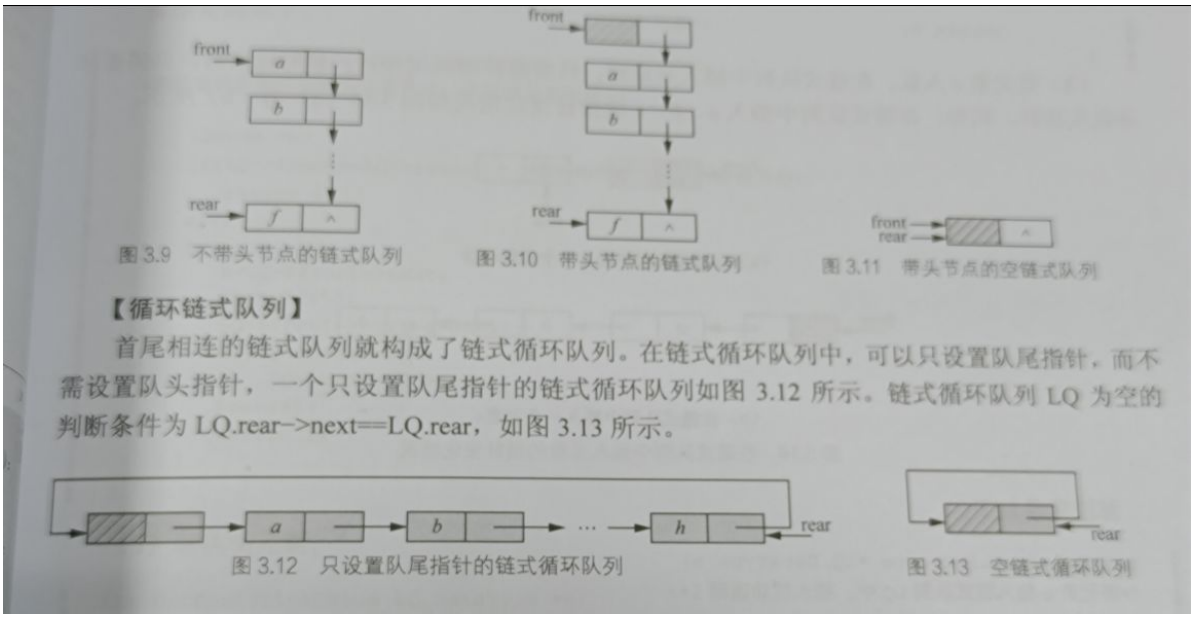
出队的核心语句`front=(front+1)%QueueSize`

简而言之，入队和出队的语句就是让rear和front分别+1，如果超出QueueSize则mod

```
int EnQueue(SeqQueue *SCQ,DataType e)
/*将元素e插入进顺序循环队列SCQ中*/
{
    if(SCQ->front==(SCQ->rear+1)%QueueSize)
    /*插入新元素前，判断队尾指针是否达到最大值，防止假溢出*/
        return 0;
    SCQ->queue[SCQ->rear]=e;             /*队尾插入e*/
    SCQ->rear=(SCQ->rear+1)%QueueSize;  /*队尾指针后移动一个位置*/
    return 1;
}
/*分割线，下面为出队*/
int DeQueue(SeqQueue *SCQ,DataType *e)
{
```

```
    if(SCQ->front==SCQ->rear)      /*出队前，检查队列是否
为空*/
        return 0;
    else
    {
        *e=SCQ->queue[SCQ->front];      /*要出队的元素值
赋给e*/
        SCQ->front=(SCQ->front+1)%QueueSize;
        /*队头指针向后移一个位置，指向新的队头*/
        return 1;
    }
}
```

循环链式队列



初始化循环链式队列

```
void InitQueue(LInkQueue *LQ)
/*初始化链式队列*/
{
    LQ->front=LQ->rear=(LQNode*)malloc(sizeof(LQNode));
    if(LQ->front==NULL)
        exit(-1);
    LQ->front->next=NULL;       /*头节点指针域为空*/
}
```

第三单元

- 树，二叉树，哈夫曼树，二叉排序树

> 递归是树的固有特性

数的基本术语

1.结点的度(Degree) :结点的子树个数

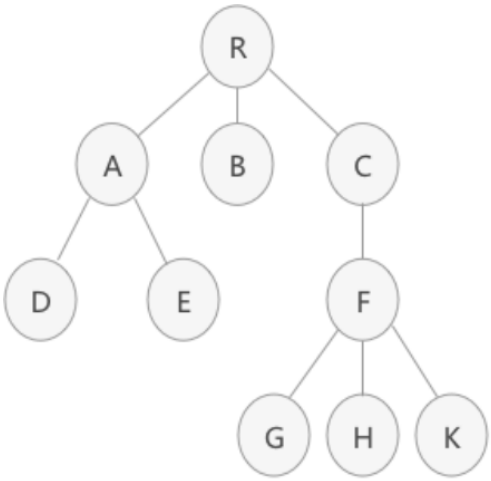
2.树的度:树的所有结点中最大的度数
3.叶结点(Leaf) :度为0的结点
4.父结点(Parent) :有子树的结点是其子树的根结点的父结点
5.子结点(Child):若A结点是B结点的父结点，则称B结点是A结点的子结点;子结点也称孩子结点。
6.兄弟结点(Sibling) :具有同一父结点的各结点彼此是兄弟结点。

7.路径和路径长度:从结点n，到nk的路径为一个结点序列n1，n2.,... ，nk ，η是n+1的父结点。路径所包含边的个数为路径的长度。
8.祖先结点(Ancestor):沿树根到某一结点路径上所有结点都是这个结点的祖先结点。
9.子孙结点(Descendant):某一结点的子树中的所有结点是这个结点的子孙。
11.结点的层次(Level) :规定根结点在1层，其它任一结点的层数是其父结点的层数加1。
12.树的深度(Depth):树中所有结点中的最大层次是这棵树的深度。

双亲表示法

节点用data来保存，双亲位置用parent来保存，parent的值=数根节点的数组下标

例如，使用双亲表示法存储图 1（A）中的树结构时，数组存储结果为（B）：



(A)

数组下标	data	parent
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

(B)

F的双亲是C，而C的数组下标是3，所以F的parent为3，适用于根据给定节点查找其双亲结点。

孩子表示法

- 和双亲表示法相反，孩子表示法的指针是指向孩子的

例如，使用孩子表示法存储图 1 （A），存储效果如图 2：

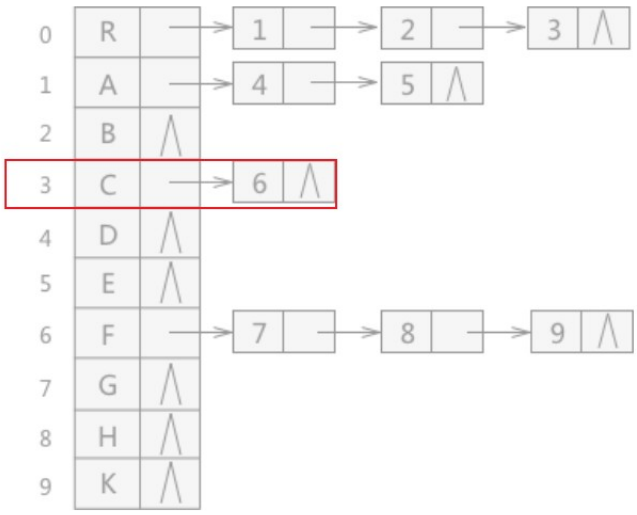


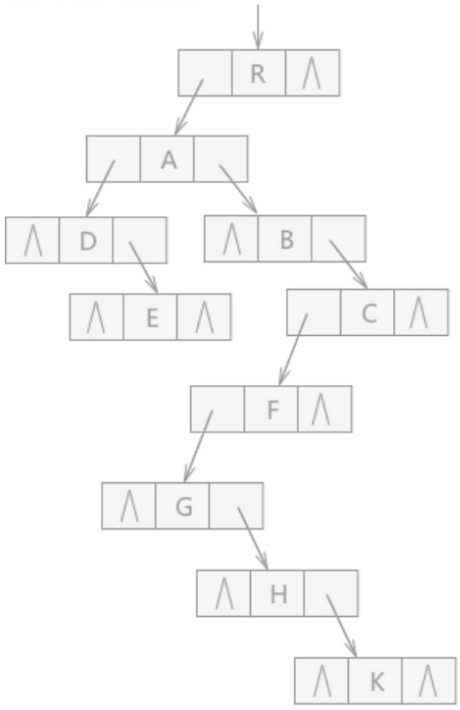
图 2 孩子表示法

使用孩子表示法存储的树结构，正好和双亲表示法相反，适用于查找某结点的孩子结点，不适用于查找其父结点。

孩子-兄弟节点表示法

通过孩子兄弟表示法，普通树转化为了二叉树，所以孩子兄弟表示法又被称为“二叉树表示法”或者“二叉链表表示法”。

例如，用孩子兄弟表示法表示图 1 （A）的普通树，存储结果为：



数据域	指向第一个子节点	指向下一个兄弟节点
data	firstchild	nextsibling

- 根的右节点指向的下一个兄弟节点不存在，所以根的右节点为非空

题型一般都是告知度、结点等，进行计算

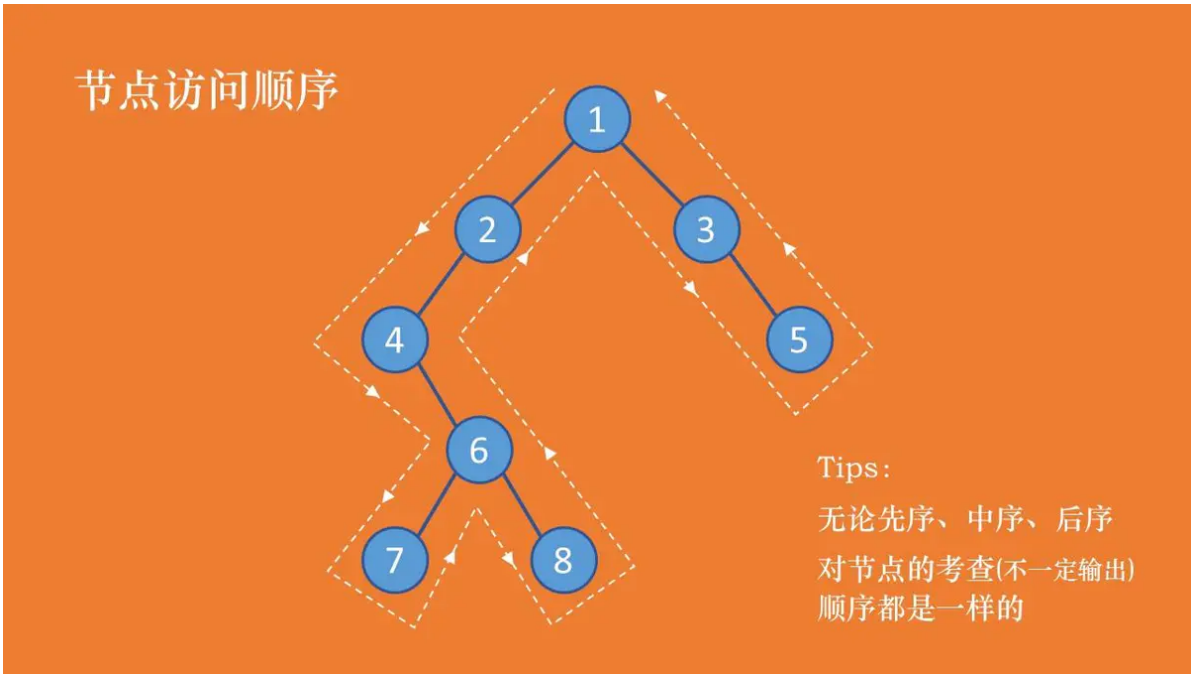
例：一棵树度为4，其中度为1，2，3，4的结点个数分别为4，2，1，1，则这棵树的叶子节点个数为多少？

解：因为任一棵树中，结点总数=度数*该度数对应的结点数+1，所以：

$$n0+4+2+1+1=(0\times n0 + 1\times 4 + 2\times 2 + 3\times 1 + 4\times 1)+1$$

二叉树的遍历

二叉树的先序遍历，即正常顺序，先把所有左边的访问完了，再访问右边(根左右)



- 先序：考察到一个节点后，即刻输出该节点的值，并继续遍历其左右子树。(根左右)

先序：1 2 4 6 7 8 3 5

- 中序：先中间根节点对称轴，然后再左最后右(左根右)

中序：4 7 6 8 2 1 3 5

- 后序：从最底下开始，遍历完左右子树后，再输出该节点的值。(左右根)

后序：7 8 6 4 2 5 3 1

哈夫曼树和哈夫曼编码

- 哈夫曼树又称最优二叉树，是一种带权路径长度最小的二叉树
- 二叉树的带权路径长度

$$WPL = \sum_{k=1}^{n} w_k l_k$$

路径×权值之和，感觉挺简单的

哈夫曼树构造过程

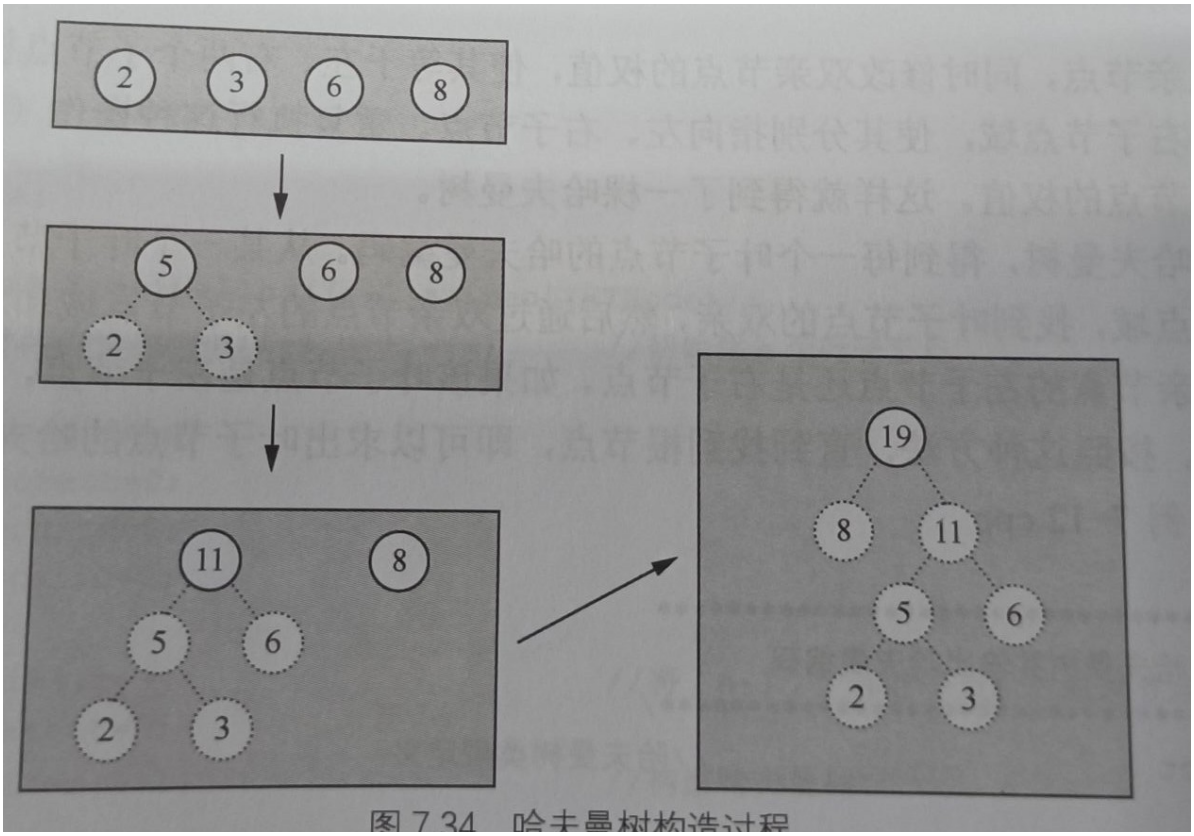


图 7.34 哈夫曼树构造过程

哈夫曼码

左为0，右为1，路径连起来就是了

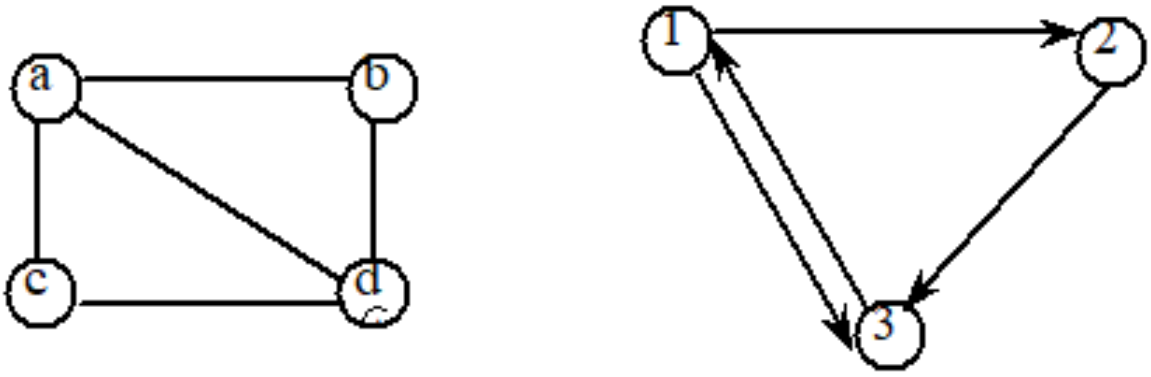
第四单元

- 图，图的基本术语，存储结构，遍历方式，连通性
- 有向图中顶点之间的关系称为弧，无向图中顶点之间的关系称为边

图的定义

G=(V,E)，V为顶点集，E为边集。设图有n个顶点，V={v1,v2,v3,.....,vn}

有向图与无向图，有向图有箭头，无向图没有箭头



(a) 无向图 G1　　(b) 有向图 G2

图 7-1　无向图和有向图

⑴具有n个顶点，n(n-1)/2条边的图，称为完全无向图，

⑵具有n个顶点，n(n-1) 条弧的有向图,称为完全有向图。

⑶完全无向图和完全有向图都称为完全图。

⑷对于一般无向图，顶点数为n，边数为e，则 0≤e≤n(n-1)/2。

⑸对于一般有向图，顶点数为n，弧数为e，则 0≤e≤n(n-1)。

⑹当一个图接近完全图时，则称它为稠密图，

⑺当一个图中含有较少的边或弧时，则称它为稀疏图。

顶点的入度与出度

度/出度/入度：

⑴在图中，一个顶点依附的边或弧的数目，称为该顶点的度。

⑵在有向图中，一个顶点依附的弧头数目，称为该顶点的入度。

⑶一个顶点依附的弧尾数目，称为该顶点的出度，

⑷某个顶点的入度和出度之和称为该顶点的度。

(5)若图中有n个顶点，e条边或弧，第i个顶点的度为di，则有e=1/2*Σ(1<=i<=n,di)

关联矩阵：

$$m_{ij}=\begin{cases}1, & v_i\text{是有向边}a_j\text{的始点}\\ -1, & v_i\text{是有向边}a_j\text{的终点}\\ 0, & v_i\text{是有向边}a_j\text{的不关联点}\end{cases}$$

邻接矩阵：简单

邻接表：

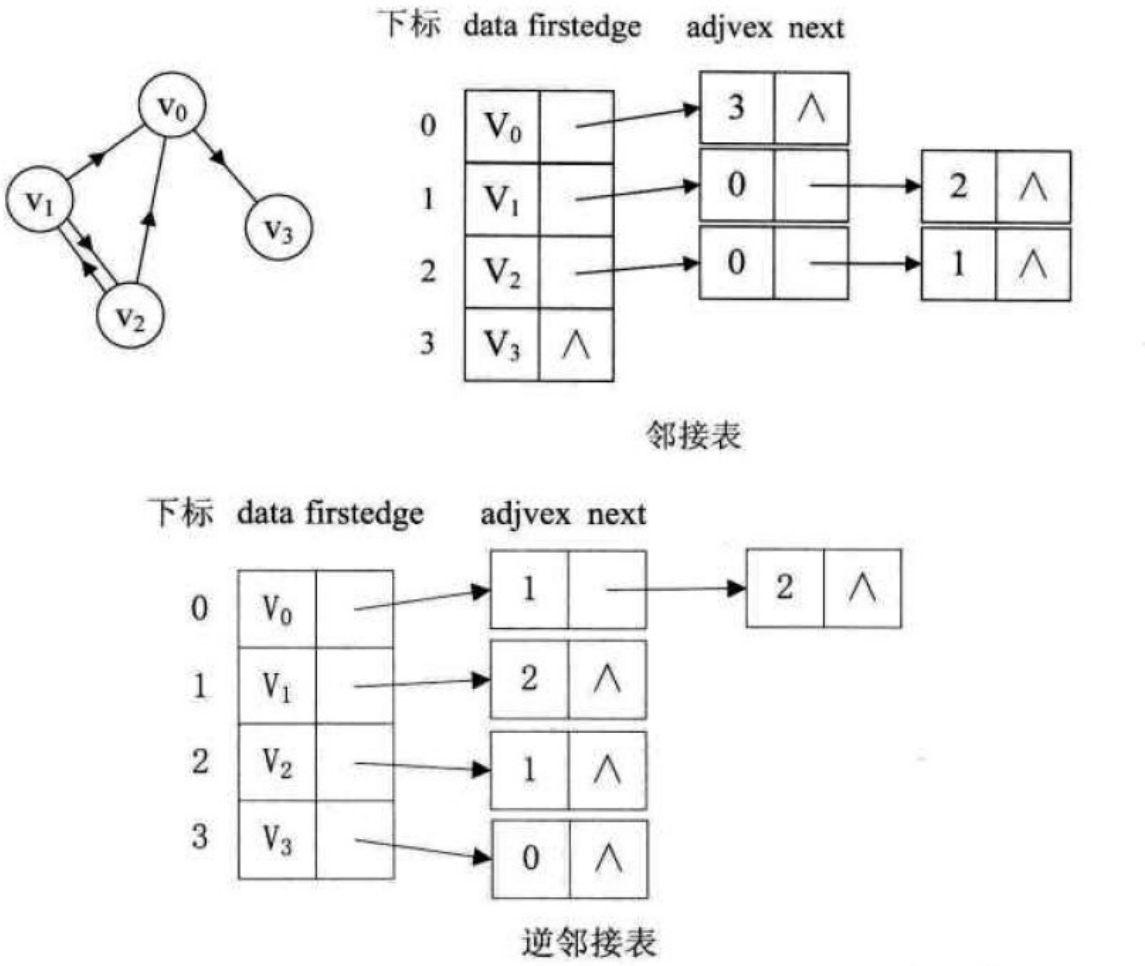


图 7-4-7

无向图：

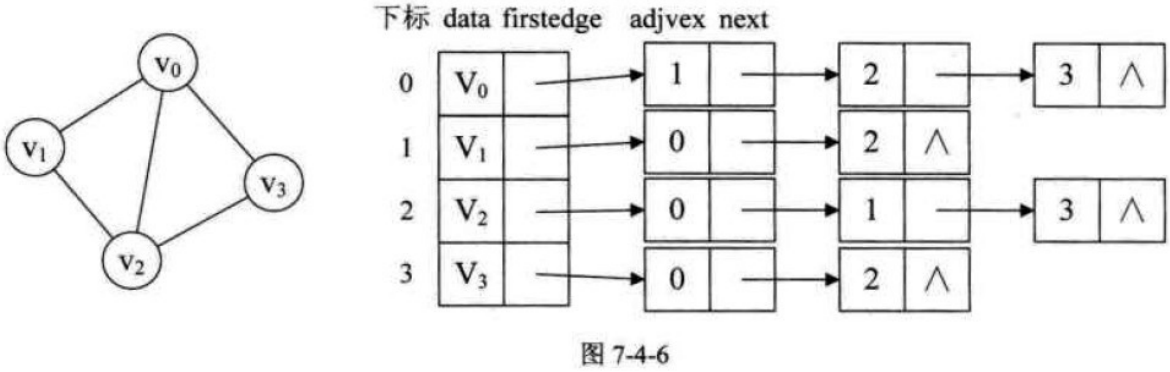


图 7-4-6

深度优先搜索类似于树的先序

广度优先搜索类似于树的层次遍历

第五单元

- 算法，排序