第 1 章 绪论

一、基本概念和术语

数据：对客观事物的符号表示，在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。

数据元素：数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。一个数据元素可由若干个**数据项**组成。数据项是数据的不可分割的最小单位。

数据对象：是性质相同的数据元素的集合，是数据的一个子集。

数据结构：是相互之间存在一种或多种特定关系的数据元素的集合。

4 类基本结构：1）集合 2）线性结构 3）树形结构 4）图状结构或网状结构

逻辑结构：结构定义中的“关系”描述的是数据元素之间的逻辑关系。集合结构、线性结构、树形结构、图形结构都属于逻辑结构。

物理结构：又称存储结构。是指数据结构在计算机中的表示（又称映像）。顺序存储结构和链接存储结构属于物理结构。

顺序映像的特点：借助元素在存储器中的相对位置来表示数据元素之间的逻辑关系。

非顺序映像的特点：借助指示元素存储地址的指针表示数据元素之间的逻辑关系。

数据类型：一个值的集合和定义在这个值集上的一组操作的总称。

抽象数据类型（ADT）：一个数学模型以及定义在该模型上的一组操作。

ADT 抽象数据类型名{

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

基本操作名（参数名）

初始条件：<初始条件描述>

操作结果：<操作结果描述>

}ADT 抽象数据类型名

三元组（D,S,P）其中 D 是数据对象，S 是对 D 上的关系集，P 是对 D 的基本操作集。

二、算法和算法分析

1、算法

算法:对特定问题求解步骤的一种描述,它是指令的有限序列,其中每一条指令表示一个或多个操作。

特性：有穷性、确定性、可行性、输入、输出

“好”算法的要求：正确性、可读性、健壮性、效率与低存储量需求

算法效率的度量

时间复杂度

O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)<O(2^n)<O(n!)<O(n^n)

像n^3，过大的 n 会使结果变得不现实。

空间复杂度

第 2 章 线性表

线性表：最常用且最简单的一种数据类型，是 n 个数据元素的有限序列。除第一元素没有直接前驱和最后一个元素没有直接后继外，其他元素都有且仅有一个直接前驱和一个直接后继。

一. 线性表的顺序表示和实现

线性表的顺序表示：指的是一组地址连续的存储单元依次存储线性表的数据元素。

线性表的第 i 个数据元素 ai 的存储位置：LOC（ai）= LOC（a1）+(i-1)*l

LOC（a1）是线性表第一个数据元素的存储位置。l 是每个元素占用 l 个存储单元。

顺序表的顺序存储结构是一种随机存储结构。特点：地址连续；依次存放；随机存储；类型相同；

线性表顺序存储的优缺点：

优点：无须为表示表中元素之间的逻辑关系而增加额外的存储空间；可以快速地存取表中任一位置的元素。

缺点：插入和删除操作需要移动大量元素；当线性表长度变化较大时，难以确定存储空间的容量；造成存储空间的“碎片”。

1、头文件引用和一些定义

```
#include <stdio.h>
#include <malloc.h>
#include <assert.h>

#define SEQLIST_INIT_SIZE 8
#define INC_SIZE 5
typedef int ElemType;

typedef struct SeqList
{
    ElemType* base;  //基地址
    int capacity;    //容量
    int size;        //线性表当前大小
}SeqList;
```

2、初始化

```
void InitSeqList(SeqList* list)
{
    list->base = (ElemType*)malloc(sizeof(ElemType) * SEQLIST_INIT_SIZE);
    assert(list->base != NULL);
    list->size = 0;
    list->capacity = SEQLIST_INIT_SIZE;
}
```

3、自动增加空间

```
_Bool Inc(SeqList* list)
{
    //自动分配空间 realloc，头文件也是 malloc.h
    ElemType* newbase = (ElemType*)realloc(list->base, sizeof(ElemType) * (list->capacity +
INC_SIZE));
    if (!newbase) return 0;  //存储分配失败
    list->base = newbase;  //新基址
    list->capacity += INC_SIZE;  //增加存储容量
    return 1;
}
```

4、尾插 push_back

```
void push_back(SeqList* list, ElemType x)
{
    if (list->size >= list->capacity && !Inc(list))
    {
        printf("顺序表已满，%d 无法插入！\n",x);
        return;
    }

    list->base[list->size] = x;
    list->size++;
}
```

5、头插 push_front

```
void push_front(SeqList* list, ElemType x)
{
    if (list->size >= list->capacity && !Inc(list))
    {
        printf("顺序表已满，无法插入！");
        return;
    }

    for (int i = list->size; i>0; i--)
    {
        list->base[i] = list->base[i-1];
    }
    list->base[0] = x;
    list->size++;
}
```

6、尾删 pop_back

```
void pop_back(SeqList* list)
{
    if (list->size == 0)
    {
        printf("表内没有元素，无法删除！");
        return;
    }
    list->size--;
}
```

7、头删 pop_front

```
void pop_front(SeqList* list)
{
    if (list->size == 0)
    {
        printf("表内没有元素，无法删除！");
        return;
    }

    for (int i =1; i<list->size; i++)
    {
        list->base[i-1] = list->base[i];
    }
    list->size--;
}
```

8.按位置插入 insert_pos

```
void insert_pos(SeqList* list, int pos, ElemType x)
{
    if (pos<0 && pos>list->size)
    {
        printf("位置不合法，不能插入！");
        return;
    }
    if (list->size >= list->capacity && !Inc(list))
    {
        printf("顺序表已满，无法插入！");
        return;
    }

    for (int i = list->size; i > pos; i--)
    {
        list->base[i] = list->base[i - 1];
    }
    list->base[pos] = x;
    list->size++;
}
```

9、按位置删除 delete_pos

```
int delete_pos(SeqList* list, int pos)
{
    if (pos<0 && pos>list->size)
    {
        printf("位置不合法，不能删除！");
        return 0;
    }

    return list->base[pos];
    for (int i = pos; i<list->size; i++)
    {
        list->base[i] = list->base[i + 1];
    }
    list->size--;

}
```

10、按值删除 delete_val

```
int delete_val(SeqList* list,ElemType x)
{
    if (list == 0)
    {
        printf("表中没有数据，无法删除！");
        return 0;
    }

    //遍历表，查找指定数据的下标，然后删除下标中的数据
    //如果没有找到指定数据的下标，显示没有该数据
    for (int i=0; i<list->size; i++)
    {
        if (list->base[i] == x)
        {
            for (int j = i; j < list->size; j++)
            {
                list->base[j] = list->base[j + 1];
            }
            list->size--;

        }
    }
    return -1;

}
```

11、遍历线性表 show_list

```
void show_list(SeqList *list)
{
```

```
    for (int i=0; i<list->size; i++)
    {
        printf("%d",list->base[i]);
        printf(" ");
    }
    printf("\n");
}
```

12、查找 find

```
int find(SeqList* list, ElemType x)
{
    for (int i = 0; i < list->size; i++)
    {
        if (x == list->base[i])
            return i;
    }
    return -1;
}
```

13、求表长 length

```
int length(SeqList* list)
{
    int cout=0;
    for (int i=0; i<list->size; i++)
    {
        cout++;
    }
    return cout;
}
```

14、排序 sort（冒泡排序）

```
void sort(SeqList* list)
{//控制排序趟数
    for (int i=0; i<list->size-1; ++i)
    {
        //控制交换次数
        for(int j=0; j<list->size-i-1; ++j)
        {
            //如果 j 大于 j+1 就把两者交换
            if (list->base[j] > list->base[j + 1])
            {
                ElemType temp;
                temp = list->base[j];
                list->base[j] = list->base[j + 1];
                list->base[j + 1] = temp;
            }
        }
    }
}
```

15、逆置 resver

```
void resver(SeqList* list)
{
    if (list->size <= 1) //如果表中没有数据或者只有一个数据则无法逆置
        return;

    int low = 0;
    int high = list->size - 1;
    ElemType temp; //中间变量
    //如果 low 小于 high 就把两者交换，直到 low 大于 high 交换结束。
    while (low < high)
    {
        temp = list->base[low];
        list->base[low] = list->base[high];
        list->base[high] = temp;
        low++;
        high--;
    }
}
```

16、清空 clear

```
void clear(SeqList* list)
{
    list->size = 0;
}
```

17、销毁表 destroy

```
void destroy(SeqList* list)
{
    free(list->base);
    list->base = NULL;
    list->capacity = 0;
    list->size = 0;
}
```

18、合并两个有序顺序表

```
void MergeList(SeqList* list, SeqList *la, SeqList *lb)
{
    //初始化 list
    list->capacity = la->size + lb->size;
    list->base = (ElemType*)malloc(sizeof(ElemType) * list->capacity);
    assert(list->base != NULL);
    list->size = la->size + lb->size;

    int ia = 0;
    int ib = 0;
    int ic = 0;
```

```
    while (ia < la->size && ib < lb->size)
    {
        if (la->base[ia] < lb->base[ib])
            list->base[ic++] = la->base[ia++];
        else
            list->base[ic++] = lb->base[ib++];
    }

    while (ia < la->size)
    {
        list->base[ic++] = la->base[ia++];
    }

    while (ib < lb->size)
    {
        list->base[ic++] = lb->base[ib++];
    }
```

二、线性表的链式表示和实现

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素(这个存储单元可以是连续的，也可以是不连续的)

结点：包含两个域，其中一个是存储数据元素信息的域叫数据域；存储直接后继存储位置的域称为指针域。

线性链表表示线性表时数据元素之间的逻辑关系是由结点中的指针指示的。

头指针和头结点的异同：

头指针：头指针是指链表指向第一个结点的指针，若链表有头结点，，则指向头结点；头指针具有标识作用，所以常用头指针冠以链表的名字；无论链表是否为空，头指针均不为空，头指针是链表的必要元素。

头结点：头结点是为了操作的统一和方便而设立的，放在第一元素的结点之前，其数据域一般无意义（也可存放链表的长度)；有了头结点，对在第一个元素结点前插入结点和删除第一个结点，其操作与其他结点的操作就统一了；头结点不一定是链表必须要素。

单链表结构和顺序存储结构优缺点：

	存储分配方式	时间性能	空间性能
顺序存储结构	顺序存储结构用一段连续的存储单元依次存储线性表的数据元素。	查找：O（1） 插入和删除：顺序存储结构需要平均移动表长一半的元素，时间为O(n)	顺序存储结构需要预先分配存储空间，分大了浪费，分小了易发生上溢。
单链表结构	单链表采用链式存储结构，用一组任意的存储单元存放线性表的元素。	查找：O（n） 插入和删除：单链表在限制某位置的指针后，插入和删除时间仅为O(1)	单链表不需要分配存储空间，只要有就能分配，元素个数也不受限制。

有无头结点的问题：

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

typedef int ElemType;

typedef struct ListNode
{
    ElemType data; //数据域
    struct ListNode* next;  //指针域
}ListNode,*list;
```

1.初始化没有头结点的链表

```
void InitList(list* head)  //ListNode **head
{
    *head = NULL;  //head 是一个指针，*head 是指向的内容
}
```

2.创建没有头结点的链表(尾插)

```
void CreatList_back(list* head)
{
    *head = (ListNode *)malloc(sizeof(ListNode));
    if (*head == NULL)
        return;
```

```
    (*head)->data = 1;
    (*head)->next = NULL;

    ListNode *p = *head;
    for (int i=2; i<=10; i++)
    {
        ListNode* s = (ListNode*)malloc(sizeof(ListNode));
        if (s == NULL)
            return;
        s->data = i;
        s->next = NULL;

        p->next = s;  //每次都让前一个结点的指针指向刚创建这个结点
        p = s;   //再让 p 指向刚创建的这个结点来进行下一轮循环
    }
}
```

3.创建没有头结点的链表(头插)

```
void CreatList_front(list* head)
{
    *head = (ListNode*)malloc(sizeof(ListNode));
    if (*head == NULL)  //每次用 malloc 分配时都要判断内存是否分配成功，否则会报“取消对 NULL 指针的‘xxx’的引用”的警告
        return;
    (*head)->data = 1;
    (*head)->next = NULL;

    for (int i = 2; i <= 10; i++)
    {
        ListNode* s = (ListNode*)malloc(sizeof(ListNode));
        if (s == NULL)
            return;
        s->data = i;
        s->next = *head; //把新创建的结点指向头指针指向的结点
        *head = s;  //再把头指针指向这个新创建的结点，实现每次插入都在头部
    }
}
```

4.遍历没有头结点的链表 ShowList

```
void ShowList(list *head)
{
    ListNode* p = *head;
    for (int i = 1; i <= 10; ++i)
    {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}
```

5.初始化有头结点的链表

```
void H_InitList(list* head)
{
    *head = (ListNode*)malloc(sizeof(ListNode));  //头指针指向称为头结点的结点
    if (*head == NULL)
        return;
    (*head)->next = NULL;  //头结点的 next 置为空
}
```

6.创建有头结点的链表(尾插)

```
void H_CreatList_back(list* head)
{
    ListNode* p = *head;
    for (int i =1; i <= 10; ++i)
    {
        ListNode *s = (ListNode*)malloc(sizeof(ListNode));
        if (s == NULL)
            return;
        s->data = i;
        s->next = NULL;

        p->next = s;
        p = s;
    }
}
```

7.创建有头结点的链表(头插)

```
void H_CreatList_front(list* head)
{
    for (int i = 1; i <= 10; ++i)
    {
        ListNode *s = (ListNode*)malloc(sizeof(ListNode));
        if (s == NULL)
            return;
        s->data = i;
        s->next = (*head)->next;  //头插的时候新结点是插在头结点和第一个结点之间的
                                  //让新结点的 next 指向头结点指向的结点，即第一个结点
        (*head)->next = s;       //再让头结点指向新结点，使得新结点变成第一个结点
    }
}
```

8.遍历有头结点的链表 ShowList

```
void H_ShowList(list *head)
{
    ListNode* p = (*head)->next;
    for (int i = 1; i <= 10; ++i)
    {
        printf("%d ", p->data);
```

```
        p = p->next;
    }
    printf("\n");
```

单链表实现：

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

typedef int ElemType;

typedef struct Node
{
    ElemType data;
    struct Node* next;
}Node,*PNode;//结点的结构体

typedef struct List
{
    PNode first;  //指向头结点
    PNode last;   //指向尾结点
    ElemType size;  //链表中的数据个数
}List;//链表的结构体
```

1.初始化

```
void InitList(List *list)
{
    list->first = list->last = (Node*)malloc(sizeof(Node));
    list->first->next = NULL;
    if (list->first == NULL)
        return;
}
```

2.尾插 push_back

```
void push_back(List* list, ElemType x)
{
    //insert(list, end(list), x);  //优化后只需要一句代码就能实现尾插
    //Node* s = buyNode(x);  //优化后也只需要一句代码就能实现申请空间的操作
    Node* s = (Node*)malloc(sizeof(Node));  //malloc 新结点
    if (s == NULL)
        return;
    s->data = x;
    s->next =NULL;

    list->last->next = s;  //让尾结点的 next 指向新结点 s
    list->last = s;     //再让尾结点 last 指向 s
    list->size++;
}
```

3.头插 push_front

```
void push_front(List* list, ElemType x)
{
    //insert(list, begin(list), x);  //优化后只需要一句代码就能实现头插
    //Node* s = buyNode(x);  //优化后也只需要一句代码就能实现申请空间的操作
    Node* s = (Node*)malloc(sizeof(Node));  //malloc 新结点
    if (s == NULL)
        return;
    s->data = x;

    s->next = list->first->next;
    list->first->next = s;
    if (list->size == 0)   //这里注意头插的时候如果插入的是第一个结点，那么 last 也要指向这个结点
    {
        list->last = s;
    }
    list->size++;
}
```

4.遍历 show_list

```
void show_list(List *list)
{
    Node* p = list->first->next;
    while (p != NULL)
    {
        printf("%d ", p->data);
                                    p = p->next;
    }
    printf("\n");
}
```

5.查找 find

```
Node* find(List* list, ElemType x)
{
    Node* p = list->first->next;
    while (p!=NULL && p->data != x)
        p = p->next;
    return p;
}
```

6.尾删 pop_back

```
void pop_back(List* list)
{
    if (list->size == 0)
        return;

    Node* p = list->first;
    while (p->next != list->last) //遍历链表找到 last 前一个结点 p，free（last）再让 last 指向 p
```

```
    {
        p = p->next;
    }
    free(list->last);
    list->last = p;
    list->last ->next = NULL;
    list->size--;
}
```

7.头删 pop_front

```
void pop_front(List* list)
{
    if (list->size == 0)
        return;

    Node* p = list->first->next; //设置 p 为第一个结点
    list->first->next = p->next; //让头结点指向 p->next
    free(p);                     //再将 p 释放
    if (list->size == 1)        //如果链表中只有一个数据，那么删除后让 last 指向 first
        list->last = list->first;
    list->size--;
}
```

8.按值插入 insert_val(按顺序的插入

```
void insert_val(List* list, ElemType x)
{
    //Node* s = buyNode(x);  //优化后也只需要一句代码就能实现申请空间的操作
    Node* s = (Node*)malloc(sizeof(Node));
    if (s == NULL)
        return;
    s->data = x;
    s->next = NULL;

    Node* p = list->first;
    while (p->next != NULL && p->next->data < s->data)//让 p 的下一个结点的数据跟 s 的数据比较，如果 p->next 的数据大就将 s 插入到 p 后
        p = p->next;
    if (p->next == NULL)  //特殊情况，如果 s 的数据比链表中的数据都大，那么就插在表尾
        list->last = s;

    s->next = p->next;
    p->next = s;
    list->size++;
}
```

9.按值删除 delete_val

```
void delete_val(List* list, ElemType x)
{
    if (list->size == 0)
```

```
        return ;

    Node* p = find(list,x);
    if (p == NULL)
    {
        printf("要删除的数据不存在\n");
        return;
    }

    if (p == list->last)  //如果 p 指向了 last 那么就不能直接用替换的方式删除了
    {
        pop_back(list);   //用尾删把 last 删除，这里 pop 函数包含了自减，下面的自减就不能写在外面了
    }
    else
    {
        Node* q = p->next;
        p->data = q->data;
        p->next = q->next;
        free(q);
        list->size--;
    }

}
```

10.求长度 length

```
int length(List* list)
{
    return list->size;
}
```

11.排序 sort

```
void sort(List* list)
{
    //将链表的第一个结点和后面的结点分开，分别用 p 和 q 指向
    //每次从断开后的链表里按顺序取一个结点，比较第一个结点和取出的结点的数据大小
    //再按情况选择是否插在第一个结点之后，实现排序
    if (list->size == 0 || list->size == 1)
        return;

    Node* s = list->first->next;
    Node* q = s->next;

    list->last = s;    //让 last 指向第一个结点
    list->last->next = NULL;  //让第一个结点的下一个结点为空，断开两个结点之间的联系

    while (q != NULL)
    {
        s = q;
        q = q->next;//让 s 指向 q 的位置，q 再指向下一个结点
```

```
        Node* p = list->first;
        while (p->next != NULL && p->next->data < s->data)//让 p 的下一个结点的数据跟 s 的数据比较，如果 p->next 的数据大就将 s 插入到 p 后
            p = p->next;
        if (p->next == NULL)  //特殊情况，如果 s 的数据比链表中的数据都大，那么就插在表尾
            list->last = s;

        s->next = p->next;
        p->next = s;
        list->size++;
    }
}
```

12.逆置 resver

```
void resver(List* list)
{
    //将链表的第一个结点和后面的结点分开，分别用 p 和 q 指向
    //每次从断开后的链表里按顺序取一个结点，插在第一个结点之前，实现逆序
    if (list->size == 0 || list->size == 1)
        return;

    Node* p = list->first->next;
    Node* q = p->next;

    list->last = p;    //让 last 指向第一个结点
    list->last->next = NULL;  //让第一个结点的下一个结点为空，断开两个结点之间的联系

    while (q != NULL)
    {
        p = q;
        q = q->next;

        p->next = list->first->next;
        list->first->next = p;
    }
}
```

13.清空 clear

```
  void clear(List* list)
{
    if (list->size == 0)
        return;

    Node* p = list->first->next;
    while (p != NULL)
    {
        list->first->next = p->next;
```

```
        free(p);
        p = list->first->next;
    }
    list->last = list->first;
    list->size = 0;
}
```

14.销毁 destroy

```
void destroy(List* list)
{
    clear(list);
    free(list->first);
    list->first = list->last = NULL;
    list->size = 0;
}
```

优化代码：

```
Node* buyNode(ElemType x)//使分配空间模块化
{
    Node* s = (Node*)malloc(sizeof(Node));
    if (s == NULL)
        return NULL;
    s->data = x;
    s->next = NULL;
    return s;
}
```

```
Node* begin(List* list)
{
    return list->first->next;
}
Node* end(List* list)
{
    return list->last->next;
}
```

```
void insert(List* list, Node* pos, ElemType x)
{
    Node* p = list->first;
    while (p->next != pos)
        p = p->next;

    Node* s = buyNode(x);
    s->next = p->next;
    p->next = s;
    if (pos == NULL)
        list->last = s;
```

```
    list->size++;
}
```

静态链表:

这种描述方法便于在不设“指针”的高级程序设计语言中使用链表结构。其中数组的一个分量表示一个结点，同时用游标（cur）代替指针指示结点在数组中的相对位置。数组的零分量可看成头结点，其指针域指示链表的第一个结点。这种链表的表示形式叫做静态链表。

静态链表的优缺点:

优点：在插入和删除操作时，只需要修改游标，不需要移动元素，从而改进了顺序存储结构中插入和删除操作需要移动大量元素的缺点。

缺点：没有解决连续存储分配带来的表长难以确定的问题；失去了顺序存储结构随机存取的特性。

```
#include<stdio.h>

#define MAX_SIZE 20
typedef char ElemType;

typedef struct ListNode  //数据分量的结构体
{
    ElemType data;
    int cur;
}ListNode;

typedef ListNode StaticList[MAX_SIZE];
```

1.初始化

```
void InitSList(StaticList& space)
{
    for (int i = 1; i < MAX_SIZE - 1; ++i)
    {
        space[i].cur = i + 1;
    }
    space[MAX_SIZE - 1].cur = 0;
    space[0].cur = -1;
}
```

2.插入

```
int Malloc_SL(StaticList &space)
{
    int i = space[1].cur;
    if (space[1].cur != 0)
```

```
        space[1].cur = space[i].cur;
    return i;
}

void insert(StaticList& space,ElemType x)
{
    int i = Malloc_SL(space);//在数组中申请一个空间
    if (i == 0)
    {
        printf("申请结点空间失败！\n");
        return;
    }

    space[i].data = x;
    if (space[0].cur == -1)//如果表中没有数据,插入时的操作
    {
        space[i].cur = -1;
    }
    else                  //表中有数据时，采用头插
    {
        space[i].cur = space[0].cur;
    }
    space[0].cur = i;
}
```

3.遍历

```
void ShowList(StaticList& space)
{
    int i = space[0].cur;
    while (i != -1)
    {
        printf("%c-->", space[i].data);
        i = space[i].cur;  //相当于 p=p-next
    }
    printf("NULL\n");
}
```

4.删除

```
void Free_SL(StaticList& space,int k)
{
    space[k].cur = space[1].cur;
    space[1].cur = k;
}

void Delete(StaticList& space)
{
    int i = space[0].cur;
    space[0].cur = space[i].cur;
```

```
    Free_SL(space, i);
}
```

单循环链表:

循环链表是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。

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

#define ElemType int

typedef struct Node
{
    ElemType data;
    struct Node* next;
}Node,*PNode;

typedef struct List
{
    PNode first;
    PNode last;
    int size;
}List;

Node* buyNode(ElemType x)
{
    Node* s = (Node*)malloc(sizeof(Node));
    if (s == NULL)
        return NULL;
    s->data = x;
    s->next = NULL;
    return s;
}
```

1.初始化

```
void InitCList(List *list)
{
    Node* s = (Node*)malloc(sizeof(Node));
    if (s == NULL)
        return;
    list->first = list->last = s;
    list->last->next = list->first;
    list->size = 0;
}
```

2.尾插

```
void push_back(List *list,ElemType x)
{
    Node* s = buyNode(x);

    list->last->next = s;
    list->last = s;
    list->last->next = list->first;
    list->size++;
}
```

3.头插 push_fornt

```
void push_front(List* list, ElemType x)
{
    Node* s = buyNode(x);
    s->next = list->first->next;
    list->first->next = s;
    if (list->first == list->last)
    {
        list->last = s;
    }
    list->size++;
}
```

4.遍历 Show_Clist

```
void Show_Clist(List* list)
{
    Node* p = list->first->next;
    while (p != list->first)
    {
        printf("%d-->", p->data);
        p = p->next;
    }
    printf("NULL\n");
}
```

5.查找 find

```
 Node* find(List* list,ElemType key)
{
    if (list->size == 0)
        return NULL;

    Node* p = list->first->next;
    while (p != list->first && p->data != key)
        p = p->next;
    if (p == list->first)
    {
        printf("没有发现指定数据! \n");
```

```
        return NULL;
    }
    return p;
}
```

6.尾删 pop_back

```
void pop_back(List* list)
{
    if (list->size == 0)
        return;

    Node* p = list->first;
    while (p->next != list->last)
        p = p->next;

    free(list->last);
    list->last = p;
    p->next = list->first;
    list->size--;
}
```

7.头删 pop_front

```
void pop_front(List* list)
{
    if (list->size == 0)
        return;

    Node* p = list->first->next;
    list->first->next = p->next;
    free(p);
    if (list->size == 1)
        list->last->next = list->first;
    list->size--;
}
```

8.按值插入 insert_val（升序）

```
void insert_val(List* list, ElemType x)
{
    Node* p = list->first;
    while (p->next != list->last && p->next->data < x)
    {
        p = p->next;
    }

    if (p->next == list->last && p->next->data < x)
    {
        push_back(list,x);
    }
    else
```

```
    {
        Node* s = buyNode(x);
        s->next = p->next;
        p->next = s;
        list->size++;
    }
}
```

9.按值删除 delete_val

```
void delete_val(List* list, ElemType key)
{
    if (list->size == 0)
        return;

    Node* p = find(list,key);
    if (p == NULL)
    {
        printf("没有找到指定数据！\n");
        return;
    }

    if (p == list->last)  //如果 p 值最后一个数据用尾删
        pop_back(list);
    else
    {
        Node* q = p->next;
        p->data = q->data;
        p->next = q->next;
        free(q);
        list->size--;
    }
}
```

10.求长度 length

```
int length(List* list)
{
    return list->size;
}
```

11.排序 sort

```
void sort(List* list)
{
    if (list->size == 0 || list->size == 1)
        return;

    Node* s = list->first->next;
    Node* q = s->next;

    list->last->next = NULL;
```

```
    list->last = s;
    list->last->next = list->first;

    while (q != NULL)
    {
        s = q;
        q = q->next;

        Node* p = list->first;
        while (p->next != list->last && p->next->data < s->data)
            p = p->next;
        if (p->next == list->last && p->next->data < s->data)
        {
            s->next = list->first;
            list->last->next = s;
            list->last = s;
        }
        else
        {
            s->next = p->next;
            p->next = s;
        }
    }
}
```

12.逆置 resver

```
void resver(List* list)
{
    if (list->size == 0 || list->size == 1)
        return;

    Node* p = list->first->next;
    Node* q = p->next;

    list->last->next = NULL;
    list->last = p;
    list->last->next = list->first;

    while (q != NULL)
    {
        p = q;
        q = q->next;

        p->next = list->first->next;
        list->first->next = p;
    }
}
```

13.清空 clear

```
void clear(List* list)
{
    Node* p = list->first->next;
    while (p != list->first)
    {
        list->first->next = p->next;  //头删
        free(p);
        p = list->first->next;
    }
    list->last = list->first;
    list->last->next = list->first;
    list->size = 0;
}
```

14.销毁 destroy

```
void destroy(List* list)
{
    clear(list);
    free(list->first);
    list->first = list->last = NULL;
}
```

双向链表:

双向链表的结点中有两个指针域，其一指向直接后继，其一指向直接前驱。

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

typedef int ElemType;

typedef struct Node
{
    ElemType data;  //数据域
    struct Node* prior;//前驱
    struct Node* next;//后驱
}Node,*PNode;

typedef struct DList
{
    PNode first;
    PNode last;
    int size;
}DList;

//创建结点的模块
Node* buyNode(ElemType x)
{
```

```
    Node* s = (Node*)malloc(sizeof(Node));
    if (s == NULL)
        return NULL;
    s->data = x;
    s->next = s->prior = NULL;
    return s;
}
```

1. 初始化

```
void InitDList(DList *dlist)
{
    dlist->first = dlist->last = (Node*)malloc(sizeof(Node));
    if (dlist->first == NULL)
        return;
    dlist->last->next = NULL;
    dlist->first->prior = NULL;
    dlist->size = 0;
}
```

2.尾插 push_back

```
void push_back(DList *dlist,ElemType x)
{
    Node* s = buyNode(x);

    s->prior = dlist->last;
    dlist->last->next = s;

    dlist->last = s;
    dlist->size++;
}
```

3.头插 push_front

```
void push_front(DList* dlist, ElemType x)
{
    Node* s = buyNode(x);

    //当链表中没有数据时，情况会和有数据时不一样
    //其 s->next->prior 是无法指向 s 的
    if (dlist->first == dlist->last)
    {
        dlist->last = s;
    }
    else
    {
        s->next = dlist->first->next;
        s->next->prior = s;
    }
    s->prior = dlist->first;
    dlist->first->next = s;
```

```
    dlist->size++;
}
```

4.遍历 show_list

```
void Show_DList(DList* dlist)
{
    Node* p = dlist->first->next;
    while (p != NULL)
    {
        printf("%d-->", p->data);
        p = p->next;
    }
    printf("NULL\n");
}
```

5.查找 find

```
Node* find(DList *dlist,ElemType key)
{
    Node* p = dlist->first;
    while (p != NULL && p->data != key)
        p = p->next;

    if (p == NULL)
        return NULL;
    return p;

}
```

6.尾删 pop_back

```
void pop_back(DList *dlist)
{
    if (dlist->size == 0)
        return;

    Node* p = dlist->last;
    dlist->last = p->prior;
    dlist->last->next = NULL;
    free(p);
    dlist->size--;
}
```

7.头删 pop_front

```
void pop_front(DList *dlist)
{
    if (dlist->size == 0)
        return;

    Node* p = dlist->first->next;
    dlist->first->next = p->next;
```

```
    p->next->prior = dlist->first;
    dlist->size--;
}
```

8.按值插入 insert_val

```
void insert_val(DList* dlist,ElemType x)
{
    Node* s = buyNode(x);
    Node* p = dlist->first;
    while (p->next != dlist->last && p->next->data < x)
    {
        p = p->next;
    }
    if (p->next == dlist->last && p->next->data < x)
    {
        s->prior = dlist->last;
        dlist->last->next = s;
        dlist->last = s;
    }
    else
    {
        s->next = p->next;
        p->next->prior = s;
        s->prior = p;
        p->next = s;
    }
    dlist->size++;

}
```

9.按值删除 delete_val

```
void delete_val(DList *dlist,ElemType key)
{
    if (dlist->size == 0)
        return;

    Node* p = find(dlist, key);
    if (p == NULL)
    {
        printf("未找到指定值! \n");
        return;
    }
    if (p == dlist->last)
    {
        dlist->last = p->prior;
    }
    else
    {
        p->next->prior = p->prior;
```

```
    }
    p->prior->next = p->next;
    free(p);
    dlist->size--;
}
```

10.求长度 length

```
int length(DList* dlist)
{
    return dlist->size;
}
```

11.排序 sort

```
void sort(DList* dlist)
{
    if (dlist->size == 0 || dlist->size == 1)
        return;

    Node* s = dlist->first;
    Node* q = s->next;

    dlist->last = s;
    dlist->last->next = NULL;

    while (q != NULL)
    {
        s = q;
        q = q->next;

        Node* p = dlist->first;
        while (p->next != NULL && p->next->data < s->data)
            p = p->next;

        if (p->next == NULL)
            dlist->last = s;

        else
        {
            p->next->prior = s;
        }
        s->next = p->next;
        s->prior = p;
        p->next = s;
    }
}
```

12.逆置 resver

```
void resver(DList* dlist)
{
```

```
    if (dlist->size == 0 || dlist->size == 1)
        return;

    Node* p = dlist->first->next;
    Node* q = p->next;

    dlist->last = p;
    dlist->last->next = NULL;

    while (q != NULL)
    {
        p = q;
        q = q->next;

        p->next = dlist->first->next;
        p->next->prior = p;
        p->prior = dlist->first;
        dlist->first->next = p;

    }
}
```

13.清空 clear

```
void clear(DList* dlist)
{
    if (dlist->size == 0)
        return;

    Node* p = dlist->first->next;
    while (p != NULL)
    {
        if (p == dlist->last)
        {
            dlist->last = dlist->first;
            dlist->last->next = NULL;
        }
        else
        {
            dlist->first->next = p->next;
            p->next->prior = dlist->first;

        }
        free(p);
        p = dlist->first->next;
    }
    dlist->size = 0;
}
```

14.销毁 destroy

```
void destroy(DList* dlist)
{
    clear(dlist);
    free(dlist->first);
    dlist->first = dlist->last = NULL;
}
```

第 3 章 栈和队列

一、栈

栈（stack）是限定仅在表尾进行插入或删除操作的线性表。对栈来说，表尾端有其特殊含义，称为栈顶，表头称为栈底。不含元素的空表称为空栈。栈又称为后进先出的线性表（LIFO）

顺序栈的实现：

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

#define STACK_INIT_SIZE 10
#define STACK_INC_SIZE 5
#define ElemType int

typedef struct SeqStack
{
    ElemType* base; //栈基地址
    int top;        //栈顶指针
    int capacity;   //容量
}SeqStack;

bool INC(SeqStack* s)
{
    //当空间满时，可以通过 INC 函数来增加空间
    ElemType       *newbase       =       (ElemType*)realloc(s->base,sizeof(ElemType)       *
(s->capacity+STACK_INC_SIZE));
    if (newbase == NULL)
        return false;

    s->base = newbase;
    s->capacity = s->capacity + STACK_INC_SIZE;
    return true;
}
```

1.初始化

```
void Initstack(SeqStack *s)
{
    s->base = (ElemType*)malloc(sizeof(ElemType)*STACK_INIT_SIZE);
    if (s->base == NULL)
        return;
    s->top = 0;
    s->capacity = STACK_INIT_SIZE;
}
```

2.判断栈是否为空

```
bool IsFull(SeqStack* s)
{
    return s->top >= s->capacity;
}
bool IsEmpty(SeqStack* s)
{
    return s->top == 0;
}
```

3.入栈 Push

```
void Push(SeqStack* s,ElemType x)
{
    if (IsFull(s) && INC(s))  //栈满，并且增加空间失败，执行模块内的语句
    {
        printf("栈满，不能入栈！\n");
        return;
    }

    s->base[s->top++] = x;
}
```

4.出栈 pop

```
void Pop(SeqStack* s)
{
    if (IsEmpty(s) && !INC(s))
    {
        printf("栈空，不能出栈！\n");
        return;
    }

    s->top--;  //top 减 1 后，原来位置上的数字其实还在，但是下次指向的又有新的数字给这个空间赋值了
}
```

5.得到栈顶位置 GetTop

```
bool GetTop(SeqStack *s,ElemType *v)
{
    if (IsEmpty(s))
    {
        printf("栈为空\n");
        return false;
    }
    *v = s->base[s->top - 1];
    return true;
}
```

6.遍历 Show_Stack

```
void Show(SeqStack* s)
```

```
{
    for (int i =s->top-1; i>=0; i--)
    {
        printf("%d ", s->base[i]);
    }
    printf("\n");
}
```

7.栈元素个数 length

```
int length(SeqStack*s)
{
    return s->top;
}
```

8.清除栈空间 clear

```
void clear(SeqStack* s)
{
    s->top = 0;
}
```

9.销毁栈空间 destroy

```
void destroy(SeqStack* s)
{
    free(s->base);
    s->base = NULL;
    s->capacity = s->top = 0;
}
```

链栈的实现：与单链表一致

栈的应用：

1. 数制转换

```
输入任意一个非负十进制数，打印出与之对应的其他进制的数（以八进制为例）
void Convert_2(int value)
{
    SeqStack st;
    Initstack(&st);

    while (value)
    {
        Push(&st, value % 2);
        value /= 2;
    }
    Show(&st);
}
```

```
int main()
{
    int value = 47183;
    Convert_2(value);
}
```

2. 括号配对的检验

```
bool Cheak(char* str)
{
    SeqStack st;
    Initstack(&st);

    char v;
    while (*str !='\0')
    {
        if (*str == '[' || *str == '(')
        {
            Push(&st, *str);
        }
        else if(*str == ']')
        {
            GetTop(&st,&v);
            if (v != '[')
                return false;
            Pop(&st);
        }else if(*str == ')')
        {
            GetTop(&st, &v);
            if (v == ')')
                return false;
            Pop(&st);
        }
        ++str;
    }
    return IsEmpty(&st);

}

int main()
{
    char *str= "[([][])]";
    bool flag = Cheak(str);
    if (flag)
    {
        printf("OK");
    }
    else
        printf("ERROR");
}
```

3.行编辑程序

```
//用户输入一行文字，若发现输入错误则补进一个退格符“#”表示前面一个字符无效，如不想要这行文字补进一个“@”表示该行字符均无效。
//对栈做一个逆序打印
void print(SeqStack* s)
{
    for (int i = 0; i < s->top; i++)
        printf("%c", s->base[i]);
    printf("\n");
}

void LineEdit()
{
    SeqStack st;
    Initstack(&st);

    printf("pleas input:");
    char ch = getchar();
    while (ch != '$')
    {
        while (ch != '$' && ch != '\n')
        {
            switch (ch)
            {
            case '#':
                Pop(&st);
                break;
            case'@':
                clear(&st);
                break;
            default:
                Push(&st, ch);
                break;
            }
            ch = getchar();  //从终端接收下一个字符
        }
        print(&st);
        ch = getchar();  //没有这句会无限循环
    }
    destroy(&st);
}

int main()
{
    LineEdit();
}
```

二、队列

与栈相反，**队列**（queue）是一种先进先出（FIFO）的线性表，他只允许在表达一端进行插入，在另一端进行删除。在队列中允许插入的一端叫做队尾（rear），允许删除的一端称为队头（front）

链队列的实现：

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

#define ElemType int

typedef struct QNode //结点结构体
{
    ElemType data;
    struct QNode* next;
}QNode,*PNode;

typedef struct LinkQueue  //队列结构体
{
    PNode front;  //队头指针
    PNode rear;   //队尾指针
}LinkQueue;

QNode* buyNode(ElemType x)
{
    QNode *s= (QNode*)malloc(sizeof(QNode));
    if (s == NULL)
        return NULL;
    s->data = x;
    s->next = NULL;
    return s;
}
```

1.初始化

```
void Initqueue(LinkQueue* q)
{
    q->front = q->rear = (QNode*)malloc(sizeof(QNode));
    if (q->front == NULL)
        return;
    q->front->next = NULL;
}
```

2.入队 EnQueue

```
void EnQueue(LinkQueue *q,ElemType x)
{
    QNode* s = buyNode(x);

    q->rear->next = s;
    q->rear = s;
}
```

3.出队 DeQueue

```
void DeQueue(LinkQueue* q)
{
    if (q->front == q->rear)
    {
        printf("队空不能出队！");
        return;
    }

    QNode* p = q->front->next;  //p 第一个结点
    q->front->next = p->next;
    if (q->rear == p) q->rear = q->front;  //如果队列中只有一个元素的情况
    free(p);
}
```

4.遍历 show

```
void Show(LinkQueue* q)
{
    QNode* p = q->front->next;
    if (p == NULL)
        printf("队空");
    while (p != NULL)
    {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}
```

5.获取队头元素

```
void GetHead(LinkQueue* q,ElemType *v)
{
    if (q->front == q->rear)
    {
        printf("队空！");
        return;
    }

    QNode* p = q->front->next;
```

```
    *v =  p->data;
}
```

6.求长度 length

```
int length(LinkQueue* q)
{
    int len=0;

    QNode* p = q->front->next;
    while (p != NULL)
    {
        len++;
        p = p->next;
    }
    return len;
}
```

7.清空 clear

```
void clear(LinkQueue* q)
{
    if (q->front == q->rear)
    {
        printf("队空! ");
        return;
    }

    QNode* p = q->front->next;
    while (p != NULL)
    {
        q->front->next = p->next;
        free(p);
        p = q->front->next;
    }
    q->rear = q->front;
}
```

8.销毁 deatroy

```
void destroy(LinkQueue* q)
{
    clear(q);
    free(q->front);
    q->front = q->rear = NULL;
}
```

队列的顺序表示和实现——循环队列:

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

```
#define ElemType int
#define MAX_SIZE 20

typedef struct SeqQueue
{
    ElemType* base;
    int front;
    int rear;
}SeqQueue;
```

1.初始化 Initqueue

```
void Initqueue(SeqQueue *q)
{
    q->base = (ElemType*)malloc(sizeof(ElemType) * MAX_SIZE);
    if (q->base == NULL)
        return;
    q->front = q->rear = 0;
}
```

2.入队 EnQueue

```
void EnQueue(SeqQueue *q,ElemType x)
{
    if (q->rear >= MAX_SIZE)
        return;
    q->base[q->rear++] = x;
}

//循环队列的入队 EnQueue
void EnQueue(SeqQueue *q,ElemType x)
{
    //rear 指向的数据的下标+1 后%MAX_SIZE
    //当 rear+1 小于 MAX_SIZE 时取模的结果还是 rear+1
    //当 rear+1 等于 MAX_SIZE 时取模是 0
    //正是利用取模的特性，编写出了循环队列这样的代码
    if ((q->rear+1)% MAX_SIZE == q->front)
        return;
    q->base[q->rear] = x;
    q->rear = (q->rear + 1) % MAX_SIZE;
}
```

3.出队 DeQueue

```
void DeQueue(SeqQueue* q)
{
    if (q->front == q->rear)
        return;

    q->front++; //直接出队
}
```

```
//循环队列的出队 DeQueue
void DeQueue(SeqQueue* q)
{
    if (q->front == q->rear)
        return;

    q->front = (q->front+1)%MAX_SIZE;
}
```

4.获取队头元素 GetHead

```
void GetHead(SeqQueue* q, ElemType* v)
{
    if (q->front == q->rear)
        return;

    *v = q->base[q->front];
}
```

5.遍历 Show

```
void Show(SeqQueue* q)
{
    for (int i = q->front; i < q->rear; i++)
    {
        printf("%d ", q->base[i]);
    }
    printf("\n");
}

//循环队列的遍历 Show
void Show(SeqQueue* q)
{
    for (int i = q->front; i!=q->rear;)
    {
        printf("%d ", q->base[i]);
        i = (i + 1) % MAX_SIZE;
    }
    printf("\n");
}
```

6.求长度 length

```
int length(SeqQueue* q)
{
    return q->rear - q->front;
}
```

7.清空 clear

```
void clear(SeqQueue* q)
{
```

```
    q->front = q->rear = 0;
}
```

8.销毁 destroy

```
void destroy(SeqQueue* q)
{
    clear(q);
    q->base =NULL;
}
```

第4章 串

一、串类型的定义

串（string)：是由零个或多个字符组成的有限序列，一般记为 s = ‘a1a2...an’(n>=0)

其中 n 表示串的长度，零个字符的串的为空串。串中任意个连续的字符组成的子序列为**子串**。包含子串的串相应的称为**主串**。通常称字符在序列中的序号为该字符在串中的**位置**。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

如果**两个串相等**，当且仅当这两个字符串的值相等，也就是说，不仅要这两个串的长度相等，两个串中对应的数据元素也要一一相等。

二、串的表示和实现

定长顺序存储表示：

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

#define MAXSIZE 20
#define u_char unsigned char

typedef u_char SString[MAXSIZE + 1]; //定义无符号字符数组，长度为 MAXSIZE+1

void print(SString s)
{
    printf("%s\n", s);
}
```

1.初始化

```
void InitString(SString s)
{
    s[0] = '\0';
}
```

2.按 str 的内容生成一个 s 串 StrAssign

```
void StrAssign(SString s,char*str)
{
    int len = strlen(str);
    for (int i = 0; i < len; i++)
    {
```

```
        s[i] = str[i];
    }
    s[len] = '\0';
}
```

3.求长度 StrLength

```
int StrLength(SString s)
{
    int len = 0;
    while (*s != '\0')
    {
        len++;
        s++;
    }
    return len;
}
```

4.复制 t 串给 s 串 StrCopy

```
void StrCopy(SString s, SString t)
{
    int len = StrLength(t);
    for (int i = 0; i < len; i++)
    {
        s[i] = t[i];
    }
    s[len] = '\0';  //不能忘记
}
```

5.比较 StrCompare

```
int StrCompare(SString s, SString t)
{
    int result = 0;
    while (*s != '\0' || *t != '\0')
    {
        result = *s - *t;  //如果 result=0 说明 s 和 t 的串长度相等且数据一致，直到串结束'\0'循环结束
        if (result != 0)
            break;
        s++;
        t++;
    }

    if (result > 0)   //如果 result! =0 说明 s 和 t 的不同，长度不同或者数据不同
        result = 1;
    else if (result < 0)
        result = -1;

    return result;
}
```

6.连接两个串 StrConcat

```
void StrConcat(SString T,SString s1, SString s2)
{
    int len1 = StrLength(s1);
    int len2 = StrLength(s2);

    if (len1 + len2 <= MAXSIZE)  //如果 s1 和 s2 的长度之和小于 MAXSIZE，说明 T 串可以完整的连接两个
串
    {
        for (int i = 0; i < len1; i++)
        {
            T[i] = s1[i];
        }
        for (int j = 0; j < len2; j++)
        {
            T[len1 + j] = s2[j];
        }
        T[len1 + len2] = '\0';
    }
    else if (len1 < MAXSIZE)  //如果 s1 和 s2 的长度之和大于 MAXSIZE，则 T 只能连接 s1 和 s2 的部分
    {
        for (int i = 0; i < len1; i++)
        {
            T[i] = s1[i];
        }
        for (int j = 0; j < MAXSIZE-len1; j++)
        {
            T[len1 + j] = s2[j];
        }
        T[MAXSIZE] = '\0';
    }
    else   //如果 T 的长度比 len1 还小，那只能存放 s1
    {
        for (int i = 0; i < len1; i++)
            T[i] = s1[i];
    }
    T[MAXSIZE] = '\0';

}
```

7.求从 pos 位置开始长度为 len 的子串 SubString

```
void SubString(SString s,SString sub,int pos,int len)
{
    int s_len = StrLength(s);
    if (pos <0 || pos>=s_len || len <=0 || len > s_len)
        return;

    int j = pos;
```

```
    for (int i = 0; i < len; i++)
    {
        sub[i] = s[j + i];
    }
    sub[len] = '\0';
}
```

8.在主串 T 的第 pos 个位置前插入子串 s StrInsert

```
void StrInsert(SString T,int pos,SString s)
{
    int T_len = StrLength(T);
    int s_len = StrLength(s);
    //if (T_len + s_len >= MAXSIZE)  //如果要 T 和 s 的长度之和大于 MAXSIZE 并且要求截取插入子串 s 的一部分插入，这个代码块就不能执行。
    //  return;
    if (T_len + s_len <= MAXSIZE)
    {
        for (int i = T_len - 1; i >= pos; i--)
        {
            T[i + s_len] = T[i];
        }
        for (int j = 0; j < s_len; j++)
        {
            T[pos + j] = s[j];
        }
        T[T_len + s_len] = '\0';

    }
    else if(T_len < MAXSIZE)
    {
        s_len = MAXSIZE - T_len;
        for (int i = T_len - 1; i >= pos; i--)
        {
            T[i + s_len] = T[i];
        }
        for (int j = 0; j < s_len; j++)
        {
            T[pos + j] = s[j];
        }
        T[T_len + s_len] = '\0';
    }
}
```

9.删除主串第 pos 个位置起长度为 len 的子串 StrDelete

```
void StrDelete(SString T, int pos, int len)
{
    int T_len = StrLength(T);
    if (len < 0 || len > T_len)
        return;
```

```
    for (int i = pos; i < T_len; i++)
    {
        T[i] = T[i + len];
    }
    T[T_len - len] = '\0';
}
```

10.清空 StrClear

```
void StrClear(SString T)
{
    T[0] = '\0';
}
```

堆分配存储表示:

这种存储方式的特点是，仍以一组地址连续的的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配而得。

```
#include<stdio.h>
#include<malloc.h>
#include<string.h>

typedef struct HString
{
    char* ch;
    int length;  //字符串长度
}HString;

void print(HString* s)
{
    for (int i = 0; i < s->length; i++)
    {
        printf("%c", s->ch[i]);
    }
    printf("\n");
}
```

1.初始化

```
void InitString(HString* s)
{
    s->ch = NULL;
    s->length = 0;
}
```

2.按 str 的内容生成一个 s 串 StrAssign

```
void StrAssign(HString *s,char *str)
{
```

```
    int len = strlen(str);
    if (s->ch != NULL)
        free(s->ch);

    s->ch = (char*)malloc(sizeof(char) * len);
    if (s->ch == NULL)
        return;

    for (int i = 0; i < len; i++)
    {
        s->ch[i] = str[i];
    }
    s->length = len;
}
```

3.求长度 StrLength

```
int StrLength(HString* s)
{
    return s->length;
}
```

4.复制 s2 串给 s1 串 StrCopy

```
void StrCopy(HString* s1, HString* s2)
{
    int len = StrLength(s2);
    if (s1->ch != NULL)  //如果 s1 中有数据，那么要先释放掉，在分配空间
        free(s1->ch);

    s1->ch = (char*)malloc(sizeof(char) * len);
    if (s1->ch == NULL)
        return;

    for (int i=0; i<len; i++)
    {
        s1->ch[i] = s2->ch[i];
    }
    s1->length = len;
}
```

5.判空 StrEmpty

```
int StrEmpty(HString* s)
{
    if (s->length == 0)
        return 1;
    else
        return 0;
}
```

6.比较 StrCompare

```
int StrCompare(HString* s1, HString* s2)
{
    if (s1->length == 0 && s2->length == 0)
        return 0;

    int result = 0;
    int i = 0, j = 0;
    while (i<s1->length && j<s2->length)
    {
        if (s1->ch[i] > s2->ch[j])
            return 1;
        else if (s1->ch[i] < s2->ch[j])
            return -1;
        else
        {
            i++;
            j++;
        }
    }
    if (i < s1->length)
        result = 1;
    if (j < s2->length)
        result = -1;

    return result;
}
```

7.连接两个串 StrConcat

```
void StrConcat(HString* T,HString* s1, HString* s2)
{
    if (T->ch != NULL)
        free(T->ch);

    int s1_len = StrLength(s1);
    int s2_len = StrLength(s2);

    T->ch = (char*)malloc(sizeof(char) * (s1_len + s2_len));
    if (T->ch == NULL)
        return;

    int i = 0;
    for (; i < s1_len; i++)
    {
        T->ch[i] = s1->ch[i];
    }
    for (int j = 0; j < s2_len; j++)
    {
        T->ch[i + j] = s2->ch[j];
    }
```

```
    T->length = s1_len + s2_len;
}
```

8.求从 pos 位置开始长度为 len 的子串 SubString

```
void SubString(HString* T, HString* sub,int pos, int len)
{
    if (pos<0 || pos>T->length || len <0 || len>T->length - pos)
        return;

    if (sub->ch != NULL)
        free(sub->ch);

    sub->ch = (char*)malloc(sizeof(char) * len);
    if (sub->ch == NULL)
        return;

    for (int i = 0; i <= len; i++)
    {
        sub->ch[i] = T->ch[pos+i];
    }
    sub->length = len;
}
```

9.在第 pos 个位置之前插入长度为 len 的子串 s StrInsert

```
void StrInsert(HString* T,int pos, int len,HString* s)
{
    if (T->length == 0)
        return;
    if (pos<0 || pos>T->length || len <0 || len>T->length - pos)
        return;

    char* ch = (char*)realloc(T->ch, sizeof(char) * (T->length + s->length));
    if (ch == NULL)
        return;
    T->ch = ch;
    T->length += s->length;

    for (int i = T->length-1; i >= pos; i--)
    {
        T->ch[i + len] = T->ch[i];
    }

    for (int i = 0; i < len; i++)
    {
        T->ch[pos + i] = s->ch[i];
    }
}
```

10.删除主串第 pos 个位置起长度为 len 的子串 StrDelete

```
void StrDelete(HString* T, int pos, int len)
{
    if (T->length == 0)
        return;
    if (pos<0 || pos>T->length || len <0 || len>T->length - pos)
        return;

    for (int i = pos; i < len; i++)
    {
        T->ch[i] = T->ch[i + len];
    }
    T->length -= len;
}
```

11.清空 StrClear

```
void StrClear(HString* T)
{
    T->ch = NULL;
    T->length = 0;
}
```

三、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。

朴素的模式匹配算法：

```
int StrIndex(SString S, SString T,int pos)
{
    int i = pos;
    int j = 0;
    while (S[i]!='\0' && T[j]!='\0')
    {
        if (S[i] == T[j])
        {
            i++;
            j++;
        }
        else
        {
            i = i - j + 1;
            j = 0;
        }
    }
```

```
    if (T[j] == '\0')
        return i - j;
    return -1;
}
```

KMP 匹配算法：

用新子串 v 代替在主串 s 中出现的子串 t 的位置 Replace

```
void StrReplace(SString S, SString T, SString V)
{
    int s_len = StrLength(S);
    int t_len = StrLength(T);
    int v_len = StrLength(V);
    int pos = 0;
    while (pos < s_len)
    {
        int index = StrIndex(S, T, pos);
        if (index == -1)
            return;

        StrDelete(S, index, t_len);
        StrInsert(S, index, V);

        pos = index + v_len;
    }
```

第5章 数组与广义表

1.矩阵的压缩存储

压缩存储：为多个值相同的元中分配一个存储空间，对零元不分配空间。

假若值相同的元素或零元素在矩阵中的分布有一定的规律，则我们称此类矩阵为特殊矩阵；无规律的称为稀疏矩阵。

2.广义表的定义

广义表是线性表的推广。一般记为：LS = （a1,a2,a3,…an）,其中 LS 是广义表的名字，n 是他的长度。在线性表中 ai 值限定为单个元素，而在广义表中 ai 可以是单个元素也可以是广义表，分别成广义表的原子和子表。一般用大写表示广义表的名称，小写表示原子。当广义表非空时，称第一个元素 a1 是表头，其余元素组成表尾。

3.广义表的存储结构

第 6 章 树和二叉树

1.树的定义和基本术语

树是 n 个结点的有限集。在任何一个非空树中：

（1）有且仅有一个特定的称为根（Root）的结点；

（2）当 n>1 时其余结点可分为 m 个不相交的有限集，其中每一个集合本身也是一棵树，并且称为根的子树。

树的**结点**包含一个数据元素及若干指向其子树的分支。

结点的**度**（Degree）：结点拥有的子树数。

叶子结点（Leaf）或终端结点：度为 0 的结点。

非终端结点或者分支结点：度不为 0 的结点。

树的度是树内各结点的度的最大值。

结点的子树的根称为该结点的**孩子**，相对应的该结点称为孩子的**双亲**。同一个双亲的孩子之间互称为**兄弟**。

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。树中结点的最大层次称为**树的深度**。

如果将树中结点的各子树看成从左至右是有次序的，则称该树为**有序树**，否则称为无序树。

森林是 m 个互不相交的树的集合。

2.二叉树

二叉树是另一种树型结构，它的特点是每个结点至多只有两个子树，并且二叉树的子树有左右之分，其次序不能左右颠倒。

性质：（1）在二叉树的第 i 层上至多有 2^{i-1} 个结点（i>=1）

（2）深度为 k 的二叉树至多有 2^k -1 个结点（k>=1）

（3）对任何一颗二叉树 T，如果其终端结点树为 n_0，度为 2 的结点数为 n_2，则 $n_0 = n_2 + 1$

（4）具有 n 个结点的完全二叉树的深度为 k=$\lfloor \log_2 n \rfloor + 1$

（5）如果对一颗有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 i（1<=i<=n）有：

① 如果 i=1，则结点 i 是二叉树的根，无双亲；如果 i>1,则其双亲为 $\lfloor i/2 \rfloor$

② 如果 2i>n，则结点 i 无左孩子（结点 i 为叶子结点）；否则其左孩子为 2i

③ 如果 2i+1>n，则结点 i 无右孩子；否则其右孩子为 2i+1.

满二叉树：一个深度为 k 且有 2^k -1 个结点的二叉树。

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 k 的满二叉树中编码从 1 至 n 的结点一一对应。

二叉树的链式存储

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

#define ElemType char

typedef struct BinTreeNode
{
    ElemType data;   //数据域
    struct BinTreeNode *Lchild;  //左孩子指针
    struct BinTreeNode* Rchild;  //右孩子指指针
}BinTreeNode;

typedef struct BinTree
{
    BinTreeNode* root;  //根结点
    ElemType refvalue;  //停止标志
}BinTree;
```

1.初始化

```
void InitBinTree(BinTree *bt,ElemType ref)
{
    bt->root = NULL;
    bt->refvalue = ref;
}
```

2.前序创建

```
void CreatBinTree_1(BinTree* bt, BinTreeNode** t)
{
    ElemType Item;
    scanf_s("%c",& Item);
    if (Item == bt->refvalue)
        (*t) = NULL;
    else
    {
        (*t) = (BinTreeNode*)malloc(sizeof(BinTreeNode));
        if ((*t) == NULL)
            return;
        CreatBinTree_1(bt, &((*t)->Lchild));
        CreatBinTree_1(bt, &((*t)->Rchild));
    }

}

void CreatBinTree_1(BinTree* bt)
```

```
{
    CreatBinTree_1(bt, &(bt->root));
}
```

3.前序遍历

```
void PreOrder_1(BinTreeNode* t)
{
    if (t != NULL)
    {
        printf("%c",t->data);
        PreOrder_1(t->Lchild);
        PreOrder_1(t->Rchild);
    }
}
void PreOrder(BinTree* bt)
{
    PreOrder_1(bt->root);
}
```

4.中序遍历

```
void InOrder_1(BinTreeNode *t)
{
    if (t != NULL)  //t 是二叉树的根结点
    {
        InOrder_1(t->Lchild);
        printf("%c", t->data);
        InOrder_1(t->Rchild);
    }
}
void InOrder(BinTree* bt)
{
    InOrder_1(bt->root);
}
```

5.后序遍历

```
void PostOrder(BinTreeNode* t)
{
    if (t != NULL)  //t 是二叉树的根结点
    {
        PostOrder(t->Lchild);
        PostOrder(t->Rchild);
        printf("%c", t->data);
    }
}
void PostOeder(BinTree* bt)
{
    PostOrder(bt->root);
}
```

6.层序遍历

```
void LevelOrder_1(BinTreeNode* t)
{
    if (t != NULL)
    {
        BinTreeNode* v;
        LinkQueue lq;

        Initqueue(&lq);
        EnQueue(&lq, t);
        while (!QueueEmpty(&lq))
        {
            GetHead(&lq, &v);
            DeQueue(&lq);
            printf("%c", v->data);
            if (v->Lchild != NULL)
                EnQueue(&lq, v->Lchild);
            if (v->Rchild != NULL)
                EnQueue(&lq, v->Rchild);
        }
    }
}
void LevelOrder(BinTree* bt)
{
    LevelOrder_1(bt->root);
}
```

二叉树应用

1.求树的结点个数

```
int Size(BinTreeNode* t)
{
    if (t == NULL) return 0;
    else
        return Size(t->Lchild) + Size(t->Rchild) + 1;
}
int Size(BinTree* bt)
{
    return Size(bt->root);
}
```

2.求树的高度

```
int Hight(BinTreeNode *t)
{
    if (t == NULL) return 0;
    else
```

```
    {
        int l_hight = Hight(t->Lchild);
        int r_hight = Hight(t->Rchild);
        if (l_hight > r_hight)
            return l_hight + 1;
        else return r_hight + 1;
    }
}
int Hight(BinTree *bt)
{
    return Hight(bt->root);
}
```

3.查找结点，并返回该结点地址

```
BinTreeNode* Search(BinTreeNode* t, ElemType key)
{
    //先判断根结点是否是目标结点，若是则直接返回
    //若不是，再访问树的左子树，
    //当结点为空时，说明左子树无目标阶段，继而访问右子树
    if (t == NULL) return NULL;
    if (t->data == key)
        return t;

    BinTreeNode* p = Search(t->Lchild, key);
    if (p != NULL)
        return p;
    return Search(t->Rchild, key);
}
BinTreeNode* Search(BinTree* bt, ElemType key)
{
    return Search(bt->root, key);
}
```

4.求目标结点的双亲结点

```
BinTreeNode* Parent(BinTreeNode*t, BinTreeNode* p)
{
    //判断根结点和目标结点是否为空，为空直接退出
    //若根结点的左右子树结点是目标结点，直接返回根结点
    //如不是，递归调用函数
    if (t == NULL || p == NULL)
        return NULL;
    if (t->Lchild == p || t->Rchild == p)
        return t;

    BinTreeNode* q = Parent(t->Lchild, p);
    if (q != NULL)
        return q;
    return Parent(t->Rchild, p);
}
```

```
BinTreeNode* Parent(BinTree* bt, BinTreeNode* p)
{
    return Parent(bt->root, p);
}
```

5.求目标结点的左右子树

```
BinTreeNode* LeftChild(BinTreeNode*p)
{
    if (p != NULL)
        return p->Lchild;
    return p->Rchild;
}

//求目标结点的右子树
BinTreeNode* RigthChild(BinTreeNode* p)
{
    if (p != NULL)
        return p->Rchild;
    return p->Lchild;
}
```

6.判断树是否是空树

```
bool BinTreeEmpty(BinTree* bt)
{
    if (bt->root == NULL)
        return true;
    return false;
}
```

7.拷贝一棵树

```
void Copy(BinTreeNode*t1,BinTreeNode*t2)
{
    //将 t2 赋值给 t1
    //判断 t2 是否为空
    //给 t1 创建一个根结点，把 t2 根结点的值赋给 t1
    //在依次将 t2 的左右子树赋值给 t1
    if (t2 == NULL)
        t1 = NULL;
    else
    {
        t1 = (BinTreeNode*)malloc(sizeof(BinTreeNode));
        if (t1 == NULL)
            return;
        t1->data = t2->data;

        Copy(t1->Lchild, t2->Lchild);
        Copy(t1->Rchild, t2->Rchild);
    }
}
```

```
void Copy(BinTree *bt1,BinTree *bt2)
{
    Copy(bt1->root,bt2->root);
}
```

8.清除二叉树

```
void BinTreeClear(BinTreeNode*t)
{
    if (t != NULL)
    {
        BinTreeClear(t->Lchild);
        BinTreeClear(t->Rchild);
        free(t);
        t = NULL;
    }
}
void BinTreeClear(BinTree*bt)
{
    BinTreeClear(bt->root);
}
```

非递归遍历

9.求前序遍历：先判断根结点是否为空，不空入栈，出栈判断右子树是否为空，不为空就入栈出栈，结束后再判断左子树是否为空，不空再入栈出栈

```
void ProOrder(BinTreeNode*t)
{
    if (t != NULL)
    {

        SeqStack st;
        Initstack(&st);

        BinTreeNode* p;
        Push(&st, t);
        while (!IsEmpty(&st))
        {
            GetTop(&st, &p);
            Pop(&st);
            printf("%c", p->data);
            if (p->Rchild != NULL)
                Push(&st, t->Rchild);
            if (p->Lchild != NULL)
                Push(&st, t->Lchild);
        }
    }
}
void ProOrder(BinTree *bt)
{
```

```
    ProOrder(bt->root);
}
```

10.中序遍历：先判断根结点是否为空，不空入栈，递归判断左子树是否为空，不为空入栈，直到左子树为空，栈顶元素出栈

```
void InOreder(BinTreeNode*t)
{
    if (t != NULL)
    {
        SeqStack st;       //创建一个栈
        Initstack(&st);

        BinTreeNode* p;
        Push(&st, t);
        while (!IsEmpty(&st))
        {
            while (t != NULL && t->Lchild!= NULL)
            {
                Push(&st, t->Lchild);
                t = t->Lchild;
            }
            GetTop(&st, &p);
            Pop(&st);
            printf("%c", p->data);

            if (p->Rchild != NULL)
            {
                t = p->Rchild;
                if (t != NULL)
                    Push(&st, t);
            }
        }

    }
}
void InOreder(BinTree*bt)
{
    InOreder(bt->root);
}
```

11. 后序遍历：比其他遍历复杂，需要重新构造一个结点，由指针域和 tag 组成，若 tag=L 则表明是结点的左子树，tag=R 表明是结点的右子树。

```
栈里定义的结点
typedef enum {L,R}Tag;

typedef struct StkNode
```

```
{
    BinTreeNode* ptr;  //存储当前指针
    Tag  tag;          //标记是左子树还是右子树
}StkNode;
```

```
void PostOrder(BinTree *bt)
{
    PostOrder(&bt->root);
}
void PostOrder(BinTreeNode*t)
{
    if (t != NULL)
    {
        SeqStack st;
        InitStack(&st);
        StkNode sn;  //结点
        BinTreeNode* p;

        do
        {
            while (t != NULL)
            {
                sn.ptr = t;
                sn.tag = L;
                Push(&st, sn);
                t = t->Lchild;
            }

            bool flag = true;  //判断是否继续访问
            while (flag && !IsEmpty(&st))  //判断 flag 是非为假，假就退出，并且判断栈是否为空
            {
                GetTop(&st,&sn);  //得到栈顶元素并出栈
                Pop(&st);
                p = sn.ptr;
                switch (sn.tag)
                {
                case L:
                    sn.tag = R;
                    Push(&st, sn);
                    flag = false;
                    t = p->Rchild;
                    break;

                case R:
                    printf("%c", p->data);
                    break;
                }
```

```
            }
        }while(!IsEmpty(&st))

    }
}
```

二叉树的恢复实现

根据一个二叉树的前序遍历序列和中序遍历序列（或者中序遍历序列和后序遍历序列）可以推出一个二叉树。(VLR--LVR--LRV)

通过前序遍历序列和中序遍历序列恢复一个二叉树

```
void CreatVLR(BinTreeNode*t, char* VLR, char* LVR, int n)
{
    if (n == 0)
        t = NULL;
    else
    {
        int k = 0;
        while (VLR[0] != LVR[k])
            k++;

        //在中序遍历序列中找到根结点后，创建新结点
        t = (BinTreeNode*)malloc(sizeof(BinTreeNode));
        if (t == NULL)
            return;
        t->data = LVR[k];

        CreatVLR(t->Lchild, VLR + 1, LVR, k);
        CreatVLR(t->Rchild, VLR + k + 1, LVR + k + 1, n - k - 1);
    }
}
void CreatVLR_1(BinTree* bt, char* VLR, char* LVR, int n)
{
    CreatVLR(bt->root, VLR, LVR, n);
}
```

通过后序遍历序列和中序遍历序列恢复一个二叉树

```
void CreatLRV_1(BinTree*bt,char*LRV,char*LVR,int n)
{
    CreatLRV(bt->root, LRV, LVR, n);
}
void CreatLRV(BinTreeNode* t, char* LVR, char* LRV, int n)
{
    if (n == 0)
        t = NULL;
    else
```

```
    {
        int k = 0;
        while (LRV[n - 1] != LVR[k])
            k++;

        //在中序遍历序列中找到根结点后，创建新结点
        t = (BinTreeNode*)malloc(sizeof(BinTreeNode));
        if (t == NULL)
            return;
        t->data = LVR[k];

        //先创建的右子树
        CreatLRV(t->Rchild, LVR+k+1, LRV+k, n-k-1);
        CreatLRV(t->Lchild, LVR,LRV, k);
    }

}
```

线索二叉树

在原来二叉树的结点结构上加两个标志域 Ltag 和 Rtag，当 Ltag 为 0 时就指针域指向其左孩子，为 1 时就指向该结点的前驱；当 Rtag 为 0 时指针域指向其右孩子，为 1 指向该结点后继。其中指向前驱的和后继的指针叫做**线索**。加上线索的二叉树叫做**线索二叉树**。

对二叉树以某种次序的遍历使其变成线索二叉树的过程叫做**线索化**。

其上所说的前驱和后继皆是指前序、中序和后序中的顺序。

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

#define ElemType char

typedef enum { LINK, THREAD }Tag;  //LINK == 0，THREAD==1

typedef struct BinThrNode  //结点结构
{
    ElemType data;   //数据域
    struct BinThrNode* Lchild;  //左右孩子结点
    struct BinThrNode* Rchild;
    Tag Ltag, Rtag;   //左右标志域
}BinThrNode;

typedef struct BinThrTree
{
    BinThrNode* root;
```

```
    ElemType refvalue;  //stop tag
}BinThrTree;

BinThrNode* buyNode(ElemType x)
{
    BinThrNode* s = (BinThrNode*)malloc(sizeof(BinThrNode));
    if (s == NULL)
        return NULL;
    s->data = x;
    s->Lchild = s->Rchild = NULL;
    s->Ltag = s->Rtag = LINK;
    return s;
}
```

1. 初始化

```
void InitThrTree(BinThrTree *bt,ElemType ref)
{
    bt->root = NULL;
    bt->refvalue = '#';
}
```

2.创建二叉树

```
void CreatBinTree(BinThrTree*bt,BinThrNode*t,char*str)
{
    if (*str == bt->refvalue)
        return;
    else
    {
        BinThrNode* s = buyNode(str);
        CreatThrTree(bt, t->Lchild, ++str);
        CreatThrTree(bt, t->Rchild, ++str);
    }
}
void CreatBinTree_1(BinThrTree* bt, char* str)
{
    CreatThrTree(bt, bt->root, str);
}
```

3.中序线索化

```
void CreatThrTree_1(BinThrTree* bt)
{

    CreatThrTree(bt->root);

}
void CreatThrTree(BinThrNode* t)
{
    if (t == NULL)
        return;
```

```
    else
    {
        BinThrNode* pre = NULL;

        CreatThrTree(t->Lchild, pre);
        if (t->Lchild == NULL)
        {
            t->Ltag = THREAD;
            t->Lchild = pre;

        }
        if (pre != NULL && pre->Rchild == NULL)
        {
            pre->Rtag = THREAD;
            pre->Rchild = t;
        }
        pre = t;
        CreatThrTree(t->Rchild, pre);
        pre->Rchild = NULL;
        pre->Rtag = THREAD;
    }

}
```

4.线索二叉树中的第一个结点

```
BinThrNode* Frist(BinThrNode* t)
{
    if (t == NULL)
        return NULL;
    //从根结点开始，如果结点的 Ltag 是 LINK，那么再指向该结点的左孩子
    //直到结点的 Ltag 是 THREAD 时，该结点为二叉树的第一个结点
    BinThrNode* p = t;
    while (p->Ltag == LINK)
        p = p->Lchild;
    return p;
}
BinThrNode* Frist_1(BinThrTree* bt)
{
    return Frist(bt->root);
}
```

5.线索二叉树中的最后一个结点

```
BinThrNode* Last(BinThrNode*t)
{
    if (t == NULL)
        return NULL;
    //从根结点开始向右子树走，如果结点的 Rtag 是 LINK，再继续指向该结点的右孩子
    //直到结点的 Rtag 是 THREAD 时，该结点是二叉树的之后一个结点
    BinThrNode* p = t;
```

```
    while (p->Rtag == LINK)
        p = p->Rchild;
    return p;
}
BinThrNode* Last_1(BinThrTree* bt)
{
    return Last(bt->root);
}
```

6.求当前结点的后继结点，cur 是当前结点

```
BinThrNode* Next(BinThrNode* t, BinThrNode* cur)
{
    if (t == NULL || cur == NULL)
        return NULL;
    if (cur->Rtag == THREAD)
        return cur->Rchild;
    return Frist(cur->Rchild);
}
BinThrNode* Next_1(BinThrTree*bt,BinThrNode*cur)
{
    Next(bt->root, cur);
}
```

7.求当前结点的前驱节点

```
BinThrNode* Prio(BinThrNode* t, BinThrNode* cur)
{
    if (t == NULL || cur == NULL)
        return NULL;
    if (cur->Ltag == THREAD)
        return cur->Lchild;
    return Last(cur->Lchild);
}
BinThrNode* Prio_1(BinThrTree* bt, BinThrNode* cur)
{
    Prio(bt->root, cur);
}
```

8.中序遍历

```
void InOrder_1(BinThrNode* t)
{
    BinThrNode* p;
    for(p=Frist(t);p!=NULL;p=Next(t,p))
    {
        printf("%c", p->data);
    }
    printf("\n");
}
void InOrder(BinThrTree* bt)
{
```

```
    InOrder_1(bt->root);
}
```

9.查找

```
BinThrNode* Sreach(BinThrNode*t,ElemType key)
{
    if (t == NULL)
        return NULL;
    if (t->data == key)   //查找到根结点，直接返回 t
        return t;
    BinThrNode* p;
    for (p = Frist(t); p != NULL; p = Next(t, p))
    {
        if (p->data == key)
            return p;
    }
    return NULL;
}
BinThrNode* Sreach_1(BinThrTree*bt,ElemType key)
{
    Sreach(bt->root,key);
}
```

10.寻找父结点

```
BinThrNode* Parent(BinThrNode*t,BinThrNode*cur)
{
    if (t == NULL || cur == NULL)
        return NULL;
    if (t == cur)
        return NULL;

    BinThrNode* p;
    if (cur->Ltag == THREAD)
    {
        p = cur->Lchild;
        if (p->Rchild == cur)
            return p;
    }
    if (cur->Rtag == THREAD)
    {
        p = cur->Rchild;
        if (p->Lchild == cur)
            return p;
    }

    p = Frist(cur->Lchild);
    p = p->Lchild;
    if (p->Rchild == cur)
        return p;
```

```
    p = Last(cur->Rchild);
    return p->Rchild;

}
BinThrNode* Parent_1(BinThrTree*bt,BinThrNode*cur)
{
    Parent(bt->root, cur);
}
```

3.树和森林

孩子兄弟表示法表示树

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

#define ElemType char

typedef struct TreeNode
{
    ElemType data;
    struct TreeNode* fristChild;     //左孩子
    struct TreeNode* nextSibling;    //右兄弟
}TreeNode;

typedef struct Tree
{
    TreeNode* root;
    ElemType refvalue;
}Tree;
```

1.初始化

```
void InitTree(Tree *t,ElemType ref)
{
    t->root = NULL;
    t->refvalue = ref;
}
```

2.创建二叉树

```
void CreatTree_1(Tree* tree, TreeNode* t, char* str)
{
    if (*str == tree->refvalue)
        t = NULL;
```

```
    else
    {
        t = (TreeNode*)malloc(sizeof(TreeNode));
        if (t == NULL)
            return;
        t->data = *str;
        CreatTree_1(tree, t->fristChild, ++str);
        CreatTree_1(tree, t->nextSibling, ++str);
    }
}
void CreatTree(Tree *tree,char*str)
{
    CreatTree_1(tree,tree->root,str);
}
```

3.返回树的根

```
TreeNode* Root(Tree*tree)
{
    return tree->root;
}
```

4.寻找第一个孩子结点

```
TreeNode* FrishtChild_1(TreeNode* t)
{
    if (t == NULL)
        return NULL;
    else {
        return t->fristChild;
    }
}
TreeNode* FrishtChild(Tree*tree)
{
    return FrishtChild_1(tree->root);
}
```

5.寻找第一个兄弟结点

```
TreeNode* NextSibling_1(TreeNode* t)
{
    if (t == NULL)
        return NULL;
    else {
        return t->nextSibling;
    }
}
TreeNode* NextSibling(Tree*tree)
{
    return NextSibling_1(tree->root);
}
```

6.查找指定结点

```
TreeNode* Fine_1(TreeNode* t, ElemType key)
{
    if (t == NULL)
        return NULL;
    if (t->data == key)
        return t;

    TreeNode* p = Fine_1(t->fristChild, key);
    if (p != NULL)
        return p;
    return Fine_1(t->nextSibling, key);
}
TreeNode*Fine(Tree*tree,ElemType key)
{
    return Fine_1(tree->root,key);
}
```

7.寻找该结点的父结点

```
TreeNode* Parent_1(TreeNode*t,TreeNode*p)
{
    if (t == NULL||p == NULL||p==t)
        return NULL;

    TreeNode* q = t->fristChild;
    TreeNode* parent;
    while (q !=NULL && q!=p)
    {
        parent = Parent(q, p);    //二叉树的左子树都是孩子结点，可直接查找父结点
        if (parent != NULL)
            return parent;

        q = q->nextSibling;       //如果左子树都没有，则查找右子树上的兄弟结点，但他们的父结点跟左子树上的不同
    }
    if (q != NULL && q == p)
        return t;
    return NULL;

}
TreeNode* Parent(Tree*tree,TreeNode*p)
{
    Parent_1(tree->root, p);
}
```

第 7 章 图

1.图的定义和术语

2.图的存储结构

邻接矩阵

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

#define Default_Vertex_Size 10
#define ElemType char

typedef struct GraphMtx
{
    int MaxVertices;  //容量
    int NumVertices;  //真实大小
    int NunEdges;     //边的大小

    ElemType* VerticesList;      //顶点列表
    int** Edge;                  //边的矩阵
}GrephMtx;
```

1. 初始化

```
void InitGraph(GraphMtx*g)
{
    g->MaxVertices = Default_Vertex_Size;
    g->NumVertices = g->NunEdges = 0;

    g->VerticesList = (ElemType*)malloc(sizeof(ElemType)*(g->MaxVertices));   //申请空间存放
顶点
    if (g->VerticesList == NULL)
        return;
    g->Edge = (int**)malloc(sizeof(int*) * (g->MaxVertices));
    if (g->Edge == NULL)
        return;
    for (int i = 0; i < g->MaxVertices; i++)
    {
        g->Edge[i] = (int*)malloc(sizeof(int) * g->MaxVertices);
    }
    for (int i = 0; i < g->MaxVertices; i++)
    {
        for (int j = 0; j < g->MaxVertices; j++)
            g->Edge[i][j] = 0;
    }
```

```
}
```

2.插入顶点

```
void InsertVertices(GraphMtx *g, ElemType v)
{
    if (g->NumVertices >= g->MaxVertices)
        return;
    g->VerticesList[g->NumVertices++] = v;
}
```

3.获取顶点的位置

```
int GetVertexPos(GraphMtx* g, ElemType v)
{
    for (int i = 0; i < g->NumVertices; i++)
    {
        if (g->VerticesList[i] == v)
            return i;
    }
    return -1;
}
```

4.插入边(在 v1 和 v2 之间插入一个边)

```
void InsertEdge(GraphMtx*g,ElemType v1, ElemType v2)
{
    int p1 = GetVertexPos(g, v1);
    int p2 = GetVertexPos(g, v2);
    if (p1 == -1 || p2 == -1)
        return;

    if (g->Edge[p1][p2] != 0)
        return;

    g->Edge[p1][p2] = g->Edge[p2][p1] = 1;
    g->NunEdges++;
}
```

5.删除顶点

```
void RemoveVertex(GraphMtx*g,ElemType v)
{
    int numedges = 0;
    int p = GetVertexPos(g, v);
    if (p == -1)
        return;

    //删除顶点列表中的指定顶点
    for (int i = p; i < g->NumVertices - 1; ++i)
    {
        g->VerticesList[i] = g->VerticesList[i + 1];
    }
```

```
    //统计矩阵中指定删除的顶点的边的个数
    for (int i = 0; i < g->NumVertices; ++i)
    {
        if (g->Edge[p][i] != 0)
        {
            numedges++;
        }
    }

    //删除矩阵中的指定顶点的行和列
    for (int i = p; i < g->NumVertices - 1; ++i)
    {
        for (int j = 0; j <g->NumVertices; ++j)
        {
            g->Edge[i][j] = g->Edge[i + 1][j];
        }
    }
    for (int i = p; i < g->NumVertices; ++i)
    {
        for (int j = 0; j < g->NumVertices; ++j)
        {
            g->Edge[j][i] = g->Edge[j][i + 1];
        }
    }
    g->NumVertices--;
    g->NunEdges -= numedges;
}
```

6.删除边(删除 v1 和 v2 之间的边

```
void RemoveEdge(GraphMtx*g,ElemType v1,ElemType v2)
{
    int p1 = GetVertexPos(g, v1);
    int p2 = GetVertexPos(g, v2);
    if (p1 == -1 || p2 == -1||g->Edge[p1][p2] == 0)
        return;

    g->Edge[p1][p2] = g->Edge[p2][p1] = 0;
    g->NunEdges--;
}
```

7.遍历图

```
void Show(GraphMtx*g)
{
    printf("  ");
    //打印横向的顶点
    for (int i = 0; i < g->NumVertices; i++)
    {
        printf("%c ", g->VerticesList[i]);
```

```
    }
    printf("\n");

    for (int i = 0; i < g->NumVertices; i++)
    {
        printf("%c ", g->VerticesList[i]);      //打印纵向的顶点
        for (int j = 0; j < g->NumVertices; j++)
        {
            printf("%d ", g->Edge[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

8.获取第一个邻接顶点

```
int GetFristNeighbor(GraphMtx* g, ElemType v)
{
    int p = GetVertexPos(g, v);
    if (p == -1)
        return -1;
    for (int i = 0; i < g->NumVertices; ++i)
    {
        if (g->Edge[p][i] == 1)
            return i;
    }
    return -1;
}
```

9.获取下一个邻接顶点(找到 v 和 w 的邻接矩阵的下一个顶点

```
int GetNextNeighbor(GraphMtx* g, ElemType v, ElemType w)
{
    int pv = GetVertexPos(g, v);
    int pw = GetVertexPos(g, w);
    if (pv == -1 || pw == -1)
        return -1;

    for (int i = pw + 1; i < g->NumVertices; ++i)
    {
        if (g->Edge[pv][i] == 1)
            return i;
    }
    return -1;
}
```

10.销毁图

```
void DestroyGraph(GraphMtx* g)
{
    free(g->VerticesList);
```

```
    g->VerticesList = NULL;
    for (int i=0;i<g->NumVertices;++i)
    {
        free(g->Edge[i]);
    }
    free(g->Edge);
    g->Edge = NULL;
    g->MaxVertices = g->NumVertices = g->NunEdges = 0;
}
```

邻接表

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

#define Default_Vertex_Size 10
#define T char

typedef struct Edge       //线结点
{
    int dest;
    struct Edge* link;
}Edge;

typedef struct Vertex //顶点结点
{
    ElemType data;
    Edge* adj; //指向线结点的指针
}Vertex;

typedef struct GraphLnk   //邻接表结构
{
    int MaxVertices;
    int NumVertices;
    int NumEdge;

    Vertex* NodeTable; //顶点表
}GraphLnk;
```

1.初始化

```
void InitGraph(GraphLnk *g)
{
    g->MaxVertices = Default_Vertex_Size;
    g->NumEdges = g->NumVertices = 0;

    g->NodeTable = (Vertex*)malloc(sizeof(Vertex) * g->MaxVertices);
    assert(g->NodeTable != NULL);
    for(int i=0; i<g->MaxVertices; ++i)
```

```
    {
        g->NodeTable[i].adj = NULL;
    }
}
```

2.获取顶点位置

```
int  GetVertexPos(GraphLnk *g, T v)
{
    for(int i=0; i<g->NumVertices; ++i)
    {
        if(g->NodeTable[i].data == v)
            return i;
    }
    return -1;
}
```

3.遍历图

```
void ShowGraph(GraphLnk *g)
{
    Edge *p;
    for(int i=0; i<g->NumVertices; ++i)
    {
        printf("%d %c:>",i,g->NodeTable[i].data);
        p = g->NodeTable[i].adj;
        while(p != NULL)
        {
            printf("%d-->",p->dest);
            p = p->link;
        }
        printf("Nul.\n");
    }
    printf("\n");
}
```

4. 插入顶点

```
void InsertVertex(GraphLnk *g, T v)
{
    if(g->NumVertices >= g->MaxVertices)
        return;
    g->NodeTable[g->NumVertices++].data = v;
}
```

5.插入边

```
void InsertEdge(GraphLnk *g, T vertex1, T vertex2)
{
    int v1 = GetVertexPos(g,vertex1);
    int v2 = GetVertexPos(g,vertex2);
    if(v1==-1 || v2==-1)
        return;
```

```
    Edge *s;
    //V1 --> V2
    s = (Edge *)malloc(sizeof(Edge));
    assert(s != NULL);
    s->dest = v2;
    s->link = g->NodeTable[v1].adj;
    g->NodeTable[v1].adj = s;

    //V2 --> V1
    s = (Edge *)malloc(sizeof(Edge));
    assert(s != NULL);
    s->dest = v1;
    s->link = g->NodeTable[v2].adj;
    g->NodeTable[v2].adj = s;

    g->NumEdges++;
}
```

6.删除边

```
void RemoveEdge(GraphLnk *g, T vertex1, T vertex2)
{
    int v1 = GetVertexPos(g,vertex1);
    int v2 = GetVertexPos(g,vertex2);

    if(v1==-1 || v2==-1)
        return;

    Edge *q = NULL;
    Edge *p;
    //v1 -- > v2
    p = g->NodeTable[v1].adj;
    while(p != NULL && p->dest != v2)
    {
        q = p;
        p = p->link;
    }
    if(p == NULL)
        return;

    if(q == NULL)
    {
        g->NodeTable[v1].adj = p->link;
    }
    else
    {
        q->link = p->link;
    }
    free(p);
```

```
    //v2 --> v1
    q = NULL;
    p = g->NodeTable[v2].adj;
    while(p->dest != v1)
    {
        q = p;
        p = p->link;
    }
    if(q==NULL)
    {
        g->NodeTable[v2].adj = p->link;
    }
    else
    {
        q->link = p->link;
    }
    free(p);
    g->NumEdges--;
}
```

7.删除顶点

```
void RemoveVertex(GraphLnk *g, T vertex)
{
    int v = GetVertexPos(g,vertex);
    if(v == -1)
        return;

    Edge *p = g->NodeTable[v].adj;

    int k;
    Edge *t = NULL;
    Edge *s;
    while(p!=NULL)
    {
        k = p->dest;
        s = g->NodeTable[k].adj;
        while(s!=NULL && s->dest!=v)
        {
            t = s;
            s = s->link;
        }
        if(s!=NULL)
        {
            if(t==NULL)
            {
                g->NodeTable[k].adj = s->link;
            }
            else
```

```
            {
                t->link = s->link;
            }
            free(s);
        }

        g->NodeTable[v].adj = p->link;
        free(p);
        p = g->NodeTable[v].adj;
    }

    g->NumVertices--;
    g->NodeTable[v].data = g->NodeTable[g->NumVertices].data;
    g->NodeTable[v].adj = g->NodeTable[g->NumVertices].adj;

    s = g->NodeTable[v].adj;
    while(s != NULL)
    {
        k = s->dest;
        p = g->NodeTable[k].adj;
        while(p != NULL)
        {
            if(p->dest == g->NumVertices)
            {
                p->dest = v;
                break;
            }
            p = p->link;
        }
        s = s->link;
    }
}
```

8.销毁图

```
void DestroyGraph(GraphLnk *g)
{
    Edge *p;
    for(int i=0; i<g->NumVertices; ++i)
    {
        p = g->NodeTable[i].adj;
        while(p != NULL)
        {
            g->NodeTable[i].adj = p->link;
            free(p);
            p = g->NodeTable[i].adj;
        }
    }
    free(g->NodeTable);
    g->NodeTable = NULL;
```

```
    g->MaxVertices = g->NumEdges = g->NumVertices = 0;
}
```

9.获取第一个邻接顶点

```
int  GetFirstNeighbor(GraphLnk *g, T vertex)
{
    int v = GetVertexPos(g,vertex);
    if(v == -1)
        return -1;
    Edge *p = g->NodeTable[v].adj;
    if(p != NULL)
        return p->dest;
    return -1;
}
```

10.获取下一个顶点

```
int  GetNextNeighbor(GraphLnk *g,  T vertex1, T vertex2)
{
    //返回 v（相对于 w 的）下一个邻接顶点，如果 w 是 v 最后一个邻接点，则返回空
    int v1 = GetVertexPos(g,vertex1);
    int v2 = GetVertexPos(g,vertex2);
    if(v1==-1 || v2==-1)
        return -1;

    Edge *p = g->NodeTable[v1].adj;
    while(p != NULL && p->dest != v2)
        p = p->link;
    if(p!=NULL && p->link!=NULL)
        return p->link->dest;
    return -1;
}
```

3.图的遍历

在遍历图时，因为各顶点之间可能是互相连通的，为了避免重复遍历需要设一个辅助数组，它的初始值为假或者零，当顶点 vi 被访问时，设置数组的第 i 个位置为真或零。

深度优先遍历（DFS）：从某个顶点 v 开始，然后访问 v 未被访问的一个邻接顶点，依次访问，直到图中所有的顶点都被访问过为止。

广度优先搜索（BFS）：类似于二叉树的层序遍历。从某个顶点 v 开始，先访问完 v 的所有邻接顶点，再访问下一个邻接顶点，依次访问。利用队列实现。

其他函数接“邻接表”

```
//获取顶点的值
T    GetVertexValue(GraphLnk *g, int v)
{
    if(v == -1)
        return 0;
    return g->NodeTable[v].data;
}
```

DFS

```
void DFS(GraphLnk *g, T vertex)
{
    int n = g->NumVertices;
    bool *visited = (bool*)malloc(sizeof(bool) * n);    //设一个数组用来标记访问过的顶点
    assert(visited != NULL);                            //未访问为假（0）访问过为真（1）
    for(int i=0; i<n; ++i)
    {
        visited[i] = false;
    }

    int v = GetVertexPos(g,vertex);
    DFS_1(g,v,visited);
    free(visited);
}

void DFS_1(GraphLnk *g, int v, bool visited[])
{
    printf("%c-->",GetVertexValue(g,v));
    visited[v] = true;
    int w = GetFirstNeighbor(g,GetVertexValue(g,v));
    while(w != -1)
    {
        if(!visited[w])
        {
            DFS_1(g,w,visited);
        }
        w = GetNextNeighbor(g,GetVertexValue(g,v),GetVertexValue(g,w));
    }
}
```

BFS

```
void BFS(GraphLnk *g, T vertex)
{
    int n = g->NumVertices;
    bool *visited = (bool*)malloc(sizeof(bool) * n);
    assert(visited != NULL);
    for(int i=0; i<n; ++i)
    {
```

```
        visited[i] = false;
    }

    int v = GetVertexPos(g,vertex);
    printf("%c-->",vertex);
    visited[v] = true;

    LinkQueue Q;   //引用"LinkQueue.h"
    InitQueue(&Q);

    EnQueue(&Q,v);
    int w;
    while(!Empty(&Q)) //判空函数在"LinkQueue.h"中实现
    {
        GetHead(&Q,&v);
        DeQueue(&Q);

        w = GetFirstNeighbor(g,GetVertexValue(g,v));
        while(w != -1)
        {
            if(!visited[w])
            {
                printf("%c-->",GetVertexValue(g,w));
                visited[w] = true;
                EnQueue(&Q,w);
            }
            w = GetNextNeighbor(g,GetVertexValue(g,v),GetVertexValue(g,w));
        }
    }
    free(visited);
}
```

```
//非连通遍历方式（针对有不连通的顶点的图的遍历方式）
void Components(GraphLnk *g)
{
    int n = g->NumVertices;
    bool *visited = (bool*)malloc(sizeof(bool) * n);
    assert(visited != NULL);
    for(int i=0; i<n; ++i)
    {
        visited[i] = false;
    }
    for(i=0; i<n; ++i)
    {
        if(!visited[i])
            DFS_1(g,i,visited);
    }
    free(visited);
}
```

4.图的连通性问题

Pirm 算法（利用邻接矩阵实现）

```
#include<stdio.h>
#include<malloc.h>
#include<assert.h>

#define Default_Vertex_Size  10

#define T char
#define E int  //权值类型
#define MAX_COST  0x7FFFFFFF  //初始化时，矩阵中的元素都是无穷大的
                              //与无向图的邻接矩阵中的 0 元素意义相同

typedef struct GraphMtx
{
    int MaxVertices;
    int NumVertices;
    int NumEdges;

    T   *VerticesList;
    int **Edge;
}GraphMtx;
```

1.初始化

```
void InitGraph(GraphMtx *g)
{
    g->MaxVertices = Default_Vertex_Size;
    g->NumVertices = g->NumEdges = 0;

    g->VerticesList = (T*)malloc(sizeof(T)*(g->MaxVertices));
    assert(g->VerticesList != NULL);

    g->Edge = (int**)malloc(sizeof(int*) * g->MaxVertices);
    assert(g->Edge != NULL);
    for(int i=0; i<g->MaxVertices; ++i)
    {
        g->Edge[i] = (int*)malloc(sizeof(int) * g->MaxVertices);
    }
    for(i=0; i<g->MaxVertices; ++i)
    {
        for(int j=0; j<g->MaxVertices; ++j)
        {
            if(i == j)
            {
                g->Edge[i][j] = 0;
            }
```

```
            else
            {
                g->Edge[i][j] = MAX_COST;
            }
        }
    }
}
```

2.打印图

```
void ShowGraph(GraphMtx *g)
{
    printf("  ");
    for(int i=0; i<g->NumVertices; ++i)
    {
        printf("%c  ",g->VerticesList[i]);
    }
    printf("\n");
    for(i=0; i<g->NumVertices; ++i)
    {
        printf("%c ",g->VerticesList[i]);
        for(int j=0; j<g->NumVertices; ++j)
        {
            if(g->Edge[i][j] == MAX_COST)
            {
                printf("%c  ",'@');
            }
            else
            {
                printf("%d  ",g->Edge[i][j]);
            }
        }
        printf("\n");
    }
    printf("\n");
}
```

3.插入边

```
void InsertEdge(GraphMtx *g, T v1, T v2, E cost)
{
    int p1 = GetVertexPos(g,v1);
    int p2 =  GetVertexPos(g,v2);
    if(p1==-1 || p2==-1)
        return;

    g->Edge[p1][p2] = g->Edge[p2][p1] = cost;
    g->NumEdges++;
}
```

#其余函数与邻接矩阵完全一样

获得权值

```
E GetWeight(GraphMtx *g, int v1, int v2)
{
    if(v1==-1 || v2==-1)
        return MAX_COST;
    return g->Edge[v1][v2];
}
```

最小生成树-prim

```
void MinSpanTree_Prim(GraphMtx *g, T vertex)
{
    //初始化
    int n = g->NumVertices;
    E *lowcost = (E*)malloc(sizeof(E)*n); //顶点的权值 lowcost[n]
    int *mst = (int *)malloc(sizeof(int)*n);//起始顶点 mst[n]
    assert(lowcost!=NULL && mst!=NULL);

    int k = GetVertexPos(g,vertex);

    for(int i=0; i<n; ++i)
    {
        if(i != k)
        {
            lowcost[i] = GetWeight(g,k,i);
            mst[i] = k;
        }
        else
        {
            lowcost[i] = 0;
        }
    }

    //寻找最小生成树
    int min,min_index;
    int begin,end;
    E cost;

    for(i=0; i<n-1; ++i)
    {
        min = MAX_COST;
        min_index = -1;
        for(int j=0; j<n; ++j)
        {
            if(lowcost[j]!=0 && lowcost[j]<min)
            {
                min = lowcost[j];
                min_index = j;
            }
```

```
        }
        begin = mst[min_index];
        end = min_index;
        printf("%c-->%c : %d\n",g->VerticesList[begin],g->VerticesList[end],min);

        lowcost[min_index] = 0;       //把路径权值最小的顶点合并
        //更新起始顶点
        for(j=0; j<n; ++j)
        {
            cost = GetWeight(g,min_index,j);
            if(cost < lowcost[j])
            {
                lowcost[j] = cost;
                mst[j] = min_index;
            }
        }
    }
}
```

Kruskal 算法

```
//边结构
typedef struct Edge
{
    int x; // start
    int y; // end
    E   cost;
}Edge;
```

快排的比较方法

```
int cmp(const void*a, const void *b)
{
    return (*(Edge*)a).cost - (*(Edge*)b).cost;
}
```

判断是否有相同的父顶点

```
bool Is_same(int *father, int i, int j)
{
    while(father[i] != i)
    {
        i = father[i];
    }
    while(father[j] != j)
    {
        j = father[j];
    }
    return i==j;
}
```

标记顶点为相同集合

```
void Mark_same(int *father, int i, int j)
{
    while(father[i] != i)
    {
        i = father[i];
    }
    while(father[j] != j)
    {
        j = father[j];
    }
    father[j] = i;
}
```

Kruskal

```
void MinSpanTree_Kruskal(GraphMtx *g)
{
    int n = g->NumVertices;
    Edge *edge = (Edge *)malloc(sizeof(Edge) * (n*(n-1)/2));
    assert(edge != NULL);

    int k = 0;
    for(int i=0; i<n; ++i)
    {
        for(int j=i; j<n; ++j)
        {
            if(g->Edge[i][j]!=0 && g->Edge[i][j]!=MAX_COST)
            {
                edge[k].x = i;
                edge[k].y = j;
                edge[k].cost = g->Edge[i][j];
                k++;
            }
        }
    }

    int v1,v2;

    qsort(edge,k,sizeof(Edge),cmp);      //#include<stdlib.h>

    int *father = (int*)malloc(sizeof(int) * n);
    assert(father != NULL);
    for(i=0; i<n; ++i)
    {
        father[i] = i;
    }

    for(i=0; i<n; ++i)
```

```
    {
        if(!Is_same(father,edge[i].x,edge[i].y))
        {
            v1 = edge[i].x;
            v2 = edge[i].y;
            printf("%c-->%c : %d\n",g->VerticesList[v1],g->VerticesList[v2],edge[i].cost);
            Mark_same(father,edge[i].x,edge[i].y);
        }
    }
}
```

5. 有向无环图及其应用

AOV 网络-拓扑排序（利用邻接表）

AOV 网：用顶点表示活动的有向图。

```
void TopologicalSort(GraphLnk *g)
{
    int n = g->NumVertices;
    int *count = (int *)malloc(sizeof(int)*n);  //记录顶点的入度
    assert(count != NULL);
    for(int i=0; i<n; ++i)
    {
        count[i] = 0;
    }

    //统计结点入度
    Edge *p;
    for(i=0; i<n; ++i)
    {
        p = g->NodeTable[i].adj;
        while(p != NULL)
        {
            count[p->dest]++;
            p = p->link;
        }
    }

    int top = -1;
    for(i=0; i<n; ++i)
    {
        if(count[i] == 0)
        {
            count[i] = top;   //Push
            top = i;
```

```
        }
    }

    int v,w;
    for(i=0; i<n; ++i)
    {
        if(top == -1)
        {
            printf("网络中有回路.\n");
            return;
        }
        else
        {
            v = top;            //Pop
            top = count[top];
            printf("%c-->",g->NodeTable[v]);
            w = GetFirstNeighbor(g,g->NodeTable[v].data);
            while(w != -1)
            {
                if(--count[w] == 0)
                {
                    count[w] = top;
                    top = w;
                }
                w = GetNextNeighbor(g,g->NodeTable[v].data,g->NodeTable[w].data);
            }
        }
    }
    free(count);
}
```

AOE 网络-关键路径

AOE 网：带权的有向无环图，顶点表示事件，弧表示活动，权表示活动持续时间。

关键路径：路径长度最长的路径，即权值之和最大的路径。

```
void CriticalPath(GraphMtx *g)
{
    int n = g->NumVertices;
    int *ve = (int*)malloc(sizeof(int) * n);
    int *vl = (int*)malloc(sizeof(int) * n);
    assert(ve!=NULL && vl!=NULL);

    for(int i=0; i<n; ++i)
    {
        ve[i] = 0;
        vl[i] = MAX_COST;
```

```
    }

    int j,w;
    //ve
    for(int i=0; i<n; ++i)
    {
        j = GetFirstNeighbor(g,g->VerticesList[i]);
        while(j != -1)
        {
            w = GetWeight(g,i,j);
            if(ve[i]+w > ve[j])
            {
                ve[j] = ve[i]+w;
            }
            j = GetNextNeighbor(g,g->VerticesList[i],g->VerticesList[j]);
        }
    }

    //vl
    vl[n-1] = ve[n-1];
    for(int i=n-2; i>0; --i)
    {
        j = GetFirstNeighbor(g,g->VerticesList[i]);
        while(j != -1)
        {
            w = GetWeight(g,i,j);
            if(vl[j]-w < vl[i])
            {
                vl[i] = vl[j]-w;
            }
            j = GetNextNeighbor(g,g->VerticesList[i],g->VerticesList[j]);
        }
    }

    int Ae, Al; //
    for(int i=0; i<n; ++i)
    {
        j = GetFirstNeighbor(g,g->VerticesList[i]);
        while(j != -1)
        {
            Ae = ve[i];
            Al = vl[j] - GetWeight(g,i,j);
            if(Ae == Al)
            {
                printf("<%c,%c>是关键路径.\n",g->VerticesList[i],g->VerticesList[j]);
            }
            j = GetNextNeighbor(g,g->VerticesList[i],g->VerticesList[j]);
        }
    }
```

```
    free(ve);
    free(vl);
}
```

6. 最短路径

带权有向图中某个源点到其他各顶点的的最短路径。利用邻接表实现。

迪杰斯特拉提出一个按路径长度递增的次序产生的最短路径。

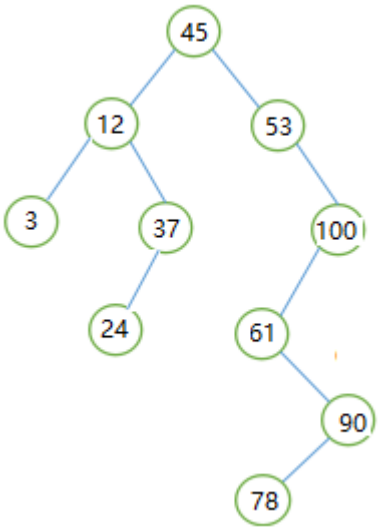
```
E dist[5];     //记录最短路径的顶点权值
int path[5];   //记录某个源点到其他各顶点的最短路径

void ShortestPath(GraphMtx *g, T vertex, E dist[], int path[])
{
    int n = g->NumVertices;
    bool *S = (bool*)malloc(sizeof(bool) * n);  //已找到最短路径终点的集合。
    assert( S != NULL);
    int v = GetVertexPos(g,vertex);

    for(int i=0; i<n; ++i)
    {
        dist[i] = GetWeight(g,v,i);
        S[i] = false;
        if(i!=v && dist[i]<MAX_COST)
        {
            path[i] = v;
        }
        else
        {
            path[i] = -1;
        }
    }

    S[v] = true;
    int min;
    int w;

    for(i=0; i<n-1; ++i)
    {
        min = MAX_COST;
        int u = v;
        for(int j=0; j<n; ++j)
        {
            if(!S[j] && dist[j]<min)
```

```
            {
                u = j;
                min = dist[j];
            }
        }

        S[u] = true;
        for(int k=0; k<n; ++k)
        {
            w = GetWeight(g,u,k);
            if(!S[k] && w<MAX_COST && dist[u]+w<dist[k])
            {
                dist[k] = dist[u]+w;
                path[k] = u;
            }
        }
    }
}
```

第 9 章

1. 动态查找

二叉排序树




```
#include<stdio.h>
#include<malloc.h>
#include<assert.h>

#define T int
#define FALSE 0
#define TRUE  1
#define BOOL  int

typedef struct BSTNode    //结点结构
{
    T data;
    BSTNode *leftChild;
    BSTNode *rightChild;
}BSTNode;

typedef struct BST//排序树结构
{
    BSTNode *root;
}BST;
```

1.初始化

```
void InitBSTree(BST *bst)
{
    bst->root = NULL;
}
```

2.插入结点

```
BOOL InsertBSTree_1(BSTNode **t, T x)
{
    if(*t == NULL)
    {
        *t = (BSTNode*)malloc(sizeof(BSTNode));
        assert(*t != NULL);
        (*t)->data = x;
        (*t)->leftChild = NULL;
        (*t)->rightChild = NULL;
        return TRUE;
    }
    else if(x < (*t)->data)  //左子树的值都小于根结点的值
    {
        InsertBSTree_1(&(*t)->leftChild,x);
    }
    else if(x > (*t)->data)  //右子树的值都大于根结点的值
    {
        InsertBSTree_1(&(*t)->rightChild,x);
    }
    return FALSE;
}
BOOL InsertBSTree(BST *bst, T x)
{
    return InsertBSTree_1(&bst->root, x);
}
```

3.求最小值

```
T Min_1(BSTNode *t)
{
    while(t->leftChild != NULL)
        t = t->leftChild;
    return t->data;
}
T Min(BST *bst)
{
    assert(bst->root != NULL);
    return Min_1(bst->root);
}
```

4.求最大值

```
T Max_1(BSTNode *t)
{
    while(t->rightChild != NULL)
        t = t->rightChild;
    return t->data;
}
T Max(BST *bst)
```

```
{
    assert(bst->root != NULL);
    return Max_1(bst->root);
}
```

5.排序（相当于中序遍历）

```
void Sort_1(BSTNode *t)
{
    if(t != NULL)
    {
        Sort_1(t->leftChild);
        printf("%d ",t->data);
        Sort_1(t->rightChild);
    }
}
void Sort(BST *bst)
{
    Sort_1(bst->root);
}
```

6.查找

```
BSTNode* Search_1(BSTNode *t, T key)
{
    if(t == NULL)
        return NULL;
    if(t->data == key)
        return t;
    if(key < t->data)
        return Search_1(t->leftChild,key);
    else
        return Search_1(t->rightChild,key);
}
BSTNode* Search(BST *bst, T key)
{
    return Search_1(bst->root, key);
}
```

7.置空

```
void MakeEmptyBSTree_1(BSTNode **t)
{
    if(*t != NULL)
    {
        MakeEmptyBSTree_1(&(*t)->leftChild);
        MakeEmptyBSTree_1(&(*t)->rightChild);
        free(*t);
        *t = NULL;
    }
}
void MakeEmptyBSTree(BST *bst)
```

```
{
    MakeEmptyBSTree_1(&bst->root);
}
```

8.删除结点

```
BOOL RemoveBSTree_1(BSTNode **t, T key)
{
    if(*t == NULL)
        return FALSE;
    if(key < (*t)->data)
        RemoveBSTree_1(&(*t)->leftChild, key);
    else if(key > (*t)->data)
        RemoveBSTree_1(&(*t)->rightChild, key);
    else
    {
        BSTNode *p = NULL;
        if((*t)->leftChild!=NULL && (*t)->rightChild!=NULL)
        {
            p = (*t)->rightChild;
            while(p->leftChild != NULL)
                p = p->leftChild;
            (*t)->data = p->data;
            RemoveBSTree_1(&(*t)->rightChild,p->data);
        }
        else
        {
            p = *t;
            if((*t)->leftChild == NULL)
                (*t) = (*t)->rightChild;
            else
                (*t) = (*t)->leftChild;
            free(p);
            p = NULL;
        }
    }
    return TRUE;
}
BOOL RemoveBSTree(BST *bst, T key)
{
    return RemoveBSTree_1(&bst->root,key);
}
```

平衡二叉树

平衡因子 BF：表示该结点的左子树的深度-右子树的深度。

平衡二叉树（AVL 树）：每个结点的平衡因子是 1,0，或-1 的二叉查找树。

```
#include<stdio.h>
#include<malloc.h>
#include<assert.h>

#define Type int
#define BOOL int
#define TRUE 1
#define FALSE 0

typedef struct AVLNode
{
    Type     data;
    AVLNode *leftChild;
    AVLNode *rightChild;
    int      bf;   //平衡因子
}AVLNode;

typedef struct AVLTree
{
    AVLNode *root;
}AVLTree;

//初始化
void InitAVLTree(AVLTree *avl)
{
    avl->root = NULL;
}

AVLNode* BuyNode(Type x)
{
    AVLNode *p = (AVLNode *)malloc(sizeof(AVLNode));
    assert(p != NULL);
    p->data = x;
    p->leftChild = p->rightChild = NULL;
    p->bf = 0;
    return p;
}
```

1.插入结点

```
BOOL InsertAVL_1(AVLNode *&t , Type x)
{
    AVLNode *p = t;
    AVLNode *parent = NULL;

    Stack st;
    InitStack(&st);

    while(p != NULL)
    {
```

```
        if(x == p->data)
            return FALSE;
        parent = p;
        Push(&st,parent);
        if(x < p->data)
            p = p->leftChild;
        else
            p = p->rightChild;
    }
    p = BuyNode(x);
    if(parent == NULL)
    {
        t = p;
        return TRUE;
    }
    if(x < parent->data)
        parent->leftChild = p;
    else
        parent->rightChild = p;

    ////////////////////////////////////////////
    //调整 BF
    while(!IsEmpty(&st))
    {
        parent = GetTop(&st);
        Pop(&st);
        if(parent->leftChild == p)
            parent->bf--;
        else
            parent->bf++;
        if(parent->bf == 0)
            break;
        if(parent->bf==1 || parent->bf==-1)
            p = parent;
        else
        {
            ////////////////////////////
            //旋转化平衡调整
            int flag = (parent->bf<0)?-1:1;
            if(p->bf == flag)   //单旋转
            {
                if(flag == -1)
                    RotateR(parent);  //   /
                else
                    RotateL(parent);  //   \

            }
            else   //双旋转
            {
```

```
                if(flag == 1)
                    RotateRL(parent); // >
                else
                    RotateLR(parent); // <
            }
            break;
        }
    }

    if(IsEmpty(&st))
        t = parent;
    else
    {
        AVLNode *q = GetTop(&st);
        if(q->data > parent->data)
            q->leftChild = parent;
        else
            q->rightChild = parent;
    }
    return TRUE;
}

BOOL InsertAVL(AVLTree *avl, Type x)
{
    return InsertAVL_1(avl->root, x);
}
```

2.删除结点

```
BOOL RemoveAVL_1(AVLNode *&t, Type key)
{
    AVLNode *ppr    = NULL;
    AVLNode *parent = NULL;
    AVLNode *p = t;
    Stack st;
    InitStack(&st);
    while(p != NULL)
    {
        if(p->data == key)
            break;
        parent = p;
        Push(&st,parent);
        if(key < p->data)
            p = p->leftChild;
        else
            p = p->rightChild;
    }
    if(p == NULL)
        return FALSE;
```

```
    AVLNode *q;
    int f = 0;   //leftChild NULL   Or  rightChild NULL
    if(p->leftChild!=NULL && p->rightChild!=NULL)
    {
        parent = p;
        Push(&st,parent);
        q = p->leftChild;
        while(q->rightChild != NULL)
        {
            parent = q;
            Push(&st,parent);
            q = q->rightChild;
        }
        p->data = q->data;
        p = q;
    }
    if(p->leftChild != NULL)
        q = p->leftChild;
    else
        q = p->rightChild;

    if(parent == NULL)
        t = parent;
    else
    {
        if(parent->leftChild == p)
        {
            parent->leftChild = q;
            f = 0; // L
        }
        else
        {
            parent->rightChild = q;
            f = 1; //R
        }

        int link_flag = 0;  //-1 leftChild
                            //1  rightChild
                            //0  no link
        while(!IsEmpty(&st))
        {
            parent = GetTop(&st);
            Pop(&st);
            if(parent->rightChild==q && f==1)
                parent->bf--;
            else
                parent->bf++;

            if(!IsEmpty(&st))
```

```
        {
            ppr = GetTop(&st);
            link_flag = (ppr->leftChild==parent)?-1 : 1;
        }
        else
        {
            link_flag = 0; //
        }

        if(parent->bf==-1 || parent->bf==1)
            break;
        if(parent->bf == 0)
            q = parent;
        else   //|2|
        {
            int flag = 0;
            if(parent->bf < 0)
            {
                flag = -1;
                q = parent->leftChild;
            }
            else
            {
                flag = 1;
                q = parent->rightChild;
            }
            if(q->bf == 0) //单旋转
            {
                if(flag == -1) //右单旋转
                {
                    RotateR(parent);
                    parent->bf = 1;
                    parent->rightChild->bf = -1;
                }
                else
                {
                    RotateL(parent);
                    parent->bf = -1;
                    parent->leftChild->bf = 1;
                }
                break;
            }
            if(q->bf == flag)
            {
                if(flag == -1) //右单旋转
                    RotateR(parent);
                else
                    RotateL(parent);
            }
```

```
                else
                {
                    if(flag == -1)
                        RotateLR(parent);
                    else
                        RotateRL(parent);
                }
                if(link_flag == 1)
                    ppr->rightChild = parent;
                else if(link_flag == -1)
                    ppr->leftChild = parent;

            }
        }

        if(IsEmpty(&st))
            t = parent;
    }

    free(p);
    return TRUE;
}

BOOL RemoveAVL(AVLTree *avl, Type key)
{
    return RemoveAVL_1(avl->root, key);
}
```

3.单旋转

```
void RotateR(AVLNode *&ptr)
{
    AVLNode *subR = ptr;
    ptr = subR->leftChild;
    subR->leftChild = ptr->rightChild;
    ptr->rightChild = subR;
    ptr->bf = subR->bf = 0;
}
void RotateL(AVLNode *&ptr)
{
    AVLNode *subL = ptr;
    ptr = subL->rightChild;
    subL->rightChild = ptr->leftChild;
    ptr->leftChild = subL;
    ptr->bf = subL->bf = 0;
}
```

4.双旋转

```
void RotateLR(AVLNode *&ptr)
{
```

```
    AVLNode *subR = ptr;
    AVLNode *subL = subR->leftChild;
    ptr = subL->rightChild;

    subL->rightChild = ptr->leftChild;
    ptr->leftChild = subL;
    if(ptr->bf <= 0)
        subL->bf = 0;
    else
        subL->bf = -1;

    subR->leftChild = ptr->rightChild;
    ptr->rightChild = subR;
    if(ptr->bf == -1)
        subR->bf = 1;
    else
        subR->bf = 0;
    ptr->bf = 0;
}
void RotateRL(AVLNode *&ptr)
{
    AVLNode *subL = ptr;
    AVLNode *subR = subL->rightChild;
    ptr = subR->leftChild;

    subR->leftChild = ptr->rightChild;
    ptr->rightChild = subR;
    if(ptr->bf >= 0)
        subR->bf = 0;
    else
        subR->bf = 1;

    subL->rightChild = ptr->leftChild;
    ptr->leftChild = subL;
    if(ptr->bf == 1)
        subL->bf = -1;
    else
        subL->bf = 0;
    ptr->bf = 0;
}
```

2. 哈希表

```
#include<stdio.h>
#include<malloc.h>
#include<assert.h>
```

```
#define ElemType int
#define P 13

typedef struct HashNode
{
    ElemType data;
    struct HashNode *link;
}HashNode;

typedef HashNode* HashTable[P];

int Hash(ElemType key)
{
    return key % P;
}
```

1.初始化

```
void InitHashTable(HashTable &ht)
{
    for(int i=0; i<P; ++i)
    {
        ht[i] = NULL;
    }
}
```

2.插入

```
void InsertHashTable(HashTable &ht, ElemType x)
{
    int index = Hash(x);
    HashNode *s = (HashNode *)malloc(sizeof(HashNode));
    assert(s != NULL);
    s->data = x;

    //头插
    s->link = ht[index];
    ht[index] = s;
}
```

3.打印

```
void ShowHashTable(HashTable &ht)
{
    for(int i=0; i<P; ++i)
    {
        printf("%d : ",i);
        HashNode *p = ht[i];
        while(p != NULL)
        {
            printf("%d-->",p->data);
            p = p->link;
```

```
        }
        printf("Nul. \n");
    }
}
```

4.查找

```
HashNode* SearchHashTable(HashTable &ht, ElemType key)
{
    int index = Hash(key);
    HashNode *p = ht[index];
    while(p!=NULL && p->data!=key)
        p = p->link;
    return p;
}
```

5.删除

```
bool RemoveHashTable(HashTable &ht, ElemType key)
{
    HashNode *p = SearchHashTable(ht,key);
    if(p == NULL)
        return false;

    int index = Hash(key);
    HashNode *q = ht[index];
    if(q == p)
    {
        ht[index] = p->link;
        free(p);
        return true;
    }

    while(q->link != p)
        q = q->link;
    q->link = p->link;
    free(p);
    return true;
}
```

Hash 表溢出桶实现

现在有一个用来存放整数的 Hash 表，Hash 表的存储单位称为桶，每个桶能放 3 个整数，当一个桶中要放的元素超过 3 个时，则要将新的元素存放在溢出桶中，每个溢出桶也能放 3 个元素，多个溢出桶使用链表串起来。此 Hash 表的基桶数目为素数 P，Hash 表的 hash 函数对 P 取模。

```
#define P 7

#define NULL_DATA -1
#define BUCKET_NODE_SIZE 3

struct bucket_node
{
    int data[BUCKET_NODE_SIZE];
    struct bucket_node *next;
};

int Hash(int key)
{
    return key % P;
}
```

1.初始化

```
void Init_bucket_node()
{
    for(int i=0; i<P; ++i)
    {
        for(int j=0; j<BUCKET_NODE_SIZE; ++j)
        {
            hash_table[i].data[j] = NULL_DATA;
        }
        hash_table[i].next = NULL;
    }
}
```

2.插入

```
int Insert_new_element(int x)
{
    int index = Hash(x);
    for(int i=0; i<BUCKET_NODE_SIZE; ++i)
    {
        if(hash_table[index].data[i] == NULL_DATA)
        {
            hash_table[index].data[i] = x;
            return 0;
        }
    }

    bucket_node *p = &hash_table[index];
    while(p->next != NULL)
    {
        p = p->next;
        for(i=0; i<BUCKET_NODE_SIZE; ++i)
        {
            if(p->data[i] == NULL_DATA)
```

```
            {
                p->data[i] = x;
                return 0;
            }
        }
    }

    bucket_node *s = (bucket_node*)malloc(sizeof(bucket_node));
    assert(s != NULL);
    for(i=0; i<BUCKET_NODE_SIZE; ++i)
    {
        s->data[i] = NULL_DATA;
    }
    s->next = NULL;

    s->data[0] = x;
    p->next = s;
    return 0;
}
```

第 10 章 内部排序

1.插入排序

直接插入排序：

是一种最简单的排序方式，它的基本操作是将一个记录插入到已经排序好的有序表中，从而得到一个新的有序表。

```
#include<malloc.h>
#include<assert.h>

#define T int
#define MAXSIZE  20

typedef T SqList[MAXSIZE];   //顺序表
```

```
 void Swap(T *a, T *b)
{
    T tmp = *a;
    *a = *b;
    *b = tmp;
}

void InsertSort(SqList &L, int n)
{
    for(int i=2; i<n; ++i)
    {
        if(L[i] < L[i-1])
        {
            L[0] = L[i];
            L[i] = L[i-1];

            for(int j=i-2; L[0] < L[j]; --j)
            {
                L[j+1] = L[j];
            }
            L[j+1] = L[0];
        }
    }
}
```

折半插入：

由于插入是在一个有序表中进行的，所以可以使用折半查找的方法来实现插入。

```
void BInsertSort(SqList &L, int n)
{
    for(int i=2; i<n; ++i)
    {
        L[0] = L[i];   //把目标值放在 0 下标里
        int low = 1;
        int high = i-1;
        while(low <= high)
        {
            int mid = (low+high)/2;
            if(L[0] >= L[mid])
            {
                low = mid+1;
            }
            else
            {
                high = mid-1;
            }
        }

        for(int j=i; j>high+1; --j)
        {
            L[j] = L[j-1];
        }
        L[high+1] = L[0];
    }
}
```

2-路插入排序：

在折半插入的基础上再次进行改进，其目的是减少排序过程中移动记录的次数，但为此需要 n 个记录的辅助空间。

```
void TWayInsertSort(SqList &L, int n)
{
    SqList TP; //辅助空间
    TP[0] = L[0];
    int head, tail;
    head = tail = 0;

    for(int i=1; i<n; ++i)
    {
        if(L[i] < TP[head])
```

```
        {
            head = (head-1+n)%n;
            TP[head] = L[i];
        }
        else if(L[i] > TP[tail])
        {
            tail++;
            TP[tail] = L[i];
        }
        else
        {
            tail++;
            TP[tail] = TP[tail-1];
            for(int j=tail-1;L[i]<TP[(j-1+n)%n]; j=(j-1+n)%n)
            {
                TP[j] = TP[(j-1+n)%n];
            }
            TP[j] = L[i];
        }
    }

    for(i=0; i<n; ++i)
    {
        L[i] = TP[head];
        head = (head+1)%n;
    }
}
```

表插入排序:

为了插入方便，将下标为 0 的元素设为最大整数 M，每个元素添加一个 link 用以表示下一个元素的下标值以达到排序的效果。不移动元素，只对下标做操作。

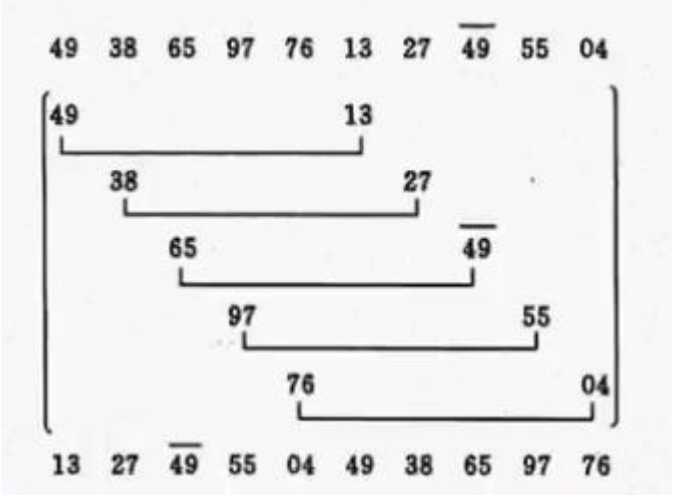
```
#define MAXVALUE 0x7fffffff
typedef struct SLNode
{
    T data;
    int link;
}SLNode;

typedef SLNode Table[MAXSIZE];
```

```
void TableInsertSort(Table &t, int n)
{
    t[0].link = 1;
    int p,q;
```

```
    for(int i=2; i<n; ++i)
    {
        p = t[0].link;
        q = 0;
        while(p!=0 && t[p].data<=t[i].data)
        {
            q = p;
            p = t[p].link;
        }
        t[i].link = t[q].link;
        t[q].link = i;
    }
}
```

希尔排序：

又称“缩小增量排序”，它也是一种属插入排序类的方法，但在时间效率上较前几种排序方法有较大的改进。其时间复杂度可提高至 O(n).基本思路是先将整个待排记录序列分割称为若干个子列分别进行排序后进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行一次直接插入排序。




```
int dlta[] = {5,3,2,1};   //每次比较元素时的增量
int t = sizeof(dlta)/sizeof(int);    //  增量个数
ShellSort(L, n, dlta, t);
```

```
void ShellInsert(SqList &L, int n, int dk)
{
    for(int i=dk+1; i<n; ++i)
    {
        if(L[i] < L[i-dk])
        {
            L[0] = L[i];
            for(int j=i-dk; j>0&&L[0]<L[j]; j-=dk)
            {
```

```
                L[j+dk] = L[j];
            }
            L[j+dk] = L[0];
        }
    }
}

void ShellSort(SqList &L, int n, int dlta[], int t)
{
    //对比较元素之间的增量进行循环，间隔 5（4,3,2,1）个元素
    for(int k=0; k<t; ++k)
    {
        ShellInsert(L, n, dlta[k]);
    }
}
```

2.交换排序

冒泡排序：

```
void BubbleSort(SqList &L, int n)
{
    for(int i=0; i<n-1; ++i)
    {
        for(int j=0; j<n-i-1; ++j)
        {
            if(L[j] > L[j+1])
            {
                Swap(&L[j],&L[j+1]);
            }
        }
    }
}
```

快速排序：

是对冒泡排序的一种改进。基本思想是通过一趟排序将排待记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两个部分继续进行排序，以达到整个序列有序。取任意一个记录作为枢轴，将比它小的记录放在左边，比它大的放在右边，由此可以该“枢轴”记录最后所落的位置 i 做分界线，将序列分为两个子序列。

```
枢轴
int Partition(SqList &L, int low, int high)
{
    T pk = L[low];
    while(low < high)
    {
        while(low<high && L[high]>=pk)
            high--;
        L[low] = L[high];   //交换高低位的记录
        while(low<high && L[low]<pk)
            low++;
        L[high] = L[low];
    }
    L[low] = pk;
    return low;
}
```

```
void QuickSort(SqList &L, int low, int high)
{
    if(low < high)
    {
        int pkloc = Partition(L,low,high);
        QuickSort(L,low,pkloc-1);
        QuickSort(L,pkloc+1,high);
    }
}
```

3. 选择排序

简单选择排序：

```
void Swap(T *a, T *b)
{
    T tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
//找到最小值，并返回下标
int SelectMinKey(SqList &L, int k, int n)
{
    T minval = L[k];
    int pos = k;
```

```
    for(int i=k+1; i<n; ++i)
    {
        if(L[i] < minval)
        {
            minval = L[i];
            pos = i;
        }
    }
    return pos;
}
```

```
void SelectSort(SqList &L, int n)
{
    for(int i=0; i<n-1; ++i)
    {
        int j = SelectMinKey(L, i, n);
        if(j != i)
        {
            Swap(&L[j],&L[i]);
        }
    }
}
```

堆排序：

```
从指定结点向下调整大小
void siftDown(T heap[], int n, int p)
{
    int i = p;
    int j = 2*i+1; //
    while(j < n)
    {
        if(j<n-1 && heap[j]>heap[j+1])
            j++;
        if(heap[i] <= heap[j])
            break;
        else
        {
            Swap(&heap[i], &heap[j]);
            i = j;
            j = 2*i+1;
        }
    }
}
```

```
T RemoveMinKey(T heap[], int n)
{
    T key = heap[0];
    heap[0] = heap[n];
    siftDown(heap,n,0);
    return key;
}
```

```
void HeapSort(SqList &L, int n)
{
    T *heap = (T *)malloc(sizeof(T) * n);
    assert(heap != NULL);
    for(int i=0; i<n; ++i)
    {
        heap[i] = L[i];
    }

    int curpos = n/2-1;       //可求得二叉树下最左边一个分支的结点
    while(curpos >= 0)
    {
        siftDown(heap, n, curpos);
        curpos--;
    }

    for(i=0; i<n; ++i)
    {
        L[i] = RemoveMinKey(heap, n-i-1);
    }

    free(heap);
    heap = NULL;
}
```

4.归并排序

```
void Merge(SqList &L, SqList &TP, int left, int mid, int right)
{
    for(int i=left; i<=right; ++i)
    {
        TP[i] = L[i];
    }
    int s1 = left;
    int s2 = mid+1;
    int k = left;
    while(s1<=mid && s2<=right)
    {
```

```
        if(TP[s1] <= TP[s2])
        {
            L[k++] = TP[s1++];
        }
        else
        {
            L[k++] = TP[s2++];
        }
    }
    while(s1 <= mid)
    {
        L[k++] = TP[s1++];
    }
    while(s2 <= right)
    {
        L[k++] = TP[s2++];
    }
}

void MergeSort(SqList &L, SqList &TP, int left, int right)
{
    if(left >= right)
        return;
    int mid = (left+right)/2;
    MergeSort(L,TP,left,mid);
    MergeSort(L,TP,mid+1,right);
    Merge(L,TP,left,mid,right);
}
```

5. 基数排序

```
#include"Sort.h"
#include"SList.h"

int getkey(T value, int k)
{
    int key;
    while(k >= 0)
    {
        key = value % 10;
        value /= 10;
        k--;
    }
    return key;
}
```

分配

```
void Distribute(SqList &L, int n, List (&lt)[10], int k)
{
    for(int i=0; i<10; ++i)
    {
        clear(&lt[i]);
    }
    int key;
    for(i=0; i<n; ++i)
    {
        key = getkey(L[i], k);
        push_back(&lt[key],L[i]);
    }
}
```

收集

```
void Collect(SqList &L, List(&lt)[10])
{
    int k = 0;
    for(int i=0; i<10; ++i)
    {
        Node *p = lt[i].first->next;
        while(p != NULL)
        {
            L[k++] = p->data;
            p = p->next;
        }
    }
}
```

```
void RadixSort(SqList &L, int n)
{
    List list[10];
    for(int i=0; i<10; ++i)
    {
        InitList(&list[i]);
    }

    for(i = 0; i<3; ++i)// 5  0  5
    {
        Distribute(L, n, list, i);
        Collect(L,list);
    }

}
```