教师签名	王铁建	
验证性实验成绩		
1.线性表	2.二叉树	3.查找与排序

重庆交通大学

信息科学与工程学院

实验报告

实验名称 1.线性表实验

课程名称 算法与数据结构

专业班级 电子信息类 2105 班

学　　号 632107030505

姓　　名 费锐

指导教师 王铁建

2023 年 4 月

一、实验内容及算法原理

1、顺序表的各种算法实现

建立顺序表，实现相关的操作：输出、插入、删除、查找等功能。

1.1、建立顺序表

a、将数组 a 中的每个元素依次放入到顺序表中，并将 n 赋给顺序表的长度域

1.2、查找操作

a、查找第一个值与 e 相等的元素的下标

b、找到返回元素的下标；若元素不存在，则返回值为 0

1.3、插入操作

a、在顺序表的第 i 个位置上插入新元素 e

b、如果 i 值正确，则将原来第 i 个元素及以后元素均后移一个位置，并从最后一个元素开始移动

c、顺序表长度加 1

1.4、删除操作

a、删除第 i 个元素

b、如果 i 值正确，则将第 i 个元素以后的元素均向前移动一个位置，并从第 i+1 个元素开始移动

c、顺序表长度减 1

1.5、初始化顺序表

a、分配存储空间

b、长度置 0

1.6、销毁顺序表：释放内存空间

1.7、判断顺序表是否为空：判断 L->length==0 是否成立

1.8、求顺序表的长度：返回 L->length

1.9、输出顺序表：和数组输出一样 for 循环依次输出

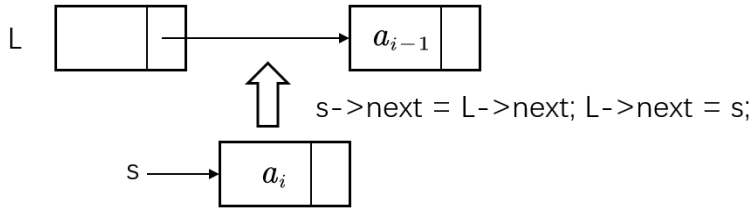
1.10、求某个数据元素值：直接寻址

2、链表的各种算法实现

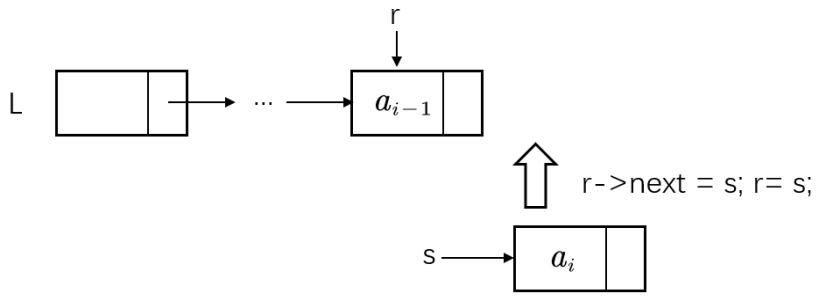
建立单链表和双链表，实现相关的操作：输出、插入、删除、查找等功能。

2.1、建立单链表

a、头插法：每次将新结点插入到头结点之后



b、尾插法：每次将新结点插入到链表的表尾



2.2、查找操作

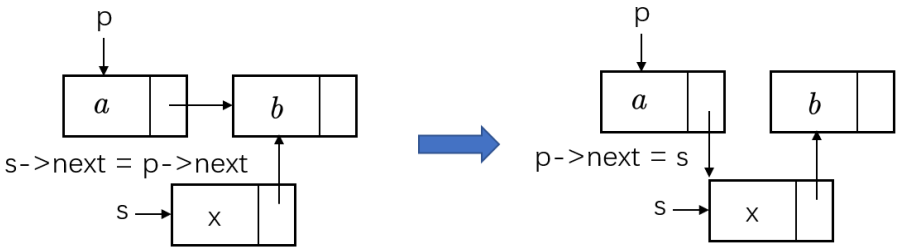
a、查找第一个值与 e 相等的元素的结点

b、找到返回逻辑符号；若元素不存在，则返回值为 0

2.3、单链表插入操作

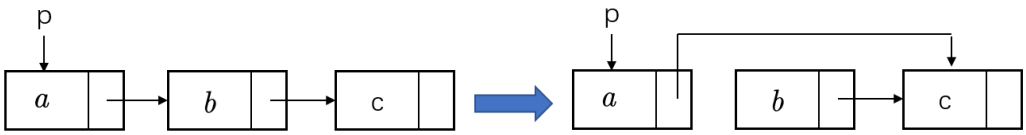
a、先找到第 i-1 个结点，p 指向它，若存在这样的结点，将值为 e 的结点

（s 指向它）插入到 p 所指的结点后面



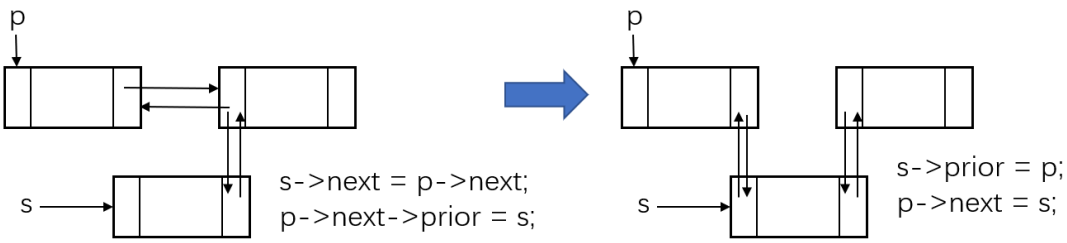
2.4、单链表删除操作

a、先找到第 i-1 个结点，p 指向它，若存在这样的结点，且也存在后继结点（q 指向它），则删除 q 所指结点



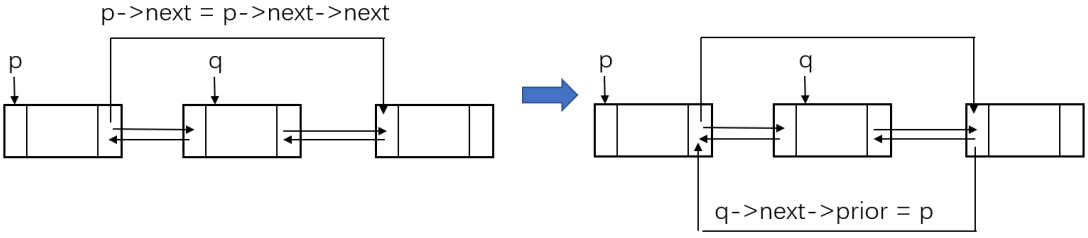
2.5、双链表插入操作

a、先找到第 i-1 个结点，p 指向它，然后在 p 所指结点之后插入一个新结点



2.6、双链表删除操作

a、先找到第 i-1 个结点，p 指向它，然后删除结点 p 的后继结点



2.7、初始化链表：头结点 next 域为空

2.8、销毁链表：逐一释放全部结点的空间

2.9、判断链表是否为空：判断 L->next == NULL 是否成立

2.10、求链表长度

a、让 p 指向头结点，n 用来累计数据结点的个数

b、p 不为空时，n++，p=p->next

2.11、输出链表：从头结点遍历

2.12、求某个数据元素值：从头开始找到第 i 个结点，若存在第 i 个数据结点，则将其 data 域赋给变量 e

3、栈的各种算法实现

建立顺序栈和链栈，实现相关的操作：进栈、出栈、判断栈是否为空等功能。

3.1、顺序栈进栈操作

a、先将栈顶指针加 1

b、然后在该位置上插入元素 e

3.2、顺序栈出栈操作

a、先将栈顶元素赋给 e

b、然后将栈顶指针减 1

3.3、顺序栈判断栈是否为空：判断条件 s->top == -1 是否成立

3.4、顺序栈初始化：s->top = -1

3.5、销毁顺序栈：释放内存空间

3.6、顺序栈取栈顶元素：data[s->top]

3.7、链栈进栈操作

a、先新建一个结点，用于存放元素 e（p 指向它）

b、然后将其作为插入到头结点之后作为新的首结点

3.8、链栈出栈操作

a、先将首结点数据域赋给 e

b、然后将其删除

3.9、链栈判断栈是否为空：判断条件 s->next == NULL 是否成立

3.10、链栈初始化：s->next = NULL

3.11、销毁链栈：逐一释放结点（同单链表）

3.12、链栈取栈顶元素：s->next->data

4、队列的各种算法实现

建立顺序队和链队，实现相关的操作：进队列、出队列、判断队列是否为空等功能。

4.1、顺序队进队列操作

a、先将队尾指针 rear 加 1

b、然后将元素 e 插入到该位置

4.2、顺序队出队列操作

a、先将队头指针 front 加 1

b、然后将该位置的元素值赋给 e

4.3、顺序队判断队列是否为空：判断 q->front == q->rear 是否成立

4.4、顺序队初始化：q->front = q->rear = -1

4.5、销毁顺序队：释放空间

4.6、链队进队列操作

a、先新建一个结点用于存放元素 e（p 指向它）

b、若原队列为空，则将链队结点的两个域均指向结点 p，否则将结点 p 链接到单链表的末尾，并让链队结点的 rear 域指向它

4.7、链队出队列操作

a、将首结点的值域赋给 e，并删除它

b、若只有一个结点，则需要链队结点的两个域均置为 NULL

4.8、链队判断队列是否为空：q->rear == NULL;

4.9、链队初始化：q->front = q->rear = NULL

4.10、销毁链队：逐一销毁所有结点

二、核心代码

1、顺序表

1.1、结构体定义

```
typedef int ElemType;
typedef struct
{   ElemType data[MaxSize];     //顺序表元素
    int length;                 //顺序表长度
}SqList;
ElemType a[MaxSize];
```

1.2、建立顺序表

```
void CreateList(SqList *&L,ElemType a[],int n)
{
    L=(SqList *)malloc(sizeof(SqList));//分配空间

    for (int i=0;i<n;i++)
        L->data[i]=a[i];//将元素存进顺序表

    L->length=n;        //将长度存进顺序表
}
```

1.3、各种算法实现

```
//初始化顺序表
```

```
void InitList(SqList *&L)
{
    L=(SqList *)malloc(sizeof(SqList)); //分配空间

    L->length=0;  //将长度赋值为0
}

//销毁顺序表
void DestroyList(SqList *&L)
{
    free(L);
}

//判断顺序表是否为空
bool ListEmpty(SqList *L)
{
    return(L->length==0); //若条件成立则为空
}

//计算顺序表长度
int ListLength(SqList *L)
{
    return(L->length);
}

//输出顺序表
void DispList(SqList *L)
{
    for (int i=0;i<L->length;i++)
        printf("%d ",L->data[i]);
    printf("\n");
}

//求顺序表中某个位置上的元素值
bool GetElem(SqList *L,int i,ElemType &e)
{
    if (i<1 || i>L->length) //判断是否出界
        return false;

    e=L->data[i-1];
    return true;
}
//按元素值查找元素位置
int LocateElem(SqList *L, ElemType e)
```

```
{
    int i=0;
    while (i<L->length && L->data[i]!=e) i++;

    if (i>=L->length)
        return 0;
    else
        return i+1;
}
//插入元素
bool ListInsert(SqList *&L,int i,ElemType e)
{
    int j;
    if (i<1 || i>L->length+1)
        return false;

    i--;                        //将顺序表位序转化为 elem 下标
    for (j=L->length;j>i;j--)     //将 data[i]及后面元素后移一个位置
        L->data[j]=L->data[j-1];

    L->data[i]=e;
    L->length++;              //顺序表长度增 1
    return true;
}

//删除元素
bool ListDelete(SqList *&L,int i,ElemType &e)
{
    int j;
    if (i<1 || i>L->length)
        return false;

    i--;                        //将顺序表位序转化为 elem 下标
    e=L->data[i];
    for (j=i;j<L->length-1;j++)   //将 data[i]之后的元素前移一个位置
        L->data[j]=L->data[j+1];
    L->length--;              //顺序表长度减 1
    return true;
}
```

2、单链表

2.1、结构体定义

```
typedef int ElemType;
```

```
typedef struct LNode        //定义单链表结点类型
{
    ElemType data;
    struct LNode *next;
} LinkNode;
ElemType a[MaxSize];
```

2.2、建立单链表

```
//头插法建立循环单链表
void CreateListF(LinkNode *&L,ElemType a[],int n)
{
    LinkNode *s;int i;
    L=(LinkNode *)malloc(sizeof(LinkNode));     //创建头结点
    L->next=NULL;
    for (i=0;i<n;i++)
    {
        s=(LinkNode *)malloc(sizeof(LinkNode));//创建新结点
        s->data=a[i];
        s->next=L->next;          //将结点 s 插在原开始结点之前,头结点之后
        L->next=s;
    }
    s=L->next;
    while (s->next!=NULL)             //查找尾结点,由 s 指向它
        s=s->next;
    s->next=L;                        //尾结点 next 域指向头结点

}

//尾插法建立循环单链表
void CreateListR(LinkNode *&L,ElemType a[],int n)
{
    LinkNode *s,*r;int i;
    L=(LinkNode *)malloc(sizeof(LinkNode));     //创建头结点
    L->next=NULL;
    r=L;                      //r 始终指向终端结点,开始时指向头结点
    for (i=0;i<n;i++)
    {
        s=(LinkNode *)malloc(sizeof(LinkNode));//创建新结点
        s->data=a[i];
        r->next=s;            //将结点 s 插入结点 r 之后
        r=s;
    }
    r->next=L;                //尾结点 next 域指向头结点
}
```

2.3、各种算法实现

```
1.  //初始化线性表
2.  void InitList(LinkNode *&L)
3.  {
4.      L=(LinkNode *)malloc(sizeof(LinkNode)); //创建头结点
5.      L->next=L;
6.  }
7.
8.  //销毁线性表
9.  void DestroyList(LinkNode *&L)
10. {
11.     LinkNode *p=L,*q=p->next;
12.     while (q!=L)
13.     {
14.         free(p);
15.         p=q;
16.         q=p->next;
17.     }
18.     free(p);
19. }
20.
21. //判断线性表是否为空
22. bool ListEmpty(LinkNode *L)
23. {
24.     return(L->next==NULL);
25. }
26.
27. //线性表长度
28. int ListLength(LinkNode *L)
29. {
30.     LinkNode *p=L;int i=0;
31.     while (p->next!=L)
32.     {
33.         i++;
34.         p=p->next;
35.     }
36.     return(i);
37. }
38.
39. //输出线性表
40. void DispList(LinkNode *L)
41. {
42.     LinkNode *p=L->next;
```

```
    while (p!=L)
    {
        printf("%d ",p->data);
        p=p->next;
    }
    printf("\n");
}

//求线性表中某个数据元素值
bool GetElem(LinkNode *L,int i,ElemType &e)
{
    int j=0;
    LinkNode *p;
    if (L->next!=L)      //单链表不为空表时
    {
        if (i==1)
        {
            e=L->next->data;
            return true;
        }
        else            //i 不为 1 时
        {
            p=L->next;
            while (j<i-1 && p!=L)
            {
                j++;
                p=p->next;
            }
            if (p==L)
                return false;
            else
            {
                e=p->data;
                return true;
            }
        }
    }
    else                //单链表为空表时
        return false;
}

//按元素值查找
int LocateElem(LinkNode *L,ElemType e)
{
```

```
87.     LinkNode *p=L->next;
88.     int n=1;
89.     while (p!=L && p->data!=e)
90.     {
91.         p=p->next;
92.         n++;
93.     }
94.     if (p==L)
95.         return(0);
96.     else
97.         return(n);
98. }
99.
100.  //插入数据元素
101.  bool ListInsert(LinkNode *&L,int i,ElemType e)
102.  {
103.      int j=0;
104.      LinkNode *p=L,*s;
105.      if (p->next==L || i==1)       //原单链表为空表或 i==1 时
106.      {
107.          s=(LinkNode *)malloc(sizeof(LinkNode)); //创建新结点 s
108.          s->data=e;
109.          s->next=p->next;       //将结点 s 插入到结点 p 之后
110.          p->next=s;
111.          return true;
112.      }
113.      else
114.      {
115.          p=L->next;
116.          while (j<i-2 && p!=L)
117.          {
118.              j++;
119.              p=p->next;
120.          }
121.          if (p==L)                //未找到第 i-1 个结点
122.              return false;
123.          else                     //找到第 i-1 个结点 p
124.          {
125.              s=(LinkNode *)malloc(sizeof(LinkNode)); //创建新结点 s
126.              s->data=e;
127.              s->next=p->next;                    //将结点 s 插入到结点 p
    之后
128.              p->next=s;
129.              return true;
```

```
        }
    }
}

//删除数据元素
bool ListDelete(LinkNode *&L,int i,ElemType &e)
{
    int j=0;
    LinkNode *p=L,*q;
    if (p->next!=L)                    //原单链表不为空表时
    {
        if (i==1)                      //i==1 时
        {
            q=L->next;                 //删除第 1 个结点
            e=q->data;
            L->next=q->next;
            free(q);
            return true;
        }
        else                           //i 不为 1 时
        {
            p=L->next;
            while (j<i-2 && p!=L)
            {
                j++;
                p=p->next;
            }
            if (p==L)                  //未找到第 i-1 个结点
                return false;
            else                       //找到第 i-1 个结点 p
            {
                q=p->next;             //q 指向要删除的结点
                e=q->data;
                p->next=q->next;   //从单链表中删除 q 结点
                free(q);               //释放 q 结点
                return true;
            }
        }
    }
    else return false;
}
```

3、双链表

3.1、结构体定义

```
typedef int ElemType;
typedef struct DNode        //定义双链表结点类型
{
    ElemType data;
    struct DNode *prior;    //指向前驱结点
    struct DNode *next;     //指向后继结点
} DLinkNode;
ElemType a[MaxSize];
```

3.2、建立双链表

```
//头插法建双链表
void CreateListF(DLinkNode *&L,ElemType a[],int n)
{
    DLinkNode *s;
    L=(DLinkNode *)malloc(sizeof(DLinkNode));   //创建头结点
    L->prior=L->next=NULL;
    for (int i=0;i<n;i++)
    {
        s=(DLinkNode *)malloc(sizeof(DLinkNode));//创建新结点
        s->data=a[i];
        s->next=L->next;          //将结点 s 插在原开始结点之前,头结点之后
        if (L->next!=NULL) L->next->prior=s;
        L->next=s;s->prior=L;
    }
}

//尾插法建双链表
void CreateListR(DLinkNode *&L,ElemType a[],int n)
{
    DLinkNode *s,*r;
    L=(DLinkNode *)malloc(sizeof(DLinkNode));   //创建头结点
    L->prior=L->next=NULL;
    r=L;                      //r 始终指向终端结点,开始时指向头结点
    for (int i=0;i<n;i++)
    {
        s=(DLinkNode *)malloc(sizeof(DLinkNode));//创建新结点
        s->data=a[i];
        r->next=s;s->prior=r; //将结点 s 插入结点 r 之后
        r=s;
    }
    r->next=NULL;                //尾结点 next 域置为 NULL
}
```

3.3、各种算法实现（由于与单链表相差不多，不再赘述）

4、顺序栈

4.1、结构体定义

```
typedef char ElemType;
typedef struct
{
    ElemType data[MaxSize];
    int top;                //栈指针
} SqStack;                  //顺序栈类型
```

4.2、各种算法实现

```
//初始化栈
void InitStack(SqStack *&s)
{
    s=(SqStack *)malloc(sizeof(SqStack));
    s->top=-1;
}

//销毁栈
void DestroyStack(SqStack *&s)
{
    free(s);
}

//判断栈是否为空
bool StackEmpty(SqStack *s)
{
    return(s->top==-1);
}

//进栈
bool Push(SqStack *&s,ElemType e)
{
    if (s->top==MaxSize-1)    //栈满的情况，即栈上溢出
        return false;
    s->top++;
    s->data[s->top]=e;
    return true;
}

//出栈
bool Pop(SqStack *&s,ElemType &e)
{
```

```
    if (s->top==-1)       //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top];
    s->top--;
    return true;
}

//取栈顶元素
bool GetTop(SqStack *s,ElemType &e)
{
    if (s->top==-1)       //栈为空的情况，即栈下溢出
        return false;
    e=s->data[s->top];
    return true;
}
```

5、链栈

5.1、结构体定义

```
typedef char ElemType;
typedef struct linknode
{
    ElemType data;              //数据域
    struct linknode *next;      //指针域
} LinkStNode;                   //链栈类型
```

5.2、各种算法实现

```
//初始化栈
void InitStack(LinkStNode *&s)
{
    s=(LinkStNode *)malloc(sizeof(LinkStNode));
    s->next=NULL;
}

//销毁栈
void DestroyStack(LinkStNode *&s)
{
    LinkStNode *p=s->next;
    while (p!=NULL)
    {
        free(s);
        s=p;
        p=p->next;
    }
```

```
    free(s);    //s 指向尾结点,释放其空间
}

//判断栈是否为空
bool StackEmpty(LinkStNode *s)
{
    return(s->next==NULL);
}

//进栈
void Push(LinkStNode *&s,ElemType e)
{   LinkStNode *p;
    p=(LinkStNode *)malloc(sizeof(LinkStNode));
    p->data=e;                //新建元素 e 对应的结点 p
    p->next=s->next;      //插入 p 结点作为开始结点
    s->next=p;
}

//出栈
bool Pop(LinkStNode *&s,ElemType &e)
{   LinkStNode *p;
    if (s->next==NULL)        //栈空的情况
        return false;
    p=s->next;                //p 指向开始结点
    e=p->data;
    s->next=p->next;      //删除 p 结点
    free(p);                //释放 p 结点
    return true;
}

//取栈顶元素
bool GetTop(LinkStNode *s,ElemType &e)
{   if (s->next==NULL)        //栈空的情况
        return false;
    e=s->next->data;
    return true;
}
```

6、顺序队

6.1、结构体定义

```
typedef char ElemType;
typedef struct
{
```

```
    ElemType data[MaxSize];
    int front,rear;                         //队头和队尾指针
} SqQueue;
```

6.2、各种算法实现

```
//初始化队列
void InitQueue(SqQueue *&q)
{   q=(SqQueue *)malloc (sizeof(SqQueue));
    q->front=q->rear=-1;
}

//销毁队列
void DestroyQueue(SqQueue *&q)          //销毁队列
{
    free(q);
}

//判断队列是否为空
bool QueueEmpty(SqQueue *q)             //判断队列是否为空
{
    return(q->front==q->rear);
}

//进队列
bool enQueue(SqQueue *&q,ElemType e)    //进队
{   if (q->rear==MaxSize-1)              //队满上溢出
        return false;                   //返回假
    q->rear++;                           //队尾增 1
    q->data[q->rear]=e;                   //rear 位置插入元素 e
    return true;                        //返回真
}

//出队列
bool deQueue(SqQueue *&q,ElemType &e)   //出队
{   if (q->front==q->rear)                //队空下溢出
        return false;
    q->front++;
    e=q->data[q->front];
    return true;
}
```

7、链队

7.1、结构体定义

```
typedef char ElemType;
typedef struct DataNode
{
    ElemType data;
    struct DataNode *next;
} DataNode;             //链队数据结点类型
typedef struct
{
    DataNode *front;
    DataNode *rear;
} LinkQuNode;           //链队类型
```

7.2、各种算法实现

```
//初始化队列
void InitQueue(LinkQuNode *&q)
{
    q=(LinkQuNode *)malloc(sizeof(LinkQuNode));
    q->front=q->rear=NULL;
}

//销毁队列
void DestroyQueue(LinkQuNode *&q)
{
    DataNode *p=q->front,*r;//p 指向队头数据结点
    if (p!=NULL)            //释放数据结点占用空间
    {   r=p->next;
        while (r!=NULL)
        {   free(p);
            p=r;r=p->next;
        }
    }
    free(p);
    free(q);                //释放链队结点占用空间
}

//判断队列是否为空
bool QueueEmpty(LinkQuNode *q)
{
    return(q->rear==NULL);
}

//进队列
void enQueue(LinkQuNode *&q,ElemType e)
{   DataNode *p;
```

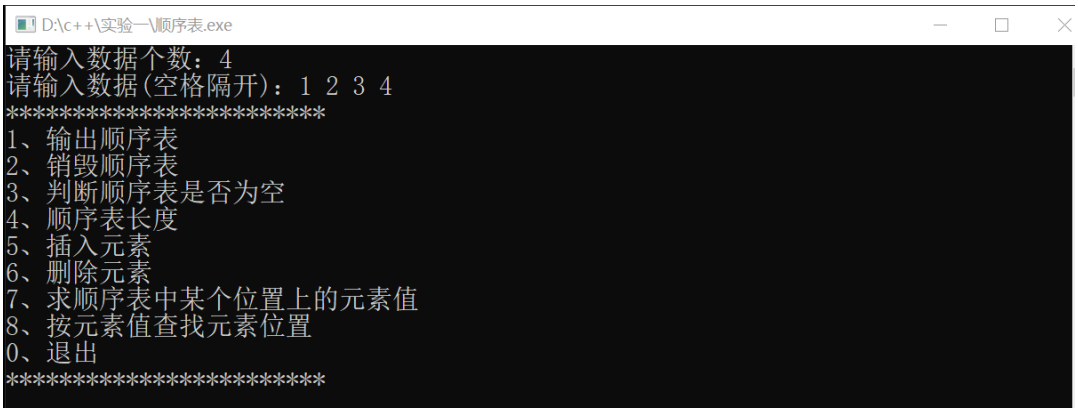
```
    p=(DataNode *)malloc(sizeof(DataNode));
    p->data=e;
    p->next=NULL;
    if (q->rear==NULL)          //若链队为空,则新结点是队首结点又是队尾结点
        q->front=q->rear=p;
    else
    {   q->rear->next=p;  //将 p 结点链到队尾,并将 rear 指向它
        q->rear=p;
    }
}

//出队列
bool deQueue(LinkQuNode *&q,ElemType &e)
{   DataNode *t;
    if (q->rear==NULL)          //队列为空
        return false;
    t=q->front;                 //t 指向第一个数据结点
    if (q->front==q->rear)  //队列中只有一个结点时
        q->front=q->rear=NULL;
    else                        //队列中有多个结点时
        q->front=q->front->next;
    e=t->data;
    free(t);
    return true;
}
```

三、实验结果及分析

1、顺序表

1.1、建立顺序表



1.2、各种算法实现

先输入 1：输出顺序表

```
1
1 2 3 4
```

输入 3：判断是否为空

```
3
表不为空
```

输入 4：求顺序表长度

```
4
顺序表长度为:4
```

输入 5：插入元素（在第 3 位插入元素 5）

```
5
请输入插入元素位置和元素值
3 5
插入成功!
1
1 2 5 3 4
```

输入 6：删除元素（删除第 4 位）

```
6
请输入删除元素位置
4
删除成功!
1
1 2 5 4
```

输入 7：求某个位置上的元素值（第 2 位）

```
7
请输入想要查看的元素位置:
2
该元素值为:2
```

输入 8：查找元素位置

```
8
请输入想要查看的元素:
5
该元素在顺序表中的下标为:3
```

2、链表

2.1、建立链表

2.2、各种算法实现

输入 1：输出链表

```
1
4 3 2 1
```

输入 3：判断链表是否为空

```
3
表不为空
```

输入 4：求链表长度

```
4
顺序表长度为:4
```

输入 5：插入元素（在第 2 位插入元素 6）

```
5
请输入插入元素位置和元素值
2 6
插入成功!
1
4 6 3 2 1
```

输入 6：删除元素（删除第 3 位）

```
6
请输入删除元素位置
3
删除成功!
1
4 6 2 1
```

输入 7：求某个位置上的元素（第 2 位）

```
7
请输入想要查看的元素位置:
2
该元素值为:6
```

输入 8：查找元素（第 4 位）

```
8
请输入想要查看的元素:
1
该元素在顺序表中的下标为:4
```

3、栈

```
D:\c++\实验一\顺序栈.exe
**************************
1、进栈
2、出栈
3、判断栈是否为空
4、销毁栈
5、取栈顶元素
0、退出
**************************
```

输入 3：判断栈是否为空

```
3
栈为空
```

输入 1：进栈（4→3→2→1）

```
1
请输入进栈的元素值:
4
进栈成功!
1
请输入进栈的元素值:
3
进栈成功!
1
请输入进栈的元素值:
2
进栈成功!
1
请输入进栈的元素值:
1
进栈成功!
```

输入 5：取栈顶元素

```
5
1
```

输入 2：出栈

```
2
出栈成功!
5
2
```

4、队列

输入 3：判断队列是否为空

```
3
队列为空
```

输入 1：进队

```
1
请输入进队的元素值：
3
进队成功!
```

输入 2：出队

```
2
出队成功!
```

重庆交通大学

信息科学与工程学院

实验报告

实验名称 2.二叉树实验

课程名称 算法与数据结构

专业班级 电子信息类2105班

学　　号 632107030505

姓　　名 费锐

指导教师 王铁建

2023 年 4 月

一、实验内容及算法原理

1、二叉树的构造

1.1、利用一颗二叉树的中序序列和先序序列构造二叉树

1.2、利用一颗二叉树的中序序列和后序序列构造二叉树

2、二叉树的四种遍历算法实现

1.1、先序遍历

a、访问根结点

b、先序遍历左子树

c、先序遍历右子树

1.2、中序遍历

a、中序遍历左子树

b、访问根结点

c、中序遍历右子树

1.3、后序遍历

a、后序遍历左子树

b、后序遍历右子树

c、访问根结点

1.4、层次遍历

a、访问根结点

b、从左到右访问第 2 层所有结点

c、从左到右访问第 3 层以至于第 h 层所有结点

二、核心代码

1、二叉树的构造

1.1、结构体定义

```
typedef char ElemType;
typedef struct node
{
    ElemType data;          //数据元素
    struct node *lchild;    //指向左孩子节点
    struct node *rchild;    //指向右孩子节点
} BTNode;
```

1.2、中序序列和先序序列构造二叉树

```
BTNode *CreateBT1(char *pre,char *in,int n)
/*pre 存放先序序列,in 存放中序序列,n 为二叉树节点个数,
本算法执行后返回构造的二叉链的根节点指针*/
{
    BTNode *s;
    char *p;
    int k;
    if (n<=0) return NULL;
    s=(BTNode *)malloc(sizeof(BTNode));     //创建二叉树节点 s
    s->data=*pre;
    for (p=in;p<in+n;p++)                    //在中序序列中找等于*ppos 的位置 k
        if (*p==*pre)                        //pre 指向根节点
            break;                           //在 in 中找到后退出循环
    k=p-in;                                  //确定根节点在 in 中的位置
    s->lchild=CreateBT1(pre+1,in,k);     //递归构造左子树
    s->rchild=CreateBT1(pre+k+1,p+1,n-k-1); //递归构造右子树
    return s;
}
```

1.3、中序序列和后序序列构造二叉树

```
BTNode *CreateBT2(char *post,char *in,int n)
/*post 存放后序序列,in 存放中序序列,n 为二叉树节点个数,
本算法执行后返回构造的二叉链的根节点指针*/
{
    BTNode *s;
    char r,*p;
    int k;
    if (n<=0) return NULL;
    r=*(post+n-1);                           //根节点值
    s=(BTNode *)malloc(sizeof(BTNode));     //创建二叉树节点 s
```

```
    s->data=r;
    for (p=in;p<in+n;p++)                    //在 in 中查找根节点
        if (*p==r)
            break;
    k=p-in;                                  //k 为根节点在 in 中的下标
    s->lchild=CreateBT2(post,in,k);          //递归构造左子树
    s->rchild=CreateBT2(post+k,p+1,n-k-1);   //递归构造右子树
    return s;
}
```

2、先序遍历

2.1、递归算法

```
void PreOrder(BTNode *b)        //先序遍历的递归算法
{
    if (b!=NULL)
    {
        printf("%c ",b->data);   //访问根结点
        PreOrder(b->lchild); //先序遍历左子树
        PreOrder(b->rchild); //先序遍历右子树
    }
}
```

2.2、非递归算法（用到栈，所以需要实验一的栈实现函数）

```
void PreOrder1(BTNode *b)           //先序非递归遍历算法 1
{
    BTNode *p;
    SqStack *st;                    //定义一个顺序栈指针 st
    InitStack(st);                  //初始化栈 st
    Push(st,b);                 //根节点进栈
    while (!StackEmpty(st))     //栈不为空时循环
    {
        Pop(st,p);              //退栈节点 p 并访问它
        printf("%c ",p->data);   //访问节点 p
        if (p->rchild!=NULL) //有右孩子时将其进栈
            Push(st,p->rchild);
        if (p->lchild!=NULL) //有左孩子时将其进栈
            Push(st,p->lchild);
    }
    printf("\n");
    DestroyStack(st);               //销毁栈
}
void PreOrder2(BTNode *b)           //先序非递归遍历算法 2
{
```

```
    BTNode *p;
    SqStack *st;                    //定义一个顺序栈指针 st
    InitStack(st);                  //初始化栈 st
    p=b;
    while (!StackEmpty(st) || p!=NULL)
    {
        while (p!=NULL)             //访问节点 p 及其所有左下节点并进栈
        {
            printf("%c ",p->data);   //访问节点 p
            Push(st,p);             //节点 p 进栈
            p=p->lchild;          //移动到左孩子
        }
        if (!StackEmpty(st))        //若栈不空
        {
            Pop(st,p);              //出栈节点 p
            p=p->rchild;          //转向处理其右子树
        }
    }
    printf("\n");
    DestroyStack(st);               //销毁栈
}
```

3、中序遍历

3.1、递归算法

```
void InOrder(BTNode *b)         //中序遍历的递归算法
{
    if (b!=NULL)
    {
        InOrder(b->lchild);      //中序遍历左子树
        printf("%c ",b->data);   //访问根结点
        InOrder(b->rchild);      //中序遍历右子树
    }
}
```

3.2、非递归算法（用到栈，所以需要实验一的栈实现函数）

```
void InOrder1(BTNode *b)                //中序非递归遍历算法
{
    BTNode *p;
    SqStack *st;                        //定义一个顺序栈指针 st
    InitStack(st);                      //初始化栈 st
    if (b!=NULL)
    {
        p=b;
```

```
        while (!StackEmpty(st) || p!=NULL)
        {
            while (p!=NULL)             //扫描节点 p 的所有左下节点并进栈
            {
                Push(st,p);             //节点 p 进栈
                p=p->lchild;          //移动到左孩子
            }
            if (!StackEmpty(st))        //若栈不空
            {
                Pop(st,p);              //出栈节点 p
                printf("%c ",p->data);   //访问节点 p
                p=p->rchild;          //转向处理其右子树
            }
        }
        printf("\n");
    }
    DestroyStack(st);               //销毁栈
}
```

4、后序遍历

4.1、递归算法

```
void PostOrder(BTNode *b)       //后序遍历的递归算法
{
    if (b!=NULL)
    {
        PostOrder(b->lchild);    //后序遍历左子树
        PostOrder(b->rchild);    //后序遍历右子树
        printf("%c ",b->data);   //访问根结点
    }
}
```

4.2、非递归算法（用到栈，所以需要实验一的栈实现函数）

```
void PostOrder1(BTNode *b)              //后序非递归遍历算法
{
    BTNode *p,*r;
    bool flag;
    SqStack *st;                        //定义一个顺序栈指针 st
    InitStack(st);                      //初始化栈 st
    p=b;
    do
    {
        while (p!=NULL)                 //扫描节点 p 的所有左下节点并进栈
        {
```

```
            Push(st,p);                    //节点 p 进栈
            p=p->lchild;                 //移动到左孩子
        }
        r=NULL;                            //r 指向刚刚访问的节点，初始时为空
        flag=true;                         //flag 为真表示正在处理栈顶节点
        while (!StackEmpty(st) && flag)
        {
            GetTop(st,p);                  //取出当前的栈顶节点 p
            if (p->rchild==r)               //若节点 p 的右孩子为空或者为刚刚访问过的节点
            {
                printf("%c ",p->data);     //访问节点 p
                Pop(st,p);
                r=p;                       //r 指向刚访问过的节点
            }
            else
            {
                p=p->rchild;           //转向处理其右子树
                flag=false;                //表示当前不是处理栈顶节点
            }
        }
    } while (!StackEmpty(st));             //栈不空循环
    printf("\n");
    DestroyStack(st);                  //销毁栈
}
```

5、层次遍历（队列实现）

```
void LevelOrder(BTNode *b)
{
    BTNode *p;
    SqQueue *qu;
    InitQueue(qu);                     //初始化队列
    enQueue(qu,b);                     //根结点指针进入队列
    while (!QueueEmpty(qu))            //队不为空循环
    {
        deQueue(qu,p);                 //出队节点 p
        printf("%c ",p->data);          //访问节点 p
        if (p->lchild!=NULL)      //有左孩子时将其进队
            enQueue(qu,p->lchild);
        if (p->rchild!=NULL)      //有右孩子时将其进队
            enQueue(qu,p->rchild);
    }
}
```

三、实验结果及分析

1、二叉树的构造

假设一颗二叉树的先序序列、中序序列和后序序列分别如下所示，

```
1.  ElemType pre[]="ABDGCEF",in[]="DGBAECF",post[]="GDBEFCA";
```

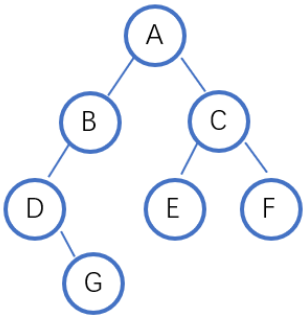
则利用先序序列和中序序列构造的二叉树如下，

```
b1:A(B(D(,G)),C(E,F))
```

则利用中序遍历和后序遍历构造的二叉树如下，

```
b2:A(B(D(,G)),C(E,F))
```

可视化后的二叉树：



2、二叉树的四种遍历算法

2.1、先序遍历、中序遍历和后序遍历

```
b:A(B(D(,G)),C(E,F))
先序遍历序列:A B D G C E F
中序遍历序列:D G B A E C F
后序遍历序列:G D B E F C A
```

2.2、层次遍历

```
b:A(B(D(,G)),C(E,F))
层次遍历序列:A B C D E F G
```

重庆交通大学

信息科学与工程学院

实验报告

实验名称 3.查找与排序实验

课程名称 算法与数据结构

专业班级 电子信息类 2105 班

学　　号 632107030505

姓　　名 费锐

指导教师 王铁建

2023 年 4 月

一、实验内容及算法原理

1、顺序查找的算法实现

从线性表的一端向另一端逐个将元素的关键字和给定的 k 值比较

2、折半查找的算法实现

2.1、前提条件：有序表

2.2、基本思路

a、确定区间中点位置 mid = (low+high)/2，将待查 k 值与 R[mid]比较

b、若 k==R[mid]，则查找成功

若 k<R[mid]，则新的查找区间在左子表 R[low , mid-1]

若 k>R[mid]，则新的查找区间在右子表 R[mid+1 , high]

3、插入排序的算法实现

3.1、直接插入排序

a、将无序区的开头元素 R[i]插入到有序区 R[0 , i-1]中的适当位置

b、先将 R[i]暂放到 temp 中，j 在有序区从后向前找，凡是关键字大于 temp 的均后移一个位置

c、若找到某个 R[j]大于等于 temp，则将 temp 放到它们后面，R[j+1]=temp

3.2、折半插入排序

a、由于有序区是有序的，可以用折半查找的方法先在 R[0 , i-1]中找到插入位置，再移动元素进行插入

3.3、希尔排序

a、先把表的全部元素分成 d_1 个组，将所有距离为 d_1 的倍数的元素放在同一个组里，各组内进行直接插入排序

b、在把表的全部元素分成 d_2（$d_2 < d_1$） 个组，重复上述操作，知道 d=1

4、交换排序的算法实现

4.1、冒泡排序

a、无序区中相邻元素减的比较和位置交换使最小的元素逐渐浮至水面

b、每趟排序都将无序区中的最小元素达到最上端

4.2、快速排序

a、在待排序的 n 个元素中任取一个元素作为基准，所有比该元素小的元素放在它的前面，所有比该元素大的元素放在它的后面

b、对分成的两个部分重复上述操作，直至每部分内只有一个元素为止

5、选择排序的算法实现

5.1、简单选择排序

a、从无序区中选出最小的元素 R[k]，将它与无序区第 1 个元素 R[i]交换

b、通过简单的两两比较方法选出无序区中最小的元素

5.2、堆排序

a、假如完全二叉树的根结点是 R[i]，它的左右子树已是大根堆，将其两个孩子的最大者与 R[i]比较，若 R[i]小，则与最大孩子交换

b、继续用上述操作构造下一级堆，直到完全变成大根堆为止

6、归并排序

将 R[0 , n-1]看成是 n 个长度为 1 的有序序列，然后两两归并，得到 n/2 个长度为 2 的有序序列，再两两归并，得到 n/4 个长度为 4 的有序序列，直到得到一个长度为 n 的有序序列

7、基数排序

分配：开始时，把 Q_0，Q_1，…，Q_{r-1} 各个队列置成空队列，然后依次考查线性表中的每一个元素 a_j，如果元素 a_j 的关键字 $k_j^i = k$，就把元素 a_j 插入到 Q_k 队列中

收集：将Q_0，Q_1，…，Q_{r-1} 各个队列中的元素依次首尾相接，得到新的元素序列，从而组成新的线性表

二、核心代码

1、顺序查找

1.1、结构体定义

```
typedef struct
{   KeyType key;        //关键字项
    InfoType data;      //其他数据项，类型为 InfoType
} RecType;              //查找元素的类型
```

1.2、算法实现

```
int SeqSearch(RecType R[],int n,KeyType k)
{
    int i=0;
    while (i<n && R[i].key!=k)   //从表头往后找
        i++;
    if (i>=n)
        return 0;
    else
        return i+1;
}

int SeqSearch1(RecType R[],int n,KeyType k)
{
    int i=0;
    R[n].key=k;
    while (R[i].key!=k) //从表头往后找
        i++;
    if (i==n)               //未找到返回 0
        return 0;
    else
        return i+1;         //找到返回逻辑序号 i+1
}
```

2、折半查找

2.1、结构体定义

```
typedef struct
{   KeyType key;        //关键字项
    InfoType data;      //其他数据项，类型为 InfoType
} RecType;              //查找元素的类型
```

2.2、算法实现

```
int BinSearch(RecType R[],int n,KeyType k) //折半查找算法
{   int low=0,high=n-1,mid;
    while (low<=high)            //当前区间存在元素时循环
    {   mid=(low+high)/2;
        if (k==R[mid].key)       //查找成功返回其逻辑序号 mid+1
            return mid+1;
        if (k<R[mid].key)        //继续在 R[low..mid-1]中查找
            high=mid-1;
        else                     //k>R[mid].key
            low=mid+1;           //继续在 R[mid+1..high]中查找
    }
    return 0;                    //未找到时返回 0（查找失败）
}
```

3、插入排序

3.1、直接插入排序

```
void InsertSort(RecType R[],int n) //对 R[0..n-1]按递增有序进行直接插入排序
{   int i, j; RecType tmp;
    for (i=1;i<n;i++)
    {
        if (R[i].key<R[i-1].key) //反序时
        {
            tmp=R[i];
            j=i-1;
            do                   //找 R[i]的插入位置
            {
                R[j+1]=R[j];     //将关键字大于 R[i].key 的记录后移
                j--;
            }  while  (j>=0 && R[j].key>tmp.key);
            R[j+1]=tmp;              //在 j+1 处插入 R[i]
        }
        printf("  i=%d: ",i); DispList(R,n);
    }
```

```
}
```

3.2、折半插入排序

```
void BinInsertSort(RecType R[],int n)
{   int i, j, low, high, mid;
    RecType tmp;
    for (i=1;i<n;i++)
    {
        if (R[i].key<R[i-1].key)     //反序时
        {
            tmp=R[i];                    //将 R[i]保存到 tmp 中
            low=0;  high=i-1;
            while (low<=high)             //在 R[low..high]中查找插入的位置
            {
                mid=(low+high)/2;        //取中间位置
                if (tmp.key<R[mid].key)
                    high=mid-1;          //插入点在左半区
                else
                    low=mid+1;           //插入点在右半区
            }                            //找位置 high
            for (j=i-1;j>=high+1;j--)     //集中进行元素后移
                R[j+1]=R[j];
            R[high+1]=tmp;               //插入 tmp
        }
        printf("  i=%d: ",i);
        DispList(R,n);
    }
}
```

3.3、希尔排序

```
void ShellSort(RecType R[],int n) //希尔排序算法
{   int i,j,d;
    RecType tmp;
    d=n/2;                   //增量置初值
    while (d>0)
    {   for (i=d;i<n;i++)     //对所有组采用直接插入排序
        {   tmp=R[i];        //对相隔 d 个位置一组采用直接插入排序
            j=i-d;
            while (j>=0 && tmp.key<R[j].key)
            {   R[j+d]=R[j];
                j=j-d;
            }
            R[j+d]=tmp;
        }
```

```
        printf("  d=%d: ",d); DispList(R,n);
        d=d/2;              //减小增量
    }
}
```

4、交换排序

4.1、冒泡排序

```
void BubbleSort(RecType R[],int n)
{
    int i,j,k;
    RecType tmp;
    for (i=0;i<n-1;i++)
    {
        for (j=n-1;j>i;j--)  //比较,找出本趟最小关键字的记录
            if (R[j].key<R[j-1].key)
            {
                tmp=R[j];  //R[j]与 R[j-1]进行交换,将最小关键字记录前移
                R[j]=R[j-1];
                R[j-1]=tmp;
            }
        printf("  i=%d: ",i);
        DispList(R,n);
    }
}
void BubbleSort1(RecType R[],int n)
{   int i,j;
    bool exchange;
    RecType tmp;
    for (i=0;i<n-1;i++)
    {   exchange=false;                 //一趟前 exchange 置为假
        for (j=n-1;j>i;j--)              //归位 R[i],循环 n-i-1 次
            if (R[j].key<R[j-1].key) //相邻两个元素反序时
            {   tmp=R[j];               //将这两个元素交换
                R[j]=R[j-1];
                R[j-1]=tmp;
                exchange=true;          //一旦有交换，exchange 置为真
            }
        printf("  i=%d: ",i);
        DispList(R,n);
        if (!exchange)                  //本趟没有发生交换，中途结束算法
            return;
    }
}
```

4.2、快速排序

```
int partition(RecType R[],int s,int t)  //一趟划分
{
    int i=s,j=t;
    RecType tmp=R[i];           //以 R[i]为基准
    while (i<j)                  //从两端交替向中间扫描,直至 i=j 为止
    {   while (j>i && R[j].key>=tmp.key)
            j--;                //从右向左扫描,找一个小于 tmp.key 的 R[j]
        R[i]=R[j];              //找到这样的 R[j],放入 R[i]处
        while (i<j && R[i].key<=tmp.key)
            i++;                //从左向右扫描,找一个大于 tmp.key 的 R[i]
        R[j]=R[i];              //找到这样的 R[i],放入 R[j]处
    }
    R[i]=tmp;
    return i;
}
void QuickSort(RecType R[],int s,int t) //对 R[s..t]的元素进行快速排序
{   int i;
    RecType tmp;
    if (s<t)                     //区间内至少存在两个元素的情况
    {   count++;
        i=partition(R,s,t);
        DispList(R,10);         //调试用
        QuickSort(R,s,i-1);     //对左区间递归排序
        QuickSort(R,i+1,t);     //对右区间递归排序
    }
}
```

5、选择排序

5.1、简单选择排序

```
void SelectSort(RecType R[],int n)
{
    int i,j,k;
    RecType temp;
    for (i=0;i<n-1;i++)          //做第 i 趟排序
    {
        k=i;
        for (j=i+1;j<n;j++)      //在当前无序区 R[i..n-1]中选 key 最小的
R[k]
            if (R[j].key<R[k].key)
                k=j;            //k 记下目前找到的最小关键字所在的位置
            if (k!=i)               //交换 R[i]和 R[k]
```

```
            {
                temp=R[i];
                R[i]=R[k];
                R[k]=temp;
            }
        printf("  i=%d: ",i); DispList(R,n);
    }
}
```

5.2、堆排序

```
void sift(RecType R[],int low,int high)
{
    int i=low,j=2*i;                        //R[j]是 R[i]的左孩子
    RecType temp=R[i];
    while (j<=high)
    {
        if (j<high && R[j].key<R[j+1].key)    //若右孩子较大,把 j 指向右孩子
            j++;                            //变为 2i+1
        if (temp.key<R[j].key)
        {
            R[i]=R[j];                      //将 R[j]调整到双亲结点位置上
            i=j;                            //修改 i 和 j 值,以便继续向下筛选
            j=2*i;
        }
        else break;                         //筛选结束
    }
    R[i]=temp;                              //被筛选结点的值放入最终位置
}

void HeapSort(RecType R[],int n)
{
    int i;
    RecType tmp;
    for (i=n/2;i>=1;i--) //循环建立初始堆,调用 sift 算法 n/2 次
        sift(R,i,n);
    printf("初始堆:"); DispList1(R,n);
    for (i=n;i>=2;i--)       //进行 n-1 趟完成推排序,每一趟堆排序的元素个数减1
    {   tmp=R[1];           //将最后一个元素与根 R[1]交换
        R[1]=R[i];
        R[i]=tmp;
        printf("第%d 趟: ",n-i+1); DispList1(R,n);
```

```
        sift(R,1,i-1);      //对 R[1..i-1]进行筛选,得到 i-1 个节点的堆
        printf("筛选为:"); DispList1(R,n);
    }
}
```

6、归并排序

```
void Merge(RecType R[],int low,int mid,int high)
{
    RecType *R1;
    int i=low,j=mid+1,k=0; //k 是 R1 的下标,i、j 分别为第 1、2 段的下标
    R1=(RecType *)malloc((high-low+1)*sizeof(RecType));  //动态分配空间
    while (i<=mid && j<=high)          //在第 1 段和第 2 段均未扫描完时循环
        if (R[i].key<=R[j].key)       //将第 1 段中的记录放入 R1 中
        {
            R1[k]=R[i];
            i++;k++;
        }
        else                              //将第 2 段中的记录放入 R1 中
        {
            R1[k]=R[j];
            j++;k++;
        }
    while (i<=mid)                    //将第 1 段余下部分复制到 R1
    {
        R1[k]=R[i];
        i++;k++;
    }
    while (j<=high)                   //将第 2 段余下部分复制到 R1
    {
        R1[k]=R[j];
        j++;k++;
    }
    for (k=0,i=low;i<=high;k++,i++) //将 R1 复制回 R 中
        R[i]=R1[k];
}
void MergeSortDC(RecType R[],int low,int high)
//对 R[low..high]进行二路归并排序
{
    int mid;
    if (low<high)
    {   mid=(low+high)/2;
        MergeSortDC(R,low,mid);
        MergeSortDC(R,mid+1,high);
        Merge(R,low,mid,high);
```

```
    }
}
void MergeSort1(RecType R[],int n)  //自顶向下的二路归并算法
{
    MergeSortDC(R,0,n-1);
}
```

7、基数排序

7.1、结构体定义

```
typedef struct node
{
    char data[MAXD];    //记录的关键字定义的字符串
    struct node *next;
} NodeType;
```

7.2、算法实现

```
void CreaLink(NodeType *&p,char *a[],int n);
void DispLink(NodeType *p);
void RadixSort(NodeType *&p,int r,int d) //实现基数排序:p 为待排序序列链表指针,r 为基数,d 为关键字位数
{
    NodeType *head[MAXR],*tail[MAXR],*t;//定义各链队的首尾指针
    int i,j,k;
    for (i=0;i<=d-1;i++)                  //从低位到高位循环
    {
        for (j=0;j<r;j++)                 //初始化各链队首、尾指针
            head[j]=tail[j]=NULL;
        while (p!=NULL)                   //对于原链表中每个结点循环
        {
            k=p->data[i]-'0';         //找第 k 个链队
            if (head[k]==NULL)            //进行分配
            {
                head[k]=p;
                tail[k]=p;
            }
            else
            {
                tail[k]->next=p;
                tail[k]=p;
            }
            p=p->next;                    //取下一个待排序的元素
        }
        p=NULL;                           //重新用 p 来收集所有结点
```

```
        for (j=0;j<r;j++)                    //对于每一个链队循环
            if (head[j]!=NULL)              //进行收集
            {
                if (p==NULL)
                {
                    p=head[j];
                    t=tail[j];
                }
                else
                {
                    t->next=head[j];
                    t=tail[j];
                }
            }
        t->next=NULL;                        //最后一个结点的 next 域置 NULL
        printf("  按%s 位排序\t",(i==0?"个":"十"));
        DispLink(p);
    }
}
void CreateLink(NodeType *&p,char a[MAXE][MAXD],int n)  //采用后插法产生链表
{
    int i;
    NodeType *s,*t;
    for (i=0;i<n;i++)
    {
        s=(NodeType *)malloc(sizeof(NodeType));
        strcpy(s->data,a[i]);
        if (i==0)
        {
            p=s;t=s;
        }
        else
        {
            t->next=s;t=s;
        }
    }
    t->next=NULL;
}
void DispLink(NodeType *p)  //输出链表
{
    while (p!=NULL)
    {
        printf("%c%c ",p->data[1],p->data[0]);
```

```
70.         p=p->next;
71.     }
72.     printf("\n");
73. }
```

三、实验结果及分析

1、顺序查找

查找序列和所要查找的数字如下，

```
1.  KeyType a[]={2,3,1,8,5,4,9,0,7,6},k=9;
```

查找结果：

```
查找表：2 3 1 8 5 4 9 0 7 6
R[7]=9
```

2、折半查找

查找序列和所要查找的数字如下，

```
1.  KeyType a[]={2,3,10,15,20,25,28,29,30,35,40},k=20;
```

查找结果：

```
查找表：2 3 10 15 20 25 28 29 30 35 40
R[5]=20
```

3、插入排序

3.1、直接插入排序

```
排序前:9 8 7 6 5 4 3 2 1 0
  i=1: 8 9 7 6 5 4 3 2 1 0
  i=2: 7 8 9 6 5 4 3 2 1 0
  i=3: 6 7 8 9 5 4 3 2 1 0
  i=4: 5 6 7 8 9 4 3 2 1 0
  i=5: 4 5 6 7 8 9 3 2 1 0
  i=6: 3 4 5 6 7 8 9 2 1 0
  i=7: 2 3 4 5 6 7 8 9 1 0
  i=8: 1 2 3 4 5 6 7 8 9 0
  i=9: 0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
```

3.2、折半插入排序

```
排序前:9 8 7 6 5 4 3 2 1 0
  i=1: 8 9 7 6 5 4 3 2 1 0
  i=2: 7 8 9 6 5 4 3 2 1 0
  i=3: 6 7 8 9 5 4 3 2 1 0
  i=4: 5 6 7 8 9 4 3 2 1 0
  i=5: 4 5 6 7 8 9 3 2 1 0
  i=6: 3 4 5 6 7 8 9 2 1 0
  i=7: 2 3 4 5 6 7 8 9 1 0
  i=8: 1 2 3 4 5 6 7 8 9 0
  i=9: 0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
```

3.3、希尔排序

```
排序前:9 8 7 6 5 4 3 2 1 0
  d=5: 4 3 2 1 0 9 8 7 6 5
  d=2: 0 1 2 3 4 5 6 7 8 9
  d=1: 0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
```

4、交换排序

4.1、冒泡排序

```
排序前:9 8 7 6 5 4 3 2 1 0
  i=0: 0 9 8 7 6 5 4 3 2 1
  i=1: 0 1 9 8 7 6 5 4 3 2
  i=2: 0 1 2 9 8 7 6 5 4 3
  i=3: 0 1 2 3 9 8 7 6 5 4
  i=4: 0 1 2 3 4 9 8 7 6 5
  i=5: 0 1 2 3 4 5 9 8 7 6
  i=6: 0 1 2 3 4 5 6 9 8 7
  i=7: 0 1 2 3 4 5 6 7 9 8
  i=8: 0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
```

4.2、快速排序

```
排序前:6 8 7 9 0 1 3 2 4 5
5 4 2 3 0 1 6 9 7 8
1 4 2 3 0 5 6 9 7 8
0 1 2 3 4 5 6 9 7 8
0 1 2 3 4 5 6 9 7 8
0 1 2 3 4 5 6 9 7 8
0 1 2 3 4 5 6 8 7 9
0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
count=7
```

5、选择排序

5.1、简单选择排序

```
排序前:9 8 7 6 5 4 3 2 1 0
  i=0: 0 8 7 6 5 4 3 2 1 9
  i=1: 0 1 7 6 5 4 3 2 8 9
  i=2: 0 1 2 6 5 4 3 7 8 9
  i=3: 0 1 2 3 5 4 6 7 8 9
  i=4: 0 1 2 3 4 5 6 7 8 9
  i=5: 0 1 2 3 4 5 6 7 8 9
  i=6: 0 1 2 3 4 5 6 7 8 9
  i=7: 0 1 2 3 4 5 6 7 8 9
  i=8: 0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
```

5.2、堆排序

```
第1趟: 0 5 9 7 6 8 4 3 2 1
筛选为:0 9 8 7 6 5 4 3 2 1
第2趟: 0 1 8 7 6 5 4 3 2 9
筛选为:0 8 6 7 2 5 4 3 1 9
第3趟: 0 1 6 7 2 5 4 3 8 9
筛选为:0 7 6 4 2 5 1 3 8 9
第4趟: 0 3 6 4 2 5 1 7 8 9
筛选为:0 6 5 4 2 3 1 7 8 9
第5趟: 0 1 5 4 2 3 6 7 8 9
筛选为:0 5 3 4 2 1 6 7 8 9
第6趟: 0 1 3 4 2 5 6 7 8 9
筛选为:0 4 3 1 2 5 6 7 8 9
第7趟: 0 2 3 1 4 5 6 7 8 9
筛选为:0 3 2 1 4 5 6 7 8 9
第8趟: 0 1 2 3 4 5 6 7 8 9
筛选为:0 2 1 3 4 5 6 7 8 9
第9趟: 0 1 2 3 4 5 6 7 8 9
筛选为:0 1 2 3 4 5 6 7 8 9
排序后:0 1 2 3 4 5 6 7 8 9
```

6、归并排序

```
排序前:9 8 7 6 5 4 3 2 1 0
排序后:0 1 2 3 4 5 6 7 8 9
```

7、基数排序

```
  初始关键字     75 23 98 44 57 12 29 64 38 82
  按个位排序     12 82 23 44 64 75 57 98 38 29
  按十位排序     12 23 29 38 44 57 64 75 82 98
  最终结果       12 23 29 38 44 57 64 75 82 98
```