《数据结构》实验指导书

计算机科学与技术学院实验中心

2020 年 11 月

目 录

《数据结构》上机实验内容和要求

通过上机实验加深对课程内容的理解，提高程序设计、开发及调试能力。本实验指导书适用于16学时《数据结构法》实验课，实验项目具体内容如下：

序号	实验名称	内容提要	每组人数	实验时数	实验类别
1	顺序表的实现及应用	掌握顺序表结构，实现其插入、删除等算法。利用顺序表将两个有序线性表合并为一个有序表。	1	2	设计
2	链表的实现及应用	掌握单链表结构，实现其插入、删除、查找等算法。利用单链表将两个有序链表合并为一个有序链表。	1	2	设计
3	栈的实现及应用	掌握栈的结构，将栈应用于表达式计算问题	1	2	设计
4	队列的实现及应用	掌握队列的结构，将队列应用于模拟服务台前的排队现象问题	1	2	设计
5	二叉树操作及应用	掌握二叉树的存储，实现三种遍历的递归算法、实现前序或中序的非递归遍历算法	1	2	设计
6	图的遍历操作及应用	实现图的存储、深度遍历和广度遍历算法	1	2	设计
7	查找算法的实现	实现顺序表的二分查找算法	1	2	设计
8	排序算法的实现	实现直接插入排序、快速排序等算法	1	2	设计

实验报告要求

请按照评分标准和报告模板要求，提交实验报告电子版文件。

实验一、顺序表的实现及应用

一、实验目的

了解和掌握线性表的顺序存储结构；掌握用 C 语言上机调试线性表的基本方法；掌握线性表的基本操作：插入、删除、查找以及线性表合并等运算在顺序存储结构和链接存储结构上的运算，以及对相应算法的性能分析。

二、实验要求

给定一段程序代码，程序代码所完成的功能为：（1）建立一个线性表；（2）依次输入数据元素 1,2,3,4,5,6,7,8,9,10；（3）删除数据元素 5；（4）依次显示当前线性表中的数据元素。假设该线性表的数据元素个数在最坏情况下不会超过 100 个，要求使用顺序表。

程序中有 3 处错误的地方，有标识，属于逻辑错误，对照书中的代码仔细分析后，要求同学们修改错误的代码，修改后上机调试得到正确的运行结果。

三、程序代码

```
#include <stdio.h>
#define MaxSize    100
typedef int DataType;

typedef struct
{
   DataType list[MaxSize];
   int size;
} SeqList;

void ListInitiate(SeqList *L)/*初始化顺序表 L*/
{
L->size = 0;/*定义初始数据元素个数*/
}

int ListLength(SeqList L)/*返回顺序表 L 的当前数据元素个数*/
{
return L.size;
}

int ListInsert(SeqList *L, int i, DataType x)
/*在顺序表 L 的位置 i（0 ≤ i ≤ size）前插入数据元素值 x*/
/*插入成功返回 1，插入失败返回 0*/
{
   int j;
```

```
    if(L->size >= MaxSize)
    {
    printf("顺序表已满无法插入! \n");
    return 0;
    }
    else if(i < 0 || i > L->size )
    {
    printf("参数 i 不合法! \n");
    return 0;
    }
    else
    { //此段程序有一处错误
    for(j = L->size; j > i; j--) L->list[j] = L->list[j];/*为插入做准备*/
    L->list[i] = x;/*插入*/
    L->size ++;/*元素个数加 1*/
    return 1;
    }
}

int ListDelete(SeqList *L, int i, DataType *x)
/*删除顺序表 L 中位置 i（0 ≤ i ≤ size - 1）的数据元素值并存放到参数 x 中*/
/*删除成功返回 1，删除失败返回 0*/
{
    int j;
    if(L->size <= 0)
    {
    printf("顺序表已空无数据元素可删! \n");
    return 0;
    }
    else if(i < 0 || i > L->size-1)
    {
    printf("参数 i 不合法");
    return 0;
    }
    else
    {//此段程序有一处错误

    *x = L->list[i];/*保存删除的元素到参数 x 中*/
    for(j = i +1; j <= L->size-1; j++) L->list[j] = L->list[j-1];/*依次前移*/
    L->size--;/*数据元素个数减 1*/
```

```
    return 1;
    }
  }

  int ListGet(SeqList L, int i, DataType *x)
  /*取顺序表 L 中第 i 个数据元素的值存于 x 中，成功则返回 1，失败返回 0*/
  {
    if(i < 0 || i > L.size-1)
    {
    printf("参数 i 不合法! \n");
    return 0;
    }
    else
    {
    *x = L.list[i];
    return 1;
    }
  }


  void main(void)
  { SeqList myList;
    int i , x;
    ListInitiate(&myList);
   for(i = 0; i < 10; i++)
   ListInsert(&myList, i, i+1);
   ListDelete(&myList, 4, &x);
   for(i = 0; i < ListLength(myList); i++)
  {
  ListGet(,i,&x); //此段程序有一处错误
  printf("%d", x);
}
  }
```

四、实验任务

1.改正上述程序中的错误。

2.编写合并函数，将两个有序线性表合并为一个有序表并在主函数中加以测试。

3.完成实验报告的撰写。

实验二、链表的实现及应用

一、实验目的

了解和掌握线性表的链式存储结构；掌握用 C 语言上机调试线性表的基本方法；掌握线性表的基本操作：插入、删除、查找以及线性表合并等运算在顺序存储结构和链接存储结构上的运算，以及对相应算法的性能分析。

二、实验要求

给定一段程序代码，程序代码所完成的功能为：（1）建立一个线性表；（2）依次输入数据元素 1,2,3,4,5,6,7,8,9,10；（3）删除数据元素 5；（4）依次显示当前线性表中的数据元素。假设该线性表的数据元素个数在最坏情况下不会超过 100 个，要求使用单链表。

程序中有 3 处错误的地方，有标识，属于逻辑错误，对照书中的代码仔细分析后，要求同学们修改错误的代码，上机调试并得到正确的运行结果。

三、程序代码：

```
#include <stdio.h>/*该文件包含 printf()等函数*/
#include <stdlib.h>/*该文件包含 exit()等函数*/
#include <malloc.h>/*该文件包含 malloc()等函数*/

typedef int DataType;/*定义 DataType 为 int*/


typedef struct Node
{
   DataType data;
   struct Node *next;
} SLNode;

void ListInitiate(SLNode **head)/*初始化*/
{
   /*如果有内存空间，申请头结点空间并使头指针 head 指向头结点*/
   if((*head = (SLNode *)malloc(sizeof(SLNode))) == NULL) exit(1);
   (*head)->next = NULL;/*置链尾标记 NULL */
   }

   int ListLength(SLNode *head)                    /*  单链表的长度*/
   {
   SLNode *p = head;/*p 指向首元结点*/
```

```
    int size = 0;/*size 初始为 0*/

    while(p->next != NULL)/*循环计数*/
    {
    p = p->next;
    size ++;
    }
    return size;
}

int ListInsert(SLNode *head, int i, DataType x)
/*在带头结点的单链表 head 的数据元素 ai（0 ≤ i ≤ size）结点前*/
/*插入一个存放数据元素 x 的结点*/
{
    SLNode *p, *q;
    int j;

    p = head; /*p 指向首元结点*/
    j = -1;/*j 初始为-1*/
    while(p->next != NULL && j < i - 1)
    /*最终让指针 p 指向数据元素 ai-1 结点*/
    {
    p = p->next;
    j++;
    }

    if(j != i - 1)
    {
    printf("插入位置参数错！ ");
    return 0;
    }

    /*生成新结点由指针 q 指示*/
    if((q = (SLNode *)malloc(sizeof(SLNode))) == NULL) exit(1);
    q->data = x;

    //此段程序有一处错误
    p->next = q->next;/*给指针 q->next 赋值*/
    p->next = q;/*给指针 p->next 重新赋值*/
    return 1;
```

```
}

int ListDelete(SLNode *head, int i, DataType *x)
/*删除带头结点的单链表 head 的数据元素 ai（0 ≤ i ≤ size - 1）结点*/
/*删除结点的数据元素域值由 x 带回。删除成功时返回 1；失败返回 0*/
{
   SLNode *p, *s;
   int j;

   p = head; /*p 指向首元结点*/
   j = -1;/*j 初始为-1*/
   while(p->next != NULL && p->next->next!= NULL && j < i - 1)
   /*最终让指针 p 指向数据元素 ai-1 结点*/
   {
   p = p->next;
   j++;
   }

   if(j != i - 1)
   {
   printf("删除位置参数错！ ");
   return 0;
   }

   //此段程序有一处错误
   s->next = p; /*指针 s 指向数据元素 ai 结点*/
   *x = s->data;/*把指针 s 所指结点的数据元素域值赋予 x*/
   p->next = s->next;/*把数据元素 ai 结点从单链表中删除*/
   free(s);/*释放指针 s 所指结点的内存空间*/
   return 1;
}

int ListGet(SLNode *head, int i, DataType *x)
/*取数据元素 ai 和删除函数类同，只是不删除数据元素 ai 结点*/
{
   SLNode *p;
   int j;

   p = head;
   j = -1;
```

```
    while(p->next != NULL && j < i)
    {
    p = p->next;j++;
    }

    if(j != i)
    {
    printf("取元素位置参数错！ ");
    return 0;
    }

    //此段程序有一处错误
    *x = p->next;
    return 1;
    }

    void Destroy(SLNode **head)
    {
    SLNode *p, *p1;

    p = *head;
    while(p != NULL)
    {
    p1 = p;
    p = p->next;
    free(p1);
    }
    *head = NULL;
}

void main(void)
{
    SLNode *head;
    int i , x;
    ListInitiate(&head);/*初始化*/
for(i = 0; i < 10; i++)
{
if(ListInsert(head, i, i+1) == 0) /*插入 10 个数据元素*/
{
printf("错误! \n");
```

```
return;
}
   }

   if(ListDelete(head, 4, &x) == 0) /*删除数据元素 5*/
   {
   printf("错误! \n");
   return;
   }

   for(i = 0; i < ListLength(head); i++)
   {
   if(ListGet(head, i, &x) == 0) /*取元素*/
   {
   printf("错误! \n");
   return;
   }
   else printf("%d        ", x);/*显示数据元素*/
   }

   Destroy(&head);
}
```

三、实验任务

1.改正上述程序中的错误。

2.编写合并函数，将两个有序的单链表合并成一个有序单链表。

3.完成实验报告的撰写。

实验三、栈的实现及应用

一、实验目的

1.掌握栈的存储表示和实现

2.掌握栈的基本操作实现。

3.掌握栈在解决实际问题中的应用。

二、实验要求

问题描述：设计一个程序，演示用算符优先法对算术表达式求值的过程。利用算符优先关系，实现对算术四则混合运算表达式的求值。

（1）输入的形式：表达式，例如 2*(3+4)#

包含的运算符只能有'+' 、'-' 、'*' 、'/' 、'('、 ')'，“#”代表输入结束符；

（2）输出的形式：运算结果，例如 2*(3+4)=14；

（3）程序所能达到的功能：对表达式求值并输出。

三、解题参考思路

为了实现用栈计算算数表达式的值，需设置两个工作栈：用于存储运算符的栈 opter，以及用于存储操作数及中间结果的栈 opnd。

算法基本思想如下：

（1）首先将操作数栈 opnd 设为空栈，而将'#'作为运算符栈 opter 的栈底元素，这样的目的是判断表达式是否求值完毕。

（2）依次读入表达式的每个字，表达式须以'#'结，读入字符若是操作数则入栈 opnd，读入字符若是运算符，则将此运算符 c 与 opter 的栈顶元素 top 比较优先级后执行相应的操作，具体操作如下：

(i)若 top 的优先级小于 c，即 top<c，则将 c 直接入栈 opter，并读入下一字符赋值给 c；

(ii)若 top 的优先级等于 c，即 top=c，则弹出 opter 的栈顶元素，并读入下一字符赋值给 c，这一步目的是进行括号操作;

（iii）若 top 优先级高于 c，即 top>c，则表明可以计算，此时弹出 opnd 的栈顶两个元素，并且弹出 opter 栈顶的的运算符，计算后将结果放入栈 opnd 中。直至 opter 的栈顶元素和当前读入的字符均为'#'，此时求值结束。

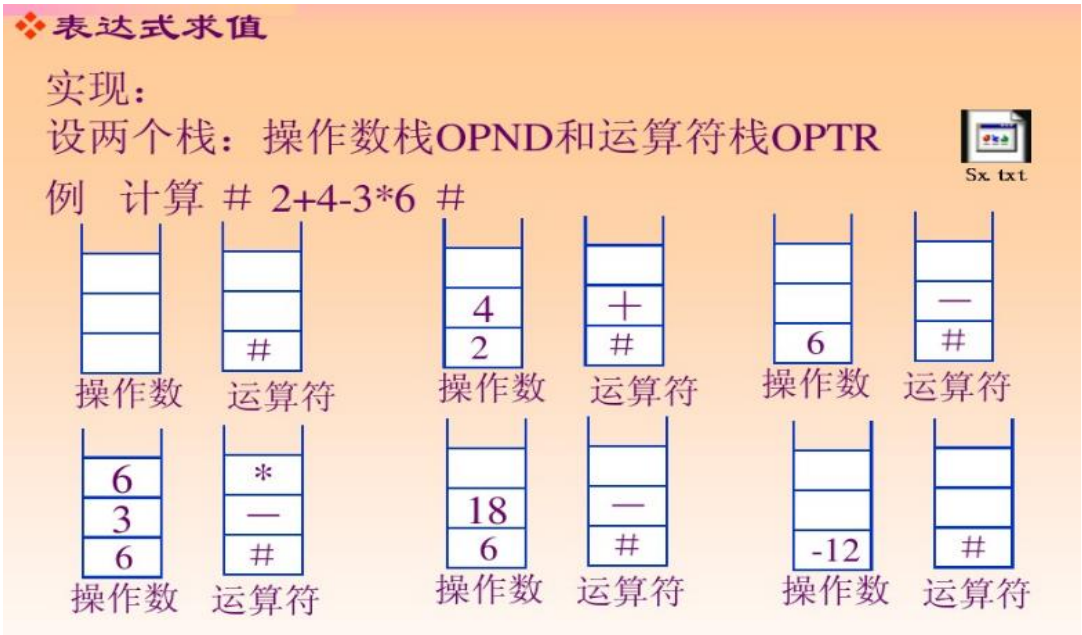
算符间的优先关系如下表所示（表来源：严蔚敏《数据结构》）：

表 3.1 算符间的优先关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

表中需要注意的是 θ 1 为 opter 的栈顶元素，θ 2 为从表达式中读取的操作符，此优先级表可以用二维数组实现。

图例：



比较算符优先关系代码示例：

```
1. int getIndex(char theta)    //获取 theta 所对应的索引
2. {
3.     int index = 0;
4.     switch (theta)
5.     {
6.     case '+':
7.         index = 0;
8.         break;
9.     case '-':
10.         index = 1;
11.         break;
12.     case '*':
13.         index = 2;
14.         break;
15.     case '/':
16.         index = 3;
17.         break;
18.     case '(':
19.         index = 4;
20.         break;
21.     case ')':
```

```
            index = 5;
            break;
        case '#':
            index = 6;
        default:break;
        }
        return index;
}

char getPriority(char theta1, char theta2)   //获取 theta1 与 theta2 之间的优先级
{
        const char priority[][7] =      //算符间的优先级关系
        {
            { '>','>','<','<','<','>','>' },
            { '>','>','<','<','<','>','>' },
            { '>','>','>','>','<','>','>' },
            { '>','>','>','>','<','>','>' },
            { '<','<','<','<','<','=','0' },
            { '>','>','>','>','0','>','>' },
            { '<','<','<','<','<','0','=' },
        };

        int index1 = getIndex(theta1);
        int index2 = getIndex(theta2);
        return priority[index1][index2];
}
```

四、实验任务

认真阅读与理解实验内容的具体要求，参考教材相关章节，结合实验内容的要求，编写实验程序并上机调试与测试，完成实验报告的撰写。

实验四、队列的实现及应用

一、实验目的

1.掌握队列的存储表示和实现。

2.掌握队列的基本操作实现。

3.掌握队列在解决实际问题中的应用。

二、实验要求

利用队列模拟服务台前的排队现象问题。

问题描述：某银行有一个客户办理业务站，在单位时间内随机地有客户到达，设每位客户的业务办理时间是某个范围的随机值。设只有一个窗口，一位业务人员，要求程序模拟统计在设定时间内，业务人员的总空闲时间和客户的平均等待时间。假定模拟数据已按客户到达的先后顺序依次存于某个正文数据文件中，对应每位客户有两个数据：到达时间和需要办理业务的时间，文本文件内容如：10 20 23 10 45 5 55 10 58 15 65 10。

三、**解题参考思路**

与栈相对应，队列是一种先进先出的线性表。它只允许在表的一端进行插入，而在另一端进行删除元素。允许插入的一端称队尾，允许删除的一端称队头。插入与删除分别称为入队与出队。队列示意图如下图所示：

【数据描述】

```
typedef struct{
  int arrive;
  int treat;//客户的信息结构
}QNODE;
typedef struct node{
QNODE data;
Struct node *next;//队列中的元素信息
}LNODE,*QueuePtr;

typedef     struct{ //链队列类型
      QueuePtr        front ;     //队头指针
      QueuePtr        rear ;      //队尾指针
} LinkQueue;
```

队列的基本操作如队列的初始化、判空、入队和出队等操作实现参考教材。

【算法描述】

```
{ 设置统计初值：业务员等待时间，客户总的待时间，客户总人数等
  设置当前时钟 clock 时间为 0；//用变量 clock 来模拟当前时间.
  打开数据文件，准备读；
```

读入第一位客户信息于暂存变量中；//文件读操作 have= fscanf(fp,"%d %d",&temp.arrive,&temp.treat);
do{//约定每轮循环，处理完一位客户
if(等待队列为空，并且还有客户)
{ //等待队列为空时
累计业务员总等待时间；
时钟推进到暂存变量中的客户的到达时间；//clock=temp.arrive
暂存变量中的客户信息进队；
读取下一位客户信息于暂存变量；
}
从等待队列出队一位客户；
累计客户人数；
将该客户的等待时间累计到客户的总等待时间；//=当前时间-客户到达时间
设定当前客户的业务办理结束时间；//=当前时间+客户办理业务所需时间
while(下一位客户的到达时间在当前客户处理结束之前，并且还有客户)
{
暂存变量中的客户信息进队；
读取下一位客户信息于暂存变量；
}
时钟推进到当前客户办理结束时间；
}while(还有未处理的客户)；//等待队列不为空或者还有客户（have==2）
计算统计结果，并输出；

附：文件操作：

```
char Fname[120];//读取文件的文件名
FILE*fp;
 if((fp=fopen(Fname,"r"))==NULL)
    {
    printf("文件打开出错");
     return 0;
    }
```

have= fscanf(fp,"%d %d",&temp.arrive,&temp.treat);//have 返回值等于从文件中一次读操作读出数据的个数，这里等于 2.

四、实验任务

认真阅读与理解实验内容的具体要求，参考教材相关章节，结合实验内容的要求，编写实验程序并上机调试与测试，完成实验报告的撰写。

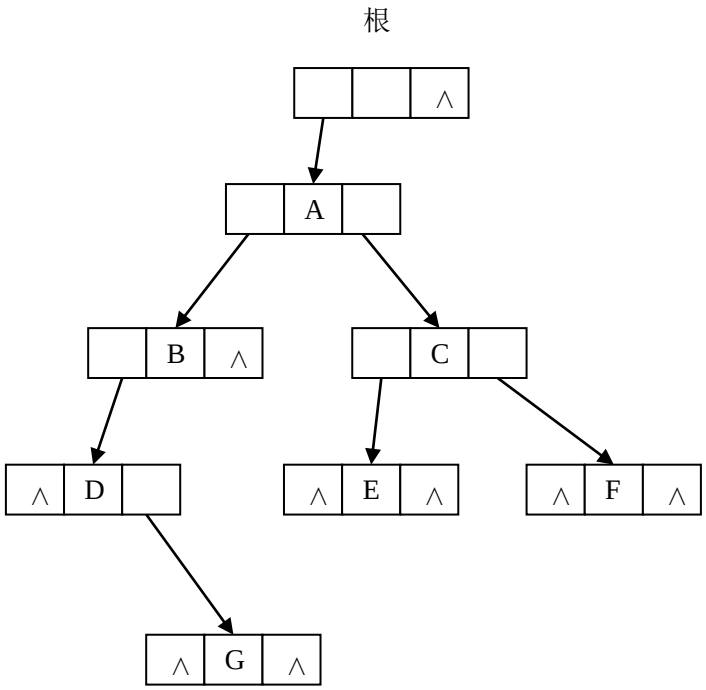
实验五、二叉树操作及应用

一、实验目的

掌握二叉树的定义、结构特征，以及各种存储结构的特点及使用范围，各种遍历算法。掌握用指针类型描述、访问和处理二叉树的运算。掌握前序或中序的非递归遍历算法。

二、实验要求

有如下二叉树：



程序代码给出了该二叉树的链式存储结构的建立、前序、中序、后序遍历的算法，同时也给出了查询“E”是否在二叉树里的代码。代码有三处错误，有标识，属于逻辑错误，对照书中的代码仔细分析后，请修改了在电脑里运行。

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

typedef char DataType;

typedef struct Node
{
DataType data;/*数据域*/
struct Node *leftChild;/*左子树指针*/
```

```
struct Node *rightChild;/*右子树指针*/
}BiTreeNode;/*结点的结构体定义*/

/*初始化创建二叉树的头结点*/
void Initiate(BiTreeNode **root)
{
   *root = (BiTreeNode *)malloc(sizeof(BiTreeNode));
   (*root)->leftChild = NULL;
   (*root)->rightChild = NULL;
}

void Destroy(BiTreeNode **root)
{
   if((*root) != NULL && (*root)->leftChild != NULL)
   Destroy(&(*root)->leftChild);

   if((*root) != NULL && (*root)->rightChild != NULL)
   Destroy(&(*root)->rightChild);

   free(*root);
}

/*若当前结点 curr 非空，在 curr 的左子树插入元素值为 x 的新结点*/
/*原 curr 所指结点的左子树成为新插入结点的左子树*/
/*若插入成功返回新插入结点的指针，否则返回空指针*/
BiTreeNode *InsertLeftNode(BiTreeNode *curr, DataType x)
{
BiTreeNode *s, *t;
   if(curr == NULL) return NULL;

   t = curr->leftChild;/*保存原 curr 所指结点的左子树指针*/
   s = (BiTreeNode *)malloc(sizeof(BiTreeNode));
   s->data = x;
   s->leftChild = t;/*新插入结点的左子树为原 curr 的左子树*/
   s->rightChild = NULL;

   curr->leftChild = s;/*新结点成为 curr 的左子树*/
   return curr->leftChild;/*返回新插入结点的指针*/
}
```

```
/*若当前结点 curr 非空，在 curr 的右子树插入元素值为 x 的新结点*/
/*原 curr 所指结点的右子树成为新插入结点的右子树*/
/*若插入成功返回新插入结点的指针，否则返回空指针*/
BiTreeNode *InsertRightNode(BiTreeNode *curr, DataType x)
{
   BiTreeNode *s, *t;

   if(curr == NULL) return NULL;

   t = curr->rightChild;/*保存原 curr 所指结点的右子树指针*/
   s = (BiTreeNode *)malloc(sizeof(BiTreeNode));
   s->data = x;
   s->rightChild = t;/*新插入结点的右子树为原 curr 的右子树*/
   s->leftChild = NULL;

   curr->rightChild = s;/*新结点成为 curr 的右子树*/
   return curr->rightChild;/*返回新插入结点的指针*/
}



void PreOrder(BiTreeNode *t, void visit(DataType item))
//使用 visit(item)函数前序遍历二叉树 t
{
   if(t != NULL)
   {//此小段有一处错误
   visit(t->data);
   PreOrder(t-> rightChild, visit);
   PreOrder(t-> leftChild, visit);
   }
}

void InOrder(BiTreeNode *t, void visit(DataType item))
//使用 visit(item)函数中序遍历二叉树 t
{
if(t != NULL)
   {//此小段有一处错误
   InOrder(t->leftChild, visit);
   InOrder(t->rightChild, visit);
   visit(t->data);
```

```
    }
}

void PostOrder(BiTreeNode *t, void visit(DataType item))
//使用 visit(item)函数后序遍历二叉树 t
{
    if(t != NULL)
    {//此小段有一处错误
    visit(t->data);
    PostOrder(t->leftChild, visit);
    PostOrder(t->rightChild, visit);
    }
}



void Visit(DataType item)
{
    printf("%c ", item);
    }

    BiTreeNode *Search(BiTreeNode *root, DataType x)//需找元素 x 是否在二叉树中
    {
    BiTreeNode *find=NULL;
    if(root!=NULL)
    {
    if(root->data==x)
    find=root;
    else
    {
    find=Search(root->leftChild,x);
    if(find==NULL)
    find=Search(root->rightChild,x);
    }
    }
    return find;
}

void main(void)
{
    BiTreeNode *root, *p, *pp,*find;
```

```
    char x='E';

    Initiate(&root);
    p = InsertLeftNode(root, 'A');
    p = InsertLeftNode(p, 'B');
    p = InsertLeftNode(p, 'D');
    p = InsertRightNode(p, 'G');
    p = InsertRightNode(root->leftChild, 'C');
    pp = p;
    InsertLeftNode(p, 'E');
    InsertRightNode(pp, 'F');

    printf("前序遍历：");
    PreOrder(root->leftChild, Visit);
    printf("\n 中序遍历：");
    InOrder(root->leftChild, Visit);
    printf("\n 后序遍历：");
    PostOrder(root->leftChild, Visit);

    find=Search(root,x);
    if(find!=NULL)
    printf("\n 数据元素%c 在二叉树中 \n",x);
    else
    printf("\n 数据元素%c 不在二叉树中 \n",x);

    Destroy(&root);

}
```

三、实验任务：

1.改正程序错误。

2.编写二叉树的前序（或中序）的非递归遍历算法并进行测试。

二叉树遍历的非递归算法

1：先序遍历的非递归算法

需要设计栈：保留结点的位置用于查找右孩子

入栈：访问结点之后要做入栈操作

出栈：某结点的左子树访问完毕之后

思路：①栈初始化，p 初始化

② 栈不空或 p 不为空

a: 若 p 不空:访问 p 所指结点，p 入栈，修改 p ,p=p->lchild

否则

b:若栈不空：出栈→p，求 p 的右孩子 p=p->rchild

算法：

```
Void    Preorder(BiTree t){
  //二叉树先序遍历非递归算法
    p=t;    InitStack(s);
    while    (p||!StackEmpty(s))
        if (p) {printf (p->data); Push(s,p); p=p->lchild}
            else {Pop(s,p); p=p->rchild }
}
```

2、中序遍历的非递归算法

需要设计栈：保留结点的位置用于查找右孩子，访问该结点

入栈：访问左子树之前要做入栈操作

出栈：某结点的左子树访问完毕之后

思路：①栈初始化，p 初始化

② 栈不空或 p 不为空

a: 若 p 不空:p 入栈，修改 p ,p=p->lchild

否则 b:若栈不空：出栈→p；访问 p 所指结点，求 p 的右孩子 p=p->rchild

```
#include<stack>
using namespace std;

stack<BiTreeNode*> s;
BiTreeNode *p;
s.push(p);
s.top();
s.pop();
s.empty()_
```

3.完成实验报告的撰写。

实验六、图的遍历操作及应用

一、实验目的

掌握有向图和无向图的概念；掌握邻接矩阵和邻接链表建立图的存储结构；掌握 DFS 及 BFS 对图的遍历操作；了解图结构在人工智能、工程等领域的广泛应用。

二、实验要求

采用邻接矩阵和邻接链表作为图的存储结构,完成有向图和无向图的 DFS 和 BFS 操作。本实验给出了示例程序，其中共有 4 处错误，错误段均有标识，属于逻辑错误。请认真理解程序，修改程序代码，并在电脑上调试运行。

三、DFS 和 BFS 的基本思想

深度优先搜索法 DFS 的基本思想：从图 G 中某个顶点 Vo 出发，首先访问 Vo，然后选择一个与 Vo 相邻且没被访问过的顶点 Vi 访问，再从 Vi 出发选择一个与 Vi 相邻且没被访问过的顶点 Vj 访问，……依次继续。如果当前被访问过的顶点的所有邻接顶点都已被访问，则回退到已被访问的顶点序列中最后一个拥有未被访问的相邻顶点的顶点 W，从 W 出发按同样方法向前遍历。直到图中所有的顶点都被访问。

广度优先算法 BFS 的基本思想：从图 G 中某个顶点 Vo 出发，首先访问 Vo，然后访问与 Vo 相邻的所有未被访问过的顶点 V1，V2，……，Vt；再依次访问与 V1，V2，……，Vt 相邻的起且未被访问过的的所有顶点。如此继续，直到访问完图中的所有顶点。

四、示例程序

1.邻接矩阵作为存储结构的程序示例

```
#include"stdio.h"
#include"stdlib.h"
#define MaxVertexNum 100     //定义最大顶点数
typedef struct{
    char vexs[MaxVertexNum];        //顶点表
    int edges[MaxVertexNum][MaxVertexNum];
      //邻接矩阵，可看作边表
    int n,e;          //图中的顶点数 n 和边数 e
}MGraph;              //用邻接矩阵表示的图的类型
//========建立邻接矩阵=======
void CreatMGraph(MGraph *G)
{
     int i,j,k;
     char a;
     printf("Input VertexNum(n) and EdgesNum(e): ");
```

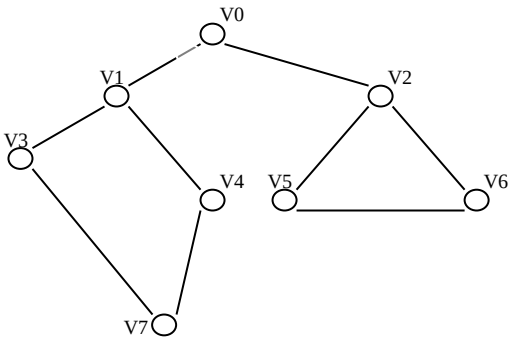


图 G 的示例

```
    scanf("%d,%d",&G->n,&G->e);          //输入顶点数和边数
    scanf("%c",&a);
    printf("Input Vertex string:");
    for(i=0;i<G->n;i++)
    {
    scanf("%c",&a);
    G->vexs[i]=a;              //读入顶点信息，建立顶点表
    }
   for(i=0;i<G->n;i++)
   for(j=0;j<G->n;j++)
        G->edges[i][j]=0;    //初始化邻接矩阵
    printf("Input edges,Creat Adjacency Matrix\n");
    for(k=0;k<G->e;k++) {       //读入 e 条边，建立邻接矩阵
        scanf("%d%d",&i,&j);        //输入边（Vi，Vj）的顶点序号
        G->edges[i][j]=1;
        G->edges[j][i]=1; //若为无向图，矩阵为对称矩阵；若建立有向图，去掉该条
   语句
    }
}
//========定义标志向量，为全局变量=======
typedef enum{FALSE,TRUE} Boolean;
Boolean visited[MaxVertexNum];
//========DFS：深度优先遍历的递归算法======
void DFSM(MGraph *G,int i)
{ //以 Vi 为出发点对邻接矩阵表示的图 G 进行 DFS 搜索，邻接矩阵是 0，1 矩阵
    int j;
    printf("%c",G->vexs[i]);     //访问顶点 Vi
    visited[i]=TRUE;             //置已访问标志
    for(j=0;j<G->n;j++)          //依次搜索 Vi 的邻接点
        if(G->edges[i][j]==1 && ! visited[j])
        DFSM(G,j);              //（Vi，Vj）∈E，且 Vj 未访问过，故 Vj 为新出发点
}
void DFS(MGraph *G)
{  //此段代码有一处错误
    int i;
    for(i=0;i<G->n;i++)
        visited[i]=FALSE;            //标志向量初始化
    for(i=0;i<G->n;i++)
        if(!visited[i])              //Vi 未访问过
        DFS(G,i);                //以 Vi 为源点开始 DFS 搜索
```

```
}
//==========BFS：广度优先遍历=======
void BFS(MGraph *G,int k)
{                  //以 Vk 为源点对用邻接矩阵表示的图 G 进行广度优先搜索
    int i,j,f=0,r=0;
    int cq[MaxVertexNum];          //定义队列
    for(i=0;i<G->n;i++)
        visited[i]=FALSE;           //标志向量初始化
    for(i=0;i<=G->n;i++)
        cq[i]=-1;                       //队列初始化
    printf("%c",G->vexs[k]);       //访问源点 Vk
    visited[k]=TRUE;
    cq[r]=k;            //Vk 已访问，将其入队。注意，实际上是将其序号入队
    while(cq[f]!=-1) {           //队非空则执行
     i=cq[f]; f=f+1;               //Vf 出队
     for(j=0;j<G->n;j++)           //依次 Vi 的邻接点 Vj
         if(G->edges[i][j]==1 && !visited[j]) {  //Vj 未访问 以下三行代码有一处
错误
         printf("%c",G->vexs[j]);          //访问 Vj
         visited[j]=FALSE;
         r=r+1; cq[r]=j;            //访问过 Vj 入队
         }
    }
}
//==========main=====
void main()
{
    MGraph *G;
    G=(MGraph *)malloc(sizeof(MGraph));   //为图 G 申请内存空间
    CreatMGraph(G);            //建立邻接矩阵
    printf("Print Graph DFS: ");
    DFS(G);                    //深度优先遍历
    printf("\n");
    printf("Print Graph BFS: ");
    BFS(G,3);               //以序号为 3 的顶点开始广度优先遍历
    printf("\n");}
```

执行顺序：

```
Input VertexNum(n) and EdgesNum(e): 8, 9
Input Vertex string: 01234567
Input edges,Creat Adjacency Matrix
```

```
0 1
0 2
1 3
1 4
2 5
2 6
3 7
4 7
5 6
Print Graph DFS: 01374256
Print Graph BFS: 31704256
```

2.邻接链表作为存储结构程序示例

```
#include"stdio.h"
#include"stdlib.h"
#define MaxVertexNum 50          //定义最大顶点数
typedef struct node{       //边表结点
     int adjvex;           //邻接点域
     struct node *next;    //链域
}EdgeNode;
typedef struct vnode{      //顶点表结点
    char vertex;           //顶点域
    EdgeNode *firstedge;   //边表头指针
}VertexNode;
typedef VertexNode AdjList[MaxVertexNum];         //AdjList 是邻接表类型
typedef struct {
    AdjList adjlist;       //邻接表
    int n,e;               //图中当前顶点数和边数
} ALGraph;                 //图类型
//=========建立图的邻接表=======
void CreatALGraph(ALGraph *G)
{
     int i,j,k;
     char a;
     EdgeNode *s;           //定义边表结点
     printf("Input VertexNum(n) and EdgesNum(e): ");
     scanf("%d,%d",&G->n,&G->e);        //读入顶点数和边数
     scanf("%c",&a);
     printf("Input Vertex string:");
     for(i=0;i<G->n;i++)          //建立顶点表
```

```
    {
        scanf("%c",&a);
        G->adjlist[i].vertex=a;        //读入顶点信息
        G->adjlist[i].firstedge=NULL;  //边表置为空表
    }
    printf("Input edges,Creat Adjacency List\n");
    for(k=0;k<G->e;k++) {          //建立边表
        scanf("%d%d",&i,&j);           //读入边（Vi，Vj）的顶点对序号
        s=(EdgeNode *)malloc(sizeof(EdgeNode));    //生成边表结点
        s->adjvex=j;                   //邻接点序号为 j
        s->next=G->adjlist[i].firstedge;
        G->adjlist[i].firstedge=s;     //将新结点*S 插入顶点 Vi 的边表头部
        s=(EdgeNode *)malloc(sizeof(EdgeNode));
        s->adjvex=i;                   //邻接点序号为 i
        s->next=G->adjlist[j].firstedge;
        G->adjlist[j].firstedge=s;     //将新结点*S 插入顶点 Vj 的边表头部
    }
}
//=========定义标志向量，为全局变量=======
typedef enum{FALSE,TRUE} Boolean;
Boolean visited[MaxVertexNum];
//========DFS：深度优先遍历的递归算法======
void DFSM(ALGraph *G,int i)
{                          //以 Vi 为出发点对邻接链表表示的图 G 进行 DFS 搜索
    EdgeNode *p;
    printf("%c",G->adjlist[i].vertex);    //访问顶点 Vi
    visited[i]=TRUE;                      //标记 Vi 已访问
    p=G->adjlist[i].firstedge;            //取 Vi 边表的头指针
    while(p) {                   //依次搜索 Vi 的邻接点 Vj，这里 j=p->adjvex
                                 //以下 3 行代码有一处错误
        if(! visited[p->adjvex])      //若 Vj 尚未被访问
            DFS(G,p->adjvex);        //则以 Vj 为出发点向纵深搜索
            p=p->next;                //找 Vi 的下一个邻接点
    }
}
void DFS(ALGraph *G)
{
    int i;
    for(i=0;i<G->n;i++)
    visited[i]=FALSE;             //标志向量初始化
```

```
    for(i=0;i<G->n;i++)
    if(!visited[i])                 //Vi 未访问过
        DFSM(G,i);                  //以 Vi 为源点开始 DFS 搜索
}
//==========BFS：广度优先遍历=========
void BFS(ALGraph *G,int k)
{                           //以 Vk 为源点对用邻接链表表示的图 G 进行广度优先搜索
    int i,f=0,r=0;
    EdgeNode *p;
    int cq[MaxVertexNum];           //定义 FIFO 队列
    for(i=0;i<G->n;i++)
        visited[i]=FALSE;               //标志向量初始化
    for(i=0;i<=G->n;i++)
        cq[i]=-1;                               //初始化标志向量
    printf("%c",G->adjlist[k].vertex); //访问源点 Vk
    visited[k]=TRUE;
    cq[r]=k;            //Vk 已访问，将其入队。注意，实际上是将其序号入队
    while(cq[f]!=-1)
        {   //队列非空则执行
        i=cq[f]; f=f+1;                 //Vi 出队
        p=G->adjlist[i].firstedge;      //取 Vi 的边表头指针
        while(p)
            {                   //依次搜索 Vi 的邻接点 Vj（令 p->adjvex=j）
            if(!visited[p->adjvex]) {           //若 Vj 未访问过
            printf("%c",G->adjlist[p->adjvex].vertex);      //访问 Vj
            visited[p->adjvex]=TRUE;
                                            //以下 3 行代码有一处错误
            r=r+1; cq[r]=p->adjvex;             //访问过的 Vj 入队
            }
        p=p->next->next;                //找 Vi 的下一个邻接点
    }
    }//endwhile
}
//==========主函数==========
void main()
{
    int i;
    ALGraph *G;
    G=(ALGraph *)malloc(sizeof(ALGraph));
    CreatALGraph(G);
```

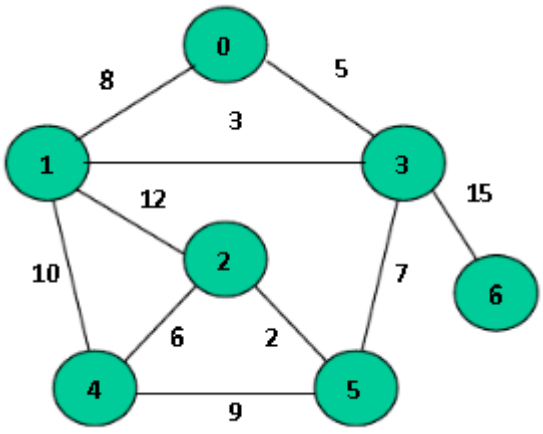
```
    printf("Print Graph DFS: ");
    DFS(G);
    printf("\n");
    printf("Print Graph BFS: ");
    BFS(G,3);
    printf("\n");
}
```

执行顺序：

```
Input VertexNum(n) and EdgesNum(e): 8, 9
Input Vertex string: 01234567
Input edges,Creat Adjacency List
0 1
0 2
1 3
1 4
2 5
2 6
3 7
4 7
5 6
Print Graph DFS: 02651473
Print Graph BFS: 37140265
```

二、实验任务

1. 改正程序中的错误并调试通过，测试并完成实验报告的撰写。
2. 选做题，编写程序输出该无向网络的最小生成树以及该最小生成树的所有边。



实验七、查找算法的实现

一、实验目的

掌握顺序和二分查找算法的基本思想及其实现方法。

二、实验要求

问题描述：对给定的任意数组（设其长度为 n），分别用顺序和二分查找方法在此数组中查找与给定值 k 相等的元素。

顺序查找基本思想：从查找表的一端开始，逐个将记录的关键字值和给定值进行比较，如果某个记录的关键字值和给定值相等，则称查找成功；否则，说明查找表中不存在关键字值为给定值的记录，则称查找失败。

二分查找基本思想：先取查找表的中间位置的关键字值与给定关键字值作比较，若它们的值相等，则查找成功；如果给定值比该记录的关键字值大，说明要查找的记录一定在查找表的后半部分，则在查找表的后半部分继续使用折半查找；若给定值比该记录的关键字值小，说明要查找的记录一定在查找表的前半部分，则在查找表的前半部分继续使用折半查找。…直到查找成功，或者直到确定查找表中没有待查找的记录为止，即查找失败。

两者比较：

（1）顺序查找的查找效率很低；但是对于待查记录的存储结构没有任何要求，既适用于顺序存储，又适用于链式存储；当待查表中的记录个数较少时，采用顺序查找法较好。

（2）二分查找的平均查找长度较小，查找速度快；但它只能用于顺序存储，不能用于链式存储；且要求表中的记录是有序的。对于不常变动的有序表，采用折半查找法是较为理想的。

三、算法思想与算法描述

1、顺序查找，在顺序表 R[0..n-1]中查找关键字为 k 的记录，成功时返回找到的记录位置，失败时返回-1，具体的算法如下所示：

```
int SeqSearch(SeqList R[],int n,KeyType k)
{
int i=0;
while(i<n&&R[i].key!=k)
{
printf("%d",R[i].key);
i++;
}
if(i>=n)
return -1;
else
{
printf("%d",R[i].key);
return i;
}
```

```
}
```

2、二分查找，在有序表 R[0..n-1]中进行二分查找，成功时返回记录的位置，失败时返回-1，具体的算法如下：

```
int BinSearch(SeqList R[],int n,KeyType k)
{
int low=0,high=n-1,mid,count=0;
while(low<=high)
{
mid=(low+high)/2;
printf(" 第 %d 次 查 找 ： 在 [ %d ,%d] 中 找 到 元 素 R[%d]:%d\n",++count,low,high,mid,R[mid].key);
if(R[mid].key==k)
return mid;
if(R[mid].key>k)
high=mid-1;
else
low=mid+1;
}
return -1;
}
```

四、实验任务

认真阅读与理解实验内容的具体要求，参考教材相关章节，编写实验程序并上机调试与测试，完成实验报告的撰写。

1.已知含有 10 个整数的查找表如下：（9，13，15，7，45，32，56，89，60，36），从键盘上输入一个整数，用顺序查找的方法在查找表中查找该整数。若存在，输出该元素的下标值，否则，给出相应的信息。

2.对有序数据表(5，7，9，12，15，18，20，22，25，30，100)，编写程序按折半查找方法查找 12 和 28。

实验八、排序算法的实现

一、实验目的

1. 掌握常用的排序方法，并掌握用高级语言实现排序算法的方法；

2. 深刻理解排序的定义和各种排序方法的特点，并能加以灵活应用；

3. 了解各种方法的排序过程及其时间复杂度的分析方法。

二、实验要求

统计成绩：给出 n 个学生的考试成绩表，每条信息由姓名和分数组成，试设计一个算法：

（1）按分数高低次序，打印出每个学生在考试中获得的名次，分数相同的为同一名次；

（2） 按名次列出每个学生的姓名与分数。

三、实验步骤

1.定义结构体。

```
Typedef   struct   student
{ char   name[8];
  int   score; }
```

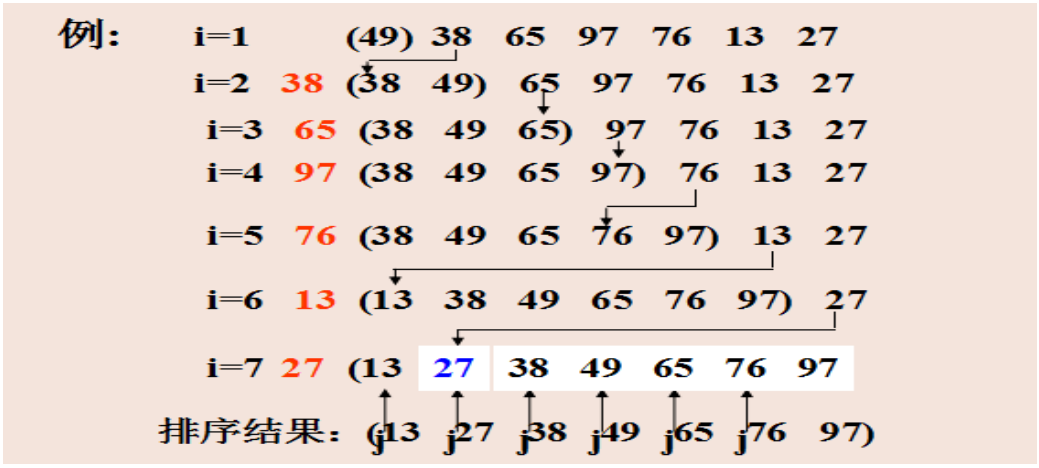
2.定义结构体数组。

3.编写主程序，对数据进行排序。

4.要求至少采用两种排序算法实现，如直接插入排序、快速排序等算法。

4.1 直接插入排序基本思想

排序过程：整个排序过程为 n-1 趟插入，即先将序列中第 1 个记录看成是一个有序子序列，然后从第 2 个记录开始，逐个进行插入，直至整个序列有序。



```
void InsertSort(RecordType r[],int n)
{
 int i,j;
 for(i=2;i<n;i++)          //执行了n-2次
  if (r[i].key<r[i-1].key)
  {  r[0]=r[i];
     j=i-1;                 //将带插入记录存放到监视哨中
        while(r[0].key<r[j].key)   //寻找插入位置
        {r[j+1]=r[j];
         j=j-1; }           //记录后移
      r[j+1]=r[0]    }      //将待插入记录插入正确位置
}
```

4.2 快速排序基本思想

基本思想：通过一趟排序，将待排序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，则可分别对这两部分记录进行排序，以达到整个序列有序。

排序过程：对 r[s……t]中记录进行一趟快速排序，附设两个指针 i 和 j，设 rp=r[s]，x=rp.key。初始时令 i=s,j=t。首先从 j 所指位置向前搜索第一个关键字小于 x 的记录，并和 rp 交换。再从 i 所指位置起向后搜索，找到第一个关键字大于 x 的记录，和 rp 交换。重复上述两步，直至 i=j 为止。再分别对两个子序列进行快速排序，直到每个子序列只含有一个记录为止。

下面举例来进行说明，主要有三个参数，i为区间的开始地址，j为区间的结束地址，X为当前的开始的值

第一步，i=0,j=9,X=21

0	1	2	3	4	5	6	7	8	9
21	32	43	98	54	45	23	4	66	86

第二步，从j开始由，后向前找，找到比X小的第一个数a[7]=4,此时i=0,j=6,X=21
进行替换

0	1	2	3	4	5	6	7	8	9
4	32	43	98	54	45	23	**21**	66	86

第三步，由前往后找，找到比X大的第一个数a[1]=32,此时i=2,j=6,X=21

0	1	2	3	4	5	6	7	8	9
4	**21**	43	98	54	45	23	32	66	86

第四步，从j=6开始由，由后向前找，找到比X小的第一个数a[0]=4,此时i=2,j=0,X=21，发现j<=i,所以第一回结束

可以发现**21**前面的数字都比21小，后面的数字都比21大
接下来对两个子区间[0,0]和[2,9]重复上面的操作即可

```
int Partition (SqList &L, int low, int high) {
    pivotkey = L.r[low].key;
    while (low < high) {
            while (low < high  && L.r[high].key >= pivotkey)  - -high;
            L.r[low] ↔ L.r[high];        //两个记录互换位置
            while (low < high  && L.r[low].key <= pivotkey)  ++low;
            L.r[low] ↔ L.r[high];        //两个记录互换位置
    }
    return low;
}// Partition
```

```
void QuickSort(SqList &L, int low, int high)  {
    if (low < high) {
            pivotloc = Partition(L, low, high);
            QuickSort(L, low, pivotloc-1);    //对低端子表递归调用本函数
            QuickSort(L, pivotloc+1, high);   //对高端子表递归调用本函数
    }
}//QuickSort
```

四、实验任务

认真阅读与理解实验内容的具体要求，参考教材相关章节，编写实验程序并上机调试与测试，完成实验报告的撰写。