深圳大学管理学院实验报告

课程名称：数据结构

实验项目（实验类型）名称：期末实验报告

学院：管理学院

专业：

指导教师：宋广为

报告人：学号：

实验时间：2024 年 6 月 26 日

实验报告提交时间：2024 年 6 月 26 日

一、实验目的与要求：（指导老师可将要求填入此部分）

内容：

1、完成顺序表的数据结构构造及相关算法；

2、完成链表的数据结构构造及相关算法；

3、完成堆栈的数据结构构造及相关算法（包括顺序栈和链栈）；

4、完成队列的数据结构构造及相关算法（包括顺序队列和链队列）；

5、完成串操作的 BF 算法和 KMP 算法

6、完成链表描述的二叉树的先序递归创建和各序遍历过程代码

7、完成一个哈夫曼二叉树的应用：给定多个字符值以及其相应的在某文本中的出现概率，求出不同字符的前缀编码。

8、完成二叉树的中序线索化算法

9、 完成某二叉树中序线索化后查找某节点的前驱和后继算法

10、完成图的深度优先遍历算法

11、完成一个特定图的 Prim 算法（选做）

要求：

1、独立完成；

2、提交实验报告，在报告中分成 10 个部分分别详细描述

3、每个部分都要在报告中贴上全部的源代码及测试结果；

4、报告使用下面提供的模板来写，其中：

实验目的和要求：题目和要求；

实验工具（实验环境、软件或语言）：描述所使用的系统和软件

实验过程和内容：直接粘贴的源代码，可以包含注释 ，同时将结果截图贴在这部分

问题分析与心得：主要讲各部分算法编写时候的心得体会。

二、实验工具（实验环境、软件或语言）：

1.系统: Windows11

2.软件：Visual Studio Code

3.语言: C++

三、实验过程及内容：

1、完成顺序表的数据结构构造及相关算法：SeqList.cpp

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

const int MAXSIZE = 100;
typedef int ElemType;
typedef struct
{
    ElemType data[MAXSIZE];
    int length;
}SqeList;

void InitSL(SqeList&); //初始化顺序表
void CreateSL(SqeList&,int); //生成顺序表
void ShowSL(SqeList); //遍历输出顺序表
int LengthSL(SqeList); //查询顺序表长度
int FindSL(SqeList,ElemType); //按值查找元素
void InsertSL(SqeList&,int,ElemType); //插入元素
void DeleteSL(SqeList&,int); //删除元素

int main(int argc, char **argv)
{
    SqeList SL;
    InitSL(SL);

    int len = 0;
    cout << "请输入顺序表长度:";
    cin >> len;
    CreateSL(SL,len);
    ShowSL(SL);

    cout << "顺序表的长度为:" << LengthSL(SL) << endl;

    ElemType Finde;
    cout << "请输入需要查询的元素:";
    cin >> Finde;
    cout <<  "该元素是顺序表中第" << FindSL(SL,Finde) << "个元
```

```
素(返回0则说明顺序表中不存在该元素)" << endl;

    int index;
    ElemType Inserte;
    cout << "请输入将在第几个位置插入新元素:";
    cin >> index;
    cout << "请输入需要插入的元素值:";
    cin >> Inserte;
    InsertSL(SL,index,Inserte);

    int DeleteIndex;
    cout << "请输入需要删除第几个元素:";
    cin >> DeleteIndex;
    DeleteSL(SL,DeleteIndex);

    return 0;
}

//初始化顺序表
void InitSL(SqeList& SL)
{
    SL.length = 0;
}

//创建顺序表
void CreateSL(SqeList& SL,int len)
{
    if(len > MAXSIZE)
        cout << "顺序表已满" << endl;
    SL.length = len;
    for(int i = 0;i < len;i++)
    {
        cout << "请输入索引为" << i << "的元素（该元素是第" << i+1
<< "个元素）:";
        cin >> SL.data[i];
    }
}
```

```cpp
//输出顺序表
void ShowSL(SqeList SL)
{
    cout << "顺序表为:";
    for(int i = 0;i< SL.length;i++)
        cout << SL.data[i] << " ";
    cout << endl;
}

//查询顺序表长度
int LengthSL(SqeList SL)
{
    return SL.length;
}

//查找元素
int FindSL(SqeList SL,ElemType e)
{
    if(SL.length == 0)
    {
        cout << "顺序表已空，操作失败";
        return -1;
    }
    else
    {
        for(int i = 0;i< SL.length;i++)
        {
            if(SL.data[i] == e)
                return i+1;
        }
        return 0;
    }
}

//插入元素
void InsertSL(SqeList& SL,int index,ElemType e)
{
    if(SL.length + 1 > MAXSIZE)
```

```
        cout << "顺序表已满，操作失败" << endl;
    else
    {
        SL.length += 1;
        for(int i = SL.length - 1;i >= index;i--)
        {
            SL.data[i] = SL.data[i - 1];
        }
        SL.data[index - 1] = e;
    }
    ShowSL(SL);
}

//删除元素
void DeleteSL(SqeList& SL,int index)
{
    if(SL.length == 0)
        cout << "顺序表已空，操作失败" << endl;
    else
    {
        for(int i = index;i < SL.length;i++)
        {
            SL.data[i - 1] = SL.data[i];
        }
        SL.length--;
        ShowSL(SL);
    }
}
```

```
请输入顺序表长度:5
请输入索引为0的元素（该元素是第1个元素）:1
请输入索引为1的元素（该元素是第2个元素）:2
请输入索引为2的元素（该元素是第3个元素）:3
请输入索引为3的元素（该元素是第4个元素）:4
请输入索引为4的元素（该元素是第5个元素）:5
顺序表为:1 2 3 4 5
顺序表的长度为:5
请输入需要查询的元素:3
该元素是顺序表中第3个元素(返回0则说明顺序表中不存在该元素)
请输入将在第几个位置插入新元素:2
请输入需要插入的元素值:66
顺序表为:1 66 2 3 4 5
请输入需要删除第几个元素:3
顺序表为:1 66 3 4 5
```

2、完成链表的数据结构构造及相关算法：LinkList.cpp

```cpp
#include <iostream>
using namespace std;

typedef int Elem;
typedef struct Node
{
    Elem data;
    struct Node* next;
}Node,*LinkList;

void Init_Link(LinkList&); //初始化链表
void Creat_Link(LinkList&, Elem datalist[], int); //创建链表
void Show_Link(LinkList&); //遍历输出链表
int Length_Link(LinkList&); //查询链表长度
int Locat_Elem(LinkList&, Elem); //按值查找元素
void Insert_Elem(LinkList&, Elem, int); //插入新元素
void Delete_Index(LinkList&, int); //按索引删除元素
void Destroy_Link(LinkList&); //销毁链表
void UpsideDown_Link(LinkList&); //逆转链表

int main(int argc, char **argv)
{
```

```
    LinkList A;
    Init_Link(A);

    Elem Data[50] = {10,2,36,55,7,20,31,1,5};
    int n;

    cout << "请输入你需要的链表长度:";
    cin >> n;
    Creat_Link(A,Data,n);


    Show_Link(A);
    cout << endl;


    cout << "链表长度为:" << Length_Link(A) << endl;
    cout << endl;


    Elem e;
    cout << "请输入你想要查找的元素值:";
    cin >> e;
    cout << "该元素所在链表中的位置为:" << Locat_Elem(A,e) <<
endl;
    cout << endl;


    cout << "请输入你想要插入的元素值:";
    cin >> e;
    cout << "请输入你想要在链表中哪个位置插入该元素:";
    cin >> n;
    Insert_Elem(A,e,n);
    cout << endl;


    cout << "请输入你想要删除的元素的索引值:";
    cin >> n;
    Delete_Index(A,n);
    cout << endl;
```

```
    char button;
    cout << "如果你想要销毁链表，输入'y' or 'Y';否则输入'n' or
'N'";
    cin >> button;
    if(button == 'Y')
        Destroy_Link(A);
    else
        Show_Link(A);
    cout << endl;


    UpsideDown_Link(A);

    return 0;
}


void Init_Link(LinkList& L)
{
    L = new Node;
    L -> next = NULL;
    cout << "链表已初始化！" << endl;
}


void Creat_Link(LinkList& L, Elem datalist[], int n)
{
    LinkList temp;
    L = new Node;
    L -> next = NULL;
    for(int i = n - 1; i >= 0; i--)
    {
        temp = new Node;
        temp -> data = datalist[i];
        temp -> next = L -> next;
        L -> next = temp;
    }
```

```
}

void Show_Link(LinkList& L)
{
    LinkList temp = L -> next;
    cout << "链表为:";
    while(temp)
    {
        cout << temp -> data << " ";
        temp = temp -> next;
    }
    cout << endl;
}

int Length_Link(LinkList& L)
{
    LinkList temp = L -> next;
    int len = 0;
    while(temp)
    {
        len++;
        temp = temp -> next;
    }
    return len;
}

int Locat_Elem(LinkList& L, Elem element)
{
    LinkList temp = L -> next;
    int index = 1;
    while(temp && temp -> data != element)
    {
        temp = temp -> next;
        index++;
    }
    if(temp)
        return index;
    else
```

```
        return 0;
}

void Insert_Elem(LinkList& L, Elem element, int index)
{
    LinkList temp = L;
    LinkList s;
    for(int i = 1;i <= index - 1;i++)
        temp = temp -> next;
    s = new Node;
    s -> data = element;
    s -> next = temp -> next;
    temp -> next = s;
    cout << "插入操作已完成!";
    Show_Link(L);
}

void Delete_Index(LinkList& L, int index)
{
    LinkList temp = L;
    LinkList qtemp;
    if(index > Length_Link(L) || index <= 0)
    {
        cout << "该输入索引非法!" << endl;
        exit(1);
    }
    int i = 1;
    while(temp && i <= index - 1)
    {
        temp = temp -> next;
        i++;
    }
    qtemp = temp -> next;
    temp -> next = qtemp -> next;
    delete qtemp;
    cout << "删除操作已完成!";
    Show_Link(L);
}
```

```
void Destroy_Link(LinkList& L)
{
    LinkList temp = L;
    LinkList qtemp;
    while(temp)
    {
        qtemp = temp;
        temp = temp -> next;
        delete qtemp;
    }
    cout << "链表已被销毁!" << endl;
}

void UpsideDown_Link(LinkList& L)
{
    LinkList temp,s;
    temp = L -> next;
    L -> next = NULL;
    while(temp)
    {
        s = temp;
        temp = temp -> next;
        s -> next = L -> next;
        L -> next = s;
    }
    cout << "链表已被逆置!";
    Show_Link(L);
}
```

```
链表已初始化!
请输入你需要的链表长度:4
链表为:10 2 36 55

链表长度为:4

请输入你想要查找的元素值:10
该元素所在链表中的位置为:1

请输入你想要插入的元素值:332
请输入你想要在链表中哪个位置插入该元素:1
插入操作已完成!链表为:332 10 2 36 55

请输入你想要删除的元素的索引值:2
删除操作已完成!链表为:332 2 36 55

如果你想要销毁链表，输入'y' or 'Y';否则输入'n' or 'N'n
链表为:332 2 36 55

链表已被逆置!链表为:55 36 2 332
```

3、完成堆栈的数据结构构造及相关算法（包括顺序栈和链栈）：顺序栈 SeqStack.cpp、

```cpp
#include <iostream>
using namespace std;
const int MAXSIZE=100;

typedef int ElemType;
typedef struct
{
    ElemType data[MAXSIZE];
    int top;
}SeqStack;

void InitSeqStack(SeqStack&); //初始化顺序栈
void PushSeqStack(SeqStack&,ElemType); //入栈
ElemType PopSeqStack(SeqStack&); //出栈
ElemType TopElemSeqStack(SeqStack&); //取栈顶元素
```

```cpp
bool EmptySeqStack(SeqStack&); //判断满栈
bool NoneSeqStack(SeqStack&); //判断空栈

int main(int argc, char **argv)
{
    SeqStack S;
    InitSeqStack(S);

    PushSeqStack(S,9);
    PushSeqStack(S,16);
    PushSeqStack(S,27);
    cout << "正在对栈顶元素" << PopSeqStack(S) << "进行出栈操作
" << endl;
    cout << "正在对栈顶元素" << PopSeqStack(S) << "进行出栈操作
" << endl;

    cout << "栈顶元素为" << TopElemSeqStack(S) << endl;

    cout << "栈是否已满(返回 1 则已满，返回 0 未满):" <<
EmptySeqStack(S) << endl;
    cout << "栈是否为空(返回 1 则已空，返回 0 未空):" <<
NoneSeqStack(S) << endl;

    return 0;
}

void InitSeqStack(SeqStack& S)
{
    S.top = -1;
    cout << "栈已初始化！" << endl;
}

void PushSeqStack(SeqStack& S,ElemType e)
{
    if(S.top == MAXSIZE -1)
    {
        cout << "该栈已满" << endl;
        exit(1);
```

```
    }
    cout << "正在对元素" << e << "进行入栈操作" << endl;
    S.top++;
    S.data[S.top] = e;
}

ElemType PopSeqStack(SeqStack& S)
{
    if(S.top == -1)
    {
        cout << "该栈已空" << endl;
        exit(1);
    }
    ElemType x = S.data[S.top];
    S.top--;
    return x;
}

ElemType TopElemSeqStack(SeqStack& S)
{
    if(S.top == -1)
    {
        cout << "该栈已空" << endl;
        exit(1);
    }
    ElemType top_element = S.data[S.top];
    return top_element;
}

bool EmptySeqStack(SeqStack& S)
{
    if(S.top == MAXSIZE - 1)
        return true;
    else
        return false;
}

bool NoneSeqStack(SeqStack& S)
```

```
{
    if(S.top == -1)
        return true;
    else
        return false;
}
```

```
栈已初始化!
正在对元素9进行入栈操作
正在对元素16进行入栈操作
正在对元素27进行入栈操作
正在对栈顶元素27进行出栈操作
正在对栈顶元素16进行出栈操作
栈顶元素为9
栈是否已满(返回1则已满，返回0未满):0
栈是否为空(返回1则已空，返回0未空):0
```

链栈 LinkStack.cpp

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

typedef int ElemType;
typedef struct Node
{
    ElemType data;
    struct Node* next;
}*LinkStack;

void InitLinkStack(LinkStack&); //初始化链栈
void PushLinkStack(LinkStack&,ElemType); //入栈
ElemType PopLinkStack(LinkStack&); //出栈
ElemType TopElemLinkStack(LinkStack&); //取栈顶元素
bool NoneLinkStack(LinkStack&); //判断栈空
void Destroy(LinkStack&); //销毁链栈
```

```cpp
int main(int argc, char **argv)
{
    LinkStack LS;
    InitLinkStack(LS);

    PushLinkStack(LS,50);
    PushLinkStack(LS,26);
    PushLinkStack(LS,16);
    PushLinkStack(LS,9);

    cout << "正在弹出栈顶元素:" << PopLinkStack(LS) << endl;
    cout << "正在弹出栈顶元素:" << PopLinkStack(LS) << endl;

    cout << "正在查询栈顶元素: " << TopElemLinkStack(LS) << endl;

    cout << "正在判断链栈是否为空(返回 1 为空，返回 0 则未空): " <<
NoneLinkStack(LS) << endl;

    cout << "正在销毁链栈" << endl;
    Destroy(LS);

    return 0;
}

void InitLinkStack(LinkStack& LStack)
{
    LStack = NULL;
}

void PushLinkStack(LinkStack& LStack,ElemType e)
{
    LinkStack temp = new Node;
    temp -> data = e;
    temp -> next = LStack;
    LStack = temp;
    cout << "正在对元素" << e << "进行入栈操作" << endl;
}
```

```
ElemType PopLinkStack(LinkStack& LStack)
{
    if(LStack == NULL)
    {
        cout << "链栈已空" << endl;
        exit(1);
    }
    ElemType x = LStack -> data;
    LinkStack temp = LStack;
    LStack = temp -> next;
    delete temp;
    return x;
}

ElemType TopElemLinkStack(LinkStack& LStack)
{
    if(LStack == NULL)
    {
        cout << "链栈已空，栈顶元素不存在！" << endl;
        exit(1);
    }
    return LStack -> data;
}

bool NoneLinkStack(LinkStack& LStack)
{
    if(LStack == NULL)
        return true;
    else
        return false;
}

void Destroy(LinkStack& LStack)
{
    LinkStack temp;
    while(LStack)
    {
```

```
        temp = LStack;
        LStack = LStack -> next;
        delete temp;
    }
    cout << "链栈已被销毁" << endl;
}
```

```
正在对元素50进行入栈操作
正在对元素26进行入栈操作
正在对元素16进行入栈操作
正在对元素9进行入栈操作
正在弹出栈顶元素:9
正在弹出栈顶元素:16
正在查询栈顶元素: 26
正在判断链栈是否为空(返回1为空，返回0则未空): 0
正在销毁链栈
链栈已被销毁
```

4、完成队列的数据结构构造及相关算法（包括顺序队列和链队列）：顺序队列 SeqQueue.cpp

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

const int MAXSIZE = 100;
typedef int ElemType;
typedef struct
{
    ElemType data[MAXSIZE];
    int front;
    int rear;
}SeqQueue;

void InitSeqQueue(SeqQueue&); //初始化队列
void EnSQueue(SeqQueue&,ElemType); //入队
ElemType DeSQueue(SeqQueue&); //出队
void ShowSQueue(SeqQueue&); //遍历输出队列
bool QueueEmpty(SeqQueue&); //判断队列是否为空
```

```cpp
bool QueueFull(SeqQueue&); //判断队列是否已满

int main(int argc, char **argv)
{
    SeqQueue S;
    InitSeqQueue(S);
    EnSQueue(S,11);
    EnSQueue(S,22);
    EnSQueue(S,33);
    ShowSQueue(S);
    DeSQueue(S);
    DeSQueue(S);
    DeSQueue(S);
    cout << "队列是否为空（输出 1 为空，输出 0 则未空）：" <<
QueueEmpty(S) << endl;
    cout << "队列是否已满（输出 1 已满，输出 0 则未满）：" <<
QueueFull(S) << endl;
    return 0;
}

void InitSeqQueue(SeqQueue& S)
{
    S.front = 0;
    S.rear = 0;
}

void EnSQueue(SeqQueue& S,ElemType e)
{
    if((S.rear+1)%MAXSIZE == S.front)
    {
        cout << "队列已满" << endl;
        exit(1);
    }
    cout << "正在入队的元素：" << e << endl;
    S.rear = (S.rear + 1) % MAXSIZE;
    S.data[S.rear] = e;
}
```

```cpp
ElemType DeSQueue(SeqQueue& S)
{
    if(S.front == S.rear)
    {
        cout << "队列已空" << endl;
        exit(1);
    }
    S.front = (S.front + 1) % MAXSIZE;
    ElemType x = S.data[S.front];
    cout << "出队的元素是: " << x << endl;
    return x;
}

void ShowSQueue(SeqQueue& S)
{
    if(S.front == S.rear)
    {
        cout << "None" << endl;
        exit(1);
    }
    cout << "队列为:";
    for(int i = S.front + 1;i <= S.rear;i++)
        cout << S.data[i] << " ";
    cout << endl;
}

bool QueueEmpty(SeqQueue &S)
{
    return S.front == S.rear;
}

bool QueueFull(SeqQueue &S)
{
    return (S.rear + 1) % MAXSIZE == S.front;
}
```

```
正在入队的元素: 11
正在入队的元素: 22
正在入队的元素: 33
队列为:11 22 33
出队的元素是: 11
出队的元素是: 22
出队的元素是: 33
队列是否为空（输出1为空，输出0则未空）: 1
队列是否已满（输出1已满，输出0则未满）: 0
```

链队列 LinkQueue.cpp

```cpp
#include <iostream>
using namespace std;

typedef int ElemType;
typedef struct Node
{
    ElemType data;
    struct Node *next;
}Node;
typedef struct
{
    Node *front;
    Node *rear;
}LinkQueue;

void InitLQueue(LinkQueue&); //初始化
void EnLQueue(LinkQueue&,ElemType); //入队
ElemType DeLQueue(LinkQueue&); //出队
bool LQueueEmpty(LinkQueue&); //判断队空
void DestroyListQueue(LinkQueue&); //销毁队列

int main(int argc, char **argv)
{
    LinkQueue L;
    InitLQueue(L);
```

```
    EnLQueue(L,11);
    EnLQueue(L,22);
    EnLQueue(L,33);
    DeLQueue(L);
    cout << "队列是否已空（返回 1 则已空，返回 0 则未空）： " <<
LQueueEmpty(L) << endl;
    DestroyListQueue(L);
    return 0;
}

void InitLQueue(LinkQueue& L)
{
    L.front = L.rear = new Node;
    L.front -> next = NULL;
}

void EnLQueue(LinkQueue& L,ElemType e)
{
    Node *temp = new Node;
    temp -> data = e;
    temp -> next = NULL;
    L.rear -> next = temp;
    L.rear = temp;
    cout << "正在入队的元素是:" << e << endl;
}

ElemType DeLQueue(LinkQueue& L)
{
    if(L.front == L.rear)
    {
        cout << "队列已空" << endl;
        exit(1);
    }
    Node *temp = L.front -> next;
    ElemType x = temp -> data;
    cout << "正在出队的元素是: " << x << endl;
    L.front = temp -> next;
    if(L.rear == temp)
```

```
        L.rear = L.front;
    delete temp;
}

bool LQueueEmpty(LinkQueue &L)
{
    return L.front == L.rear;
}

void DestroyListQueue(LinkQueue &L)
{
    while(L.front)
    {
      L.rear=L.front->next;
      delete L.front;
      L.front=L.rear;
    }
    cout << "队列已销毁" << endl;
}
```

```
正在入队的元素是:11
正在入队的元素是:22
正在入队的元素是:33
正在出队的元素是: 11
队列是否已空（返回1则已空，返回0则未空）: 0
队列已销毁
```

5、完成串操作的 BF 算法和 KMP 算法

BF 算法 BF.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int MaxSize = 100;

int BF(char *major, char *key)
{
```

```
    int i = 0;
    int j = 0;
    while(i < strlen(major) && j < strlen(key))
    {
        if(major[i] == key[j])
        {
            //开始匹配，标记点移动
            i++;
            j++;
        }
        else
        {
            //匹配中断，i 回到上一轮匹配开始的位置+1，j 置 0
            i = i - j + 1;
            j = 0;
        }
    }
    if(j >= strlen(key))
        return i - j;  //匹配成功，返回开始匹配的初始位置
    else
        return -1;  //匹配不成功，返回-1
}

int main(int argc, char **argv)
{
    char major[MaxSize],key[MaxSize];
    int flag;

    cout << "请输入主串：";
    cin >> major;
    cout << "请输入需要匹配的子串：";
    cin >> key;

    flag = BF(major,key);
    if(flag == -1)
        cout << "匹配失败\n";
    else
        cout << "匹配成功，匹配位置为：" << flag << endl;
```

```
    return 0;
}
```

```
请输入主串：aabcaabbac
请输入需要匹配的子串：bcaa
匹配成功，匹配位置为：2
```

```
请输入主串：aaavbasafasfa
请输入需要匹配的子串：sdvasadsaxz
匹配失败
```

KMP 算法 KMP.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int MaxSize = 100;

void get_next(char *key,int *next)
{

    next[0] = -1;
    int j = 0,k = -1;
    while(j < strlen(key))
    {
        if(k == -1 || key[j] ==key[k])
        {
            j++;
            k++;
            if(key[j] != key[k])
                next[j] = k;
            else
                next[j] = next[k];
        }
        else
            k = next[k];
    }
}

int KMP(char *major,char *key)
```

```
{
    int next[MaxSize];
    int i = 0,j = 0;
    get_next(key,next);
    while(i < strlen(major) && j < strlen(key))
    {
        if(j == -1 || major[i] == key[j])
        {
            i++;
            j++;
        }
        else
            j = next[j];
    }
    if(j >= strlen(key))
        return i - strlen(key);
    else
        return -1;

}


int main(int argc, char **argv)
{
    char major[MaxSize],key[MaxSize];

    cout << "请输入主串：";
    cin >> major;

    cout << "请输入需要匹配的子串：";
    cin >> key;

    int flag = KMP(major,key);
    if(flag == -1)
        cout << "匹配失败\n";
    else
        cout << "匹配成功，匹配位置为：" << flag << endl;
```

```
    return 0;
}
```

```
请输入主串：aabaabacab
请输入需要匹配的子串：abac
匹配成功，匹配位置为：4
```

6、完成链表描述的二叉树的先序递归创建和各序遍历过程代码 BinaryTree.cpp

```
#include <bits/stdc++.h>
using namespace std;

typedef char ElemType;
typedef struct BiNode
{
    ElemType data;
    BiNode *left,*right;
}*Bitree;

void Create(Bitree& T) //先序创建二叉树
{
    static int i = 0;
    char input[] =
{'a','b','c','#','d','#','#','#','e','f','#','#','g','h','#'
,'#','i','#','#'};

    char ch = input[i];
    i++;
    if(ch == '#')
        //如果输入的是#，则二叉树的节点为空节点
        T = NULL;
    else
    {
        T = new BiNode;
        T -> data = ch;
        Create(T -> left);
        Create(T -> right);
```

```
    }
}

void PreOrder(Bitree T) //先序遍历
{
    if(T != NULL){
        cout << T -> data << " ";
        PreOrder(T -> left);
        PreOrder(T -> right);
    }
}

void InOrder(Bitree T) //中序遍历
{
    if(T != NULL)
    {
        InOrder(T -> left);
        cout << T -> data << " ";
        InOrder(T -> right);
    }
}

void PostOrder(Bitree T) //后序遍历
{
    if(T != NULL)
    {
        PostOrder(T -> left);
        PostOrder(T -> right);
        cout << T -> data <<" ";
    }
}

int main(int argc, char **argv)
{
    Bitree T = NULL; //初始化二叉树
    Create(T);

    cout << endl << "先序遍历结果: " << endl;
```

```
    PreOrder(T);

    cout << endl << "中序遍历结果: " << endl;
    InOrder(T);

    cout << endl << "后序遍历结果: " << endl;
    PostOrder(T);

    return 0;
}
```

```
先序遍历结果:
a b c d e f g h i
中序遍历结果:
c d b a f e h g i
后序遍历结果:
d c b f h i g e a
```

7、完成一个哈夫曼二叉树的应用：给定多个字符值以及其相应的在某文本中的出现概率，求出不同字符的前缀编码 Huffman.cpp

```
#include <bits/stdc++.h>
using namespace std;

typedef struct HuffmanNode
{
    char data;
    int weight;
    int lchild,rchild,parent;
}HTNode,*HuffmanTree;

void select(HuffmanTree t,int i,int& s1,int& s2)
{
    int j;
```

```
    //定位第一个根节点下标起点
    for(j = 0;j < i;j++)
        if(t[j].parent == -1)
            break;
    //求 s1
    for(s1 = j,j++;j < i;j++)
        if(t[j].parent == -1 && t[j].weight < t[s1].weight)
            s1 = j;

    //定位第二个根节点下标起点
    for(j = 0;j < i;j++)
        if(t[j].parent == -1 && j != s1)//避开 s1
            break;
    //求 s2
    for(s2 = j,j++;j < i;j++)
        if(t[j].parent == -1 && j != s1 && t[j].weight <
t[s2].weight)
            s2 = j;
}

void CreateHT(HuffmanTree& HT,int n)
{
    //输入数据及权重
    char* input_char = new char[n];
    int*  input_weight = new int[n];
    cout << "please input char:" << endl;
    for(int i = 0;i < n;i++)
        cin >> input_char[i];
    cout << "please input weight:" << endl;
    for(int i = 0;i < n;i++)
        cin >> input_weight[i];

    int m = 2 * n - 1;
    HT = new HTNode[m];

    //初始化哈夫曼树
    for(int i = 0;i < n;i++)
    {
```

```
        HT[i].data = input_char[i];
        HT[i].weight = input_weight[i];
        HT[i].parent = -1;
        HT[i].lchild = -1;
        HT[i].rchild = -1;
    }
    for(int i = 0;i < m;i++)
        HT[i].parent = -1;

    //创建哈夫曼树
    int s1,s2;
    for(int i = n;i < m;i++)
    {
        select(HT,i,s1,s2);
        HT[s1].parent = HT[s2].parent = i;
        HT[i].lchild = s1;
        HT[i].rchild = s2;
        HT[i].weight = HT[s1].weight + HT[s2].weight;
    }

    delete input_char;
    delete input_weight;
}

void Code(HuffmanTree HT,int n,int i,char* code)
{
    char *temp = new char[n];

    //从后往前回溯对字符进行编码
    temp[n-1] = '\0';
    int start = n - 1;

    int parent = HT[i].parent;
    int p = i;//初始从当前点出发
    while(parent != -1)
    {
        if(HT[parent].lchild == p)
            temp[--start] = '0';
```

```
        else
            temp[--start] = '1';
        p = parent;
        parent = HT[parent].parent;//从父亲上溯
    }
    strcpy_s(code,strlen(&temp[start])+1,&(temp[start]));

    delete temp;
}

int main(int argc, char **argv)
{
    int n;
    HuffmanTree HT;
    cout << "please input the number of char:";
    cin >> n;

    CreateHT(HT,n);
    //显示数组
    for(int t = 0;t < 2 * n - 1;t++)
        cout << HT[t].weight << " " << HT[t].parent << " " <<
HT[t].lchild << " " << HT[t].rchild << endl;

    //求第 i 个字符的编码
    char* code = new char[n];

    for(int i = 0;i < n;i++)
    {
        Code(HT,n,i,code);
        cout << HT[i].data << "->" << code << endl;
    }

    delete code;
    delete HT;

    return 0;
}
```

```
please input the number of char:3
please input char:
a
b
c
please input weight:
2
3
5
2 3 -1 -1
3 3 -1 -1
5 4 -1 -1
5 4 0 1
10 -1 2 3
a->10
b->11
c->0
```

8、完成二叉树的中序线索化算法

9、完成某二叉树中序线索化后查找某节点的前驱和后继算法

ThreadTree.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef char ElemType;
enum flag{Child,Thread};
typedef struct BiNode
{
    ElemType data;
    BiNode *left,*right;
    flag ltype,rtype;
}*Bitree;

void Create(Bitree& t) //先序创建二叉树
{
    static int i =0;
```

```
    char input[] =
{'a','b','c','#','d','#','#','#','e','f','#','#','g','h','#'
,'#','i','#','#'};
    char ch = input[i];
    i++;

    if(ch == '#')
    {
        //如果输入的是#，则二叉树的节点为空节点
        t=NULL;
        return;
    }
    else
        //如果输入不是#，则分配节点空间
        if(!(t = new BiNode))
        {
            cout << "分配失败" << endl;
            return;
        }
        else
        {
            //分配空间成功
            t -> data = ch;

            //先将左右指针类型设为孩子类型
            t -> ltype = Child;
            t -> rtype = Child;

            Create(t -> left);
            Create(t -> right);
        }
}

void PreOrder(Bitree t) //先序遍历
{
    if(t != NULL)
    {
        cout << t -> data << " ";
```

```
        PreOrder(t -> left);
        PreOrder(t -> right);
    }
}

void InOrder(Bitree t) //中序遍历
{

    if(t != NULL)
    {
        InOrder(t -> left);
        cout << t -> data << " ";
        InOrder(t -> right);
    }
}
void PostOrder(Bitree t) //后序遍历
{
    if(t != NULL)
    {
        PostOrder(t -> left);
        PostOrder(t -> right);
        cout << t -> data << " ";
    }
}

void InThreaded(Bitree &t) //中序线索化
{
    //prenode 变量设置为静态变量，确保每次递归正确
    static Bitree prenode = NULL;
    if(t)
    {
        InThreaded(t -> left); //处理左子树

        //处理根节点
        if(!t -> left)
        {
            t -> ltype = Thread;
            t -> left = prenode;
```

```
        }

        if(prenode && !prenode -> right)
        {
            prenode -> rtype = Thread;
            prenode -> right = t;
        }
        prenode = t;

        InThreaded(t -> right); //处理右子树
    }
}


BiNode* InOrder_Post(BiNode *p) //线索化后查找某节点的后继
{
    BiNode *q;
    if(p -> rtype == Thread)
        return p -> right;
    else
    {
        q = p -> right;
        while(q -> ltype == Child)
            q = q -> left;
        return q;
    }
}

BiNode* InOrder_Pre(BiNode *p) //线索化后查找某节点的前驱
{
    BiNode *q;
    if(p -> ltype == Thread)
        return p -> left;
    else
    {
        q = p -> left;
        while(q -> rtype == Child)
            q = q -> right;
```

```
        return q;
    }
}

void DestroyTree(Bitree t) //销毁二叉树
{
    if(t)
    {
        if(t -> ltype == Child)
            DestroyTree(t -> left);
        if(t -> rtype = Child)
            DestroyTree(t -> right);
        delete t;
        t = NULL;
    }
}

int main(int argc, char **argv)
{
    Bitree tree = NULL;
    Create(tree);

    cout << endl << "先序遍历" << endl;
    PreOrder(tree);
    cout << endl << "中序遍历" << endl;
    InOrder(tree);
    cout << endl << "后序遍历" << endl;
    PostOrder(tree);

    InThreaded(tree);
    BiNode *temp = InOrder_Pre(tree);
    cout << endl << "根节点的前驱是" << temp -> data << endl;
    temp=InOrder_Post(tree);
    cout << "根节点的后继是" << temp -> data << endl;

    DestroyTree(tree);

    return 0;
```

```
}
```

```
先序遍历
a b c d e f g h i
中序遍历
c d b a f e h g i
后序遍历
d c b f h i g e a
根节点的前驱是b
根节点的后继是f
```

10、完成图的深度优先遍历算法 DFS.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

const int MAX = 100;
enum GraphType{DG,UG,DN,UN}; //图的类型定义:有向图，无向图，有向网，无向网
typedef char VertexType;
bool visited[MAX];

typedef struct
{
    VertexType vexs[MAX]; //顶点表
    int arcs[MAX][MAX]; //邻接矩阵
    int vexnum,arcnum; //顶点数和边数
    GraphType kind; //图的类型
}MGraph;

void CreateMatrix(MGraph& G)
{
    G.vexnum = 7; //7 个顶点
    G.arcnum = 12; //12 条边
    char temp[] = {'A','B','C','D','E','F','G'};
```

```
    for(int i = 0;i < G.vexnum;i++)
        G.vexs[i] = temp[i];

    //初始化邻接矩阵
    for(int i = 0;i < G.vexnum;i++)
        for(int j = 0;j < G.vexnum;j++)
            G. arcs[i][j] = 0;

    G.arcs[0][1] = G.arcs[1][0] = 1;
    G.arcs[0][5] = G.arcs[5][0] = 1;
    G.arcs[0][6] = G.arcs[6][0] = 1;
    G.arcs[1][2] = G.arcs[2][1] = 1;
    G.arcs[1][5] = G.arcs[5][1] = 1;
    G.arcs[2][5] = G.arcs[5][2] = 1;
    G.arcs[2][4] = G.arcs[4][2] = 1;
    G.arcs[2][3] = G.arcs[3][2] = 1;
    G.arcs[3][4] = G.arcs[4][3] = 1;
    G.arcs[4][6] = G.arcs[6][4] = 1;
    G.arcs[4][5] = G.arcs[5][4] = 1;
    G.arcs[6][5] = G.arcs[5][6] = 1;

    G.kind=UG;
}

void DFS(MGraph G,int v)
{
    cout << G.vexs[v];
    visited[v] = true;
    for(int i = 0;i<G.vexnum;i++)
    {
        if(G.arcs[v][i] != 0 && !visited[i])
            DFS(G,i);
    }
}

void DFS_Graph(MGraph G)
{
    cout << "Output is:";
```

```
    for(int i = 0;i < G.vexnum;i++)
        visited[i] = false;

    for(int i= 0;i < G.vexnum;i++)
    {
        if(!visited[i])
            DFS(G, i);
    }
}

int main(int argc, char **argv)
{
    MGraph G;
    CreateMatrix(G);
    DFS_Graph(G);

    return 0;
}
```

```
Output is:ABCDEFG
```

11、完成一个特定图的 Prim 算法（选做）

四、问题分析与心得：

1.在对顺序表做插入操作时要注意顺序表的长度是否会超过最大限度，同时要注意将插入位置后的元素进行移动。

2.在对链表进行插入、删除操作时，要注意指针的指向，保证链表未处理的部分不断开。

3.对栈进行操作时，要注意栈的元素是先进的后出，先入栈的会到达栈底，先出栈的是栈顶。

4.在对队列元素进行操作时，是头出尾入，即先进入的会先出。对于顺序队列，因为数组是静态申请的，故存在假上溢问题，利用循环队列解决该问题。

5.在计算 **KMP** 的 **next** 数组时，要注意每次递归时的位置和当前字符值是否匹配。

6.创建二叉树时，对于输入为“**#**”，则该节点为空节点，非空节点要分配空间并写入数据；创建和遍历时都用到了递归操作。

7.在对哈夫曼树进行编码时，对 **HT[parent].lchild(rchild)**进行判断来确定该步编码为‘**0**’还是‘**1**’。

8.在中序线索化二叉树时，利用两个指针 **p** 和 **pre**，让其在中序遍历的过程中分别指向一前一后，也就是说始终保持 **p** 是 **pre** 的后继，**pre** 是 **p** 的前驱。

9.在完成某二叉树中序线索化后查找某节点的前驱和后继时，对中序而言，查找前驱：从左开始，一路向右，最后一个右子树为空时，这个节点就是前驱；对于查找后继：从右开始，一路向左，最后一个左子树为空时，这个节点就是后继。

10.在对图进行深度优先遍历时，是从未被访问的邻接点中选取一个顶点 ***i***，从 ***i*** 出发进行深度优先遍历。深度优先遍历是树的先序遍历的推广。

指导教师批阅意见：

成绩评定：

指导教师签字：

年　　月　　日

备注：