第二章线性表

一、顺序表：SeqList.h，SeqList.cpp

**1**、**SeqList.h**

#ifndef SEQLIST\_H\_

#define SEQLIST\_H\_

#include <iostream>

using namespace std;

const int defaultSize = 100;

template <class T>

class SeqList {

private:

T \*data; // 存放数组

int maxSize; // 最大可容纳表项的项数

int last; // 当前已存表项数

public:

SeqList(int sz = defaultSize); // 构造函数

~SeqList( ) { delete[ ] data;} // 析构函数

int Length( ) const {return last + 1;} // 计算表长度

void makeEmpty( ) { last = -1; } // 把表置空

int Search(T x) const; // 在表中搜索x 的位置，返回位置序号

bool Insert(int i, T x); // 插入

T\* getData(const int i)const;

void output( )const;

};

#include "SeqList.cpp"

#endif

**2**、**SeqList.cpp**

#ifndef SEQLIST\_CPP\_

#define SEQLIST\_CPP\_

#include <iostream>

using namespace std;

#include "SeqList.h"

template <class T>

SeqList<T>::SeqList(int sz) { // 构造函数

if (sz > 0) {

maxSize = sz;

last = -1;

data = new T[maxSize]; // 创建表存储数组

if (data == NULL) { // 动态分配失败

cerr <<"存储分配错误！"<< endl;

exit(1);

}

}

}

template <class T>

int SeqList<T>::Search(T x) const { // 在表中搜索x 的位置，返回位置序号

for (int i = 0; i <= last; i++) // 顺序搜索

if ( data[i] == x )

return i + 1; // 表项序号和表项位置差1

return 0; // 搜索失败

}

template <class T>

bool SeqList<T>::Insert(int i, const T x) { // 插入

if (last == maxSize - 1) // 表满

return false;

if (i < 0 || i > last + 1) // 参数i 不合理

return false;

for (int j = last; j >= i; j--) // 依次后移

data[j + 1] = data[j];

data[i] = x; last++; // 插入

return true; // 插入成功

}

template <class T>

void SeqList<T>::output( )const {

cout <<"元素个数为"<< last + 1 <<" ："<< endl;

for(int i = 0; i <= last; i++)

cout << '#' << i+1 <<":"<< data[i] <<"; ";

cout << endl;

}

template <class T>

inline T\* SeqList<T>::getData(const int i)const {

return (i > 0 && i <= last + 1) ? &data[i - 1] : NULL;

}

#endif

二、单链表：List.h，List.cpp

**1**、**List.h**

#ifndef LIST\_H\_

#define LIST\_H\_

#include <iostream>

#include <cstdlib>

using namespace std;

template <class T>

struct LinkNode { // 链表结点类的定义

T data; // 数据域

LinkNode<T> \*link; // 链指针域

LinkNode( ) { link = NULL; } // 构造函数

LinkNode(T item, LinkNode<T> \*ptr = NULL) { // 构造函数

data = item;

link = ptr;

}

bool operator== (T x) { return data == x; } // 重载函数，判相等

bool operator!= (T x) { return data != x; }

};

template <class T>

class List { // 单链表类定义, 不用继承也可实现

private:

LinkNode<T> \*first; // 表头指针

public:

List( ); // 构造函数

List(const T x);

List(List<T>& L); // 复制构造函数

~List( ) { makeEmpty( ); delete first; } // 析构函数

void makeEmpty( ); // 将链表置为空表

int Length( ) const; // 计算链表的长度

LinkNode<T> \*Search(const T x)const; // 搜索含x 元素

LinkNode<T> \*Locate(const int i)const; // 定位第i 个元素

T \*getData(const int i)const; // 取出第i 元素值

bool setData(const int i, const T x); // 更新第i 元素值

bool Insert (const int i, const T x); // 在第i 元素后插入

bool Remove(const int i, T& x); // 删除第i 个元素

bool IsEmpty( ) const { return first->link == NULL;} // 判表空否

LinkNode<T> \*getHead( ) const { return first; }

void setHead(LinkNode<T> \*f ) { first = f; }

// void Sort( ); // 排序

void output( );

List<T>& operator=(List<T>& L) {

LinkNode<T> \*srcptr = L.first;

LinkNode<T> \*destptr = first = new LinkNode<T>(srcptr->data);

if(first == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

while(srcptr->link != NULL) {

destptr->link = new LinkNode<T>(srcptr->link->data);

if(destptr->link == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

destptr = destptr->link;

srcptr = srcptr->link;

}

destptr->link = NULL;

return \*this;

}

};

#include "List.cpp"

#endif

**2**、**List.cpp**

#ifndef LIST\_CPP\_

#define LIST\_CPP\_

#include <iostream>

#include <cstdlib>

using namespace std;

#include "List.h"

template <class T>

List<T>::List( ) {

first = new LinkNode<T>;

if(first == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

}

template <class T>

List<T>::List(const T x) {

first = new LinkNode<T>(x);

if(first == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

}

template <class T>

List<T>::List(List<T>& L) {

LinkNode<T> \*srcptr = L.first;

LinkNode<T> \*destptr = first = new LinkNode<T>(srcptr->data);

if(first == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

while(srcptr->link != NULL) {

destptr->link = new LinkNode<T>(srcptr->link->data);

if(destptr->link == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

destptr = destptr->link;

srcptr = srcptr->link;

}

destptr->link = NULL;

}

template <class T>

void List<T>::makeEmpty( ) {

LinkNode<T> \*q;

while (first->link != NULL) {

q = first->link; // 保存被删结点

first->link = q->link; // 从链上摘下该结点

delete q; // 删除

}

}

template <class T>

int List<T>::Length( ) const {

LinkNode<T> \*p = first->link;

int count = 0;

while ( p != NULL ) {// 逐个结点检测

p = p->link;

count++;

}

return count;

}

template <class T>

LinkNode<T>\* List<T>::Search(const T x)const {

// 在表中搜索含数据x 的结点, 搜索成功时函数返回该结点地址; 否则返回NULL。

LinkNode<T> \*current = first->link;

while (current != NULL && current->data != x ) // 沿着链找含x 结点

current = current->link;

return current;

}

template <class T>

LinkNode<T>\* List<T>::Locate (const int i)const {

// 函数返回表中第i 个元素的地址。若i < 0 或i 超出表中结点个数，则返回NULL。

if (i < 0) return NULL; // i 不合理

LinkNode<T> \*current = first;

int k = 0;

while ( current != NULL && k < i ) {

current = current->link;

k++;

}

return current; // 返回第i 号结点地址或NULL

}

template <class T>

bool List<T>::Insert (const int i, const T x) {

// 将新元素x 插入在链表中第i 个结点之后

LinkNode<T> \*current = Locate(i);

if (current == NULL) return false; // 无插入位置

LinkNode<T> \*newNode = new LinkNode<T>(x, current->link); // 创建新结点

if(newNode == NULL) {

cerr <<"存储分配失败!"<< endl;

exit(1);

}

// newNode->link = current->link; // 链入

current->link = newNode;

return true; // 插入成功

}

template <class T>

bool List<T>::Remove (const int i, T& x ) {

// 删除链表第i 个元素, 通过引用参数x 返回元素值

LinkNode<T> \*current = Locate(i-1);

if ( current == NULL || current->link == NULL)

return false; // 删除不成功

LinkNode<T> \*del = current->link;

current->link = del->link;

x = del->data;

delete del;

return true;

}

template <class T>

T\* List<T>::getData(const int i) const {

// 取出表中第i 个元素的值

if(i <= 0) return NULL;

LinkNode<T> \*current = Locate(i);

if( current != NULL )

return &(current->data);

else

return NULL;

}

template <class T>

bool List<T>::setData(const int i, const T x) {

// 更新第i 元素值

if(i <= 0) return false;

LinkNode<T> \*current = Locate(i);

if(current != NULL) {

current -> data = x;

return true;

}

else return false;

}

template <class T>

void List<T>::output( ) {

LinkNode<T> \*p = first->link;

int i = 0;

cout <<"表中元素为："<< endl;

while(p) {

cout << '#' << i++ <<":"<< p->data <<"";

p = p->link;

}

cout << endl;

}

#endif

第三章栈和队列

一、顺序栈：SeqStackt.h，SeqStack.cpp

**1**、**SeqStackt.h**

#ifndef \_STACK\_H\_

#define \_STACK\_H\_

#include <iostream>

using namespace std;

const int DefaultSize = 50; // 栈的默认大小

const int IncreamentSize = 20; // 当栈满时增加的量

template <class T>

class SeqStack { // 顺序栈类定义

private:

T \*elements; // 栈元素存放数组

int top; // 栈顶指针

int maxSize; // 栈最大容量

void overflowProcess( ); // 栈的溢出处理

public:

SeqStack(int sz = 50); // 构造函数

~SeqStack( ); // 析构函数

void Push(const T x); // 进栈

bool Pop(T& x); // 出栈

bool getTop(T& x)const; // 取栈顶内容

bool IsEmpty( ) const;

bool IsFull( ) const;

int getSize( ) const;

void MakeEmpty( );

void Traverse( )const;

};

template <class T>

void SeqStack<T>::overflowProcess( ) {

// 私有函数：当栈满则执行扩充栈存储空间处理

T \*newArray = new T[maxSize + IncreamentSize]; // 创建更大的存储数组

for (int i = 0; i <= top; i++)

newArray[i] = elements[i];

maxSize += IncreamentSize;

delete [ ]elements;

elements = newArray; // 改变elements 指针

};

#include "SeqStack.cpp"

#endif

**2**、**SeqStack.cpp**

#ifndef \_STACK\_CPP\_

#define \_STACK\_CPP\_

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class T>

SeqStack<T>::SeqStack(int size): top(-1), maxSize(size) {

if(maxSize <= 0) return;

elements = new T[maxSize];

if(elements == NULL) {

cerr <<"初始化失败！"<< endl;

exit(1);

}

}

template <class T>

SeqStack<T>::~SeqStack( ) { // 析构函数

delete [ ]elements;

}

template <class T>

void SeqStack<T>::Push(const T x) {

if (IsFull( ) == true) overflowProcess( ); // 栈满

elements[++top] = x; // 栈顶指针先加1, 再进栈

}

template <class T>

bool SeqStack<T>::Pop(T& x) { // 函数退出栈顶元素并返回栈顶元素的值

if (IsEmpty( ) == true) return false;

x = elements[top--]; // 栈顶指针退1

return true; // 退栈成功

}

template <class T>

bool SeqStack<T>::getTop(T& x) const { // 若栈不空则函数返回该栈栈顶元素的地址

if (IsEmpty( ) == true) return false;

x = elements[top];

return true;

}

template <class T>

inline bool SeqStack<T>::IsEmpty( ) const {

return top == -1;

}

template <class T>

inline bool SeqStack<T>::IsFull( ) const {

return top == maxSize-1;

}

template <class T>

inline int SeqStack<T>::getSize( ) const {

return top + 1;

}

template <class T>

inline void SeqStack<T>::MakeEmpty( ) {

top = -1;

}

template <class T>

void SeqStack<T>::Traverse( )const {

for(int i = 0; i <= top; i++)

cout << elements[i] <<"";

cout << endl;

}

#endif

二、链式栈：LinkedStack.h，LinkedStack.cpp

**1**、**LinkedStack.h**

#ifndef \_LINKED\_STACK\_H\_

#define \_LINKED\_STACK\_H\_

#include<iostream>

using namespace std;

template<class T>

struct LinkNode {

T data;

LinkNode<T> \*link;

LinkNode(LinkNode<T> \*ptr = NULL) { link = ptr; }

LinkNode(const T& item, LinkNode<T> \*ptr = NULL): data(item), link(ptr) { }

};

template<class T>

class LinkedStack {

private:

LinkNode<T> \*top;

public:

LinkedStack( );

~LinkedStack( );

void Push(const T x);

bool Pop(T& x);

bool getTop(T& x)const;

bool IsEmpty( )const;

int getSize( )const;

void makeEmpty( );

};

#include "LinkedStack.cpp"

#endif

**2**、**LinkedStack.cpp**

#ifndef \_LINKED\_STACK\_CPP\_

#define \_LINKED\_STACK\_CPP\_

#include "LinkedStack.h"

#include <iostream>

#include <cassert>

using namespace std;

template<class T>

LinkedStack<T>::LinkedStack( ): top(NULL) { }

template<class T>

LinkedStack<T>::~LinkedStack( ) {

makeEmpty( );

}

template<class T>

void LinkedStack<T>::makeEmpty( ) {

while(top != NULL) {

LinkNode<T> \*p = top;

top = top -> link;

delete p;

}

}

template<class T>

void LinkedStack<T>::Push(const T x) {

top = new LinkNode<T>(x, top); // 创建新的含x 的结点

assert(top != NULL);

}

template<class T>

bool LinkedStack<T>::Pop(T& x) {

if(IsEmpty( )) return false;

LinkNode<T> \*p = top;

top = top -> link;

x = p -> data;

delete p;

return true;

}

template<class T>

bool LinkedStack<T>::getTop (T& x)const {

if(IsEmpty( ) == true) return false;

x = top -> data;

return true;

}

template<class T>

inline bool LinkedStack<T>::IsEmpty( )const {

return (top == NULL) ? true : false;

}

template<class T>

int LinkedStack<T>::getSize( )const {

LinkNode<T> \*p = top;

int k = 0;

while( p != NULL) {

p = p -> link;

k++;

}

return k;

}

#endif

三、循环队列：SeqQueue.h，SeqQueue.cpp

**1**、**SeqQueue.h**

#ifndef \_SEQ\_QUEUE\_H\_

#define \_SEQ\_QUEUE\_H\_

#include <iostream>

using namespace std;

const int DefaultSize = 5;

template <class T>

class SeqQueue { // 循环队列类定义

private:

int rear, front; // 队尾与队头指针

T \*elements; // 队列存放数组

int maxSize; // 队列最大容量

void Increase( ); // 当队列满是增大一倍

void Decrease( ); // 当队列中元素个数低于空间的1/4 时缩减一半

public:

SeqQueue(int sz = DefaultSize); // 构造函数

~SeqQueue( ); // 析构函数

bool EnQueue(const T x); // 新元素进队列

bool DeQueue(T& x); // 退出队头元素

bool getFront(T& x)const; // 取队头元素值

void makeEmpty( );

bool IsEmpty( )const;

bool IsFull( )const;

int getSize( )const;

void output( )const;

};

#include "SeqQueue.cpp"

#endif

**2**、**SeqQueue.cpp**

#ifndef \_SEQ\_QUEUE\_CPP\_

#define \_SEQ\_QUEUE\_CPP\_

#include <iostream>

#include <cassert>

using namespace std;

#include "SeqQueue.h"

template <class T>

SeqQueue<T>::SeqQueue(int sz) : front(0), rear(0), maxSize(sz) {// 构造函数

assert(elements = new T[maxSize]);

if( elements == NULL ) {

cerr <<"初始化队列失败！"<< endl;

exit(1);

}

}

template <class T>

SeqQueue<T>::~SeqQueue( ) { // 析构函数

delete[ ] elements;

}

template <class T>

bool SeqQueue<T>::EnQueue(const T x) {

// 若队列不满, 则将x 插入到该队列队尾, 并返回true, 否则返回false

if (IsFull( ) == true) Increase( ); // return false;

elements[rear] = x; // 先存入

rear = (rear+1) % maxSize; // 尾指针加一

return true;

}

template <class T>

bool SeqQueue<T>::DeQueue(T& x) {

// 若队列不空则函数退队头元素并返回其值

if (IsEmpty( ) == true) return false;

x = elements[front]; // 先取队头

front = (front+1) % maxSize; // 再队头指针加一

if ( 4 \* getSize( ) <= maxSize ) Decrease( );

return true;

}

template <class T>

bool SeqQueue<T>::getFront(T& x) const {

// 若队列不空则函数返回该队列队头元素的值

if (IsEmpty( ) == true) return false; // 队列空

x = elements[front]; // 返回队头元素

return true;

}

template <class T>

inline void SeqQueue<T>::makeEmpty( ) {

front = rear = 0;

}

template <class T>

inline bool SeqQueue<T>::IsEmpty( ) const {

return front == rear;

}

template <class T>

inline bool SeqQueue<T>::IsFull( ) const {

return (rear + 1) % maxSize == front;

}

template <class T>

inline int SeqQueue<T>::getSize( ) const {

return (rear - front + maxSize) % maxSize;

}

template <class T>

void SeqQueue<T>::output( )const {

for( int i = front; i != rear; i = (i + 1) % maxSize)

cout << elements[i] <<"";

cout <<"还可容纳"<< maxSize - getSize( ) - 1 <<" 个元素"<< endl;

}

template <class T>

void SeqQueue<T>::Increase( ) {

T \*newArray = new T[2 \* maxSize];

if( newArray == NULL ) {

cerr <<"初始化队列失败！"<< endl;

exit(1);

}

int k = 0;

for( int i = front; i != rear; i = (i + 1) % maxSize, k++)

newArray[k] = elements[i];

delete [ ]elements;

elements = newArray;

front = 0;

rear = k;

maxSize \*= 2;

}

template <class T>

void SeqQueue<T>::Decrease( ) {

if( maxSize <= DefaultSize ) return;

T \*newArray = new T[maxSize/2];

if( newArray == NULL ) {

cerr <<"初始化队列失败！"<< endl;

exit(1);

}

int k = 0;

for( int i = front; i != rear; i = (i + 1) % maxSize, k++)

newArray[k] = elements[i];

delete [ ]elements;

elements = newArray;

front = 0;

rear = k;

maxSize /= 2;

}

#endif

第四章数组

一、对称矩阵上三角存储：SymmetryMatrix.h，SymmetryMatrix.cpp

**1**、**SymmetryMatrix.h**

#ifndef \_SYMMETRY\_MATRIX\_H\_

#define \_SYMMETRY\_MATRIX\_H\_

#include <cassert>

using namespace std;

template <class T>

class SMatrix { // 对称矩阵类：只存储上三角

private:

T \*Base; // 存储矩阵的一维数组的首地址

int n; // 矩阵的阶数

public:

SMatrix( ); // 构造函数

SMatrix(SMatrix<T> &p); // 拷贝构造函数

~SMatrix( ) { // 析构函数

if(Base != NULL) delete [ ]Base;

}

T& getElement(int i, int j)const; // 返回第i 行，第j 列元素，i, j 从1 开始

void output( )const; // 输出矩阵

void input( ); // 输入矩阵

SMatrix<T> operator=(SMatrix<T> &p);

SMatrix<T> operator+(SMatrix<T> &p);

};

#include "SymmetryMatrix.cpp"

#endif

**2**、**SymmetryMatrix.cpp**

#ifndef \_SYMMETRY\_MATRIX\_CPP\_

#define \_SYMMETRY\_MATRIX\_CPP\_

#include "SymmetryMatrix.h"

template <class T>

SMatrix<T>::SMatrix( ) : Base(NULL), n(0) { } // 构造函数

template <class T>

T& SMatrix<T>::getElement(int i, int j)const {// 返回第i 行，第j 列元素

int k;

if(i > j) { k = i; i = j; j = k; }

k = (2\*n-i)\*(i-1)/2 + j - 1; // (2n-i)\*(i-1)/2 + j -1

return Base[k];

}

template <class T>

void SMatrix<T>::output( )const { // 输出矩阵

for(int i = 1; i <= n; i++) {

for(int j = 1; j <= n; j++)

cout << getElement(i, j) << " ";

cout << endl;

}

}

template <class T>

void SMatrix<T>::input( ) { // 输入矩阵

int k;

cout << "输入矩阵的阶数：";

cin >> k;

if(k > 0) {

n = k;

Base = new T[n\*(n+1)/2];

assert(Base != NULL);

}

else {

n = 0;

Base = NULL;

return;

}

k = 0;

for(int i = 1; i <= n; i++) {

for(int j = i; j <= n; j++) {

cout << "第" << i << " 行，第" << j << " 列：";

cin >> Base[k++];

}

}

}

template <class T>

SMatrix<T>::SMatrix(SMatrix<T> &p) {

n = p.n;

Base = new T[n\*(n+1)/2];

assert(Base != NULL);

for(int k = 0; k < n\*(n+1)/2; k++)

Base[k] = p.Base[k];

}

template <class T>

SMatrix<T> SMatrix<T>::operator=(SMatrix<T> &p) {

if(Base != NULL)

delete [ ]Base;

n = p.n;

if(n <= 0) {

Base = NULL;

return \*this;

}

Base = new T[n\*(n+1)/2];

assert(Base != NULL);

for(int k = 0; k < n\*(n+1)/2; k++)

Base[k] = p.Base[k];

return \*this;

}

template <class T>

SMatrix<T> SMatrix<T>::operator+(SMatrix<T> &p) {

SMatrix<T> sm3;

if(n != p.n)

return sm3;

sm3.Base = new T[n\*(n+1)/2];

assert(sm3.Base != NULL);

sm3.n = n;

for(int k = 0; k < n\*(n+1)/2; k++)

sm3.Base[k] = Base[k] + p.Base[k];

return sm3;

}

#endif

二、三对角矩阵：TriDiagonalMatrix.h，TriDiagonalMatrix.cpp

**1**、**TriDiagonalMatrix.h**

#ifndef \_TRI\_DIAGONAL\_MATRIX\_H\_

#define \_TRI\_DIAGONAL\_MATRIX\_H\_

#include <cassert>

using namespace std;

template <class T>

class TDMatrix { // 三对角矩阵类：只存储三条主对角线上的元素

private:

T \*Base; // 存储矩阵的一维数组的首地址

int n; // 矩阵的阶数

public:

TDMatrix( ); // 构造函数

TDMatrix(TDMatrix<T> &p); // 拷贝构造函数

~TDMatrix( ) { // 析构函数

if(Base != NULL)

delete [ ]Base;

}

T& getElement(int i, int j)const; // 返回第i 行，第j 列元素，i, j 从1 开始

void output( )const; // 输出矩阵

void input( ); // 输入矩阵

TDMatrix<T> operator=(TDMatrix<T> &p);

};

#include "TriDiagonalMatrix.cpp"

#endif

**2**、**TriDiagonalMatrix.cpp**

#ifndef \_TRI\_DIAGONAL\_MATRIX\_CPP\_

#define \_TRI\_DIAGONAL\_MATRIX\_CPP\_

#include "TriDiagonalMatrix.h"

template <class T>

TDMatrix<T>::TDMatrix( ) : Base(NULL), n(0) { } // 构造函数

template <class T>

TDMatrix<T>::TDMatrix(TDMatrix<T> &p) { // 拷贝构造函数

n = p.n;

Base = new T[3\*n-1];

assert(Base != NULL);

for(int i = 0; i < 3\*n-1; i++)

Base[i] = p.Base[i];

}

template <class T>

T& TDMatrix<T>::getElement(int i, int j)const { // 返回第i 行，第j 列元素

int k = i - j;

if(k >= -1 && k <= 1) return Base[2 \* i + j - 3];

else return Base[3\*n-2];

}

template <class T>

void TDMatrix<T>::output( )const { // 输出矩阵

for(int i = 1; i <= n; i++) {

for(int j = 1; j <= n; j++)

cout << getElement(i, j) << " ";

cout << endl;

}

}

template <class T>

void TDMatrix<T>::input( ) { // 输入矩阵

int i, j, k;

cout << "输入矩阵的阶数：";

cin >> k;

if(k > 0) {

n = k;

Base = new T[3\*n-1];

assert(Base != NULL);

}

else {

n = 0;

Base = NULL;

return;

}

k = 0;

for(i = j = 1; j <= 2; j++) {

cout << "第" << i << " 行，第" << j << " 列：";

cin >> Base[k++];

}

for(i = 2; i < n; i++) {

for(j = i - 1; j <= i + 1; j++) {

cout << "第" << i << " 行，第" << j << " 列：";

cin >> Base[k++];

}

}

for(j = i - 1; j <= n; j++) {

cout << "第" << i << " 行，第" << j << " 列：";

cin >> Base[k++];

}

Base[k] = 0;

}

template <class T>

TDMatrix<T> TDMatrix<T>::operator=(TDMatrix<T> &p) {

if(Base != NULL)

delete [ ]Base;

n = p.n;

if(n <= 0) {

Base = NULL;

return \*this;

}

Base = new T[3\*n-1];

assert(Base != NULL);

for(int i = 0; i < 3\*n-1; i++)

Base[i] = p.Base[i];

return \*this;

}

#endif

第五章树与二叉树

一、二叉链表：BinaryTree.h，BinaryTree.cpp

**1**、**BinaryTree.h**

#ifndef \_BINARY\_TREE\_H

#define \_BINARY\_TREE\_H

#include <fstream>

#include <iostream>

using namespace std;

template <class T>

struct BinTreeNode { // 二叉树结点类定义

T data; // 数据域

BinTreeNode<T> \*leftChild, \*rightChild; // 左子女、右子女链域

BinTreeNode ( ): leftChild(NULL), rightChild(NULL) { } // 构造函数

BinTreeNode (T x, BinTreeNode<T> \*l = NULL, BinTreeNode<T> \*r = NULL): data(x), leftChild(l), rightChild(r) { }

};

template <class T>

class BinaryTree { // 二叉树类定义

private:

BinTreeNode<T> \*root; // 二叉树的根指针

T RefValue; // 数据输入停止标志

void CreateBinTree (ifstream& in, BinTreeNode<T>\* &subTree); // 从文件读入建树

void destroy (BinTreeNode<T> \*subTree);

int Height (BinTreeNode<T> \*subTree)const; // 返回树高度

int Size (BinTreeNode<T> \*subTree)const; // 返回结点数

BinTreeNode<T> \*Parent (BinTreeNode<T>\* subTree, BinTreeNode<T> \*t); // 返回父结点

void PreOrder (BinTreeNode<T>\* subTree, void (\*visit)(BinTreeNode<T> \*t)); // 前序遍历

void InOrder (BinTreeNode<T>\* subTree, void (\*visit)(BinTreeNode<T> \*t)); // 中序遍历

void PostOrder (BinTreeNode<T>\* subTree, void (\*visit)(BinTreeNode<T> \*t)); // 后序遍历

public:

BinaryTree ( ) : root (NULL) { } // 构造函数

BinaryTree (T value) : RefValue(value), root(NULL) { }// 构造函数

// BinaryTree (BinaryTree<T>& s); // 复制构造函数

void CreatePreOrder (ifstream& in) {CreateBinTree(in, root);}

~BinaryTree ( ) { destroy(root); } // 析构函数

bool IsEmpty ( ) { return root == NULL;} // 判二叉树空否

int Height ( ) { return Height(root); } // 求树高度

int Size ( ) { return Size(root); } // 求结点数

BinTreeNode<T> \*Parent (BinTreeNode <T> \*t) { // 返回双亲结点

return (root == NULL || root == t) ? NULL : Parent (root, t);

}

BinTreeNode<T> \*LeftChild (BinTreeNode<T> \*t) { // 返回左子女

return (t != NULL)?t->leftChild : NULL;

}

BinTreeNode<T> \*RightChild (BinTreeNode<T> \*t) { // 返回右子女

return (t != NULL)?t->rightChild : NULL;

}

BinTreeNode<T> \*getRoot( ) const { return root; }// 取根

void PreOrder (void (\*visit) (BinTreeNode<T> \*t)) { PreOrder (root, visit); } // 前序遍历

void InOrder (void (\*visit) (BinTreeNode<T> \*t)) { InOrder (root, visit); } // 中序遍历

void PostOrder (void (\*visit) (BinTreeNode<T> \*t)) { PostOrder (root, visit); }// 后序遍历

};

#include "BinaryTree.cpp"

#endif

**2**、**BinaryTree.cpp**

#ifndef \_BINARY\_TREE\_CPP

#define \_BINARY\_TREE\_CPP

#include <iostream>

#include <fstream>

#include <assert.h>

#include "BinaryTree.h"

template <class T>

BinTreeNode<T> \*BinaryTree<T>::Parent(BinTreeNode <T> \*subTree, BinTreeNode <T> \*t) {

// 私有函数: 从结点subTree 开始, 搜索结点t 的双亲, 若找到则返回双亲结点地址, 否则返回NULL

if (subTree == NULL) return NULL;

if (subTree->leftChild == t || subTree->rightChild == t )

return subTree; // 找到, 返回父结点地址

BinTreeNode <T> \*p;

if ((p = Parent (subTree->leftChild, t)) != NULL) // 递归在左子树中搜索

return p;

else return Parent (subTree->rightChild, t); // 递归在左子树中搜

}

template<class T>

void BinaryTree<T>::destroy (BinTreeNode<T> \* subTree) {

// 私有函数: 删除根为subTree 的子树

if (subTree != NULL) {

destroy (subTree->leftChild); // 删除左子树

destroy (subTree->rightChild); // 删除右子树

delete subTree; // 删除根结点

}

}

template <class T>

void BinaryTree<T>::InOrder (BinTreeNode<T> \* subTree, void (\*visit) (BinTreeNode<T> \*t)) {

// 私有函数: 中序遍历（visit）二叉树subTree

if (subTree != NULL) {

InOrder (subTree->leftChild, visit); // 遍历左子树

visit (subTree); // 访问根结点

InOrder (subTree->rightChild, visit); // 遍历右子树

}

}

template <class T>

void BinaryTree<T>::PreOrder (BinTreeNode<T> \* subTree, void (\*visit) (BinTreeNode<T> \*t)) {

// 私有函数: 前序遍历（visit）二叉树subTree

if (subTree != NULL) {

visit (subTree); // 访问根结点

PreOrder (subTree->leftChild, visit); // 遍历左子树

PreOrder (subTree->rightChild, visit); // 遍历右子树

}

}

template <class T>

void BinaryTree<T>::PostOrder (BinTreeNode<T> \* subTree, void (\*visit) (BinTreeNode<T> \*t)) {

// 私有函数: 后序遍历（visit）二叉树subTree

if (subTree != NULL ) {

PostOrder (subTree->leftChild, visit); // 遍历左子树

PostOrder (subTree->rightChild, visit); // 遍历右子树

visit (subTree); // 访问根结点

}

}

template <class T>

int BinaryTree<T>::Size (BinTreeNode<T> \* subTree) const {

// 私有函数：利用二叉树后序遍历算法计算二叉树的结点个数

if (subTree == NULL) return 0; // 空树

else return 1 + Size (subTree->leftChild) + Size (subTree->rightChild);

}

template <class T>

int BinaryTree<T>::Height ( BinTreeNode<T> \* subTree) const {

// 私有函数：利用二叉树后序遍历算法计算二叉树的高度或深度

if (subTree == NULL) return 0;// 空树高度为0

else {

int i = Height (subTree->leftChild);

int j = Height (subTree->rightChild);

return (i < j) ? j+1 : i+1;

}

}

template<class T>

void BinaryTree<T>::CreateBinTree (ifstream& in, BinTreeNode<T>\* &subTree) {

// 私有函数: 以递归方式建立二叉树。

T item;

if ( !in.eof ( ) ) { // 未读完, 读入并建树

in >> item; // 读入根结点的值

if (item != RefValue) {

subTree = new BinTreeNode<T>(item); // 建立根结点

if (subTree == NULL) { cerr <<"存储分配错!"<< endl; exit (1);}

CreateBinTree (in, subTree->leftChild); // 递归建立左子树

CreateBinTree (in, subTree->rightChild);// 递归建立右子树

}

else subTree = NULL; // 封闭指向空子树的指针

}

};

#endif

二、堆：Heap.h，Heap.cpp

**1**、**Heap.h**

#ifndef \_BINARY\_TREE\_H

#ifndef \_HEAP\_H\_

#define \_HEAP\_H\_

#include <iostream>

using namespace std;

const int DefaultSize = 10;

template <class T>

class MinHeap { // 最小堆类定义

private:

T \*heap; // 最小堆元素存储数组

int currentSize; // 最小堆当前元素个数

int maxHeapSize; // 最小堆最大容量

void siftDown (int start, int m); // 调整算法：从start 开始向下调整到m

void siftUp (int start); // 调整算法：从start 开始向上调整

public:

MinHeap(int sz = DefaultSize); // 构造函数：长度为sz

MinHeap(T arr[ ], int n); // 构造函数：用数组arr[ ] 里的元素创建堆

~MinHeap( ) { delete [ ]heap; } // 析构函数

bool Insert (const T d); // 插入

bool Remove (T& d); // 删除

bool IsEmpty ( )const; // 判堆空否

bool IsFull ( ) const; // 判堆满否

void MakeEmpty ( ); // 置空堆

};

template <class T>

inline bool MinHeap<T>::IsEmpty ( ) const {

// 判堆空否

return currentSize == 0;

}

template <class T>

inline bool MinHeap<T>::IsFull ( ) const {

// 判堆满否

return currentSize == maxHeapSize;

}

template <class T>

inline void MinHeap<T>::MakeEmpty ( ) {

// 置空堆

currentSize = 0;

}

#include "Heap.cpp"

#endif

**2**、**BinaryTree.cpp**

#ifndef \_HEAP\_CPP\_

#define \_HEAP\_CPP\_

#include <iostream>

using namespace std;

#include "Heap.h"

template <class T>

void MinHeap<T>::siftDown (int start, int m ) {

// 私有函数：从结点start 开始到m 为止，自上向下比较，

// 如果子女的值小于父结点的值，则关键码小的上浮，

// 继续向下层比较，将一个集合局部调整为最小堆。

int i = start, j = 2\*i+1; // j 是i 的左子女位置

T temp = heap[i];

while (j <= m) { // 检查是否到最后位置

if (j < m && heap[j] > heap[j+1]) // 让j 指向两子女中的小者

j++;

if (temp <= heap[j]) // 小则不做调整

break;

else { // 否则小者上移, i, j 下降

heap[i] = heap[j];

i = j;

j = 2\*j+1;

}

}

heap[i] = temp; // 回放temp 中暂存的元素

}

template <class T>

void MinHeap<T>::siftUp(int start) {

// 私有函数：从结点start 开始到结点0 为止，自下向上比较，

// 如果子女的值小于父结点的值，则相互交换，

// 这样将集合重新调整为最小堆。关键码比较符<=在E 中定义。

int j = start, i = (j - 1) / 2;

T temp = heap[j];

while (j > 0) { // 沿父结点路径向上直达根

if (heap[i] <= temp) break; // 父结点值小, 不调整

else { // 父结点结点值大, 调整

heap[j] = heap[i];

j = i;

i = (i-1)/2;

}

}

heap[j] = temp; // 回送

}

template <class T>

MinHeap<T>::MinHeap(int sz) {

// 构造函数：建立大小为sz 的堆

maxHeapSize = (DefaultSize < sz) ? sz : DefaultSize;

heap = new T[maxHeapSize]; // 创建堆空间

if (heap == NULL) {

cerr <<"堆存储分配失败！"<< endl; exit(1);

}

currentSize = 0; // 建立当前大小

}

template <class T>

MinHeap<T>::MinHeap (T arr[ ], int n) {

// 构造函数：用数组arr[ ] 里的元素创建堆

maxHeapSize = (DefaultSize < n) ? n : DefaultSize;

heap = new T[maxHeapSize];

if (heap == NULL) {

cerr <<"堆存储分配失败！"<< endl; exit(1);

}

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

heap[i] = arr[i];

currentSize = n; // 复制堆数组, 建立当前大小

int currentPos = (currentSize - 2) / 2; // 找最初调整位置：即最后分支结点

while (currentPos >= 0) { // 逐步向上扩大堆

siftDown (currentPos, currentSize - 1); // 局部自上向下下滑调整

currentPos--;

}

}

template <class T>

bool MinHeap<T>::Insert (const T x ) {

// 将x 插入到最小堆中

if (currentSize == maxHeapSize) { // 堆满

cout <<"Heap Full"<< endl;

return false;

}

heap[currentSize] = x; // 插入

siftUp (currentSize); // 向上调整

currentSize++; // 堆计数加1

return true;

}

template <class T>

bool MinHeap<T>::Remove (T& x) {

// 从堆中删除元素，并用x 返回

if (currentSize <= 0) // 堆空, 返回false

{ cout <<"Heap empty"<< endl; return false; }

x = heap[0];

heap[0] = heap[--currentSize];

siftDown(0, currentSize-1); // 自上向下调整为堆

return true; // 返回最小元素

}

#endif

三、Huffman 树（实现一）：Huffman.h，Huffman.cpp，Heap.h，Heap.cpp

**1**、**Huffman.h**

#ifndef \_HUFFMANTREE\_H\_

#define \_HUFFMANTREE\_H\_

#include <iostream>

using namespace std;

#include "heap.h"

#include "SeqStack.h"

#include "SeqQueue.h"

template <class T, class E>

struct HuffmanNode { // 树结点的类定义

T data; // 结点数据

E weight; // 结点的权值

HuffmanNode<T, E> \*parent; // 父结点指针

HuffmanNode<T, E> \*leftChild, \*rightChild; // 左、右子女

HuffmanNode( ) : parent(NULL), leftChild(NULL), rightChild(NULL) { } // 构造函数

HuffmanNode (T d, int w, HuffmanNode<T, E> \*p = NULL,

HuffmanNode<T, E> \*l = NULL, HuffmanNode<T, E> \*r = NULL)

: data (d), weight(w), parent (p), leftChild (l), rightChild (r) { }

friend bool operator>(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2);

friend bool operator<(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2);

friend bool operator>=(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2);

friend bool operator<=(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2);

};

template <class T, class E>

bool operator>(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2) {

return R1.weight > R2.weight;

}

template <class T, class E>

bool operator<(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2) {

return R1.weight < R2.weight;

}

template <class T, class E>

bool operator>=(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2) {

return R1.weight >= R2.weight;

}

template <class T, class E>

bool operator<=(HuffmanNode<T, E> R1, HuffmanNode<T, E> R2) {

return R1.weight <= R2.weight;

}

template <class T, class E>

class HuffmanTree { // Huffman 树类定义

protected:

HuffmanNode<T, E> \*root; // 树的根

HuffmanNode<T, E> \*NodeList;

int N; // 叶子结点个数

char \*\*code; // 编码

void deleteTree (HuffmanNode<T, E> \*t); // 删除以t 为根的子树

// 把树ht1 和ht2 合并到树ht 中

void mergeTree (HuffmanNode<T, E>\* &ht1, HuffmanNode<T, E>\* &ht2, HuffmanNode<T, E>\* &ht);

void HuffmanCode( ); // Huffman 编码

public:

HuffmanTree (T v[ ], E w[ ], int n); // 构造函数

~HuffmanTree( ); // 析构函数

void DispHuffmanCode( )const;

// 对字符串s 进行Huffman 编码，并输出到队列q 中

void HuffmanEnCode(SeqQueue<char>&q, const char \*s = "")const;

// 对q1 中的Huffman 码解码到q2 中

void HuffmanDeCode(SeqQueue<char>&q1, SeqQueue<char>&q2)const;

};

template <class T, class E>

void HuffmanTree<T, E>::HuffmanEnCode(SeqQueue<char>&q, const char \*s)const {

// 对字符串s 进行Huffman 编码，并输出到队列q 中

char \*p;

// q.makeEmpty( );

while(\*s) {

if(\*s == ' ') {

// cout << code[26];

for(p = code[26]; \*p; p++)

q.EnQueue(\*p);

}

else {

// cout << code[\*s - 'A'];

for(p = code[\*s - 'A']; \*p; p++)

q.EnQueue(\*p);

}

s++;

}

}

template <class T, class E>

void HuffmanTree<T, E>::HuffmanDeCode(SeqQueue<char>&q1, SeqQueue<char>&q2)const {

// 对q1 中的Huffman 码解码到q2 中

char item;

HuffmanNode<T, E> \*r;

while(!q1.IsEmpty( )) {

r = root;

while(r->leftChild != NULL || r->rightChild != NULL) {

q1.DeQueue(item);

if(item == '0')

r = r->leftChild;

else

r = r->rightChild;

}

q2.EnQueue(r->data);

}

}

#include "HuffmanTree.cpp"

#endif

**2**、**Huffman.cpp**

#ifndef \_HUFFMANTREE\_CPP\_

#define \_HUFFMANTREE\_CPP\_

#include <iostream>

using namespace std;

#include "heap.h"

#include "SeqStack.h"

template <class T, class E>

HuffmanTree<T, E>::HuffmanTree (T v[ ], E w[ ], int n) {

// 给出n 个权值w[0] ～ w[n-1], 构造Huffman 树

MinHeap<HuffmanNode<T, E> \*> hp(n);

HuffmanNode<T, E> \*parent, \*first, \*second;

NodeList = new HuffmanNode<T, E>[n];

int i;

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

NodeList[i].data = v[i];

NodeList[i].weight = w[i];

hp.Insert(&NodeList[i]); // 插入最小堆中

}

for (i = 0; i < n-1; i++) { // n-1 趟, 建Huffman

hp.Remove (first); // 根权值最小的树

hp.Remove (second); // 根权值次小的树

mergeTree (first, second, parent); // 合并

hp.Insert (parent); // 重新插入堆中

}

root = parent; // 建立根结点

N = n;

HuffmanCode( );

}

template <class T, class E>

HuffmanTree<T, E>::~HuffmanTree( ) {

// 析构函数

deleteTree(root);

delete NodeList;

for(int i = 0; i < N; i++)

delete code[i];

delete code;

}

template <class T, class E>

void HuffmanTree<T, E>::DispHuffmanCode( )const {

for(int i = 0; i < N - 1; i++) {

cout << NodeList[i].data <<"："<< code[i] << endl;

}

cout << ' ' <<"："<< code[i] << endl;

}

template <class T, class E>

void HuffmanTree<T, E>::HuffmanCode( ) {

// 私有函数

code = new char\*[N];

SeqStack<char> s;

for(int i = 0; i < N; i++) {

HuffmanNode<T, E> \*p = NodeList[i].parent;

HuffmanNode<T, E> \*q = &NodeList[i];

while(p) {

if(p->leftChild == q)

s.Push('0');

else

s.Push('1');

q = p;

p = p->parent;

}

code[i] = new char[s.getSize( )+1];

int k = 0;

while(!s.IsEmpty( ))

s.Pop(code[i][k++]);

code[i][k] = '\0';

}

}

template <class T, class E>

void HuffmanTree<T, E>::mergeTree (HuffmanNode<T, E>\* &ht1,

HuffmanNode<T, E>\* &ht2, HuffmanNode<T, E>\* &ht) {

// 私有函数：

ht = new HuffmanNode<T, E>;

ht->leftChild = ht1;

ht->rightChild = ht2;

ht->weight = ht1->weight + ht2->weight;

ht1->parent = ht2->parent = ht;

}

template <class T, class E>

void HuffmanTree<T, E>::deleteTree(HuffmanNode<T, E> \*t) {

// 私有函数：删除以t 为根的子树

if(t == NULL) return;

if(t->leftChild == NULL) return;

deleteTree(t->leftChild);

deleteTree(t->rightChild);

if(t->parent != NULL) {

if(t->parent->leftChild == t)

t->parent->leftChild = NULL;

else

t->parent->rightChild = NULL;

}

delete t;

}

#endif

**3**、**Heap.h**（此**Heap** 与前面的**Heap** 相比有较小变动）

#ifndef \_HEAP\_H\_

#define \_HEAP\_H\_

#include <iostream>

using namespace std;

const int DefaultSize = 10;

template <class T>

class MinHeap { // 最小堆类定义

private:

T \*heap; // 最小堆元素存储数组

int currentSize; // 最小堆当前元素个数

int maxHeapSize; // 最小堆最大容量

void siftDown (int start, int m); // 调整算法：从start 开始向下调整到m

void siftUp (int start); // 调整算法：从start 开始向上调整

public:

MinHeap(int sz = DefaultSize); // 构造函数：长度为sz

MinHeap(T arr[ ], int n); // 构造函数：用数组arr[ ] 里的元素创建堆

~MinHeap( ) { delete [ ]heap; } // 析构函数

bool Insert (const T d); // 插入

bool Remove (T& d); // 删除

bool IsEmpty ( )const; // 判堆空否

bool IsFull ( ) const; // 判堆满否

void MakeEmpty ( ); // 置空堆

};

template <class T>

inline bool MinHeap<T>::IsEmpty ( ) const {

// 判堆空否

return currentSize == 0;

}

template <class T>

inline bool MinHeap<T>::IsFull ( ) const {

// 判堆满否

return currentSize == maxHeapSize;

}

template <class T>

inline void MinHeap<T>::MakeEmpty ( ) {

// 置空堆

currentSize = 0;

}

#include "Heap.cpp"

#endif

**4**、**Heap.cpp**（此**Heap** 与前面的**Heap** 相比有较小变动）

#ifndef \_HEAP\_CPP\_

#define \_HEAP\_CPP\_

#include <iostream>

using namespace std;

#include "Heap.h"

template <class T>

void MinHeap<T>::siftDown (int start, int m ) {

// 私有函数：从结点start 开始到m 为止，自上向下比较，

// 如果子女的值小于父结点的值，则关键码小的上浮，

// 继续向下层比较，将一个集合局部调整为最小堆。

int i = start, j = 2\*i+1; // j 是i 的左子女位置

T temp = heap[i];

while (j <= m) { // 检查是否到最后位置

if (j < m && \*heap[j] > \*heap[j+1])// 让j 指向两子女中的小者

j++;

if (\*temp <= \*heap[j]) // 小则不做调整

break;

else { // 否则小者上移, i, j 下降

heap[i] = heap[j];

i = j;

j = 2\*j+1;

}

}

heap[i] = temp; // 回放temp 中暂存的元素

}

template <class T>

void MinHeap<T>::siftUp(int start) {

// 私有函数：从结点start 开始到结点0 为止，自下向上比较，

// 如果子女的值小于父结点的值，则相互交换，

// 这样将集合重新调整为最小堆。关键码比较符<=在E 中定义。

int j = start, i = (j - 1) / 2;

T temp = heap[j];

while (j > 0) { // 沿父结点路径向上直达根

if (\*heap[i] <= \*temp) break; // 父结点值小, 不调整

else { // 父结点结点值大, 调整

heap[j] = heap[i];

j = i;

i = (i-1)/2;

}

}

heap[j] = temp; // 回送

}

template <class T>

MinHeap<T>::MinHeap(int sz) {

// 构造函数：建立大小为sz 的堆

maxHeapSize = (DefaultSize < sz) ? sz : DefaultSize;

heap = new T[maxHeapSize]; // 创建堆空间

if (heap == NULL) {

cerr <<"堆存储分配失败！"<< endl; exit(1);

}

currentSize = 0; // 建立当前大小

}

template <class T>

MinHeap<T>::MinHeap (T arr[ ], int n) {

// 构造函数：用数组arr[ ] 里的元素创建堆

maxHeapSize = (DefaultSize < n) ? n : DefaultSize;

heap = new T[maxHeapSize];

if (heap == NULL) {

cerr <<"堆存储分配失败！"<< endl; exit(1);

}

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

heap[i] = arr[i];

currentSize = n; // 复制堆数组, 建立当前大小

int currentPos = (currentSize - 2) / 2; // 找最初调整位置：即最后分支结点

while (currentPos >= 0) { // 逐步向上扩大堆

siftDown (currentPos, currentSize - 1); // 局部自上向下下滑调整

currentPos--;

}

}

template <class T>

bool MinHeap<T>::Insert (const T x ) {

// 将x 插入到最小堆中

if (currentSize == maxHeapSize) // 堆满

{ cout <<"Heap Full"<< endl; return false; }

heap[currentSize] = x; // 插入

siftUp (currentSize); // 向上调整

currentSize++; // 堆计数加1

return true;

}

template <class T>

bool MinHeap<T>::Remove (T& x) {

// 从堆中删除元素，并用x 返回

if (currentSize <= 0) // 堆空, 返回false

{ cout <<"Heap empty"<< endl; return false; }

x = heap[0];

heap[0] = heap[--currentSize];

siftDown(0, currentSize-1); // 自上向下调整为堆

return true; // 返回最小元素

}

#endif

三、Huffman 树（实现二）：Huffman.h，Huffman.cpp

**1**、**Huffman.h**

# #ifndef \_HUFFMANTREE\_H\_

#define \_HUFFMANTREE\_H\_

#include <iostream>

#include <cassert>

using namespace std;

#include "SeqStack.h"

#include "SeqQueue.h"

template <class T>

struct HTNode {

T weight;

int parent, lChild, rChild;

HTNode( ) : parent(-1), lChild(-1), rChild(-1) { } // 构造函数

HTNode (T w, HTNode<T> p = -1, HTNode<T> l = -1, HTNode<T> r = -1)

: weight(w), parent(p), lChild(l), rChild (r) { }

};

template <class E, class T>

class HuffmanTree { // Huffman 树类定义

protected:

HTNode<T> \*ht; // Huffman 树数组指针

E \*NodeList; // 结点中的值

int N; // 叶子结点个数

char \*\*code; // 编码

void HuffmanCode( ); // Huffman 编码

public:

HuffmanTree (E v[ ], T w[ ], int n); // 构造函数

~HuffmanTree( ); // 析构函数

void DispHuffmanCode( )const;

// 对字符串s 进行Huffman 编码，并输出到队列q 中

void HuffmanEnCode(SeqQueue<char>&q, const char \*s = "")const;

// 对q1 中的Huffman 码解码到q2 中

void HuffmanDeCode(SeqQueue<char>&q1, SeqQueue<char>&q2)const;

};

#include "HuffmanTree.cpp"

#endif

**2**、**Huffman.cpp**

#ifndef \_HUFFMANTREE\_CPP\_

#define \_HUFFMANTREE\_CPP\_

#include <iostream>

using namespace std;

#include "SeqStack.h"

template <class E, class T>

HuffmanTree<E, T>::HuffmanTree (E v[ ], T w[ ], int n) {

// 给出n 个权值w[0] ～ w[n-1], 构造Huffman 树

int i, m;

if(n <= 0) {

ht = NULL;

NodeList = NULL;

N = 0;

code = NULL;

return;

}

// 申请空间

m = 2\*n-1;

assert(ht = new HTNode<T>[m]);

assert(NodeList = new E[n]);

N = n;

// 把权值和结点值填到树中

T MaxNum = T(0);

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

ht[i].weight = w[i];

NodeList[i] = v[i];

MaxNum += w[i];

}

// 重复n - 1 次，建立Huffman 树

for (i = n; i < m; i++) { // 求n-1 次根

T min1 = MaxNum, min2;

int pos1 = -1, pos2;

for (int j = 0; j < i; j++) { // 检测前i 棵树

if (ht[j].parent == -1) // 可参选的树根

if (ht[j].weight < min1) { // 选最小

pos2 = pos1; min2 = min1;

pos1 = j; min1 = ht[j].weight;

}

else if (ht[j].weight < min2) { // 选次小

pos2 = j; min2 = ht[j].weight;

}

}

ht[i].lChild = pos1; ht[i].rChild = pos2;

ht[i].weight = ht[pos1].weight + ht[pos2].weight;

ht[pos1].parent = ht[pos2].parent = i;

}

HuffmanCode( );

}

template <class E, class T>

HuffmanTree<E, T>::~HuffmanTree( ) {

// 析构函数

delete [ ]ht;

delete [ ]NodeList;

for(int i = 0; i < N; i++)

delete [ ]code[i];

delete [ ]code;

}

template <class E, class T>

void HuffmanTree<E, T>::DispHuffmanCode( )const {

for(int i = 0; i < N - 1; i++) {

cout << NodeList[i] <<"："<< code[i] << endl;

}

cout << ' ' <<"："<< code[i] << endl;

}

template <class E, class T>

void HuffmanTree<E, T>::HuffmanCode( ) {

// 私有函数

code = new char\*[N];

SeqStack<char> s;

for(int i = 0; i < N; i++) {

int q = i;

int p = ht[q].parent;

while(p > -1) {

if(ht[p].lChild == q)

s.Push('0');

else

s.Push('1');

q = p;

p = ht[q].parent;

}

code[i] = new char[s.getSize( )+1];

int k = 0;

while(!s.IsEmpty( ))

s.Pop(code[i][k++]);

code[i][k] = '\0';

}

}

template <class E, class T>

void HuffmanTree<E, T>::HuffmanEnCode(SeqQueue<char>&q, const char \*s)const {

// 对字符串s 进行Huffman 编码，并输出到队列q 中

char \*p;

q.makeEmpty( );

while(\*s) {

if(\*s == ' ') {

// cout << code[26];

for(p = code[26]; \*p; p++)

q.EnQueue(\*p);

}

else {

// cout << code[\*s - 'A'];

for(p = code[\*s - 'A']; \*p; p++)

q.EnQueue(\*p);

}

s++;

}

}

template <class E, class T>

void HuffmanTree<E, T>::HuffmanDeCode(SeqQueue<char>&q1, SeqQueue<char>&q2)const {

// 对q1 中的Huffman 码解码到q2 中

char item;

int r;

while(!q1.IsEmpty( )) {

r = 2\*N - 2;

while(ht[r].lChild != -1 || ht[r].rChild != -1) {

q1.DeQueue(item);

if(item == '0')

r = ht[r].lChild;

else

r = ht[r].rChild;

}

q2.EnQueue(NodeList[r]);

}

}

#endif

第六章集合与字典

一、并查极：UFSet.h，UFSet.cpp

**1**、**UFSet.h**

#ifndef \_UFSETS\_H\_

#define \_UFSETS\_H\_

const int DefaultSize = 10;

class UFSets { // 集合中的各个子集合互不相交

private:

int \*parent; // 集合元素数组(双亲表示)

int size; // 集合元素的数目

public:

UFSets (int sz = DefaultSize); // 构造函数

~UFSets( ) { delete [ ]parent; } // 析构函数

UFSets& operator = (UFSets& R); // 集合赋值

void Union (int Root1, int Root2); // 子集合并

int Find (int x); // 查找x 的根

void WeightedUnion (int Root1, int Root2); // 改进例程: 带权的合并算法

};

#include "UFSets.cpp"

#endif

**2**、**UFSet.cpp**

#ifndef \_UFSETS\_CPP\_

#define \_UFSETS\_CPP\_

#include "UFSets.h"

UFSets::UFSets (int sz) {

// 构造函数：sz 是集合元素个数，双亲数组的范围为parent[0]～parent[size-1]。

size = sz; // 集合元素个数

parent = new int[size]; // 创建双亲数组

for (int i = 0; i < size; i++)

parent[i] = -1; // 每个自成单元素集合

}

int UFSets::Find (int x) {

// 函数搜索并返回包含元素x 的树的根。

// if (parent[x] < 0)

// return x; // 根的parent[ ]值小于0

// else

// return (Find (parent[x]));

// 尾部递归，可以用迭代实现

while(parent[x] >= 0)

x = parent[x];

return x;

}

void UFSets::Union (int Root1, int Root2) { // 这段代码被修改，书上有错误

// 求两个不相交集合Root1 与Root2 的并

Root1 = Find(Root1);

Root2 = Find(Root2);

if (Root1 != Root2) {

parent[Root1] += parent[Root2];

parent[Root2] = Root1; // 将Root2 连接到Root1 下面

}

}

void UFSets::WeightedUnion (int Root1, int Root2) {

// 按Union 的加权规则改进的算法

Root1 = Find(Root1);

Root2 = Find(Root2);

if (Root1 != Root2) {

int temp = parent[Root1] + parent[Root2];

if (parent[Root2] < parent[Root1]) {

parent[Root1] = Root2; // Root2 中结点数多

parent[Root2] = temp; // Root1 指向Root2

}

else {

parent[Root2] = Root1; // Root1 中结点数多

parent[Root1] = temp; // Root2 指向Root1

}

}

}

#endif