**顺序表：**

SeqList.h

const int DefaultSize=100;

///SeqList///

template <typename T>

class SeqList{

public:

SeqList(int sz=DefaultSize)

:MaxSize(sz),Size(0){

if(sz>0){

\_elements=new T[MaxSize];

}

}

~SeqList(){

delete[] \_elements;

}

int Length() const{

return Size;

}

int Find(T x) const;

int Is\_Elements(T x) const;

int Insert(T x,int i);

int Remove(T x);

int IsEmpty(){

return Size==0;

}

int IsFull(){

return Size==MaxSize;

}

T Get(int i){

return i<0||i>Size?(cout<<"can't find the \_elements"<<endl,0):\_elements[i];

}

void Print();

private:

T \*\_elements;

const int MaxSize;

int Size;

};

///Find///

template <typename T>

int SeqList<T>::Find(T x) const{

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

if(\_elements[i]==x)

return i;

cout<<"can't find the \_elements you want to find"<<endl;

return -1;

}

///Is\_Elements///

template <typename T>

int SeqList<T>::Is\_Elements(T x) const{

if(Find(x)==-1)

return 0;

return 1;

}

///Insert///

template <typename T>

int SeqList<T>::Insert(T x, int i){

if(Size!=MaxSize)

Size++;

else

cout<<"The List is full!"<<endl;

if(i<0||i>Size-1){//坐标小于0或者坐标比当前元素数大1或当前元素数等于最大元素数减1

cout<<"the operate is illegal"<<endl;

return 0;

}

for(int j=Size-1;j>i;j--){

\_elements[j]=\_elements[j-1];

}//将第i位上的值赋值给i+1位

\_elements[i]=x;//将要插入的值赋值给第i位

return 1;

}

///Remove///

template <typename T>

int SeqList<T>::Remove(T x){

int size=0;

while(\_elements[size]!=x&&size!=Size)

size++;

if(size==Size)

{

cout<<"can't find the element:"<<x<<" you want to remove"<<endl;

cout<<"Size:"<<Size<<endl;

return 1;

}

else

{

for(int i=size;i<Size-1;i++){

\_elements[i]=\_elements[i+1];

}

Size--;

cout<<"remove("<<x<<")"<<"at size "<<size<<" Then Size is "<<Size<<endl;

return 0;

}

}

///Print///

template <typename T>

void SeqList<T>::Print(){

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

cout<<i+1<<":\t"<<\_elements[i]<<endl;

cout<<endl<<endl;

}

test.cpp

#include <iostream>

#include "SeqList.h"

using namespace std;

int main()

{

SeqList<int> test(11);

for(int i=0;i<10;i++){

test.Insert(i,0);

}//倒序插入

cout<<test.Length()<<endl;

for(i=0;i<10;i++){

test.Remove(i);

}

test.Print();

cout<<"-----------"<<endl;

for(i=0;i<10;i++){

test.Insert(i,i);

}

test.Print();

if(!test.IsFull())

cout<<"Not Full"<<endl;

else

cout<<"Full"<<endl;

return 0;

}

**单链表**

ListNode.h

template<typename Type>

class SingleList;

///ListNode///

template<typename Type>

class ListNode{

private:

friend typename SingleList<Type>;

ListNode():\_next(NULL){}

ListNode(const Type item,ListNode<Type> \*next=NULL):data(item),\_next(next){}

~ListNode(){

\_next=NULL;

}

public:

Type GetData();

friend ostream& operator<< (ostream& ,ListNode<Type>&);

private:

Type data;

ListNode \*\_next;

};

///GetData///

template<typename Type>

Type ListNode<Type>::GetData(){

return this->data;

}

///operator<<///

template<typename Type>

ostream& operator<<(ostream& os,ListNode<Type>& out){

os<<out.data;

return os;

}

SingleList.h

#include "ListNode.h"

template<typename T>

class SingleList{

public:

SingleList():\_head(new ListNode<T>()){}

~SingleList(){

MakeEmpty();

delete \_head;

}

public:

void MakeEmpty();

int Length();

ListNode<T> \*Find(int n);

bool Insert(T item,int n=0);

T Remove(int n=0);

bool RemoveAll(T item);

T Get(int n);

void Print();

private:

ListNode<T> \*\_head;

};

///MakeEmpty///

template<typename T>

void SingleList<T>::MakeEmpty(){

ListNode<T> \*\_delete;

while(\_head->\_next!=NULL){

\_delete=\_head->\_next;

\_head->\_next=\_delete->\_next;

delete \_delete;

}

}

///Length///

template<typename T>

int SingleList<T>::Length(){

ListNode<T> \*\_move=\_head->\_next;

int count=0;

while(\_move!=NULL){

\_move=\_move->\_next;

count++;

}

return count;

}

///Find///

template<typename T>

ListNode<T>\* SingleList<T>::Find(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

return NULL;

}

ListNode<T> \*\_move=\_head->\_next;

for(int i=0;i<n&&\_move;i++){

\_move=\_move->\_next;

}

if(\_move==NULL){

cout<<"The n is out of boundary"<<endl;

return NULL;

}

return \_move;

}

///Insert///

template<typename T>

bool SingleList<T>::Insert(T item, int n){

if(n<0){

cout<<"The n is illegal"<<endl;

return 0;

}

ListNode<T> \*\_move=\_head;

ListNode<T> \*pnode=new ListNode<T>(item);

if(pnode==NULL){

cout<<"Application error!"<<endl;

return 0;

}

for(int i=0;i<n&&\_move;i++){

\_move=\_move->\_next;

}

if(\_move==NULL){

cout<<"the n is illegal"<<endl;

return 0;

}

pnode->\_next=\_move->\_next;

\_move->\_next=pnode;

return 1;

}

///RemoveAll///

template<typename T>

bool SingleList<T>::RemoveAll(T item){//删去所有值为item的元素

ListNode<T> \*\_move=\_head;

ListNode<T> \*\_delete=\_head->\_next;

while(\_delete!=NULL){

if(\_delete->data==item){

\_move->\_next=\_delete->\_next;

delete \_delete;

\_delete=\_move->\_next;

continue;

}

\_move=\_move->\_next;

\_delete=\_delete->\_next;

}

return 1;

}

///Remove///

template<typename T>

T SingleList<T>::Remove(int n){

if(n<0){

cout<<"can't find the element"<<endl;

exit(1);

}

ListNode<T> \*\_move=\_head,\*\_delete;

for(int i=0;i<n&&\_move->\_next;i++){

\_move=\_move->\_next;

}

if(\_move->\_next==NULL){

cout<<"can't find the element"<<endl;

exit(1);

}

\_delete=\_move->\_next;

\_move->\_next=\_delete->\_next;

T temp=\_delete->data;

delete \_delete;

return temp;

}

///Get///

template<typename T>

T SingleList<T>::Get(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

ListNode<T> \*\_move=\_head->\_next;

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

\_move=\_move->\_next;

if(NULL==\_move){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

}

return \_move->data;

}

///Print///

template<typename T>

void SingleList<T>::Print(){

ListNode<T> \*\_move=\_head->\_next;

cout<<"head";

while(\_move){

cout<<"—>"<<\_move->data;

\_move=\_move->\_next;

}

cout<<"—>over"<<endl;

}

test.cpp

#include <iostream>

using namespace std;

#include "SingleList.h"

int main()

{

SingleList<int> list;

cout<<list.Length()<<endl;

for(int i=0;i<10;i++)

list.Insert(i,i);

list.Print();

list.Remove(3);

list.Print();

cout<<list.Get(3)<<endl;

cout<<\*list.Find(4)<<endl;

cout<<\*list.Find(6)<<endl;

list.Insert(3,5);

list.Insert(3,7);

list.Print();

list.RemoveAll(3);

list.Print();

list.MakeEmpty();

list.Print();

cout<<list.Length()<<endl;

return 0;

}

**双向链表**

ListNode

template<typename T>

class DoubleList;

///ListNode///

template<typename T>

class ListNode{

private:

friend class DoubleList<T>;//做此声明是为了让DoubleList类能够访问ListNode的私有成员

ListNode():\_prior(NULL),\_next(NULL){}

ListNode(const T item,ListNode<T> \*pprior=NULL,ListNode<T> \*pnext=NULL)

:data(item),\_prior(pprior),\_next(pnext){}

~ListNode(){

\_prior=NULL;

\_next=NULL;

}

public:

T GetData();

private:

T data;

ListNode \*\_prior;

ListNode \*\_next;

};

///GetData///

template<typename T>

T ListNode<T>::GetData(){

return this->data;

}

DoubleList.h

#include "ListNode.h"

//Ctrl+E，能够匹配两个括号

template<typename T>

class DoubleList{

public:

DoubleList():\_head(new ListNode<T>()){

\_head->\_prior=\_head;

\_head->\_next=\_head;

}

~DoubleList(){

MakeEmpty();

delete \_head;

}

void MakeEmpty();

int Length();

ListNode<T> \* PositionFind(int n=0);

ListNode<T> \* DataFind(T item);

bool Insert(T item,int n=0);

T Remove(int n=0);

T Get(int n=0);

void Print();

private:

ListNode<T> \*\_head;//带头结点

//带不带头结点的区别是：

//有头结点，第一个节点是head->next->data，没有头结点，第一个节点是head->data

};

///MakeEmpty///

template<typename T>

void DoubleList<T>::MakeEmpty(){

ListNode<T> \*\_move=\_head->\_next,\*\_delete;

while(\_move!=\_head){

\_delete=\_move;

\_move=\_delete->\_next;

delete \_delete;

}

\_head->\_next=\_head;

\_head->\_prior=\_head;

}

///Length///

template<typename T>

int DoubleList<T>::Length(){

/\* ListNode<T> \*pprior=\_head->\_prior,\*pnext=\_head->\_next;

int count=0;

while(true){

if(pprior->\_next==pnext){

break;

}

if(pprior==pnext&&pprior!=\_head){

count++;

break;

}

count+=2;

pprior=pprior->\_prior;

pnext=pnext->\_next;

}\*/

ListNode<T> \*p=\_head->\_next;

int count=0;

while(p!=\_head)

{

count++;

p=p->\_next;

}

return count;

}

///DataFind///

template<typename T>

ListNode<T>\* DoubleList<T>::DataFind(T item){

ListNode<T> \*pprior=\_head->\_prior,\*pnext=\_head->\_next;

while(pprior->\_next!=pnext && pprior!=pnext){ //find the data in the two direction

if(pprior->data==item){

return pprior;

}

if(pnext->data==item){

return pnext;

}

pprior=pprior->\_prior;

pnext=pnext->\_next;

}

cout<<"can't find the element"<<endl;

return NULL;

}

///PositionFind///

template<typename T>

ListNode<T>\* DoubleList<T>::PositionFind(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

return NULL;

}

ListNode<T> \*\_move=\_head->\_next;

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

\_move=\_move->\_next;

if(\_move==\_head){

cout<<"The n is out of boundary"<<endl;

return NULL;

}

}

return \_move;

}

///Insert///

template<typename T>

bool DoubleList<T>::Insert(T item,int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

return 0;

}

ListNode<T> \*\_node=new ListNode<T>(item),\*\_move=\_head;

if(\_node==NULL){

cout<<"Application Erorr!"<<endl;

exit(1);

}

for(int i=0;i<n;i++){ //find the position for insert

\_move=\_move->\_next;

if(\_move==\_head){

cout<<"The n is out of boundary"<<endl;

return 0;

}

}

\_node->\_next=\_move->\_next;

\_node->\_prior=\_move;

\_move->\_next=\_node;

\_node->\_next->\_prior=\_node;

return 1;

}

///Remove///

template<typename T>

T DoubleList<T>::Remove(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

ListNode<T> \*\_move=\_head,\*\_delete;

for(int i=0;i<n;i++){ //find the position for delete

\_move=\_move->\_next;

if(\_move==\_head){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

}

\_delete=\_move;

\_move->\_prior->\_next=\_delete->\_next;

\_move->\_next->\_prior=\_delete->\_prior;

T temp=\_delete->data;

delete \_delete;

return temp;

}

///Get///

template<typename T>

T DoubleList<T>::Get(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

ListNode<T> \*\_move=\_head;

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

\_move=\_move->\_next;

if(\_move==\_head){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

}

return \_move->data;

}

///Print///

template<typename T>

void DoubleList<T>::Print(){

ListNode<T> \*\_move=\_head->\_next;

cout<<"\_head";

while(\_move!=\_head){

cout<<"—>"<<\_move->data;

\_move=\_move->\_next;

}

cout<<"—>over"<<endl<<endl;

}

test.cpp

#include <iostream>

#include "DoubleList.h"

using namespace std;

int main()

{

DoubleList<int> list;

for(int i=0;i<10;i++){

list.Insert(i+1,i);

}

cout<<list.Length()<<endl;

cout<<(\*list.PositionFind(3)).GetData()<<endl;

cout<<(\*list.DataFind(4)).GetData()<<endl;

list.Remove(5);

list.Print();

cout<<list.Get(5)<<endl;

return 0;

}

**循环链表**

ListNode.h

template<typename Type> class CircularList;

///ListNode///

template<typename Type> class ListNode{

private:

friend class CircularList<Type>;

ListNode():\_next(NULL){}

ListNode(const Type item,ListNode<Type> \*next=NULL):data(item),\_next(next){}

~ListNode(){

\_next=NULL;

}

public:

Type GetData(){return data;}

ListNode\* GetNext(){return \_next;}

private:

Type data;

ListNode \*\_next;

};

CircularList.h

#include "ListNode.h"

///CircularList///

template<typename T> class CircularList{

public:

CircularList():head(new ListNode<T>()){

head->\_next=head;

}

~CircularList(){

MakeEmpty();

delete head;

}

public:

void MakeEmpty();

int Length();

ListNode<T> \*Find(int n);

bool Insert(T item,int n=0);

T Remove(int n=0);

bool RemoveAll(T item);

T Get(int n);

void Print();

private:

ListNode<T> \*head;

};

///MakeEmpty///

template<typename T> void CircularList<T>::MakeEmpty(){

ListNode<T> \*pdel,\*pmove=head;

while(pmove->\_next!=head){

pdel=pmove->\_next;

pmove->\_next=pdel->\_next;

delete pdel;

}

}

///Length///

template<typename T>

int CircularList<T>::Length(){

ListNode<T> \*pmove=head;

int count=0;

while(pmove->\_next!=head){

pmove=pmove->\_next;

count++;

}

return count;

}

///Find///

template<typename T>

ListNode<T>\* CircularList<T>::Find(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

return NULL;

}

ListNode<T> \*pmove=head->\_next;

for(int i=0;i<n&&pmove!=head;i++){

pmove=pmove->\_next;

}

if(pmove==head){

cout<<"The n is out of boundary"<<endl;

return NULL;

}

return pmove;

}

///Insert///

template<typename T>

bool CircularList<T>::Insert(T item, int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

return 0;

}

ListNode<T> \*pmove=head;

ListNode<T> \*pnode=new ListNode<T>(item);

if(pnode==NULL){

cout<<"Application error!"<<endl;

exit(1);

}

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

pmove=pmove->\_next;

if(pmove==head){

cout<<"The n is out of boundary"<<endl;

return 0;

}

}

pnode->\_next=pmove->\_next;

pmove->\_next=pnode;

return 1;

}

///RemoveAll///

template<typename T>

bool CircularList<T>::RemoveAll(T item){

ListNode<T> \*pmove=head;

ListNode<T> \*pdel=head->\_next;

while(pdel!=head){

if(pdel->data==item){

pmove->\_next=pdel->\_next;

delete pdel;

pdel=pmove->\_next;

continue;

}

pmove=pmove->\_next;

pdel=pdel->\_next;

}

return 1;

}

///Remove///

template<typename T>

T CircularList<T>::Remove(int n){

if(n<0){

cout<<"can't find the element"<<endl;

exit(1);

}

ListNode<T> \*pmove=head,\*pdel;

for(int i=0;i<n&&pmove->\_next!=head;i++){

pmove=pmove->\_next;

}

if(pmove->\_next==head){

cout<<"can't find the element"<<endl;

exit(1);

}

pdel=pmove->\_next;

pmove->\_next=pdel->\_next;

T temp=pdel->data;

delete pdel;

return temp;

}

///Get///

template<typename T>

T CircularList<T>::Get(int n){

if(n<0){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

ListNode<T> \*pmove=head->\_next;

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

pmove=pmove->\_next;

if(pmove==head){

cout<<"The n is out of boundary"<<endl;

exit(1);

}

}

return pmove->data;

}

///Print///

template<typename T>

void CircularList<T>::Print(){

ListNode<T> \*pmove=head->\_next;

cout<<"head";

while(pmove!=head){

cout<<"—>"<<pmove->data;

pmove=pmove->\_next;

}

cout<<"—>over"<<endl;

}

test.cpp

#include <iostream>

#include "CircularList.h"

using namespace std;

int main()

{

CircularList<int> list;

for(int i=0;i<5;i++)

list.Insert(i+1,i);

list.Print();

cout<<list.Length()<<endl;

cout<<(\*(\*list.Find(3)).GetNext()).GetData()<<endl;

list.Remove(3);

list.Print();

cout<<list.Get(3)<<endl;

list.Print();

list.Insert(2,3);

list.Insert(2);

list.Print();

list.RemoveAll(2);

list.Print();

list.MakeEmpty();

cout<<list.Length()<<endl;

return 0;

}

**顺序栈**

SeqStack.h

template<typename T> class SeqStack{

public:

SeqStack(int sz):top(-1),MaxSize(sz){

elements=new T[sz];

if(elements==NULL){

cout<<"Application Error!"<<endl;

exit(1);

}

}

~SeqStack(){

delete[] elements;

}

public:

void Push(const T item);

T Pop();

T GetTop() const;

void Print();

void MakeEmpty(){

top=-1;

}

bool IsEmpty() const{

return top==-1;

}

bool IsFull() const{

return top==MaxSize-1;

}

private:

int top;

T \*elements;

int MaxSize;

};

///Push///

template<typename T> void SeqStack<T>::Push(const T item){

if(IsFull()){

cout<<"The stack is full!"<<endl;

return;

}

elements[++top]=item;

}

///Pop///

template<typename T> T SeqStack<T>::Pop(){

if(IsEmpty()){

cout<<"There is no elements!"<<endl;

exit(1);

}

return elements[top--];

}

///GetTop///

template<typename T> T SeqStack<T>::GetTop() const{

if(IsEmpty()){

cout<<"There is no elements!"<<endl;

exit(1);

}

return elements[top];

}

///Print///

template<typename T> void SeqStack<T>::Print(){

cout<<"bottom";

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

cout<<"—>"<<elements[i];

}

cout<<"—>top"<<endl;

}

test.cpp

#include<iostream>

using namespace std;

#include "SeqStack.h"

int main(){

SeqStack<int> stack(10);

for(int i=0;i<10;i++)

stack.Push(i);

stack.Print();

if(stack.IsFull())

cout<<"Stack Full!"<<endl;

else

cout<<"not full"<<endl;

cout<<stack.GetTop()<<endl;

stack.Pop();

cout<<stack.GetTop()<<endl;

stack.MakeEmpty();

if(stack.IsEmpty())

cout<<"Empty stack!"<<endl;

else

cout<<"Sth. in stack"<<endl;

return 0;

}

**链式栈**

StackNode.h

template<typename T>

class LinkStack;

///StackNode///

template<typename T>

class StackNode{

private:

friend class LinkStack<T>;

StackNode(T dt,StackNode<T> \*next=NULL):data(dt),\_next(next){}

private:

T data;

StackNode<T> \*\_next;

};

LinkStack.h

#include "StackNode.h"

///LinkStack

template<typename T> class LinkStack{

public:

LinkStack():\_top(NULL){}

~LinkStack(){

MakeEmpty();

}

public:

void MakeEmpty();

void Push(const T item);

T Pop();

T GetTop() const;

void Print();

bool IsEmpty() const{

return \_top==NULL;

}

private:

StackNode<T> \*\_top;

};

///MakeEmpty///

template<typename T> void LinkStack<T>::MakeEmpty(){

StackNode<T> \*\_move;

while(\_top!=NULL){

\_move=\_top;

\_top=\_top->\_next;

delete \_move;

}

}

///Push///

template<typename T> void LinkStack<T>::Push(const T item){

\_top=new StackNode<T>(item,\_top);

}

///GetTop///

template<typename T> T LinkStack<T>::GetTop() const{

if(IsEmpty()){

cout<<"There is no elements!"<<endl;

exit(1);

}

return \_top->data;

}

///Pop///

template<typename T> T LinkStack<T>::Pop(){

if(IsEmpty()){

cout<<"There is no elements!"<<endl;

exit(1);

}

StackNode<T> \*pdel=\_top;

\_top=\_top->\_next;

T temp=pdel->data;

delete pdel;

return temp;

}

///Print///

template<typename T> void LinkStack<T>::Print(){

StackNode<T> \*\_move=\_top;

cout<<"top";

while(\_move!=NULL){

cout<<"<—"<<\_move->data;

\_move=\_move->\_next;

}

cout<<"<—bottom"<<endl;

}

test.cpp

#include <iostream>

using namespace std;

#include "LinkStack.h"

int main(){

LinkStack<int> stack;

stack.Push(1);

stack.Push(2);

stack.Push(3);

stack.Push(4);//先前添加失败，不知是何原因，如若检错，尝试一上来就用数组初始化栈

const int a[5]={5,6,7,8,9};

for(int i=0;i<5;i++)

stack.Push(a[i]);

stack.Print();

if(stack.IsEmpty())

cout<<"Empty!"<<endl;

else

cout<<"Not Empty"<<endl;

cout<<stack.Pop()<<endl;

cout<<stack.GetTop()<<endl;

stack.MakeEmpty();

if(stack.IsEmpty())

cout<<"Empty!"<<endl;

else

cout<<"Not Empty"<<endl;

int init[4]={1,2,3};

for(int in=0;in<3;in++)

stack.Push(init[in]);

stack.Print();

return 0;

}

**顺序队列**

SeqQueue.h

template<typename T>

class SeqQueue{

public:

SeqQueue(int sz):rear(0),front(0),count(0),MaxSize(sz){

\_elements=new T[sz];

if(\_elements==NULL){

cout<<"Application Error!"<<endl;

exit(1);

}

}

~SeqQueue(){

delete[] \_elements;

}

void MakeEmpty();

bool IsEmpty();

bool IsFull();

bool Append(const T item);

T Delete();

T Get();

void Print();

private:

int rear;

int front;

int count;

int MaxSize;

T \*\_elements;

};

///MakeEmpty///

template<typename T>

void SeqQueue<T>::MakeEmpty(){

this->count=0;

this->front=0;

this->rear=0;

}

///IsEmpty///

template<typename T>

bool SeqQueue<T>::IsEmpty(){

return count==0;

}

///IsFull///

template<typename T>

bool SeqQueue<T>::IsFull(){

return count==MaxSize;

}

///Append///

template<typename T>

bool SeqQueue<T>::Append(const T item){

if(IsFull()){

cout<<"The queue is full!"<<endl;

return 0;

}

\_elements[rear]=item;

rear=(rear+1)%MaxSize;

count++;

return 1;

}

///Delete///

template<typename T>

T SeqQueue<T>::Delete(){

if(IsEmpty()){

cout<<"There is no element!"<<endl;

exit(1);

}

T temp=\_elements[front];

front=(front+1)%MaxSize;

count--;

return temp;

}

///Get///

template<typename T>

T SeqQueue<T>::Get(){

if(IsEmpty()){

cout<<"There is no element!"<<endl;

exit(1);

}

return \_elements[front];

}

///Print///

template<typename T>

void SeqQueue<T>::Print(){

cout<<"front";

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

cout<<"—>"<<\_elements[(front+i+MaxSize)%MaxSize];

}

cout<<"—>rear"<<endl;

}

test.cpp

#include <iostream>

using namespace std;

#include "SeqQueue.h"

int main(){

SeqQueue<int> queue(10);

int init[10]={1,6,9,0,2,5,8,3,7,4};

for(int i=0;i<5;i++){

queue.Append(init[i]);

}

if(queue.IsFull())

cout<<"Full!"<<endl;

else

cout<<"Not Full!"<<endl;

queue.Print();

queue.Delete();

queue.Print();

cout<<endl;

cout<<queue.Get()<<endl;

queue.Print();

queue.MakeEmpty();

if(queue.IsEmpty())

cout<<"Empty!"<<endl;

else

cout<<"Not Empty!"<<endl;

return 0;

}

**链式队列**

QueueNode.h

template<typename T>

class LinkQueue;

///QueueNode///

template<typename T>

class QueueNode{

private:

friend class LinkQueue<T>;

QueueNode(const T item,QueueNode<T> \*next=NULL)

:data(item),\_next(next){}

private:

T data;

QueueNode<T> \*\_next;

};

LinkQueue.h

#include "QueueNode.h"

///LinkQueue///

template<typename T> class LinkQueue{

public:

LinkQueue():\_rear(NULL),\_front(NULL){}

~LinkQueue(){

MakeEmpty();

}

void Append(const T item);

T Delete();

T GetFront();

void MakeEmpty();

void Print();

bool IsEmpty() const{

return \_front==NULL;

}

private:

QueueNode<T> \*\_rear,\*\_front;

};

///MakeEmpty///

template<typename T> void LinkQueue<T>::MakeEmpty(){

QueueNode<T> \*\_del;

while(\_front){

\_del=\_front;

\_front=\_front->\_next;

delete \_del;

}

}

///Append///

template<typename T> void LinkQueue<T>::Append(const T item){

if(\_front==NULL){

\_front=\_rear=new QueueNode<T>(item);

}

else{

\_rear=\_rear->\_next=new QueueNode<T>(item);

}

}

///Delete///

template<typename T> T LinkQueue<T>::Delete(){

if(IsEmpty()){

cout<<"There is no element!"<<endl;

exit(1);

}

QueueNode<T> \*\_del=\_front;

T temp=\_front->data;

\_front=\_front->\_next;

delete \_del;

return temp;

}

///GetFront///

template<typename T> T LinkQueue<T>::GetFront(){

if(IsEmpty()){

cout<<"There is no element!"<<endl;

exit(1);

}

return \_front->data;

}

///Print///

template<typename T> void LinkQueue<T>::Print(){

QueueNode<T> \*\_move=\_front;

cout<<"front";

while(\_move){

cout<<"—>"<<\_move->data;

\_move=\_move->\_next;

}

cout<<"—>rear"<<endl;

}

test.cpp

#include <iostream>

using namespace std;

#include "LinkQueue.h"

int main(){

LinkQueue<int> queue;

int init[10]={1,3,6,8,9,2,0,5,4,7};

for(int i=0;i<10;i++){

queue.Append(init[i]);

}

queue.Print();

queue.Delete();

queue.Print();

cout<<queue.GetFront()<<endl;

queue.Print();

queue.MakeEmpty();

queue.Print();

queue.MakeEmpty();

if(queue.IsEmpty())

cout<<"Empty!"<<endl;

return 0;

}

**串**

MyString.h

#include <iostream>

#include <cstring>

using namespace std;

const int MAXSIZE=100;

///CMyString///

class CMyString

{

public:

CMyString(const CMyString& copy){ //initialize the string with string

\_str=new char[MAXSIZE+1];

if(!\_str){

cerr<<"Allocation Error"<<endl;

exit(1);

}

this->curlen=copy.curlen;

strcpy(\_str,copy.\_str);

}

CMyString(const char \*init){ //initialize the string with char\*

\_str=new char[MAXSIZE+1];

if(!\_str){

cerr<<"Allocation Error"<<endl;

exit(1);

}

this->curlen=strlen(init);

strcpy(\_str,init);

}

CMyString(){ //create empty string

\_str=new char[MAXSIZE+1];

if(!\_str){

cerr<<"Allocation Error"<<endl;

exit(1);

}

this->curlen=0;

\_str[0]='\0';

}

~CMyString(){

delete[] \_str;

}

int Length() const{

return curlen;

}

int Find(CMyString part) const;

char\* GetBuffer() const{ //get the char\* from string

return this->\_str;

}

public:

CMyString& operator()(int pos,int len);

bool operator==(const CMyString cmp\_str) const;

bool operator!=(const CMyString cmp\_str) const;

bool operator<(const CMyString cmp\_str) const;

bool operator>(const CMyString cmp\_str) const;

bool operator!() const{

return curlen==0;

}

CMyString& operator=(const CMyString &copy);

CMyString& operator+=(const CMyString &add);

char& operator[](int i);

friend ostream& operator<<(ostream& os,CMyString& str){

os<<str.\_str;

return os;

}

friend istream& operator>>(istream& is,CMyString& str){

is>>str.\_str;

return is;

}

private:

void Next();

private:

int curlen;

char \*\_str;

int \*\_next;

};

///Find///

int CMyString::Find(CMyString part) const{ //string match :KMP

int posP=0,posT=0;

int lengthP=part.curlen,lengthT=this->curlen;

part.Next();

while(posP<lengthP&&posT<lengthT){

if(part.\_str[posP]==this->\_str[posT]){

posP++;

posT++;

}

else{

if(posP==0){

posT++;

}

else{

posP=part.\_next[posP-1];

}

}

}

delete[] part.\_next;

if(posP<lengthP){

return 0;

}

else{

return 1;

}

}

///operator()///

CMyString& CMyString::operator()(int pos, int len){ //get len char with the begining of pos

CMyString \*temp=new CMyString; //dangerous operation

if(pos<0||pos+len-1>MAXSIZE||len<0){

temp->curlen=0;

temp->\_str[0]='\0';

}

else{

if(pos+len-1>=curlen){

len=curlen-pos;

}

temp->curlen=len;

for(int i=0,j=pos;i<len;i++,j++){

temp->\_str[i]=\_str[j];

}

temp->\_str[len]='\0';

}

return \*temp;

}

///operator==///

bool CMyString::operator==(const CMyString cmp\_str) const{

if(this->curlen!=cmp\_str.curlen){

return 0;

}

for(int i=0;i<this->curlen;i++){

if(this->\_str[i]!=cmp\_str.\_str[i])

return 0;

}

return 1;

}

///operator!=///

bool CMyString::operator!=(const CMyString cmp\_str) const{

if(\*this==cmp\_str)

return 0;

return 1;

}

///operator<///

bool CMyString::operator<(const CMyString cmp\_str) const{

if(this->curlen!=cmp\_str.curlen){

//Perhaps here it is also compared not by length but by order in dictonary. Hanlian 20100117

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*hanlian fix start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#ifdef HAN\_FIX

int shortter = this->curlen<cmp\_str.curlen? curlen: cmp\_str.curlen;

int flag\_cmp = shortter < curlen? 2:1; //specific which string is shortter.

for (int index = 0; index < shortter; index++)

{

if (this->\_str[index] == cmp\_str.\_str[index])

{

continue;

}

else if (this->\_str[index] < cmp\_str.\_str[index])

{

return 1;

}

else

{

return 0;

}

}

if (index == shortter) //the long string starts with the short string.

{

if (flag\_cmp == 2)

{

return 0;

}

else

{

returen 1;

}

}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*hanlian fix stop\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

return this->curlen<cmp\_str.curlen;

}

for(int i=0;i<this->curlen;i++){

if(this->\_str[i]!=cmp\_str.\_str[i]){

return this->\_next[i]<cmp\_str.\_next[i];

}

}

return 0;

}

///operator>///

bool CMyString::operator>(const CMyString cmp\_str) const{

if(\*this<cmp\_str||\*this==cmp\_str){

return 0;

}

return 1;

}

///operator=///

CMyString& CMyString::operator=(const CMyString &copy){ //赋值操作

delete[] this->\_str;

this->\_str=new char[copy.curlen+1];

strcpy(this->\_str,copy.\_str);

return \*this;

}

///operator+=///

CMyString& CMyString::operator+=(const CMyString &add){ //字符串追加

int length=this->curlen+add.curlen;

int n=this->curlen;

CMyString temp(\*this);

delete[] this->\_str;

this->\_str=new char[length+1];

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

this->\_str[i]=temp[i];

}

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

this->\_str[i]=add.\_str[i-n];

}

this->\_str[length]='\0';

return \*this;

}

char& CMyString::operator[](int i){ //取元素

if(i<0||i>=this->curlen){

cout<<"out of boundary!"<<endl;

exit(1);

}

return this->\_str[i];

}

///Next///

void CMyString::Next(){ //get the next char for matching : KMP

int length=this->curlen;

this->\_next=new int[length];

this->\_next[0]=0;

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

int j=this->\_next[i-1];

while(\*(this->\_str+i)!=\*(this->\_str+j)&&j>0){

j=this->\_next[j-1];

}

if(\*(this->\_str+i)==\*(this->\_str+j)){

this->\_next[i]=j+1;

}

else{

this->\_next[i]=0;

}

}

// for(int i=0;i<length;i++)

// cout<<i<<":\t"<<\_next[i]<<endl;

}

test.cpp

#include <iostream>

using namespace std;

#include "MyString.h"

int main(){

CMyString test1("babc");

CMyString test2("abababcdefb");

cout<<test2.Find(test1)<<endl;

cout<<test2(2,3)<<endl;

if(test1<test2){

cout<<test1<<"<"<<test2<<endl ;

}

else{

if(test1==test2){

cout<<test1<<"=="<<test2<<endl;

}

else{

if(test1>test2){

cout<<test1<<">"<<test2<<endl;

}

}

}

int length=test2.Length();

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

cout<<test2[i];

}

cout<<endl;

test1+=test2;

cout<<test1<<endl;

test1=test2;

cout<<test1<<endl;

return 0;

}

**二叉树**

BinTreeNode.h

template<typename T> class BinaryTree;

///BinTreeNode///

template<typename T> class BinTreeNode{

public:

friend class BinaryTree<T>;

BinTreeNode():\_left(NULL),\_right(NULL){}

BinTreeNode(T item,BinTreeNode<T> \*left=NULL,BinTreeNode<T> \*right=NULL)

:data(item),\_left(left),\_right(right){}

T GetData() const;

BinTreeNode<T> \*GetLeft() const;

BinTreeNode<T> \*GetRight() const;

void SetData(const T data);

void SetLeft(const BinTreeNode<T> \*left);

void SetRight(const BinTreeNode<T> \*right);

void InOrder();

void PreOrder();

void PostOrder();

int Size();

int Height();

BinTreeNode<T> \*Copy(const BinTreeNode<T> \*copy);

void Destroy(){

if(this!=NULL){

this->\_left->Destroy();

this->\_right->Destroy();

delete this;

}

}

friend bool equal(const BinTreeNode<T> \*s,const BinTreeNode<T> \*t){//<T>

if(s==NULL&&t==NULL){

return 1;

}

if(s&&t&&s->data==t->data&&equal(s->\_left,t->\_left)&&equal(s->\_right,t->\_right)){

return 1;

}

return 0;

}

private:

BinTreeNode<T> \*\_left,\*\_right;

T data;

};

///GetData///

template<typename T> T BinTreeNode<T>::GetData() const{

return this!=NULL?data:-1;

}

///GetLeft///

template<typename T> BinTreeNode<T>\* BinTreeNode<T>::GetLeft() const{

return this!=NULL?\_left:NULL;

}

///GetRight///

template<typename T> BinTreeNode<T>\* BinTreeNode<T>::GetRight() const{

return this!=NULL?\_right:NULL;

}

///SetData///

template<typename T> void BinTreeNode<T>::SetData(const T data){

if(this!=NULL){

data=data;

}

}

///SetLeft///

template<typename T> void BinTreeNode<T>::SetLeft(const BinTreeNode<T> \*left){

if(this!=NULL){

\_left=left;

}

}

///SetRight///

template<typename T> void BinTreeNode<T>::SetRight(const BinTreeNode<T> \*right){

if(this!=NULL){

\_right=right;

}

}

///Copy///

template<typename T> BinTreeNode<T>\* BinTreeNode<T>::Copy(const BinTreeNode<T> \*copy){

if(copy==NULL){

return NULL;

}

BinTreeNode<T> \*temp=new BinTreeNode<T>(copy->data);

temp->\_left=Copy(copy->\_left);

temp->\_right=Copy(copy->\_right);

return temp;

}

///InOrder///

template<typename T> void BinTreeNode<T>::InOrder(){

if(this!=NULL){

this->\_left->InOrder();

cout<<"--->"<<this->data;

this->\_right->InOrder();

}

}

///PreOrder///

template<typename T> void BinTreeNode<T>::PreOrder(){

if(this!=NULL){

cout<<"--->"<<this->data;

this->\_left->PreOrder();

this->\_right->PreOrder();

}

}

///PostOrder///

template<typename T> void BinTreeNode<T>::PostOrder(){

if(this!=NULL){

this->\_left->PostOrder();

this->\_right->PostOrder();

cout<<"--->"<<this->data;

}

}

///Size///

template<typename T> int BinTreeNode<T>::Size(){

if(this==NULL){

return 0;

}

return 1+this->\_left->Size()+this->\_right->Size(); // this line has protential danger. Hanlian, should be fixed.

}

///Height///

template<typename T> int BinTreeNode<T>::Height(){

if(this==NULL){

return -1;

}

int lheight,rheight;

lheight=this->\_left->Height();

rheight=this->\_right->Height();

return 1+(lheight>rheight?lheight:rheight);

}

BinaryTree.h

#include "BinTreeNode.h"

///BinaryTree///

template<typename Type>

class BinaryTree{

public:

BinaryTree():\_root(NULL){}

BinaryTree(const Type Stop):stop(Stop),\_root(NULL){}

BinaryTree(BinaryTree<Type>& copy);

virtual ~BinaryTree(){

\_root->Destroy();

}

virtual bool IsEmpty(){

return \_root==NULL;

}

virtual BinTreeNode<Type> \*GetLeft(BinTreeNode<Type> \*\_current);

virtual BinTreeNode<Type> \*GetRight(BinTreeNode<Type> \*\_current);

virtual BinTreeNode<Type> \*GetParent(BinTreeNode<Type> \*\_current);

const BinTreeNode<Type> \*GetRoot() const;

virtual bool Insert(const Type item);

virtual BinTreeNode<Type> \*Find(const Type item) const;

void InOrder();

void PreOrder();

void PostOrder();

int Size();

int Height();

BinaryTree<Type>& operator=(const BinaryTree<Type> copy);

friend bool operator== (const BinaryTree<Type> s,const BinaryTree<Type> t){

return equal(s.\_root,t.\_root);

}

friend ostream& operator<< (ostream& os,BinaryTree<Type>& out){

out.Print(out.\_root);

return os;

}

friend istream& operator>>(istream& is,BinaryTree<Type>& in){

Type item;

cout<<"initialize the tree:"

<<endl

<<"Input data(end with "<<in.stop<<"!):";

is>>item;

while(item!=in.stop){

in.Insert(item);

is>>item;

}

return is;

}

private:

Type stop;

BinTreeNode<Type> \*\_root;

BinTreeNode<Type> \*GetParent(BinTreeNode<Type> \*\_start,BinTreeNode<Type> \*\_current);

void Print(BinTreeNode<Type> \*\_start,int n=0);

};

///Constructor///

template<typename Type>

BinaryTree<Type>::BinaryTree(BinaryTree<Type>& copy){

if(copy.\_root){

this->stop=copy.stop;

}

\_root=\_root->Copy(copy.\_root);

}

///GetLeft///

template<typename Type>

BinTreeNode<Type>\* BinaryTree<Type>::GetLeft(BinTreeNode<Type> \*\_current){

return \_root&&\_current?\_current->\_left:NULL;

}

///GetRight///

template<typename Type>

BinTreeNode<Type>\* BinaryTree<Type>::GetRight(BinTreeNode<Type> \*\_current){

return \_root&&\_current?\_current->\_right:NULL;

}

///GetRoot///

template<typename Type>

const BinTreeNode<Type>\* BinaryTree<Type>::GetRoot() const{

return \_root;

}

///GetParent///

template<typename Type>

BinTreeNode<Type>\* BinaryTree<Type>::GetParent(BinTreeNode<Type> \*\_start, BinTreeNode<Type> \*\_current){

if(\_start==NULL||\_current==NULL){

return NULL;

}

if(\_start->\_left==\_current||\_start->\_right==\_current){

return \_start;

}

BinTreeNode<Type> \*pmove;

if((pmove=GetParent(\_start->\_left,\_current))!=NULL){

return pmove;

}

else{

return GetParent(\_start->\_right,\_current);

}

}

///GetParent///

template<typename Type>

BinTreeNode<Type>\* BinaryTree<Type>::GetParent(BinTreeNode<Type> \*\_current){

return \_root==NULL||\_current==\_root?NULL:GetParent(\_root,\_current);

}

///Insert///

template<typename Type>

bool BinaryTree<Type>::Insert(const Type item){

BinTreeNode<Type> \*pstart=\_root,\*newnode=new BinTreeNode<Type>(item);

if(\_root==NULL){

\_root=newnode;

return 1;

}

while(1){

if(item==pstart->data){

cout<<"The item "<<item<<" is exist!"<<endl;

return 0;

}

if(item<pstart->data){

if(pstart->\_left==NULL){

pstart->\_left=newnode;

return 1;

}

pstart=pstart->\_left;

}

else{

if(pstart->\_right==NULL){

pstart->\_right=newnode;

return 1;

}

pstart=pstart->\_right;

}

}

}

///Find///

template<typename Type>

BinTreeNode<Type>\* BinaryTree<Type>::Find(const Type item) const{

BinTreeNode<Type> \*pstart=\_root;

while(pstart){

if(item==pstart->data){

return pstart;

}

if(item<pstart->data){

pstart=pstart->\_left;

}

else{

pstart=pstart->\_right;

}

}

return NULL;

}

///Print///

template<typename Type>

void BinaryTree<Type>::Print(BinTreeNode<Type> \*\_start, int n){

if(\_start==NULL){

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

cout<<" ";

}

cout<<"NULL"<<endl;

return;

}

Print(\_start->\_right,n+1);

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

cout<<" ";

}

if(n>=0){

cout<<\_start->data<<"—>"<<endl;

}

Print(\_start->\_left,n+1);

}

///Operator=///

template<typename Type>

BinaryTree<Type>& BinaryTree<Type>::operator=(const BinaryTree<Type> copy){

if(copy.\_root){

this->stop=copy.stop;

}

\_root=\_root->Copy(copy.\_root);

return \*this;

}

///InOrder///

template<typename Type>

void BinaryTree<Type>::InOrder(){

this->\_root->InOrder();

}

///PreOrder///

template<typename Type>

void BinaryTree<Type>::PreOrder(){

this->\_root->PreOrder();

}

///PostOrder///

template<typename Type>

void BinaryTree<Type>::PostOrder(){

this->\_root->PostOrder();

}

///Size///

template<typename Type>

int BinaryTree<Type>::Size(){

return this->\_root->Size();

}

///Height///

template<typename Type>

int BinaryTree<Type>::Height(){

return this->\_root->Height();

}

test.cpp

#include <iostream>

using namespace std;

#include "BinaryTree.h"

int main(){

BinaryTree<int> tree(-1);

int init[30]={17,6,22,29,14,0,21,13,27,18,2,28,8

,26,3,12,20,4,9,23,15,1,11,5,19,24,16,7,10,25};

for(int i=0;i<30;i++){

tree.Insert(init[i]);

}

// cin>>tree; 本语句可以输入节点值

cout<<tree<<endl;

cout<<tree.GetParent(tree.Find(20))->GetData()<<endl;

cout<<tree.Find(15)->GetRight()->GetData()<<endl;

cout<<"size="<<tree.Size()<<endl;

cout<<"height="<<tree.Height()<<endl;

tree.InOrder();

cout<<endl<<endl;

tree.PreOrder();

cout<<endl<<endl;

tree.PostOrder();

cout<<endl<<endl;

BinaryTree<int> tree2=tree;

cout<<tree2<<endl;

cout<<tree2.GetParent(tree2.Find(20))->GetData()<<endl;

cout<<tree2.Find(15)->GetRight()->GetData()<<endl;

cout<<(tree==tree2)<<endl;

return 0;

}

**线索二叉树**

ThreadNode.h

template<typename T> class ThreadTree;

template<typename T> class ThreadInorderIterator;

///ThreadNode///

template<typename T> class ThreadNode{

public:

friend class ThreadTree<T>;

friend class ThreadInorderIterator<T>;

ThreadNode():leftThread(1),rightThread(1){

\_left=this;

\_right=this;

}

ThreadNode(const T item):data(item),\_left(NULL),\_right(NULL)

,leftThread(0),rightThread(0){}

private:

int leftThread,rightThread;

ThreadNode<T> \*\_left,\*\_right;

T data;

};

ThreadTree.h

#include "ThreadNode.h"

template<typename T> class ThreadInorderIterator;

///ThreadTree///

template<typename T> class ThreadTree{

public:

friend class ThreadInorderIterator<T>;

ThreadTree():\_root(new ThreadNode<T>()){}

private:

ThreadNode<T> \*\_root;

};

ThreadInorderIterator.h

#include "ThreadTree.h"

#include<iostream>

using std::cout;

using std::endl;

template<typename Type>

class ThreadInorderIterator{

public:

ThreadInorderIterator(ThreadTree<Type> &tree):tree\_(tree),\_current(tree.\_root){

//InThread(tree\_.\_root->\_left,tree\_.\_root);

}

ThreadNode<Type> \*First();

ThreadNode<Type> \*Prior();

ThreadNode<Type> \*Next();

void Print();

void Print(ThreadNode<Type> \*start, int n=0);

void InOrder();

void InsertLeft(ThreadNode<Type> \*left);

void InsertRight(ThreadNode<Type> \*right);

ThreadNode<Type> \*GetParent(ThreadNode<Type> \*pcurrent);

private:

ThreadTree<Type> &tree\_;

ThreadNode<Type> \*\_current;

void InThread(ThreadNode<Type> \*pcurrent,ThreadNode<Type> \*pre);

};

///InThread///

template<typename Type>

void ThreadInorderIterator<Type>::InThread(

ThreadNode<Type> \*pcurrent, ThreadNode<Type> \*pre){

if(pcurrent!=tree\_.\_root){

InThread(pcurrent->\_left,pre);

if(pcurrent->\_left==NULL){

pcurrent->\_left=pre;

pcurrent->leftThread=1;

}

if(pre->\_right==NULL){

pre->\_right=pcurrent;

pre->rightThread=1;

}

pre=pcurrent;

InThread(pcurrent->\_right,pre);

}

}

///First///

template<typename Type>

ThreadNode<Type>\* ThreadInorderIterator<Type>::First(){

while(\_current->leftThread==0){

\_current=\_current->\_left;

}

return \_current;

}

///Prior///

template<typename Type>

ThreadNode<Type>\* ThreadInorderIterator<Type>::Prior(){

ThreadNode<Type> \*pmove=\_current->\_left;

if(0==\_current->leftThread){

while(0==pmove->rightThread){

pmove=pmove->\_right;

}

}

\_current=pmove;

if(\_current==tree\_.\_root){

return NULL;

}

return \_current;

}

///Next///

template<typename Type>

ThreadNode<Type>\* ThreadInorderIterator<Type>::Next(){

ThreadNode<Type> \*pmove=\_current->\_right;

if(0==\_current->rightThread){

while(0==pmove->leftThread){

pmove=pmove->\_left;

}

}

\_current=pmove;

if(\_current==tree\_.\_root){

return NULL;

}

return \_current;

}

///InOrder///

template<typename Type>

void ThreadInorderIterator<Type>::InOrder(){

ThreadNode<Type> \*pmove=tree\_.\_root;

while(pmove->\_left!=tree\_.\_root){

pmove=pmove->\_left;

}

\_current=pmove;

cout<<"root";

while(pmove!=tree\_.\_root&&pmove){

cout<<"—>"<<pmove->data;

pmove=this->Next();

}

cout<<"—>end";

}

///InsertLeft///

template<typename Type>

void ThreadInorderIterator<Type>::InsertLeft(ThreadNode<Type> \*left){

left->\_left=\_current->\_left;

left->leftThread=\_current->leftThread;

left->\_right=\_current;

left->rightThread=1;

\_current->\_left=left;

\_current->leftThread=0;

if(0==left->leftThread){

\_current=left->\_left;

ThreadNode<Type> \*temp=First();

temp->\_right=left;

}

\_current=left;

}

///InsertRight///

template<typename Type>

void ThreadInorderIterator<Type>::InsertRight(ThreadNode<Type> \*right){

right->\_right=\_current->\_right;

right->rightThread=\_current->rightThread;

right->\_left=\_current;

right->leftThread=1;

\_current->\_right=right;

\_current->rightThread=0;

if(0==right->rightThread){

\_current=right->\_right;

ThreadNode<Type> \*temp=First();

temp->\_left=right;

}

\_current=right;

}

///Getparent///

template<typename Type>

ThreadNode<Type>\* ThreadInorderIterator<Type>::GetParent(

ThreadNode<Type> \*pcurrent){

ThreadNode<Type> \*pmove=pcurrent;

while(0==pmove->leftThread){

pmove=pmove->\_left;

}

pmove=pmove->\_left;

if(pmove==tree\_.\_root){

if(pmove->\_left==pcurrent){

return NULL;

}

}

if(pmove->\_right==pcurrent){

return pmove;

}

pmove=pmove->\_right;

while(pmove->\_left!=pcurrent){

pmove=pmove->\_left;

}

return pmove;

}

///Print///

template<typename Type>

void ThreadInorderIterator<Type>::Print(ThreadNode<Type> \*start, int n){

if(start->leftThread&&start->rightThread){

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

cout<<" ";

}

if(n>=0){

cout<<start->data<<"—>"<<endl;

}

return;

}

if(start->rightThread==0){

Print(start->\_right,n+1);

}

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

cout<<" ";

}

if(n>=0){

cout<<start->data<<"—>"<<endl;

}

if(start->leftThread==0){

Print(start->\_left,n+1);

}

}

///Print///

template<typename Type>

void ThreadInorderIterator<Type>::Print(){

Print(tree\_.\_root->\_left);

}

test.cpp

#include <iostream>

using namespace std;

#include "ThreadInorderIterator.h"

int main(){

ThreadTree<int> tree;

ThreadInorderIterator<int> threadtree(tree);

int init[10]={3,6,0,2,8,4,9,1,5,7};

for(int i=0;i<10;){

threadtree.InsertLeft(new ThreadNode<int>(init[i++]));

threadtree.InsertRight(new ThreadNode<int>(init[i++]));

}

threadtree.Print();

cout<<endl<<endl;

threadtree.InOrder();

return 0;

}

**堆**

MinHeap.h

#include<iostream>

using namespace std;

template<typename T>

class MinHeap{

public:

MinHeap(int size):maxSize(size > defaultSize ? size : defaultSize)

,\_heap(new T[maxSize]),currentSize(0){}

MinHeap(T heap[],int n);

~MinHeap(){

delete[] \_heap;

}

public:

bool Insert(const T item);

bool Delete(const T item);

bool IsEmpty() const{

return currentSize == 0;

}

bool IsFull() const{

reutrn currentSize == maxSize;

}

void Print(const int start=0, int n=0);

private:

void Adjust(const int start, const int end);

private:

static const int defaultSize;

const int maxSize;

T \*\_heap;

int currentSize;

};

//静态常量在类外初始化

template<typename T>

const MinHeap<T>::defaultSize=100;

///Constructor///

template<typename T>

MinHeap<T>::MinHeap(T heap[], int n):maxSize(

n > defaultSize ? n : defaultSize){

\_heap = new T[maxSize];

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

\_heap[i] = heap[i];

}

currentSize = n;

int pos=(n-2)/2;

while(pos>=0){

Adjust(pos, n-1);

pos--;

}

}

///Adjust///

template<typename T>

void MinHeap<T>::Adjust(const int start, const int end){

int i = start,j = i\*2+1;

T temp=\_heap[i];

while(j <= end){

if(j<end && \_heap[j]>\_heap[j+1]){

j++;

}

if(temp <= \_heap[j]){

break;

}

else{

\_heap[i] = \_heap[j];

i = j;

j = 2\*i+1;

}

}

\_heap[i] = temp;

}

///Insert///

template<typename T>

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

if(currentSize == maxSize){

cerr<<"Heap Full!"<<endl;

return 0;

}

\_heap[currentSize] = item;

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

T temp = \_heap[j];

while(j > 0){

if(\_heap[i] <= temp){

break;

}

else{

\_heap[j] = \_heap[i];

j = i;

i = (j-1)/2;

}

}

\_heap[j] = temp;

currentSize++;

return 1;

}

///Delete///

template<typename T>

bool MinHeap<T>::Delete(const T item){

if(0 == currentSize){

cerr<<"Heap Empty!"<<endl;

return 0;

}

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

if(\_heap[i] == item){

\_heap[i] = \_heap[currentSize-1];

Adjust(i,currentSize-2);

currentSize--;

i=0;

}

}

return 1;

}

///Print///

template<typename T>

void MinHeap<T>::Print(const int start, int n){

if(start >= currentSize){

return;

}

Print(start\*2+2, n+1);

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

cout<<" ";

}

cout<< \_heap[start] << "—>" << endl;

Print(start\*2+1, n+1);

}

test.cpp

#include <iostream>

using namespace std;

#include "MinHeap.h"

int main(){

int init[30]={17,6,22,29,14,0,21,13,27,18,2,28,8

,26,3,12,20,4,9,23,15,1,11,5,19,24,16,7,10,25};

MinHeap<int> heap(init,30);

heap.Print();

cout<<"-------------------------"<<endl;

heap.Insert(20);

heap.Print();

cout<<"-------------------------"<<endl;

heap.Delete(20);

heap.Print();

cout<<"-------------------------"<<endl;

return 0;

}

**哈夫曼树（未能运行）**

BinTreeNode.h

#pragma once

#include "BinTreeNode.h"

template<typename T>

void Huffman(T \*, int, BinaryTree<T> &);

template<typename T>

class BinaryTree{

public:

BinaryTree(BinaryTree<T> &bt1, BinaryTree<T> &bt2){

\_root = new BinTreeNode<T>(bt1.\_root->m\_data

+ bt2.\_root->m\_data, bt1.\_root, bt2.\_root);

}

BinaryTree(T item){

\_root = new BinTreeNode<T>(item);

}

BinaryTree(const BinaryTree<T> &copy){

this->\_root = copy.\_root;

}

BinaryTree(){

\_root = NULL;

}

void Destroy(){

\_root->Destroy();

}

~BinaryTree(){

// \_root->Destroy();

}

BinaryTree<T>& operator=(BinaryTree<T> copy){

\_root=\_root->Copy(copy.\_root);

return \*this;

}//evaluate node

friend void Huffman(T \*, int, BinaryTree<T> &);

friend bool operator <(BinaryTree<T> &l, BinaryTree<T> &r){

return l.\_root->GetData() < r.\_root->GetData();

}

friend bool operator >(BinaryTree<T> &l, BinaryTree<T> &r){

return l.\_root->GetData() > r.\_root->GetData();

}

friend bool operator <=(BinaryTree<T> &l, BinaryTree<T> &r){

return l.\_root->GetData() <= r.\_root->GetData();

}

friend ostream& operator<<(ostream& os,BinaryTree<T>& out){

out.Print(out.\_root);

return os;

};//output the data

private:

BinTreeNode<T> \*\_root;

void Print(BinTreeNode<T> \*start,int n=0); //print the tree with the root of start

};

template<typename T> void BinaryTree<T>::Print(BinTreeNode<T> \*start, int n){

if(start==NULL){

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

cout<<" ";

}

cout<<"NULL"<<endl;

return;

}

Print(start->m\_pright,n+1); //print the right subtree

for(int i=0;i<n;i++){ //print blanks with the height of the node

cout<<" ";

}

if(n>=0){

cout<<start->m\_data<<"--->"<<endl;//print the node

}

Print(start->m\_pleft,n+1); //print the left subtree

}

BinaryTree.h

#pragma once

#include "BinTreeNode.h"

template<typename T>

void Huffman(T \*, int, BinaryTree<T> &);

template<typename T>

class BinaryTree{

public:

BinaryTree(BinaryTree<T> &bt1, BinaryTree<T> &bt2){

\_root = new BinTreeNode<T>(bt1.\_root->m\_data

+ bt2.\_root->m\_data, bt1.\_root, bt2.\_root);

}

BinaryTree(T item){

\_root = new BinTreeNode<T>(item);

}

BinaryTree(const BinaryTree<T> &copy){

this->\_root = copy.\_root;

}

BinaryTree(){

\_root = NULL;

}

void Destroy(){

\_root->Destroy();

}

~BinaryTree(){

// \_root->Destroy();

}

BinaryTree<T>& operator=(BinaryTree<T> copy){

\_root=\_root->Copy(copy.\_root);

return \*this;

}//evaluate node

friend void Huffman(T \*, int, BinaryTree<T> &);

friend bool operator <(BinaryTree<T> &l, BinaryTree<T> &r){

return l.\_root->GetData() < r.\_root->GetData();

}

friend bool operator >(BinaryTree<T> &l, BinaryTree<T> &r){

return l.\_root->GetData() > r.\_root->GetData();

}

friend bool operator <=(BinaryTree<T> &l, BinaryTree<T> &r){

return l.\_root->GetData() <= r.\_root->GetData();

}

friend ostream& operator<<(ostream& os,BinaryTree<T>& out){

out.Print(out.\_root);

return os;

};//output the data

private:

BinTreeNode<T> \*\_root;

void Print(BinTreeNode<T> \*start,int n=0); //print the tree with the root of start

};

template<typename T> void BinaryTree<T>::Print(BinTreeNode<T> \*start, int n){

if(start==NULL){

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

cout<<" ";

}

cout<<"NULL"<<endl;

return;

}

Print(start->m\_pright,n+1); //print the right subtree

for(int i=0;i<n;i++){ //print blanks with the height of the node

cout<<" ";

}

if(n>=0){

cout<<start->m\_data<<"--->"<<endl;//print the node

}

Print(start->m\_pleft,n+1); //print the left subtree

}

Huffman.h

#pragma once

#include "BinaryTree.h"

#include "MinHeap.h"

template<typename Type>

void Huffman(Type \*elements, int n, BinaryTree<Type> &tree){

BinaryTree<Type> first, second;

BinaryTree<Type> node[20];

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

node[i].m\_proot = new BinTreeNode<Type>(elements[i]);

}

MinHeap<BinaryTree<Type> > heap(node, n);

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

heap.DeleteMin(first);

heap.DeleteMin(second);

//using the first and the second minimize element create new tree

if (first.m\_proot->GetData() == second.m\_proot->GetData()){

tree = \*(new BinaryTree<Type>(second, first));

}

else {

tree = \*(new BinaryTree<Type>(first, second));

}

heap.Insert(tree);

}

}

test.cpp

#include <iostream>

#include "Huffman.h"

using namespace std;

int main(){

BinaryTree<int> tree;

int init[10]={3,6,0,2,8,4,9,1,5,7};

Huffman(init,10,tree);

cout << tree;

tree.Destroy();

return 0;

}

别的版本：

#include<iostream>

using namespace std;

#define MaxLeaf 50 //叶子数最大值

#define MaxNode 500 //最大节点数目

#define MaxCode 50 //最大编码位数

#define MaxValue 100000 //最大权值

typedef struct node //节点类型定义

{

　　int weight;

　　char letter[10];

　　int parent;

　　int lchild;

　　int rchild;

}huffnode;

typedef struct code //编码类型定义

{

　　int bits[MaxCode];

　　int start;

}huffcode;

void HuffManTree(huffnode a[],int n) //构造哈夫曼树

{

　　 int i,j,m1,m2,x1,x2;

　　　　for(i=0;i<n-1;i++)

　　　　{

　　　　　　m1=m2=MaxValue;

　　　　　　x1=x2=0;

　　　　　　for(j=0;j<n+i;j++) //找出最小的两个权值节点

　　　　　　{

　　　　　　　　if(a[j].parent==-1&&a[j].weight<m1)

　　　　　　　　{

　　　　　　　　　　m2=m1;

　　　　　　　　　　x2=x1;

　　　　　　　　　　m1=a[j].weight;

　　　　　　　　　　x1=j;

　　　　　　　　}

　　　　　　　　else

　　　　　　　　　　if(a[j].parent==-1&&a[j].weight<m2)

　　　　　　　　　　{

　　　　　　　　　　　　m2=a[j].weight;

　　　　　　　　　　　　x2=j;

　　　　　　　　　　}

　　　　　　}

　　　　　　a[x1].parent=n+i;

　　　　　　a[x2].parent=n+i;

　　　　　　a[n+i].lchild=x1;

　　　　　　a[n+i].rchild=x2;

　　　　　　a[n+i].weight=a[x1].weight+a[x2].weight;

　　　　}

}

void HuffManCode(huffnode a[],int n) //哈夫曼编码

{

　　int i,j,c,p;

　　　　huffcode cd ,code[MaxNode];

　　　　HuffManTree(a,n);

　　　　for(i=0;i<n;i++)

　　　　{

　　　　　　cd.start=n;

　　　　　　c=i;

　　　　　　p=a[c].parent;

　　　　　　while(p!=-1)

　　　　　　{

　　　　　　　　if(a[p].lchild==c) //左孩子为0

　　　　　　　　　　cd.bits[cd.start]=0;

　　　　　　　　else

cd.bits[cd.start]=1; //右孩子为1

　　　　　　　　cd.start--;

　　　　　　　　c=p;

　　　　　　　　p=a[p].parent;

　　　　　　}

　　　　　　cd.start++;

　　　　　　for(j=cd.start;j<=n;j++)

　　　　　　　　code[i].bits[j]=cd.bits[j];

　　　　　　code[i].start=cd.start;

　　　　}

　　　　cout<<'\t'<<"字符"<<'\t'<<"权值"<<'\t'<<"哈夫曼编码"<<endl;

　　　　for(i=0;i<n;i++) //输出哈夫曼编码

　　　　{

　　　　　　cout<<" "<<i+1<<'\t';

　　　　　　cout<<a[i].letter<<'\t';

　　　　　　cout<<a[i].weight<<'\t';

　　　　　　for(j=code[i].start;j<=n;j++)

　　　　　　cout<<code[i].bits[j];

　　　　　　cout<<endl;

　　　　}

}

void main()

{

　　int i,n;

　　　　huffnode huff\_node[MaxNode];

　　　　cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl;

　　　　cout<<'\t'<<"哈夫曼编码的程序设计与实现"<<'\n';

　　　　cout<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<'\n'<<'\n';

　　　　cout<<"-->>请输入叶子节点的数目:";

　　　　cin>>n;

　　　　for(i=0;i<2\*n-1;i++)

　　　　{

　　　　　　huff\_node[i].weight=0;

　　　　　　huff\_node[i].rchild=-1;

　　　　　　huff\_node[i].parent=-1;

　　　　　　huff\_node[i].lchild=-1;

　　　　}

　　　　cout<<"请输入"<<n<<"个叶子节点的字母和权值:"<<endl;

　　　　for(i=0;i<n;i++)

　　　　{

　　　　　　cout<<"请输入第"<<i+1<<"个字母和权值：";

　　　　　　cin>>huff\_node[i].letter>>huff\_node[i].weight;

　　　　}

　　　　cout<<endl;

　　　　cout<<"<=========叶子节点的字母和权值===========>"<<endl;

　　　　cout<<" "<<"字符"<<" "<<"权值"<<endl;

　　　　for(i=0;i<n;i++)

　　　　{

　　　　　　cout<<" "<<huff\_node[i].letter;

　　　　　　cout<<'\t'<<" "<<huff\_node[i].weight<<endl;

　　　　}

　　　　cout<<endl;

　　　　cout<<"<=============输出哈夫曼树===============>"<<endl;

　　　　HuffManCode(huff\_node,n);

}

**树**

QueueNode.h

template<typename T>

class LinkQueue;

template<typename T>

class QueueNode{

private:

friend class LinkQueue<T>;

QueueNode(const T item,QueueNode<T> \*next=NULL)

:data(item),\_next(next){}

private:

T data;

QueueNode<T> \*\_next;

};

LinkQueue.h

#include "QueueNode.h"

///LinkQueue///

template<typename T>

class LinkQueue{

public:

LinkQueue():\_rear(NULL),\_front(NULL){}

~LinkQueue(){

MakeEmpty();

}

void Append(const T item);

T Delete();

T GetFront();

void MakeEmpty();

bool IsEmpty() const{

return \_front==NULL;

}

void Print();

private:

QueueNode<T> \*\_rear,\*\_front;

};

///MakeEmpty///

template<typename T>

void LinkQueue<T>::MakeEmpty(){

QueueNode<T> \*pdel;

while(\_front){

pdel=\_front;

\_front=\_front->\_next;

delete pdel;

}

}

///Append///

template<typename T>

void LinkQueue<T>::Append(const T item){

if(\_front==NULL){

\_front=\_rear=new QueueNode<T>(item);

}

else{

\_rear=\_rear->\_next=new QueueNode<T>(item);

}

}

///Delete///

template<typename T>

T LinkQueue<T>::Delete(){

if(IsEmpty()){

cout<<"There is no element!"<<endl;

exit(1);

}

QueueNode<T> \*pdel=\_front;

T temp=\_front->data;

\_front=\_front->\_next;

delete pdel;

return temp;

}

///GetFront///

template<typename T>

T LinkQueue<T>::GetFront(){

if(IsEmpty()){

cout<<"There is no element!"<<endl;

exit(1);

}

return \_front->data;

}

///Print///

template<typename T>

void LinkQueue<T>::Print(){

QueueNode<T> \*pmove=\_front;

cout<<"front";

while(pmove){

cout<<"—>"<<pmove->data;

pmove=pmove->\_next;

}

cout<<"—>rear"<<endl<<endl<<endl;

}

TreeNode.h

template<typename T>

class Tree;

template<typename T>

class TreeNode{

public:

friend class Tree<T>;

private:

T data;

TreeNode<T> \*\_first,\*\_next;

TreeNode():\_first(NULL), \_next(NULL){}

TreeNode(T item, TreeNode<T> \*first = NULL, TreeNode<T> \*next = NULL)

:data(item), \_first(first), \_next(next){}

};

Tree.h

#include "TreeNode.h"

#include "LinkQueue.h"

///Tree///

template<typename T>

class Tree{

public:

Tree():\_root(NULL), \_current(NULL){}

public:

TreeNode<T> \*GetCurrent(){

return \_current;

}

void SetCurrent(TreeNode<T> \*pcurrent){

\_current = pcurrent;

}

bool Insert(T item);

void Remove(T item);

void Remove(TreeNode<T> \*pcurrent);

bool Find(T item);

void PrintChild(TreeNode<T> \*pcurrent);

TreeNode<T> \*Parent(TreeNode<T> \*pcurrent);

void Print();

void PreOrder(TreeNode<T> \*proot);

void PostOrder(TreeNode<T> \*proot);

void LevelOrder(TreeNode<T> \*proot);

void PreOrder();

void PostOrder();

void LevelOrder();

private:

TreeNode<T> \*\_root,\*\_current;

bool Find(TreeNode<T> \*proot, T item);

void Remove(TreeNode<T> \*proot, T item);

TreeNode<T> \*Parent(TreeNode<T> \*proot, TreeNode<T> \*pcurrent);

void Print(TreeNode<T> \*\_start, int n=0);

};

///Insert///

template<typename T>

bool Tree<T>::Insert(T item){

TreeNode<T> \*newnode = new TreeNode<T>(item);

if (NULL == newnode){

cout << "Application Error!" <<endl;

exit(1);

}

if (NULL == \_root){

\_root = newnode;

\_current = \_root;

return 1;

}

if (NULL == \_current){

cerr << "insert error!" <<endl;

return 0;

}

if(NULL == \_current->\_first){

\_current->\_first = newnode;

\_current = newnode;

return 1;

}

TreeNode<T> \*pmove = \_current->\_first;

while(pmove->\_next){

pmove = pmove->\_next;

}

pmove->\_next = newnode;

\_current = newnode;

return 1;

}

///Remove///

template<typename T>

void Tree<T>::Remove(TreeNode<T> \*pcurrent){

if(NULL == pcurrent){

return;

}

TreeNode<T> \*temp = Parent(pcurrent);

if(NULL == temp){

TreeNode<T> \*pmove = pcurrent->\_first;

if(NULL != pmove->\_first){

pmove=pmove->\_first;

while(pmove->\_next){

pmove = pmove->\_next;

}

pmove->\_next = pcurrent->\_first->\_next;

pcurrent->\_first->\_next = NULL;

}

else{

pmove->\_first = pmove->\_next;

}

\_root = pcurrent->\_first;

}

else{

if(temp->\_first == pcurrent){

TreeNode<T> \*pmove = pcurrent->\_first;

if (pmove){

while (pmove->\_next){

pmove = pmove->\_next;

}

pmove->\_next = pcurrent->\_next;

}

else{

pcurrent->\_first = pcurrent->\_next;

}

}

else{

TreeNode<T> \*pmove = temp->\_first;

while(pmove->\_next != pcurrent){

pmove = pmove->\_next;

}

pmove->\_next = pcurrent->\_next;

while(pmove->\_next){

pmove = pmove->\_next;

}

pmove->\_next = pcurrent->\_first;

}

}

delete pcurrent;

}

///Remove///

template<typename T>

void Tree<T>::Remove(TreeNode<T> \*proot, T item){

if(NULL == proot){

return;

}

if(proot->\_first){

TreeNode<T> \*pmove=proot->\_first;

while(pmove){

Remove(pmove, item);

pmove = pmove->\_next;

}

}

if(proot->data == item){

Remove(proot);

}

}

///Remove///

template<typename T>

void Tree<T>::Remove(T item){

Remove(\_root, item);

}

///Parent///

template<typename T>

TreeNode<T>\* Tree<T>::Parent(

TreeNode<T> \*proot, TreeNode<T> \*pcurrent){

if(NULL == proot){

return NULL;

}

TreeNode<T> \*pmove=proot->\_first,\*temp;

if(NULL != pmove){

while(pmove){

if(pmove == pcurrent){

return proot;

}

pmove = pmove->\_next;

}

}

pmove = proot->\_first;

while(pmove){

temp = Parent(pmove, pcurrent);

if(temp){

return temp;

}

pmove = pmove->\_next;

}

return NULL;

}

///Parent///

template<typename T>

TreeNode<T>\* Tree<T>::Parent(TreeNode<T> \*pcurrent){

return Parent(\_root,pcurrent);

}

///PrintChild///

template<typename T>

void Tree<T>::PrintChild(TreeNode<T> \*pcurrent){

TreeNode<T> \*pmove = pcurrent->\_first;

cout<<"first";

if(NULL != pmove){

cout<<"--->"<<pmove->data;

}

while(pmove->\_next){

cout<<"--->"<<pmove->data;

pmove = pmove->\_next;

}

}

///Find///

template<typename T>

bool Tree<T>::Find(TreeNode<T> \*proot, T item){

if (proot->data == item){

return 1;

}

if (NULL == proot){

return 0;

}

TreeNode<T> \*pmove=proot->\_first;

if (NULL == pmove){

return 0;

}

while (pmove){

if (Find(pmove, item)){

return 1;

}

pmove = pmove->\_next;

}

return 0;

}

///Find///

template<typename T>

bool Tree<T>::Find(T item){

return Find(\_root,item);

}

///Print///

template<typename T>

void Tree<T>::Print(TreeNode<T> \*\_start, int n){

if (NULL == \_start){

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

cout << " ";

}

cout << "NULL" << endl;

return;

}

TreeNode<T> \*pmove = \_start->\_first;

Print(pmove, n+1);

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

cout << " ";

}

cout << \_start->data << "--->" <<endl;

if (NULL == pmove){

return;

}

pmove = pmove->\_next;

while (pmove){

Print(pmove, n+1);

pmove = pmove->\_next;

}

}

///Print///

template<typename T>

void Tree<T>::Print(){

Print(\_root);

}

///PreOrder///

template<typename T>

void Tree<T>::PreOrder(TreeNode<T> \*proot){

if (NULL == proot){

return;

}

cout << proot->data;

TreeNode<T> \*pmove = proot->\_first;

while (pmove){

PreOrder(pmove);

pmove = pmove->\_next;

}

}

///PostOrder///

template<typename T>

void Tree<T>::PostOrder(TreeNode<T> \*proot){

if (NULL == proot){

return;

}

TreeNode<T> \*pmove = proot->\_first;

while (pmove){

PostOrder(pmove);

pmove = pmove->\_next;

}

cout << proot->data;

}

///PreOrder///

template<typename T>

void Tree<T>::PreOrder(){

PreOrder(\_root);

}

///PostOrder///

template<typename T>

void Tree<T>::PostOrder(){

PostOrder(\_root);

}

///LevelOrder///

template<typename T>

void Tree<T>::LevelOrder(TreeNode<T> \*proot){ //using queue

LinkQueue<TreeNode<T> \*> queue;

TreeNode<T> \*pmove, \*ptemp;

if (proot != NULL){

queue.Append(proot);

while (!queue.IsEmpty()){

ptemp = queue.Delete();

cout << ptemp->data;

pmove = ptemp->\_first;

while(pmove){

queue.Append(pmove);

pmove = pmove->\_next;

}

}

}

}

///LevelOrder///

template<typename T>

void Tree<T>::LevelOrder(){

LevelOrder(\_root);

}

test.cpp

#include <iostream>

using namespace std;

#include "Tree.h"

int main(){

Tree<int> tree;

int init[10]={3,6,0,2,8,4,9,1,5,7};

for (int i=0; i<10; i++){

tree.Insert(init[i]);

if (1 == i % 2){

tree.SetCurrent(tree.Parent(tree.GetCurrent()));

}

}

tree.Print();

cout << endl <<endl << endl;

tree.Remove(3);

tree.Print();

cout << endl <<endl << endl;

cout << tree.Find(5) << endl << tree.Find(11) <<endl;

tree.PreOrder();

cout << endl;

tree.PostOrder();

cout << endl;

tree.LevelOrder();

return 0;

}

**B+树**

BTreeNode.h

template<typename T>

class BTree;

///BTreeNode///

template<typename T>

class BTreeNode{

public:

friend BTree<T>;

BTreeNode(): maxSize(0), \_\_tr(NULL), \_parent(NULL){}

BTreeNode(int size): size(0), maxSize(size), \_parent(NULL){

\_key = new T[size+1];

\_\_tr = new BTreeNode<T> \*[size+1];

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

\_\_tr[i] = NULL;

\_key[i] = this->infinity;

}

}

void Destroy(BTreeNode<T> \*proot);

~BTreeNode(){

if (maxSize){

delete[] \_key;

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

\_\_tr[i] = NULL;

}

}

}

bool IsFull(){

return size == maxSize;

}

T GetKey(int i){

if (this){

return this->\_key[i];

}

return -1;

}

private:

int size;

int maxSize; //the Max Size of key

T \*\_key;

BTreeNode<T> \*\_parent;

BTreeNode<T> \*\*\_\_tr;

static const T infinity;

};

template<typename T>

const T BTreeNode<T>::infinity=10000;

///Triple///

template<typename T>

struct Triple{

BTreeNode<T> \*m\_pfind;

int m\_nfind;

bool m\_ntag;

};

///Destroy///

template<typename T>

void BTreeNode<T>::Destroy(BTreeNode<T> \*proot){

if (NULL == proot){

return;

}

for (int i=0; i<proot->size; i++){

Destroy(proot->\_\_tr[i]);

}

delete proot;

}

BTree.h

#include "BTreeNode.h"

///BTree///

template<typename T>

class BTree{

public:

BTree(int size): maxSize(size), \_root(NULL){}

~BTree();

Triple<T> Search(const T item);

int Size();

int Size(BTreeNode<T> \*root);

bool Insert(const T item);

bool Remove(const T item);

void Print();

BTreeNode<T> \*GetParent(const T item);

private:

void InsertKey(BTreeNode<T> \*pinsert, int n, const T item, BTreeNode<T> \*pright);

void PreMove(BTreeNode<T> \*root, int n);

void Merge(BTreeNode<T> \*pleft, BTreeNode<T> \*pparent, BTreeNode<T> \*pright, int n);

void LeftAdjust(BTreeNode<T> \*pright, BTreeNode<T> \*pparent, int min, int n);

void RightAdjust(BTreeNode<T> \*pleft, BTreeNode<T> \*pparent, int min, int n);

void Print(BTreeNode<T> \*start, int n = 0);

private:

BTreeNode<T> \*\_root;

const int maxSize;

};

///析构函数///

template<typename T>

BTree<T>::~BTree(){

\_root->Destroy(\_root);

}

///Search///

template<typename T>

Triple<T> BTree<T>::Search(const T item){

Triple<T> result;

BTreeNode<T> \*pmove = \_root, \*parent = NULL;

int i = 0;

while (pmove){

i = -1;

while (item > pmove->\_key[++i]);

if (pmove->\_key[i] == item){

result.m\_pfind = pmove;

result.m\_nfind = i;

result.m\_ntag = 1;

return result;

}

parent = pmove;

pmove = pmove->\_\_tr[i];

}

result.m\_pfind = parent;

result.m\_nfind = i;

result.m\_ntag = 0;

return result;

}

///InsertKey///

template<typename T>

void BTree<T>::InsertKey(BTreeNode<T> \*pinsert, int n, const T item, BTreeNode<T> \*pright){

pinsert->size++;

for (int i=pinsert->size; i>n; i--){

pinsert->\_key[i] = pinsert->\_key[i-1];

pinsert->\_\_tr[i+1] = pinsert->\_\_tr[i];

}

pinsert->\_key[n] = item;

pinsert->\_\_tr[n+1] = pright;

if (pinsert->\_\_tr[n+1]){

pinsert->\_\_tr[n+1]->\_parent = pinsert;

for (int i=0; i<=pinsert->\_\_tr[n+1]->size; i++){

if (pinsert->\_\_tr[n+1]->\_\_tr[i]){

pinsert->\_\_tr[n+1]->\_\_tr[i]->\_parent = pinsert->\_\_tr[n+1];

}

}

}

}

///Insert///

template<typename T>

bool BTree<T>::Insert(const T item){

if (NULL == \_root){

\_root = new BTreeNode<T>(maxSize);

\_root->size = 1;

\_root->\_key[1] = \_root->\_key[0];

\_root->\_key[0] = item;

\_root->\_\_tr[0] = \_root->\_\_tr[1] =NULL;

return 1;

}

Triple<T> find = this->Search(item);

if (find.m\_ntag){

cerr << "The item is exist!" << endl;

return 0;

}

BTreeNode<T> \*pinsert = find.m\_pfind, \*newnode;

BTreeNode<T> \*pright = NULL, \*pparent;

T key = item;

int n = find.m\_nfind;

while (1){

if (pinsert->size < pinsert->maxSize-1){

InsertKey(pinsert, n, key, pright);

return 1;

}

int m = (pinsert->size + 1) / 2;

InsertKey(pinsert, n, key, pright);

newnode = new BTreeNode<T>(this->maxSize);

for (int i=m+1; i<=pinsert->size; i++){

newnode->\_key[i-m-1] = pinsert->\_key[i];

newnode->\_\_tr[i-m-1] = pinsert->\_\_tr[i];

pinsert->\_key[i] = pinsert->infinity;

pinsert->\_\_tr[i] = NULL;

}

newnode->size = pinsert->size - m - 1;

pinsert->size = m;

for (i=0; i<=newnode->size; i++){

if (newnode->\_\_tr[i]){

newnode->\_\_tr[i]->\_parent = newnode;

for (int j=0; j<=newnode->\_\_tr[i]->size; j++){

if (newnode->\_\_tr[i]->\_\_tr[j]){

newnode->\_\_tr[i]->\_\_tr[j]->\_parent = newnode->\_\_tr[i];

}

}

}

}

for (i=0; i<=pinsert->size; i++){

if (pinsert->\_\_tr[i]){

pinsert->\_\_tr[i]->\_parent = pinsert;

for (int j=0; j<=pinsert->size; j++){

if (pinsert->\_\_tr[i]->\_\_tr[j]){

pinsert->\_\_tr[i]->\_\_tr[j]->\_parent = pinsert->\_\_tr[i];

}

}

}

}

key = pinsert->\_key[m];

pright = newnode;

if (pinsert->\_parent){

pparent = pinsert->\_parent;

n = -1;

pparent->\_key[pparent->size] = pparent->infinity;

while (key > pparent->\_key[++n]);

newnode->\_parent = pinsert->\_parent;

pinsert = pparent;

}

else {

\_root = new BTreeNode<T>(this->maxSize);

\_root->size = 1;

\_root->\_key[1] = \_root->\_key[0];

\_root->\_key[0] = key;

\_root->\_\_tr[0] = pinsert;

\_root->\_\_tr[1] = pright;

newnode->\_parent = pinsert->\_parent = \_root;

return 1;

}

}

}

///PreMove///

template<typename T>

void BTree<T>::PreMove(BTreeNode<T> \*root, int n){

root->\_key[root->size] = root->infinity;

for (int i=n; i<root->size; i++){

root->\_key[i] = root->\_key[i+1];

root->\_\_tr[i+1] = root->\_\_tr[i+2];

}

root->size--;

}

///Merge///

template<typename T>

void BTree<T>::Merge(BTreeNode<T> \*pleft, BTreeNode<T> \*pparent, BTreeNode<T> \*pright, int n){

pleft->\_key[pleft->size] = pparent->\_key[n];

BTreeNode<T> \*ptemp;

for (int i=0; i<=pright->size; i++){

pleft->\_key[pleft->size+i+1] = pright->\_key[i];

pleft->\_\_tr[pleft->size+i+1] = pright->\_\_tr[i];

ptemp = pleft->\_\_tr[pleft->size+i+1];

if (ptemp){

ptemp->\_parent = pleft;

for (int j=0; j<=ptemp->size; j++){

if (ptemp->\_\_tr[j]){

ptemp->\_\_tr[j]->\_parent = ptemp;

}

}

}

}

pleft->size = pleft->size + pright->size + 1;

delete pright;

PreMove(pparent, n);

// this->Print();

}

///LeftAdjust///

template<typename T>

void BTree<T>::LeftAdjust(BTreeNode<T> \*pright, BTreeNode<T> \*pparent, int min, int n){

BTreeNode<T> \*pleft = pparent->\_\_tr[n-1], \*ptemp;

if (pleft->size > min-1){

for (int i=pright->size+1; i>0; i--){

pright->\_key[i] = pright->\_key[i-1];

pright->\_\_tr[i] = pright->\_\_tr[i-1];

}

pright->\_key[0] = pparent->\_key[n-1];

pright->\_\_tr[0] = pleft->\_\_tr[pleft->size];

ptemp = pright->\_\_tr[0];

if (ptemp){ //change the tree's parent which is moved

ptemp->\_parent = pright;

for (int i=0; i<ptemp->size; i++){

if (ptemp->\_\_tr[i]){

ptemp->\_\_tr[i]->\_parent = ptemp;

}

}

}

pparent->\_key[n-1] = pleft->\_key[pleft->size-1];

pleft->\_key[pleft->size] = pleft->infinity;

pleft->size--;

pright->size++;

}

else {

Merge(pleft, pparent, pright, n-1);

}

// this->Print();

}

///RightAdjust///

template<typename T>

void BTree<T>::RightAdjust(BTreeNode<T> \*pleft, BTreeNode<T> \*pparent, int min, int n){

BTreeNode<T> \*pright = pparent->\_\_tr[1], \*ptemp;

if (pright && pright->size > min-1){

pleft->\_key[pleft->size] = pparent->\_key[0];

pparent->\_key[0] = pright->\_key[0];

pleft->\_\_tr[pleft->size+1] = pright->\_\_tr[0];

ptemp = pleft->\_\_tr[pleft->size+1];

if (ptemp){ //change the tree's parent which is moved

ptemp->\_parent = pleft;

for (int i=0; i<ptemp->size; i++){

if (ptemp->\_\_tr[i]){

ptemp->\_\_tr[i]->\_parent = ptemp;

}

}

}

pright->\_\_tr[0] = pright->\_\_tr[1];

pleft->size++;

PreMove(pright,0);

}

else {

Merge(pleft, pparent, pright, 0);

}

}

///Remove///

template<typename T>

bool BTree<T>::Remove(const T item){

Triple<T> result = this->Search(item);

if (!result.m\_ntag){

return 0;

}

BTreeNode<T> \*pdel, \*pparent, \*pmin;

int n = result.m\_nfind;

pdel = result.m\_pfind;

if (pdel->\_\_tr[n+1] != NULL){ //change into delete leafnode

pmin = pdel->\_\_tr[n+1];

pparent = pdel;

while (pmin != NULL){

pparent = pmin;

pmin = pmin->\_\_tr[0];

}

pdel->\_key[n] = pparent->\_key[0];

pdel = pparent;

n = 0;

}

PreMove(pdel, n); //delete the node

int min = (this->maxSize + 1) / 2;

while (pdel->size < min-1){ //if it is not a BTree, then adjust

n = 0;

pparent = pdel->\_parent;

if (NULL == pparent)

{

return 1;

}

while (n<= pparent->size && pparent->\_\_tr[n]!=pdel){

n++;

}

if (!n){

RightAdjust(pdel, pparent, min, n); //adjust with the parent and the right child tree

}

else {

LeftAdjust(pdel, pparent, min, n); //adjust with the parent and the left child tree

}

pdel = pparent;

if (pdel == \_root){

break;

}

}

if (!\_root->size){ //the root is merged

pdel = \_root->\_\_tr[0];

delete \_root;

\_root = pdel;

\_root->\_parent = NULL;

for (int i=0; i<\_root->size; i++){

if (\_root->\_\_tr[i]){

\_root->\_\_tr[i]->\_parent = \_root;

}

}

}

return 1;

}

///Print///

template<typename T>

void BTree<T>::Print(BTreeNode<T> \*start, int n){

if (NULL == start){

return;

}

if (start->\_\_tr[0]){

Print(start->\_\_tr[0], n+1); //print the first child tree

}

else {

for (int j=0; j<n; j++){

cout << " ";

}

cout << "NULL" << endl;

}

for (int i=0; i<start->size; i++){ //print the orther child tree

for (int j=0; j<n; j++){

cout << " ";

}

cout << start->\_key[i] << "--->" <<endl;

if (start->\_\_tr[i+1]){

Print(start->\_\_tr[i+1], n+1);

}

else {

for (int j=0; j<n; j++){

cout << " ";

}

cout << "NULL" << endl;

}

}

}

///Print///

template<typename T>

void BTree<T>::Print(){

Print(\_root);

}

///Size///

template<typename T>

int BTree<T>::Size(BTreeNode<T> \*root){

if (NULL == root){

return 0;

}

int size=root->size;

for (int i=0; i<=root->size; i++){

if (root->\_\_tr[i]){

size += this->Size(root->\_\_tr[i]);

}

}

return size;

}

///Size///

template<typename T>

int BTree<T>::Size(){

return this->Size(this->\_root);

}

///GetParent///

template<typename T>

BTreeNode<T>\* BTree<T>::GetParent(const T item){

Triple<T> result = this->Search(item);

return result.m\_pfind->\_parent;

}

test.cpp

#include <iostream>

#include <cstdlib>

using namespace std;

#include "BTree.h"

int main(){

BTree<int> btree(3);

int init[]={1,3,5,7,4,2,8,0,6,9,29,13,25,11,32,55,34,22,76,45

,14,26,33,88,87,92,44,54,23,12,21,99,19,27,57,18,72,124,158,234

,187,218,382,122,111,222,333,872,123};

for (int i=0; i<49; i++){

btree.Insert(init[i]);

}

btree.Print();

cout << endl << endl << endl;

Triple<int> result = btree.Search(13);

cout << result.m\_pfind->GetKey(result.m\_nfind) << endl;

cout << endl << endl << endl;

for (i=0; i<49; i++){

btree.Remove(init[i]);

btree.Print();

cout << endl << endl << endl;

}

return 0;

}

**图**

MinHeap.h//略，与前一样

Edge.h

template<typename T>

struct Edge{

public:

Edge(int dest, T cost): dest(dest), cost(cost), \_next(NULL){}

public:

int dest;

T cost;

Edge<T> \*\_next;

};

Vertex.h

#include "Edge.h"

template<typename NameType, typename DistType>

struct Vertex{

public:

Vertex(): \_adj(NULL){}

NameType data;

Edge<DistType> \*\_adj;

~Vertex();

};

template<typename NameType, typename DistType>

Vertex<NameType, DistType>::~Vertex(){

Edge<DistType> \*pmove = \_adj;

while (pmove){

\_adj = pmove->\_next;

delete pmove;

pmove = \_adj;

}

}

Graph.h

#include "Vertex.h"

///Graph///

template<typename NameType, typename DistType>

class Graph{

public:

Graph(int size = defaultSize);

~Graph();

bool GraphEmpty() const{

return 0 == numVertex;

}

bool GraphFull() const{

return maxNum == numVertex;

}

int NumberOfVertex() const{

return numVertex;

}

int NumberOfEdge() const{

return numEdges;

}

NameType GetValue(int v);

DistType GetWeight(int v1, int v2);

int GetFirst(int v);

int GetNext(int v1, int v2);

bool InsertVertex(const NameType vertex);

bool RemoveVertex(int v);

bool InsertEdge(int v1, int v2, DistType weight=infinity);

bool RemoveEdge(int v1, int v2);

void Print();

Edge<DistType> \*GetMin(int v, int \*visited);

void Prim(Graph<NameType, DistType> &graph);

void DFS(int v, int \*visited);

void DFS();

void Dijkstra(int v, DistType \*shotestpath);

private:

Vertex<NameType, DistType> \*\_nodeTable;

int numVertex;

const int maxNum;

static const int defaultSize;

static const DistType infinity;

int numEdges;

int GetVertexPos(const NameType vertex);

};

template<typename NameType, typename DistType>

const int Graph<NameType, DistType>::defaultSize=10;

template<typename NameType, typename DistType>

const DistType Graph<NameType, DistType>:: infinity=100000;

///Constuctor///

template<typename NameType, typename DistType>

Graph<NameType, DistType>::Graph(int size)

: numVertex(0), maxNum(size), numEdges(0){

\_nodeTable = new Vertex<NameType, DistType>[size];

}

///析构函数///

template<typename NameType, typename DistType>

Graph<NameType, DistType>::~Graph(){

Edge<DistType> \*pmove;

for (int i=0; i<this->numVertex; i++){

pmove = this->\_nodeTable[i].\_adj;

if (pmove){

this->\_nodeTable[i].\_adj = pmove->\_next;

delete pmove;

pmove = this->\_nodeTable[i].\_adj;

}

}

delete[] \_nodeTable;

}

///GetFirst///

template<typename NameType, typename DistType>

int Graph<NameType, DistType>::GetFirst(int v){

if (v<0 || v>=this->numVertex){

return -1;

}

Edge<DistType> \*ptemp = this->\_nodeTable[v].\_adj;

return \_nodeTable[v].\_adj ? \_nodeTable[v].\_adj->dest : -1;

}

///GetNext///

template<typename NameType, typename DistType>

int Graph<NameType, DistType>::GetNext(int v1, int v2){

if (-1 != v1){

Edge<DistType> \*pmove = this->\_nodeTable[v1].\_adj;

while (NULL != pmove->\_next){

if (pmove->dest==v2){

return pmove->\_next->dest;

}

pmove = pmove->\_next;

}

}

return -1;

}

///GetValue///

template<typename NameType, typename DistType>

NameType Graph<NameType, DistType>::GetValue(int v){

if (v<0 || v>=this->numVertex){

cerr << "The vertex is not exsit" <<endl;

exit(1);

}

return \_nodeTable[v].data;

}

///GetVertextPos///

template<typename NameType, typename DistType>

int Graph<NameType, DistType>::GetVertexPos(const NameType vertex){

for (int i=0; i<this->numVertex; i++){

if (vertex == \_nodeTable[i].data){

return i;

}

}

return -1;

}

///GetWeight///

template<typename NameType, typename DistType>

DistType Graph<NameType, DistType>::GetWeight(int v1, int v2){

if (v1>=0 && v1<this->numVertex && v2>=0 && v2<this->numVertex){

if (v1 == v2){

return 0;

}

Edge<DistType> \*pmove = \_nodeTable[v1].\_adj;

while (pmove){

if (pmove->dest == v2){

return pmove->cost;

}

pmove = pmove->\_next;

}

}

return infinity;

}

///InsertEdge///

template<typename NameType, typename DistType>

bool Graph<NameType, DistType>::InsertEdge(int v1, int v2, DistType weight){

if (v1>=0 && v1<this->numVertex && v2>=0 && v2<this->numVertex){

Edge<DistType> \*pmove = \_nodeTable[v1].\_adj;

if (NULL == pmove){

\_nodeTable[v1].\_adj = new Edge<DistType>(v2, weight);

return 1;

}

while (pmove->\_next){

if (pmove->dest == v2){

break;

}

pmove = pmove->\_next;

}

if (pmove->dest == v2){

pmove->cost = weight;

return 1;

}

else{

pmove->\_next = new Edge<DistType>(v2, weight);

return 1;

}

}

return 0;

}

///InesrtVertex///

template<typename NameType, typename DistType>

bool Graph<NameType, DistType>::InsertVertex(const NameType vertex){

int i = this->GetVertexPos(vertex);

if (-1 != i){

this->\_nodeTable[i].data = vertex;

}

else{

if (!this->GraphFull()){

this->\_nodeTable[this->numVertex].data = vertex;

this->numVertex++;

}

else{

cerr << "The Graph is Full" <<endl;

return 0;

}

}

return 1;

}

///RemoveEdge///

template<typename NameType, typename DistType>

bool Graph<NameType, DistType>::RemoveEdge(int v1, int v2){

if (v1>=0 && v1<this->numVertex && v2>=0 && v2<this->numVertex){

Edge<DistType> \*pmove = this->\_nodeTable[v1].\_adj, \*pdel;

if (NULL == pmove){

cerr << "the edge is not exist!" <<endl;

return 0;

}

if (pmove->dest == v2){

this->\_nodeTable[v1].\_adj = pmove->\_next;

delete pmove;

return 1;

}

while (pmove->\_next){

if (pmove->\_next->dest == v2){

pdel = pmove->\_next;

pmove->\_next = pdel->\_next;

delete pdel;

return 1;

}

pmove = pmove->\_next;

}

}

cerr << "the edge is not exist!" <<endl;

return 0;

}

///RemoveVertex///

template<typename NameType, typename DistType>

bool Graph<NameType, DistType>::RemoveVertex(int v){

if (v<0 || v>=this->numVertex){

cerr << "the vertex is not exist!" << endl;

return 0;

}

Edge<DistType> \*pmove, \*pdel;

for (int i=0; i<this->numVertex; i++){

pmove = this->\_nodeTable[i].\_adj;

if (i != v){

if (NULL == pmove){

continue;

}

if (pmove->dest == v){

this->\_nodeTable[i].\_adj = pmove->\_next;

delete pmove;

continue;

}

else {

if (pmove->dest > v){

pmove->dest--;

}

}

while (pmove->\_next){

if (pmove->\_next->dest == v){

pdel = pmove->\_next;

pmove->\_next = pdel->\_next;

delete pdel;

}

else {

if (pmove->\_next->dest > v){

pmove->\_next->dest--;

pmove = pmove->\_next;

}

}

}

}

else {

while (pmove){

this->\_nodeTable[i].\_adj = pmove->\_next;

delete pmove;

pmove = this->\_nodeTable[i].\_adj;

}

}

}

this->numVertex--;

for (i=v; i<this->numVertex; i++)

{

this->\_nodeTable[i].\_adj = this->\_nodeTable[i+1].\_adj;

this->\_nodeTable[i].data = this->\_nodeTable[i+1].data;

}

this->\_nodeTable[this->numVertex].\_adj = NULL;

return 1;

}

///Print///

template<typename NameType, typename DistType>

void Graph<NameType, DistType>::Print(){

Edge<DistType> \*pmove;

for (int i=0; i<this->numVertex; i++){

cout << this->\_nodeTable[i].data << "—>";

pmove = this->\_nodeTable[i].\_adj;

while (pmove){

cout << pmove->cost << "—>" << this->\_nodeTable[pmove->dest].data << "—>";

pmove = pmove->\_next;

}

cout << "NULL" << endl;

}

}

///Prim///

template<typename NameType, typename DistType>

void Graph<NameType, DistType>::Prim(Graph<NameType, DistType> &graph){

int \*node = new int[this->numVertex];

int \*visited = new int[this->numVertex];

int count = 0;

Edge<DistType> \*ptemp, \*ptemp2 = new Edge<DistType>(0, this->infinity), \*pmin;

int min;

for (int i=0; i<this->numVertex; i++){

graph.InsertVertex(this->\_nodeTable[i].data);

node[i] = 0;

visited[i] = 0;

}

visited[0] = 1;

while(++count < this->numVertex){

pmin = ptemp2;

pmin->cost = this->infinity;

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

ptemp = GetMin(node[i], visited);

if (NULL == ptemp){

continue;

}

if (pmin->cost > ptemp->cost){

pmin = ptemp;

min = node[i];

}

}

node[count] = pmin->dest;

visited[node[count]] = 1;

graph.InsertEdge(pmin->dest, min, pmin->cost);

graph.InsertEdge(min, pmin->dest, pmin->cost);

}

graph.DFS();

delete ptemp2;

delete[] node;

delete[] visited;

}

///DFS///

template<typename NameType, typename DistType>

void Graph<NameType, DistType>::DFS(int v, int \*visited){

cout << "—>" << this->GetValue(v);

visited[v] = 1;

int weight = this->GetFirst(v);

while (-1 != weight){

if (!visited[weight]){

cout << "—>" << this->GetWeight(v, weight);

DFS(weight, visited);

}

weight = this->GetNext(v, weight);

}

}

///DFS///

template<typename NameType, typename DistType>

void Graph<NameType, DistType>::DFS(){

int \*visited = new int[this->numVertex];

for (int i=0; i<this->numVertex; i++){

visited[i] = 0;

}

cout << "head";

DFS(0, visited);

cout << "—>end";

}

///GetMin///

template<typename NameType, typename DistType>

Edge<DistType>\* Graph<NameType, DistType>::GetMin(int v, int \*visited){

Edge<DistType> \*pmove = this->\_nodeTable[v].\_adj, \*ptemp = new Edge<DistType>(0, this->infinity), \*pmin = ptemp;

while (pmove){

if (!visited[pmove->dest] && pmin->cost>pmove->cost){

pmin = pmove;

}

pmove = pmove->\_next;

}

if (pmin == ptemp){

delete ptemp;

return NULL;

}

delete ptemp;

return pmin;

}

///Dijkstra///

template<typename NameType, typename DistType>

void Graph<NameType, DistType>::Dijkstra(int v, DistType \*shotestpath){

int \*visited = new int[this->numVertex];

int \*node = new int[this->numVertex];

for (int i=0; i<this->numVertex; i++){

visited[i] = 0;

node[i] = 0;

shotestpath[i] = this->GetWeight(v, i);

}

visited[v] = 1;

for (i=1; i<this->numVertex; i++){

DistType min = this->infinity;

int u=v;

for (int j=0; j<this->numVertex; j++){

if (!visited[j] && shotestpath[j]<min){

min = shotestpath[j];

u = j;

}

}

visited[u] = 1;

for (int w=0; w<this->numVertex; w++){

DistType weight = this->GetWeight(u, w);

if (!visited[w] && weight!=this->infinity

&& shotestpath[u]+weight<shotestpath[w]){

shotestpath[w] = shotestpath[u] + weight;

}

}

}

delete[] visited;

delete[] node;

}

test.cpp

#include <iostream>

using namespace std;

#include "Graph.h"

int main(){

Graph<char \*, int> graph,graph2;

int shotestpath[7];

char \*vertex[] = {"地大", "武大", "华科", "交大", "北大", "清华", "复旦"};

int edge[][3] = {{0, 1, 43}, {0, 2, 12}, {1, 2, 38}, {2, 3 ,1325}

,{3, 6, 55}, {4, 5, 34}, {4, 6, 248}};

for (int i=0; i<7; i++){

graph.InsertVertex(vertex[i]);

}

graph.Print();

cout << endl << endl <<endl;

for (i=0; i<7; i++){

graph.InsertEdge(edge[i][0], edge[i][1], edge[i][2]);

graph.InsertEdge(edge[i][1], edge[i][0], edge[i][2]);

}

graph.Print();

cout << endl << endl <<endl;

graph.Dijkstra(0, shotestpath);

for (i=0; i<7; i++){

cout << graph.GetValue(0) << "--->" << graph.GetValue(i)

<< ": " << shotestpath[i] <<endl;

}

cout << endl << endl <<endl;

graph.Prim(graph2);

cout << endl << endl <<endl;

graph.RemoveVertex(2);

graph.Print();

return 0;

}

**排序**

QueueNode.h//略

LinkQueue.h//略

Data.h

template<typename T>

class Element{

public:

T GetKey(){

return key;

}

void SetKey(T item){

key = item;

}

public:

Element<T>& operator =(Element<T> copy){

key = copy.key;

return \*this;

}

bool operator ==(Element<T> item){

return this->key == item.key;

}

bool operator !=(Element<T> item){

return this->key != item.key;

}

bool operator <(Element<T> item){

return this->key < item.key;

}

bool operator >(Element<T> item){

return this->key > item.key;

}

bool operator >=(Element<T> item){

return this->key >= item.key;

}

bool operator <=(Element<T> item){

return this->key <= item.key;

}

private:

T key;

};

template<typename T> class Sort;

///DataList///

template<typename T>

class DataList{

public:

friend class Sort<T>;

DataList(int size=defaultSize): maxSize(size), currentSize(0){

\_vector = new Element<T>[size];

}

DataList(T \*data, int size);

bool Insert(T item);

~DataList(){

delete[] \_vector;

}

int Size(){

return this->currentSize;

}

void Swap(Element<T> &left, Element<T> &right){

Element<T> temp = left;

left = right;

right = temp;

}

void Print();

private:

static const int defaultSize;

Element<T> \*\_vector;

const int maxSize;

int currentSize;

};

template<typename T>

const int DataList<T>::defaultSize=10;

///Constuctor///

template<typename T>

DataList<T>::DataList(T \*data, int size)

: maxSize(size > defaultSize ? size : defaultSize), currentSize(0){

this->\_vector = new Element<T>[size];

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

this->\_vector[i].SetKey(data[i]);

}

this->currentSize += size;

}

///Insert///

template<typename T>

bool DataList<T>::Insert(T item){

if (this->currentSize == this->maxSize){

cerr << "The list is full!" <<endl;

return 0;

}

this->\_vector[this->currentSize++].SetKey(item);

}

///Print///

template<typename T>

void DataList<T>::Print(){

cout << "The list is:";

for (int i=0; i<this->currentSize; i++){

cout << " " << this->\_vector[i].GetKey();

}

}

Sort.h

#include "Data.h"

#include "LinkQueue.h"

///Sort///

template<typename T>

class Sort{

public:

void InsertSort(DataList<T> &list, int n=-1);

void BinaryInsertSort(DataList<T> &list, int n=-1);

void ShellSort(DataList<T> &list, const int gap=-1);

void BubbleSort(DataList<T> &list);

void QuickSort(DataList<T> &list, int left=0, int right=-1);

void SelectSort(DataList<T> &list);

void HeapSort(DataList<T> &list);

void MergeSort(DataList<T> &list);

void RadixSort(DataList<int> &list, int m, int d);

private:

void BubbleSwap(DataList<T> &list, const int n, int &flag);

void SelectChange(DataList<T> &list, const int n);

void HeapAdjust(DataList<T> &list, const int start, const int end);

void Merge(DataList<T> &list, DataList<T> &mergedlist, const int len);

void MergeDouble(DataList<T> &list, DataList<T> &mergedlist, const int start, const int part, const int end);

};

///InsertSort///

template<typename T>

void Sort<T>::InsertSort(DataList<T> &list, int n){

if (-1 == n){

for (int i=1; i<list.currentSize; i++){

InsertSort(list, i);

}

return;

}

Element<T> temp = list.\_vector[n];

int i;

for (i=n; i>0; i--){

if (temp > list.\_vector[i-1]){

break;

}

else{

list.\_vector[i] = list.\_vector[i-1];

}

}

list.\_vector[i] = temp;

}

///BinaryInsertSort///

template<typename T>

void Sort<T>::BinaryInsertSort(DataList<T> &list, int n){

if (-1 == n){

for (int i=1; i<list.currentSize; i++){

BinaryInsertSort(list, i);

}

return;

}

Element<T> temp = list.\_vector[n];

int left = 0, right = n-1;

while(left <= right){

int middle = (left + right) / 2;

if (temp < list.\_vector[middle]){

right = middle - 1;

}

else {

left = middle + 1;

}

}

for (int i=n-1; i>=left; i--){

list.\_vector[i+1] = list.\_vector[i];

}

list.\_vector[left] = temp;

}

///ShellSort///

template<typename T>

void Sort<T>::ShellSort(DataList<T> &list, const int gap){

if (-1 == gap){

int gap = list.currentSize / 2;

while (gap){

ShellSort(list, gap);

gap = (int)(gap / 2);

}

return;

}

for (int i=gap; i<list.currentSize; i++){

InsertSort(list, i);

}

}

///BubbleSwap///

template<typename T>

void Sort<T>::BubbleSwap(DataList<T> &list, const int n, int &flag){

flag = 0;

for (int i=list.currentSize-1; i>=n; i--){

if (list.\_vector[i-1] > list.\_vector[i]){

list.Swap(list.\_vector[i-1], list.\_vector[i]);

flag = 1;

}

}

}

///BubbleSort///

template<typename T>

void Sort<T>::BubbleSort(DataList<T> &list){

int flag = 1, n = 0;

while (++n<list.currentSize && flag){

BubbleSwap(list, n, flag);

}

}

///QuickSort///

template<typename T>

void Sort<T>::QuickSort(DataList<T> &list, int left, int right){

if (-3 == right){

right = list.currentSize - 1;

}

if (left < right){

int pivotpos = left;

Element<T> pivot = list.\_vector[left];

for (int i=left+1; i<=right; i++){

if (list.\_vector[i]<pivot && ++pivotpos!=i){

list.Swap(list.\_vector[pivotpos], list.\_vector[i]);

}

list.Swap(list.\_vector[left], list.\_vector[pivotpos]);

}

QuickSort(list, left, pivotpos-1);

QuickSort(list, pivotpos+1, right);

}

}

///SelectChange///

template<typename T>

void Sort<T>::SelectChange(DataList<T> &list, const int n){

int j = n;

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

if (list.\_vector[i] < list.\_vector[j]){

j = i;

}

}

if (j != n){

list.Swap(list.\_vector[n], list.\_vector[j]);

}

}

///SelectSort///

template<typename T>

void Sort<T>::SelectSort(DataList<T> &list){

for (int i=0; i<list.currentSize-1; i++){

SelectChange(list, i);

}

}

///heapAdjust///

template<typename T>

void Sort<T>::HeapAdjust(DataList<T> &list, const int start, const int end){

int current = start, child = 2 \* current + 1;

Element<T> temp = list.\_vector[start];

while (child <= end){

if (child<end && list.\_vector[child]<list.\_vector[child+1]){

child++;

}

if (temp >= list.\_vector[child]){

break;

}

else {

list.\_vector[current] = list.\_vector[child];

current = child;

child = 2 \* current + 1;

}

}

list.\_vector[current] = temp;

}

///HeapSort///

template<typename T>

void Sort<T>::HeapSort(DataList<T> &list){

for (int i=(list.currentSize-2)/2; i>=0; i--){

HeapAdjust(list, i, list.currentSize-1);

}

for (int j=list.currentSize-1; j>=1; j--){

list.Swap(list.\_vector[0], list.\_vector[j]);

HeapAdjust(list, 0, j-1);

}

}

///MergeDouble///

template<typename T>

void Sort<T>::MergeDouble(DataList<T> &list, DataList<T> &mergedlist, const int start, const int part, const int end){

int i = start, j = part + 1, k = start;

while (i<=part && j<=end){

if (list.\_vector[i] <= list.\_vector[j]){

mergedlist.\_vector[k++] = list.\_vector[i++];

}

else {

mergedlist.\_vector[k++] = list.\_vector[j++];

}

}

if (i <= part){

for (int m=i; m<=part && k<=end;){

mergedlist.\_vector[k++] = list.\_vector[m++];

}

}

else {

for (int m=j; m<=end && k<=end; m++){

mergedlist.\_vector[k++] = list.\_vector[m];

}

}

}

///Merge///

template<typename T>

void Sort<T>::Merge(DataList<T> &list, DataList<T> &mergedlist, const int len){

int n = 0;

while (n+2\*len < list.currentSize){

MergeDouble(list, mergedlist, n, n+len-1, n+2\*len-1);

n += 2\*len;

}

if (n+len < list.currentSize){

MergeDouble(list, mergedlist, n, n+len-1, list.currentSize-1);

}

else {

for (int i=n; i<list.currentSize; i++){

mergedlist.\_vector[i] = list.\_vector[i];

}

}

}

///MergeSort///

template<typename T>

void Sort<T>::MergeSort(DataList<T> &list){

DataList<T> temp(list.maxSize);

temp.currentSize = list.currentSize;

int len = 1;

while (len < list.currentSize){

Merge(list, temp, len);

len \*= 2;

Merge(temp, list, len);

len \*= 2;

}

}

///RadixSort///

template<typename T>

void Sort<T>::RadixSort(DataList<int> &list, int m, int d){

LinkQueue<int> \*queue = new LinkQueue<int>[d];

int power = 1;

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

if (i){

power = power \* d;

}

for (int j=0; j<list.currentSize; j++){

int k = (list.\_vector[j].GetKey() / power) % d;

queue[k].Append(list.\_vector[j].GetKey());

}

for (int h=0,l=0; h<d; h++){

while (!queue[h].IsEmpty()){

list.\_vector[l++].SetKey(queue[h].Delete());

}

}

}

}

test.cpp

#include <iostream>

using namespace std;

#include "Sort.h"

int main(){

int init[15]={1,3,5,7,4,2,8,0,6,9,29,13,25,11,32};

DataList<int> data(init, 15);

Sort<int> sort;

data.Print();

cout << endl << endl <<endl;

sort.InsertSort(data);

sort.BinaryInsertSort(data);

sort.ShellSort(data);

sort.BubbleSort(data);

sort.QuickSort(data);

sort.SelectSort(data);

sort.HeapSort(data);

sort.MergeSort(data);

sort.RadixSort(data, 2, 10);

data.Print();

return 0;

}