## 目 录

[**1、顺序表 1**](#_Toc208152380)

[**Seqlist.h 1**](#_Toc208152381)

[**Test.cpp 6**](#_Toc208152382)

[**2、单链表 8**](#_Toc208152383)

[**ListNode.h 8**](#_Toc208152384)

[**SingleList.h 10**](#_Toc208152385)

[**test.cpp 20**](#_Toc208152386)

[**3、双向链表 22**](#_Toc208152387)

[**NodeList.h 22**](#_Toc208152388)

[**DoubleList.h 24**](#_Toc208152389)

[**Test.cpp 34**](#_Toc208152390)

[**4、循环链表 36**](#_Toc208152391)

[**ListNode.h 36**](#_Toc208152392)

[**CircularList.h 37**](#_Toc208152393)

[**Test.cpp 47**](#_Toc208152394)

[**5、顺序栈 49**](#_Toc208152395)

[**SeqStack.h 49**](#_Toc208152396)

[**Test.cpp 54**](#_Toc208152397)

[**6、链式栈 55**](#_Toc208152398)

[**StackNode.h 55**](#_Toc208152399)

[**LinkStack.h 56**](#_Toc208152400)

[**Test.cpp 60**](#_Toc208152401)

[**7、顺序队列 62**](#_Toc208152402)

[**SeqQueue.h 63**](#_Toc208152403)

[**Test.cpp 68**](#_Toc208152404)

[**8、链式队列 70**](#_Toc208152405)

[**QueueNode.h 70**](#_Toc208152406)

[**LinkQueue.h 71**](#_Toc208152407)

[**Test.cpp 75**](#_Toc208152408)

[**9、优先级队列 77**](#_Toc208152409)

[**QueueNode.h 77**](#_Toc208152410)

[**Compare.h 78**](#_Toc208152411)

[**PriorityQueue.h 80**](#_Toc208152412)

[**Test.cpp 85**](#_Toc208152413)

[**10、串 88**](#_Toc208152414)

[**MyString.h 88**](#_Toc208152415)

[**MyString.cpp 90**](#_Toc208152416)

[**test.cpp 101**](#_Toc208152417)

[**11、二叉树 104**](#_Toc208152418)

[**BinTreeNode.h 104**](#_Toc208152419)

[**BinaryTree.h 112**](#_Toc208152420)

[**Test.cpp 124**](#_Toc208152421)

[**12、线索二叉树 126**](#_Toc208152422)

[**ThreadNode.h 126**](#_Toc208152423)

[**ThreadTree.h 128**](#_Toc208152424)

[**ThreadInorderIterator.h 128**](#_Toc208152425)

[**test.cpp 139**](#_Toc208152426)

[**13、堆 140**](#_Toc208152427)

[**MinHeap.h 140**](#_Toc208152428)

[**test.cpp 147**](#_Toc208152429)

[**14、哈夫曼树 149**](#_Toc208152430)

[**BinTreeNode.h 149**](#_Toc208152431)

[**BinaryTree.h 151**](#_Toc208152432)

[**MinHeap.h 156**](#_Toc208152433)

[**Huffman.h 161**](#_Toc208152434)

[**Test.cpp 163**](#_Toc208152435)

[**15、树 164**](#_Toc208152436)

[**QueueNode.h 164**](#_Toc208152437)

[**LinkQueue.h 165**](#_Toc208152438)

[**TreeNode.h 169**](#_Toc208152439)

[**Tree.h 170**](#_Toc208152440)

[**test.cpp 187**](#_Toc208152441)

[**16、B+树 189**](#_Toc208152442)

[**BTreeNode.h 189**](#_Toc208152443)

[**BTree.h 192**](#_Toc208152444)

[**test.cpp 215**](#_Toc208152445)

[**17、图 217**](#_Toc208152446)

[**MinHeap.h 217**](#_Toc208152447)

[**Edge.h 222**](#_Toc208152448)

[**Vertex.h 223**](#_Toc208152449)

[**Graph.h 224**](#_Toc208152450)

[**test.cpp 246**](#_Toc208152451)

[**18、排序 249**](#_Toc208152452)

[**Data.h 249**](#_Toc208152453)

[**QueueNode.h 255**](#_Toc208152454)

[**LinkQueue.h 259**](#_Toc208152455)

[**Sort.h 263**](#_Toc208152456)

[**test.cpp 278**](#_Toc208152457)

## 1、顺序表

### Seqlist.h

**const int DefaultSize=100;**

**template <typename Type> class SeqList{**

**public:**

**SeqList(int sz=DefaultSize)**

**:m\_nmaxsize(sz),m\_ncurrentsize(-1){**

**if(sz>0){**

**m\_elements=new Type[m\_nmaxsize];**

**}**

**}**

**~SeqList(){**

**delete[] m\_elements;**

**}**

**int Length() const{ //get the length**

**return m\_ncurrentsize+1;**

**}**

**int Find(Type x) const; //find the position of x**

**int IsElement(Type x) const; //is it in the list**

**int Insert(Type x,int i); //insert data**

**int Remove(Type x); //delete data**

**int IsEmpty(){**

**return m\_ncurrentsize==-1;**

**}**

**int IsFull(){**

**return m\_ncurrentsize==m\_nmaxsize-1;**

**}**

**Type Get(int i){ //get the ith data**

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

**}**

**void Print();**

**private:**

**Type \*m\_elements;**

**const int m\_nmaxsize;**

**int m\_ncurrentsize;**

**};**

**template <typename Type> int SeqList<Type>::Find(Type x) const{**

**for(int i=0;i<m\_ncurrentsize;i++)**

**if(m\_elements[i]==x)**

**return i;**

**cout<<"can't find the element you want to find"<<endl;**

**return -1;**

**}**

**template <typename Type> int SeqList<Type>::IsElement(Type x) const{**

**if(Find(x)==-1)**

**return 0;**

**return 1;**

**}**

**template <typename Type> int SeqList<Type>::Insert(Type x, int i){**

**if(i<0||i>m\_ncurrentsize+1||m\_ncurrentsize==m\_nmaxsize-1){**

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

**return 0;**

**}**

**m\_ncurrentsize++;**

**for(int j=m\_ncurrentsize;j>i;j--){**

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

**}**

**m\_elements[i]=x;**

**return 1;**

**}**

**template <typename Type> int SeqList<Type>::Remove(Type x){**

**int size=m\_ncurrentsize;**

**for(int i=0;i<m\_ncurrentsize;){**

**if(m\_elements[i]==x){**

**for(int j=i;j<m\_ncurrentsize;j++){**

**m\_elements[j]=m\_elements[j+1];**

**}**

**m\_ncurrentsize--;**

**continue;**

**}**

**i++;**

**}**

**if(size==m\_ncurrentsize){**

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

**return 0;**

**}**

**return 1;**

**}**

**template <typename Type> void SeqList<Type>::Print(){**

**for(int i=0;i<=m\_ncurrentsize;i++)**

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

**cout<<endl<<endl;**

**}**

### Test.cpp

**#include <iostream>**

**#include "SeqList.h"**

**using namespace std;**

**int main()**

**{**

**SeqList<int> test(15);**

**int array[15]={2,5,8,1,9,9,7,6,4,3,2,9,7,7,9};**

**for(int i=0;i<15;i++){**

**test.Insert(array[i],0);**

**}**

**test.Insert(1,0);**

**cout<<(test.Find(0)?"can't be found ":"Be found ")<< 0 << endl<<endl;**

**test.Remove(7);**

**test.Print();**

**test.Remove(9);**

**test.Print();**

**test.Remove(0);**

**test.Print();**

**return 0;**

**}**

## 单链表

### ListNode.h

**template<typename Type> class SingleList;**

**template<typename Type> class ListNode{**

**private:**

**friend typename SingleList<Type>;**

**ListNode():m\_pnext(NULL){}**

**ListNode(const Type item,ListNode<Type> \*next=NULL):m\_data(item),m\_pnext(next){}**

**~ListNode(){**

**m\_pnext=NULL;**

**}**

**public:**

**Type GetData();**

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

**private:**

**Type m\_data;**

**ListNode \*m\_pnext;**

**};**

**template<typename Type> Type ListNode<Type>::GetData(){**

**return this->m\_data;**

**}**

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

**os<<out.m\_data;**

**return os;**

**}**

### SingleList.h

**#include "ListNode.h"**

**template<typename Type> class SingleList{**

**public:**

**SingleList():head(new ListNode<Type>()){}**

**~SingleList(){**

**MakeEmpty();**

**delete head;**

**}**

**public:**

**void MakeEmpty(); //make the list empty**

**int Length(); //get the length**

**ListNode<Type> \*Find(Type value,int n); //find thd nth data which is equal to value**

**ListNode<Type> \*Find(int n); //find the nth data**

**bool Insert(Type item,int n=0); //insert the data in the nth position**

**Type Remove(int n=0); //remove the nth data**

**bool RemoveAll(Type item); //remove all the data which is equal to item**

**Type Get(int n); //get the nth data**

**void Print(); //print the list**

**private:**

**ListNode<Type> \*head;**

**};**

**template<typename Type> void SingleList<Type>::MakeEmpty(){**

**ListNode<Type> \*pdel;**

**while(head->m\_pnext!=NULL){**

**pdel=head->m\_pnext;**

**head->m\_pnext=pdel->m\_pnext;**

**delete pdel;**

**}**

**}**

**template<typename Type> int SingleList<Type>::Length(){**

**ListNode<Type> \*pmove=head->m\_pnext;**

**int count=0;**

**while(pmove!=NULL){**

**pmove=pmove->m\_pnext;**

**count++;**

**}**

**return count;**

**}**

**template<typename Type> ListNode<Type>\* SingleList<Type>::Find(int n){**

**if(n<0){**

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

**return NULL;**

**}**

**ListNode<Type> \*pmove=head->m\_pnext;**

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

**pmove=pmove->m\_pnext;**

**}**

**if(pmove==NULL){**

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

**return NULL;**

**}**

**return pmove;**

**}**

**template<typename Type> ListNode<Type>\* SingleList<Type>::Find(Type value,int n){**

**if(n<1){**

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

**return NULL;**

**}**

**ListNode<Type> \*pmove=head;**

**int count=0;**

**while(count!=n&&pmove){**

**pmove=pmove->m\_pnext;**

**if(pmove->m\_data==value){**

**count++;**

**}**

**}**

**if(pmove==NULL){**

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

**return NULL;**

**}**

**return pmove;**

**}**

**template<typename Type> bool SingleList<Type>::Insert(Type item, int n){**

**if(n<0){**

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

**return 0;**

**}**

**ListNode<Type> \*pmove=head;**

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

**if(pnode==NULL){**

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

**return 0;**

**}**

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

**pmove=pmove->m\_pnext;**

**}**

**if(pmove==NULL){**

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

**return 0;**

**}**

**pnode->m\_pnext=pmove->m\_pnext;**

**pmove->m\_pnext=pnode;**

**return 1;**

**}**

**template<typename Type> bool SingleList<Type>::RemoveAll(Type item){**

**ListNode<Type> \*pmove=head;**

**ListNode<Type> \*pdel=head->m\_pnext;**

**while(pdel!=NULL){**

**if(pdel->m\_data==item){**

**pmove->m\_pnext=pdel->m\_pnext;**

**delete pdel;**

**pdel=pmove->m\_pnext;**

**continue;**

**}**

**pmove=pmove->m\_pnext;**

**pdel=pdel->m\_pnext;**

**}**

**return 1;**

**}**

**template<typename Type> Type SingleList<Type>::Remove(int n){**

**if(n<0){**

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

**exit(1);**

**}**

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

**for(int i=0;i<n&&pmove->m\_pnext;i++){**

**pmove=pmove->m\_pnext;**

**}**

**if(pmove->m\_pnext==NULL){**

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

**exit(1);**

**}**

**pdel=pmove->m\_pnext;**

**pmove->m\_pnext=pdel->m\_pnext;**

**Type temp=pdel->m\_data;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type SingleList<Type>::Get(int n){**

**if(n<0){**

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

**exit(1);**

**}**

**ListNode<Type> \*pmove=head->m\_pnext;**

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

**pmove=pmove->m\_pnext;**

**if(NULL==pmove){**

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

**exit(1);**

**}**

**}**

**return pmove->m\_data;**

**}**

**template<typename Type> void SingleList<Type>::Print(){**

**ListNode<Type> \*pmove=head->m\_pnext;**

**cout<<"head";**

**while(pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->over"<<endl<<endl<<endl;**

**}**

### test.cpp

**#include <iostream>**

**using namespace std;**

**#include "SingleList.h"**

**int main()**

**{**

**SingleList<int> list;**

**for(int i=0;i<20;i++){**

**list.Insert(i\*3,i);**

**}**

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

**list.Insert(3,i\*3);**

**}**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**list.Remove(5);**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**list.RemoveAll(3);**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**cout<<"The third element is "<<list.Get(3)<<endl;**

**cout<<\*list.Find(18,1)<<endl;**

**list.Find(100);**

**list.MakeEmpty();**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**return 0;**

**}**

## 双向链表

### NodeList.h

**template<typename Type> class DoublyList;**

**template<typename Type> class ListNode{**

**private:**

**friend class DoublyList<Type>;**

**ListNode():m\_pprior(NULL),m\_pnext(NULL){}**

**ListNode(const Type item,ListNode<Type> \*prior=NULL,ListNode<Type> \*next=NULL)**

**:m\_data(item),m\_pprior(prior),m\_pnext(next){}**

**~ListNode(){**

**m\_pprior=NULL;**

**m\_pnext=NULL;**

**}**

**public:**

**Type GetData();**

**private:**

**Type m\_data;**

**ListNode \*m\_pprior;**

**ListNode \*m\_pnext;**

**};**

**template<typename Type> Type ListNode<Type>::GetData(){**

**return this->m\_data;**

**}**

### DoubleList.h

**#include "ListNode.h"**

**template<typename Type> class DoublyList{**

**public:**

**DoublyList():head(new ListNode<Type>()){ //the head node point to itself**

**head->m\_pprior=head;**

**head->m\_pnext=head;**

**}**

**~DoublyList(){**

**MakeEmpty();**

**delete head;**

**}**

**public:**

**void MakeEmpty(); //make the list empty**

**int Length(); //get the length of the list**

**ListNode<Type> \*Find(int n=0); //find the nth data**

**ListNode<Type> \* FindData(Type item); //find the data which is equal to item**

**bool Insert(Type item,int n=0); //insert item in the nth data**

**Type Remove(int n=0); //delete the nth data**

**Type Get(int n=0); //get the nth data**

**void Print(); //print the list**

**private:**

**ListNode<Type> \*head;**

**};**

**template<typename Type> void DoublyList<Type>::MakeEmpty(){**

**ListNode<Type> \*pmove=head->m\_pnext,\*pdel;**

**while(pmove!=head){**

**pdel=pmove;**

**pmove=pdel->m\_pnext;**

**delete pdel;**

**}**

**head->m\_pnext=head;**

**head->m\_pprior=head;**

**}**

**template<typename Type> int DoublyList<Type>::Length(){**

**ListNode<Type> \*pprior=head->m\_pprior,\*pnext=head->m\_pnext;**

**int count=0;**

**while(1){**

**if(pprior->m\_pnext==pnext){**

**break;**

**}**

**if(pprior==pnext&&pprior!=head){**

**count++;**

**break;**

**}**

**count+=2;**

**pprior=pprior->m\_pprior;**

**pnext=pnext->m\_pnext;**

**}**

**return count;**

**}**

**template<typename Type> ListNode<Type>\* DoublyList<Type>::Find(int n = 0){**

**if(n<0){**

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

**return NULL;**

**}**

**ListNode<Type> \*pmove=head->m\_pnext;**

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

**pmove=pmove->m\_pnext;**

**if(pmove==head){**

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

**return NULL;**

**}**

**}**

**return pmove;**

**}**

**template<typename Type> bool DoublyList<Type>::Insert(Type item,int n){**

**if(n<0){**

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

**return 0;**

**}**

**ListNode<Type> \*newnode=new ListNode<Type>(item),\*pmove=head;**

**if(newnode==NULL){**

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

**exit(1);**

**}**

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

**pmove=pmove->m\_pnext;**

**if(pmove==head){**

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

**return 0;**

**}**

**}**

**//insert the data**

**newnode->m\_pnext=pmove->m\_pnext;**

**newnode->m\_pprior=pmove;**

**pmove->m\_pnext=newnode;**

**newnode->m\_pnext->m\_pprior=newnode;**

**return 1;**

**}**

**template<typename Type> Type DoublyList<Type>::Remove(int n = 0){**

**if(n<0){**

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

**exit(1);**

**}**

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

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

**pmove=pmove->m\_pnext;**

**if(pmove==head){**

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

**exit(1);**

**}**

**}**

**//delete the data**

**pdel=pmove;**

**pmove->m\_pprior->m\_pnext=pdel->m\_pnext;**

**pmove->m\_pnext->m\_pprior=pdel->m\_pprior;**

**Type temp=pdel->m\_data;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type DoublyList<Type>::Get(int n = 0){**

**if(n<0){**

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

**exit(1);**

**}**

**ListNode<Type> \*pmove=head;**

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

**pmove=pmove->m\_pnext;**

**if(pmove==head){**

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

**exit(1);**

**}**

**}**

**return pmove->m\_data;**

**}**

**template<typename Type> void DoublyList<Type>::Print(){**

**ListNode<Type> \*pmove=head->m\_pnext;**

**cout<<"head";**

**while(pmove!=head){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->over"<<endl<<endl<<endl;**

**}**

**template<typename Type> ListNode<Type>\* DoublyList<Type>::FindData(Type item){**

**ListNode<Type> \*pprior=head->m\_pprior,\*pnext=head->m\_pnext;**

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

**if(pprior->m\_data==item){**

**return pprior;**

**}**

**if(pnext->m\_data==item){**

**return pnext;**

**}**

**pprior=pprior->m\_pprior;**

**pnext=pnext->m\_pnext;**

**}**

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

**return NULL;**

**}**

### Test.cpp

**#include <iostream>**

**#include "DoublyList.h"**

**using namespace std;**

**int main()**

**{**

**DoublyList<int> list;**

**for(int i=0;i<20;i++){**

**list.Insert(i\*3,i);**

**}**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

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

**list.Insert(3,i\*3);**

**}**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**list.Remove(5);**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**cout<<list.FindData(54)->GetData()<<endl;**

**cout<<"The third element is "<<list.Get(3)<<endl;**

**list.MakeEmpty();**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**return 0;**

**}**

## 循环链表

### ListNode.h

**template<typename Type> class CircularList;**

**template<typename Type> class ListNode{**

**private:**

**friend class CircularList<Type>;**

**ListNode():m\_pnext(NULL){}**

**ListNode(const Type item,ListNode<Type> \*next=NULL):m\_data(item),m\_pnext(next){}**

**~ListNode(){**

**m\_pnext=NULL;**

**}**

**private:**

**Type m\_data;**

**ListNode \*m\_pnext;**

**};**

### CircularList.h

**#include "ListNode.h"**

**template<typename Type> class CircularList{**

**public:**

**CircularList():head(new ListNode<Type>()){**

**head->m\_pnext=head;**

**}**

**~CircularList(){**

**MakeEmpty();**

**delete head;**

**}**

**public:**

**void MakeEmpty(); //clear the list**

**int Length(); //get the length**

**ListNode<Type> \*Find(Type value,int n); //find the nth data which is equal to value**

**ListNode<Type> \*Find(int n); //find the nth data**

**bool Insert(Type item,int n=0); //insert the data into the nth data of the list**

**Type Remove(int n=0); //delete the nth data**

**bool RemoveAll(Type item); //delete all the datas which are equal to value**

**Type Get(int n); //get the nth data**

**void Print(); //print the list**

**private:**

**ListNode<Type> \*head;**

**};**

**template<typename Type> void CircularList<Type>::MakeEmpty(){**

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

**while(pmove->m\_pnext!=head){**

**pdel=pmove->m\_pnext;**

**pmove->m\_pnext=pdel->m\_pnext;**

**delete pdel;**

**}**

**}**

**template<typename Type> int CircularList<Type>::Length(){**

**ListNode<Type> \*pmove=head;**

**int count=0;**

**while(pmove->m\_pnext!=head){**

**pmove=pmove->m\_pnext;**

**count++;**

**}**

**return count;**

**}**

**template<typename Type> ListNode<Type>\* CircularList<Type>::Find(int n){**

**if(n<0){**

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

**return NULL;**

**}**

**ListNode<Type> \*pmove=head->m\_pnext;**

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

**pmove=pmove->m\_pnext;**

**}**

**if(pmove==head){**

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

**return NULL;**

**}**

**return pmove;**

**}**

**template<typename Type> ListNode<Type>\* CircularList<Type>::Find(Type value,int n){**

**if(n<1){**

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

**return NULL;**

**}**

**ListNode<Type> \*pmove=head;**

**int count=0;**

**while(count!=n){**

**pmove=pmove->m\_pnext;**

**if(pmove->m\_data==value){**

**count++;**

**}**

**if(pmove==head){**

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

**return NULL;**

**}**

**}**

**return pmove;**

**}**

**template<typename Type> bool CircularList<Type>::Insert(Type item, int n){**

**if(n<0){**

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

**return 0;**

**}**

**ListNode<Type> \*pmove=head;**

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

**if(pnode==NULL){**

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

**exit(1);**

**}**

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

**pmove=pmove->m\_pnext;**

**if(pmove==head){**

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

**return 0;**

**}**

**}**

**pnode->m\_pnext=pmove->m\_pnext;**

**pmove->m\_pnext=pnode;**

**return 1;**

**}**

**template<typename Type> bool CircularList<Type>::RemoveAll(Type item){**

**ListNode<Type> \*pmove=head;**

**ListNode<Type> \*pdel=head->m\_pnext;**

**while(pdel!=head){**

**if(pdel->m\_data==item){**

**pmove->m\_pnext=pdel->m\_pnext;**

**delete pdel;**

**pdel=pmove->m\_pnext;**

**continue;**

**}**

**pmove=pmove->m\_pnext;**

**pdel=pdel->m\_pnext;**

**}**

**return 1;**

**}**

**template<typename Type> Type CircularList<Type>::Remove(int n){**

**if(n<0){**

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

**exit(1);**

**}**

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

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

**pmove=pmove->m\_pnext;**

**}**

**if(pmove->m\_pnext==head){**

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

**exit(1);**

**}**

**pdel=pmove->m\_pnext;**

**pmove->m\_pnext=pdel->m\_pnext;**

**Type temp=pdel->m\_data;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type CircularList<Type>::Get(int n){**

**if(n<0){**

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

**exit(1);**

**}**

**ListNode<Type> \*pmove=head->m\_pnext;**

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

**pmove=pmove->m\_pnext;**

**if(pmove==head){**

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

**exit(1);**

**}**

**}**

**return pmove->m\_data;**

**}**

**template<typename Type> void CircularList<Type>::Print(){**

**ListNode<Type> \*pmove=head->m\_pnext;**

**cout<<"head";**

**while(pmove!=head){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->over"<<endl<<endl<<endl;**

**}**

### Test.cpp

**#include <iostream>**

**#include "CircularList.h"**

**using namespace std;**

**int main()**

**{**

**CircularList<int> list;**

**for(int i=0;i<20;i++){**

**list.Insert(i\*3,i);**

**}**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

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

**list.Insert(3,i\*3);**

**}**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**list.Remove(5);**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**list.RemoveAll(3);**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**cout<<"The third element is "<<list.Get(3)<<endl;**

**list.MakeEmpty();**

**cout<<"the Length of the list is "<<list.Length()<<endl;**

**list.Print();**

**return 0;**

**}**

## 顺序栈

### SeqStack.h

**template<typename Type> class SeqStack{**

**public:**

**SeqStack(int sz):m\_ntop(-1),m\_nMaxSize(sz){**

**m\_pelements=new Type[sz];**

**if(m\_pelements==NULL){**

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

**exit(1);**

**}**

**}**

**~SeqStack(){**

**delete[] m\_pelements;**

**}**

**public:**

**void Push(const Type item); //push data**

**Type Pop(); //pop data**

**Type GetTop() const; //get data**

**void Print(); //print the stack**

**void MakeEmpty(){ //make the stack empty**

**m\_ntop=-1;**

**}**

**bool IsEmpty() const{**

**return m\_ntop==-1;**

**}**

**bool IsFull() const{**

**return m\_ntop==m\_nMaxSize-1;**

**}**

**private:**

**int m\_ntop;**

**Type \*m\_pelements;**

**int m\_nMaxSize;**

**};**

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

**if(IsFull()){**

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

**return;**

**}**

**m\_pelements[++m\_ntop]=item;**

**}**

**template<typename Type> Type SeqStack<Type>::Pop(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pelements[m\_ntop--];**

**}**

**template<typename Type> Type SeqStack<Type>::GetTop() const{**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pelements[m\_ntop];**

**}**

**template<typename Type> void SeqStack<Type>::Print(){**

**cout<<"bottom";**

**for(int i=0;i<=m\_ntop;i++){**

**cout<<"--->"<<m\_pelements[i];**

**}**

**cout<<"--->top"<<endl<<endl<<endl;**

**}**

### Test.cpp

**#include<iostream>**

**using namespace std;**

**#include "SeqStack.h"**

**int main(){**

**SeqStack<int> stack(10);**

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

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

**stack.Push(init[i]);**

**}**

**stack.Print();**

**stack.Push(88);**

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

**stack.Print();**

**stack.MakeEmpty();**

**stack.Print();**

**stack.Pop();**

**return 0;**

**}**

## 链式栈

### StackNode.h

**template<typename Type> class LinkStack;**

**template<typename Type> class StackNode{**

**private:**

**friend class LinkStack<Type>;**

**StackNode(Type dt,StackNode<Type> \*next=NULL):m\_data(dt),m\_pnext(next){}**

**private:**

**Type m\_data;**

**StackNode<Type> \*m\_pnext;**

**};**

### LinkStack.h

**#include "StackNode.h"**

**template<typename Type> class LinkStack{**

**public:**

**LinkStack():m\_ptop(NULL){}**

**~LinkStack(){**

**MakeEmpty();**

**}**

**public:**

**void MakeEmpty(); //make the stack empty**

**void Push(const Type item); //push the data**

**Type Pop(); //pop the data**

**Type GetTop() const; //get the data**

**void Print(); //print the stack**

**bool IsEmpty() const{**

**return m\_ptop==NULL;**

**}**

**private:**

**StackNode<Type> \*m\_ptop;**

**};**

**template<typename Type> void LinkStack<Type>::MakeEmpty(){**

**StackNode<Type> \*pmove;**

**while(m\_ptop!=NULL){**

**pmove=m\_ptop;**

**m\_ptop=m\_ptop->m\_pnext;**

**delete pmove;**

**}**

**}**

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

**m\_ptop=new StackNode<Type>(item,m\_ptop);**

**}**

**template<typename Type> Type LinkStack<Type>::GetTop() const{**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_ptop->m\_data;**

**}**

**template<typename Type> Type LinkStack<Type>::Pop(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**StackNode<Type> \*pdel=m\_ptop;**

**m\_ptop=m\_ptop->m\_pnext;**

**Type temp=pdel->m\_data;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> void LinkStack<Type>::Print(){**

**StackNode<Type> \*pmove=m\_ptop;**

**cout<<"buttom";**

**while(pmove!=NULL){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->top"<<endl<<endl<<endl;**

**}**

### Test.cpp

**#include <iostream>**

**using namespace std;**

**#include "LinkStack.h"**

**int main(){**

**LinkStack<int> stack;**

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

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

**stack.Push(init[i]);**

**}**

**stack.Print();**

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

**stack.Print();**

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

**stack.Print();**

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

**stack.Print();**

**stack.MakeEmpty();**

**stack.Print();**

**stack.Pop();**

**return 0;**

**}**

## 7.顺序队列

### SeqQueue.h

**template<typename Type> class SeqQueue{**

**public:**

**SeqQueue(int sz):m\_nrear(0),m\_nfront(0),m\_ncount(0),m\_nMaxSize(sz){**

**m\_pelements=new Type[sz];**

**if(m\_pelements==NULL){**

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

**exit(1);**

**}**

**}**

**~SeqQueue(){**

**delete[] m\_pelements;**

**}**

**void MakeEmpty(); //make the queue empty**

**bool IsEmpty();**

**bool IsFull();**

**bool Append(const Type item); //insert data**

**Type Delete(); //delete data**

**Type Get(); //get data**

**void Print(); //print the queue**

**private:**

**int m\_nrear;**

**int m\_nfront;**

**int m\_ncount;**

**int m\_nMaxSize;**

**Type \*m\_pelements;**

**};**

**template<typename Type> void SeqQueue<Type>::MakeEmpty(){**

**this->m\_ncount=0;**

**this->m\_nfront=0;**

**this->m\_nrear=0;**

**}**

**template<typename Type> bool SeqQueue<Type>::IsEmpty(){**

**return m\_ncount==0;**

**}**

**template<typename Type> bool SeqQueue<Type>::IsFull(){**

**return m\_ncount==m\_nMaxSize;**

**}**

**template<typename Type> bool SeqQueue<Type>::Append(const Type item){**

**if(IsFull()){**

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

**return 0;**

**}**

**m\_pelements[m\_nrear]=item;**

**m\_nrear=(m\_nrear+1)%m\_nMaxSize;**

**m\_ncount++;**

**return 1;**

**}**

**template<typename Type> Type SeqQueue<Type>::Delete(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**Type temp=m\_pelements[m\_nfront];**

**m\_nfront=(m\_nfront+1)%m\_nMaxSize;**

**m\_ncount--;**

**return temp;**

**}**

**template<typename Type> Type SeqQueue<Type>::Get(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pelements[m\_nfront];**

**}**

**template<typename Type> void SeqQueue<Type>::Print(){**

**cout<<"front";**

**for(int i=0;i<m\_ncount;i++){**

**cout<<"--->"<<m\_pelements[(m\_nfront+i+m\_nMaxSize)%m\_nMaxSize];**

**}**

**cout<<"--->rear"<<endl<<endl<<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]);**

**}**

**queue.Print();**

**cout<<queue.Delete()<<endl;**

**queue.Print();**

**for(int i=5;i<10;i++){**

**queue.Append(init[i]);**

**}**

**queue.Print();**

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

**queue.MakeEmpty();**

**queue.Print();**

**queue.Append(1);**

**queue.Print();**

**return 0;**

**}**

## 8、链式队列

### QueueNode.h

**template<typename Type> class LinkQueue;**

**template<typename Type> class QueueNode{**

**private:**

**friend class LinkQueue<Type>;**

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

**:m\_data(item),m\_pnext(next){}**

**private:**

**Type m\_data;**

**QueueNode<Type> \*m\_pnext;**

**};**

### LinkQueue.h

**#include "QueueNode.h"**

**template<typename Type> class LinkQueue{**

**public:**

**LinkQueue():m\_prear(NULL),m\_pfront(NULL){}**

**~LinkQueue(){**

**MakeEmpty();**

**}**

**void Append(const Type item); //insert data**

**Type Delete(); //delete data**

**Type GetFront(); //get data**

**void MakeEmpty(); //make the queue empty**

**void Print(); //print the queue**

**bool IsEmpty() const{**

**return m\_pfront==NULL;**

**}**

**private:**

**QueueNode<Type> \*m\_prear,\*m\_pfront;**

**};**

**template<typename Type> void LinkQueue<Type>::MakeEmpty(){**

**QueueNode<Type> \*pdel;**

**while(m\_pfront){**

**pdel=m\_pfront;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**}**

**}**

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

**if(m\_pfront==NULL){**

**m\_pfront=m\_prear=new QueueNode<Type>(item);**

**}**

**else{**

**m\_prear=m\_prear->m\_pnext=new QueueNode<Type>(item);**

**}**

**}**

**template<typename Type> Type LinkQueue<Type>::Delete(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**QueueNode<Type> \*pdel=m\_pfront;**

**Type temp=m\_pfront->m\_data;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type LinkQueue<Type>::GetFront(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pfront->m\_data;**

**}**

**template<typename Type> void LinkQueue<Type>::Print(){**

**QueueNode<Type> \*pmove=m\_pfront;**

**cout<<"front";**

**while(pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->rear"<<endl<<endl<<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.Delete();**

**return 0;**

**}**

## 9、优先级队列

### QueueNode.h

**template<typename Type,typename Cmp> class PriorityQueue;**

**template<typename Type,typename Cmp> class QueueNode{**

**private:**

**friend class PriorityQueue<Type,Cmp>;**

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

**:m\_data(item),m\_pnext(next){}**

**private:**

**Type m\_data;**

**QueueNode<Type,Cmp> \*m\_pnext;**

**};**

### Compare.h

**template<typename Type> class Compare{ //处理一般比较大小**

**public:**

**static bool lt(Type item1,Type item2);**

**};**

**template<typename Type> bool Compare<Type>::lt(Type item1, Type item2){**

**return item1<item2;**

**}**

**struct SpecialData{**

**friend ostream& operator<<(ostream& ,SpecialData &);**

**int m\_ntenor;**

**int m\_npir;**

**};**

**ostream& operator<<(ostream& os,SpecialData &out){**

**os<<out.m\_ntenor<<" "<<out.m\_npir;**

**return os;**

**}**

**class SpecialCmp{ //处理特殊比较大小,用户可添加适当的类**

**public:**

**static bool lt(SpecialData item1,SpecialData item2);**

**};**

**bool SpecialCmp::lt(SpecialData item1, SpecialData item2){**

**return item1.m\_npir<item2.m\_npir;**

**}**

### PriorityQueue.h

**#include "QueueNode.h"**

**#include "Compare.h"**

**template<typename Type,typename Cmp> class PriorityQueue{ //Cmp is Designed for compare**

**public:**

**PriorityQueue():m\_prear(NULL),m\_pfront(NULL){}**

**~PriorityQueue(){**

**MakeEmpty();**

**}**

**void MakeEmpty(); //make the queue empty**

**void Append(const Type item); //insert data**

**Type Delete(); //delete data**

**Type GetFront(); //get data**

**void Print(); //print the queue**

**bool IsEmpty() const{**

**return m\_pfront==NULL;**

**}**

**private:**

**QueueNode<Type,Cmp> \*m\_prear,\*m\_pfront;**

**};**

**template<typename Type,typename Cmp> void PriorityQueue<Type,Cmp>::MakeEmpty(){**

**QueueNode<Type,Cmp> \*pdel;**

**while(m\_pfront){**

**pdel=m\_pfront;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**}**

**}**

**template<typename Type,typename Cmp> void PriorityQueue<Type,Cmp>::Append(const Type item){**

**if(m\_pfront==NULL){**

**m\_pfront=m\_prear=new QueueNode<Type,Cmp>(item);**

**}**

**else{**

**m\_prear=m\_prear->m\_pnext=new QueueNode<Type,Cmp>(item);**

**}**

**}**

**template<typename Type,typename Cmp> Type PriorityQueue<Type,Cmp>::Delete(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**QueueNode<Type,Cmp> \*pdel=m\_pfront,\*pmove=m\_pfront;**

**while(pmove->m\_pnext){ //get the minimize priority's data**

**//cmp:: lt is used for compare the two data, if the front one**

**// is less than the back, then return 1**

**if(Cmp::lt(pmove->m\_pnext->m\_data,pdel->m\_pnext->m\_data)){**

**pdel=pmove;**

**}**

**pmove=pmove->m\_pnext;**

**}**

**pmove=pdel;**

**pdel=pdel->m\_pnext;**

**pmove->m\_pnext=pdel->m\_pnext;**

**Type temp=pdel->m\_data;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type,typename Cmp> Type PriorityQueue<Type,Cmp>::GetFront(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**QueueNode<Type,Cmp> \*pdel=m\_pfront,\*pmove=m\_pfront->m\_pnext;**

**while(pmove){ //get the minimize priority's data**

**if(Cmp::lt(pmove->m\_data,pdel->m\_data)){**

**pdel=pmove;**

**}**

**pmove=pmove->m\_pnext;**

**}**

**return pdel->m\_data;**

**}**

**template<typename Type,typename Cmp> void PriorityQueue<Type,Cmp>::Print(){**

**QueueNode<Type,Cmp> \*pmove=m\_pfront;**

**cout<<"front";**

**while(pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->rear"<<endl<<endl<<endl;**

**}**

### Test.cpp

**#include <iostream>**

**#include <cstdlib>**

**using namespace std;**

**#include "PriorityQueue.h"**

**int main(){**

**PriorityQueue<int,Compare<int> > queue;**

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

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

**queue.Append(init[i]);**

**}**

**queue.Print();**

**queue.Delete();**

**queue.Print();**

**system("pause");**

**system("cls");**

**PriorityQueue<SpecialData,SpecialCmp> spe\_queue;**

**int init2[5][2]={{34,2},{64,1},{18,3},{24,2},{55,4}};**

**SpecialData data[5];**

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

**data[i].m\_npir=init2[i][1];**

**data[i].m\_ntenor=init2[i][0];**

**}**

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

**spe\_queue.Append(data[i]);**

**}**

**spe\_queue.Print();**

**cout<<spe\_queue.GetFront()<<endl<<endl;**

**spe\_queue.Delete();**

**spe\_queue.Print();**

**return 0;**

**}**

## 10、串

### MyString.h

**const int MAXSIZE=100;**

**class CMyString**

**{**

**public:**

**CMyString(const CMyString& copy);**

**CMyString(const char \*init);**

**CMyString();**

**~CMyString(){**

**delete[] m\_pstr;**

**}**

**int Length() const{**

**return m\_ncurlen;**

**}**

**int Find(CMyString part) const;**

**char\* GetBuffer() const;**

**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 m\_ncurlen==0;**

**}**

**CMyString& operator=(const CMyString &copy);**

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

**char& operator[](int i);**

**friend ostream& operator<<(ostream& ,CMyString&);**

**friend istream& operator>>(istream& ,CMyString&);**

**private:**

**void Next();**

**private:**

**int m\_ncurlen;**

**char \*m\_pstr;**

**int \*m\_pnext;**

**};**

### MyString.cpp

**#include <iostream>**

**#include <cstring>**

**using namespace std;**

**#include "MyString.h"**

**CMyString::CMyString(){ //create empty string**

**m\_pstr=new char[MAXSIZE+1];**

**if(!m\_pstr){**

**cerr<<"Allocation Error"<<endl;**

**exit(1);**

**}**

**this->m\_ncurlen=0;**

**m\_pstr[0]='\0';**

**}**

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

**m\_pstr=new char[MAXSIZE+1];**

**if(!m\_pstr){**

**cerr<<"Allocation Error"<<endl;**

**exit(1);**

**}**

**this->m\_ncurlen=strlen(init);**

**strcpy(m\_pstr,init);**

**}**

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

**m\_pstr=new char[MAXSIZE+1];**

**if(!m\_pstr){**

**cerr<<"Allocation Error"<<endl;**

**exit(1);**

**}**

**this->m\_ncurlen=copy.m\_ncurlen;**

**strcpy(m\_pstr,copy.m\_pstr);**

**}**

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

**int posP=0,posT=0;**

**int lengthP=part.m\_ncurlen,lengthT=this->m\_ncurlen;**

**part.Next();**

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

**if(part.m\_pstr[posP]==this->m\_pstr[posT]){**

**posP++;**

**posT++;**

**}**

**else{**

**if(posP==0){**

**posT++;**

**}**

**else{**

**posP=part.m\_pnext[posP-1];**

**}**

**}**

**}**

**delete[] part.m\_pnext;**

**if(posP<lengthP){**

**return 0;**

**}**

**else{**

**return 1;**

**}**

**}**

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

**int length=this->m\_ncurlen;**

**this->m\_pnext=new int[length];**

**this->m\_pnext[0]=0;**

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

**int j=this->m\_pnext[i-1];**

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

**j=this->m\_pnext[j-1];**

**}**

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

**this->m\_pnext[i]=j+1;**

**}**

**else{**

**this->m\_pnext[i]=0;**

**}**

**}**

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

**// cout<<i<<":\t"<<m\_pnext[i]<<endl;**

**}**

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

**return this->m\_pstr;**

**}**

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

**CMyString \*temp=new CMyString;**

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

**temp->m\_ncurlen=0;**

**temp->m\_pstr[0]='\0';**

**}**

**else{**

**if(pos+len-1>=m\_ncurlen){**

**len=m\_ncurlen-pos;**

**}**

**temp->m\_ncurlen=len;**

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

**temp->m\_pstr[i]=m\_pstr[j];**

**}**

**temp->m\_pstr[len]='\0';**

**}**

**return \*temp;**

**}**

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

**if(this->m\_ncurlen!=cmp\_str.m\_ncurlen){**

**return 0;**

**}**

**for(int i=0;i<this->m\_ncurlen;i++){**

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

**return 0;**

**}**

**return 1;**

**}**

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

**if(\*this==cmp\_str)**

**return 0;**

**return 1;**

**}**

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

**if(this->m\_ncurlen!=cmp\_str.m\_ncurlen){**

**return this->m\_ncurlen<cmp\_str.m\_ncurlen;**

**}**

**for(int i=0;i<this->m\_ncurlen;i++){**

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

**return this->m\_pnext[i]<cmp\_str.m\_pnext[i];**

**}**

**}**

**return 0;**

**}**

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

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

**return 0;**

**}**

**return 1;**

**}**

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

**delete[] this->m\_pstr;**

**this->m\_pstr=new char[copy.m\_ncurlen+1];**

**strcpy**

**(this->m\_pstr,copy.m\_pstr);**

**return \*this;**

**}**

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

**int length=this->m\_ncurlen+add.m\_ncurlen;**

**int n=this->m\_ncurlen;**

**CMyString temp(\*this);**

**delete[] this->m\_pstr;**

**this->m\_pstr=new char[length+1];**

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

**this->m\_pstr[i]=temp[i];**

**}**

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

**this->m\_pstr[i]=add.m\_pstr[i-n];**

**}**

**this->m\_pstr[length]='\0';**

**return \*this;**

**}**

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

**if(i<0||i>=this->m\_ncurlen){**

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

**exit(1);**

**}**

**return this->m\_pstr[i];**

**}**

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

**os<<str.m\_pstr;**

**return os;**

**}**

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

**is>>str.m\_pstr;**

**return is;**

**}**

### 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;**

**}**

## 11、二叉树

### BinTreeNode.h

**template<typename Type> class BinaryTree;**

**template<typename Type> class BinTreeNode{**

**public:**

**friend class BinaryTree<Type>;**

**BinTreeNode():m\_pleft(NULL),m\_pright(NULL){}**

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

**:m\_data(item),m\_pleft(left),m\_pright(right){}**

**Type GetData() const; //get thd data**

**BinTreeNode<Type> \*GetLeft() const; //get the left node**

**BinTreeNode<Type> \*GetRight() const; //get the right node**

**void SetData(const Type data); //change the data**

**void SetLeft(const BinTreeNode<Type> \*left); //change thd left node**

**void SetRight(const BinTreeNode<Type> \*right); //change the right node**

**void InOrder(); //inorder the tree with the root of the node**

**void PreOrder(); //perorder the tree with the root of the node**

**void PostOrder(); //postoder the tree with the root of the node**

**int Size(); //get size**

**int Height(); //get height**

**BinTreeNode<Type> \*Copy(const BinTreeNode<Type> \*copy); //copy the node**

**void Destroy(){ //destroy the tree with the root of the node**

**if(this!=NULL){**

**this->m\_pleft->Destroy();**

**this->m\_pright->Destroy();**

**delete this;**

**}**

**}**

**friend bool equal<Type>(const BinTreeNode<Type> \*s,const BinTreeNode<Type> \*t); //is equal?**

**private:**

**BinTreeNode<Type> \*m\_pleft,\*m\_pright;**

**Type m\_data;**

**};**

**template<typename Type> Type BinTreeNode<Type>::GetData() const{**

**return this!=NULL?m\_data:-1;**

**}**

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

**return this!=NULL?m\_pleft:NULL;**

**}**

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

**return this!=NULL?m\_pright:NULL;**

**}**

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

**if(this!=NULL){**

**m\_data=data;**

**}**

**}**

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

**if(this!=NULL){**

**m\_pleft=left;**

**}**

**}**

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

**if(this!=NULL){**

**m\_pright=right;**

**}**

**}**

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

**if(copy==NULL){**

**return NULL;**

**}**

**BinTreeNode<Type> \*temp=new BinTreeNode<Type>(copy->m\_data);**

**temp->m\_pleft=Copy(copy->m\_pleft);**

**temp->m\_pright=Copy(copy->m\_pright);**

**return temp;**

**}**

**template<typename Type> bool equal(const BinTreeNode<Type> \*s,const BinTreeNode<Type> \*t){**

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

**return 1;**

**}**

**if(s&&t&&s->m\_data==t->m\_data&&equal(s->m\_pleft,t->m\_pleft)&&equal(s->m\_pright,t->m\_pright)){**

**return 1;**

**}**

**return 0;**

**}**

**template<typename Type> void BinTreeNode<Type>::InOrder(){**

**if(this!=NULL){**

**this->m\_pleft->InOrder();**

**cout<<"--->"<<this->m\_data;**

**this->m\_pright->InOrder();**

**}**

**}**

**template<typename Type> void BinTreeNode<Type>::PreOrder(){**

**if(this!=NULL){**

**cout<<"--->"<<this->m\_data;**

**this->m\_pleft->PreOrder();**

**this->m\_pright->PreOrder();**

**}**

**}**

**template<typename Type> void BinTreeNode<Type>::PostOrder(){**

**if(this!=NULL){**

**this->m\_pleft->PostOrder();**

**this->m\_pright->PostOrder();**

**cout<<"--->"<<this->m\_data;**

**}**

**}**

**template<typename Type> int BinTreeNode<Type>::Size(){**

**if(this==NULL){**

**return 0;**

**}**

**return 1+this->m\_pleft->Size()+this->m\_pright->Size();**

**}**

**template<typename Type> int BinTreeNode<Type>::Height(){**

**if(this==NULL){**

**return -1;**

**}**

**int lheight,rheight;**

**lheight=this->m\_pleft->Height();**

**rheight=this->m\_pright->Height();**

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

**}**

### BinaryTree.h

**#include "BinTreeNode.h"**

**template<typename Type> class BinaryTree{**

**public:**

**BinaryTree():m\_proot(NULL){}**

**BinaryTree(const Type stop):m\_stop(stop),m\_proot(NULL){}**

**BinaryTree(BinaryTree<Type>& copy);**

**virtual ~BinaryTree(){**

**m\_proot->Destroy();**

**}**

**virtual bool IsEmpty(){ //is empty?**

**return m\_proot==NULL;**

**}**

**virtual BinTreeNode<Type> \*GetLeft(BinTreeNode<Type> \*current); //get the left node**

**virtual BinTreeNode<Type> \*GetRight(BinTreeNode<Type> \*current);//get the right node**

**virtual BinTreeNode<Type> \*GetParent(BinTreeNode<Type> \*current);//ghe thd parent**

**const BinTreeNode<Type> \*GetRoot() const; //get root**

**virtual bool Insert(const Type item); //insert a new node**

**virtual BinTreeNode<Type> \*Find(const Type item) const; //find thd node with the data**

**void InOrder();**

**void PreOrder();**

**void PostOrder();**

**int Size(); //get size**

**int Height(); //get height**

**BinaryTree<Type>& operator=(const BinaryTree<Type> copy); //evaluate node**

**friend bool operator== <Type>(const BinaryTree<Type> s,const BinaryTree<Type> t);//is equal?**

**friend ostream& operator<< <Type>(ostream& ,BinaryTree<Type>&); //output the data**

**friend istream& operator>> <Type>(istream& ,BinaryTree<Type>&); //input the data**

**private:**

**Type m\_stop; //just using for input the data;**

**BinTreeNode<Type> \*m\_proot;**

**//find the parent of current in the tree with the root of start**

**BinTreeNode<Type> \*GetParent(BinTreeNode<Type> \*start,BinTreeNode<Type> \*current);**

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

**};**

**template<typename Type> BinaryTree<Type>::BinaryTree(BinaryTree<Type>& copy){**

**if(copy.m\_proot){**

**this->m\_stop=copy.m\_stop;**

**}**

**m\_proot=m\_proot->Copy(copy.m\_proot);**

**}**

**template<typename Type> BinTreeNode<Type>\* BinaryTree<Type>::GetLeft(BinTreeNode<Type> \*current){**

**return m\_proot&&current?current->m\_pleft:NULL;**

**}**

**template<typename Type> BinTreeNode<Type>\* BinaryTree<Type>::GetRight(BinTreeNode<Type> \*current){**

**return m\_proot&&current?current->m\_pright:NULL;**

**}**

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

**return m\_proot;**

**}**

**template<typename Type> BinTreeNode<Type>\* BinaryTree<Type>::GetParent(BinTreeNode<Type> \*start, BinTreeNode<Type> \*current){**

**if(start==NULL||current==NULL){**

**return NULL;**

**}**

**if(start->m\_pleft==current||start->m\_pright==current){**

**return start;**

**}**

**BinTreeNode<Type> \*pmove;**

**if((pmove=GetParent(start->m\_pleft,current))!=NULL){//find the parent in the left subtree**

**return pmove;**

**}**

**else{**

**return GetParent(start->m\_pright,current); //find the parent in the right subtree**

**}**

**}**

**template<typename Type> BinTreeNode<Type>\* BinaryTree<Type>::GetParent(BinTreeNode<Type> \*current){**

**return m\_proot==NULL||current==m\_proot?NULL:GetParent(m\_proot,current);**

**}**

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

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

**if(m\_proot==NULL){**

**m\_proot=newnode;**

**return 1;**

**}**

**while(1){**

**if(item==pstart->m\_data){**

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

**return 0;**

**}**

**if(item<pstart->m\_data){**

**if(pstart->m\_pleft==NULL){**

**pstart->m\_pleft=newnode;**

**return 1;**

**}**

**pstart=pstart->m\_pleft; //if less than the node then insert to the left subtree**

**}**

**else{**

**if(pstart->m\_pright==NULL){**

**pstart->m\_pright=newnode;**

**return 1;**

**}**

**pstart=pstart->m\_pright;//if more than the node then insert to the right subtree**

**}**

**}**

**}**

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

**BinTreeNode<Type> \*pstart=m\_proot;**

**while(pstart){**

**if(item==pstart->m\_data){**

**return pstart;**

**}**

**if(item<pstart->m\_data){**

**pstart=pstart->m\_pleft; //if less than the node then find in the left subtree**

**}**

**else{**

**pstart=pstart->m\_pright;//if more than the node then find in the right subtree**

**}**

**}**

**return NULL;**

**}**

**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->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**

**}**

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

**if(copy.m\_proot){**

**this->m\_stop=copy.m\_stop;**

**}**

**m\_proot=m\_proot->Copy(copy.m\_proot);**

**return \*this;**

**}**

**template<typename Type> ostream& operator<<(ostream& os,BinaryTree<Type>& out){**

**out.Print(out.m\_proot);**

**return os;**

**}**

**template<typename Type> istream& operator>>(istream& is,BinaryTree<Type>& in){**

**Type item;**

**cout<<"initialize the tree:"<<endl<<"Input data(end with "<<in.m\_stop<<"!):";**

**is>>item;**

**while(item!=in.m\_stop){ //m\_stop is the end of input**

**in.Insert(item);**

**is>>item;**

**}**

**return is;**

**}**

**template<typename Type> bool operator==(const BinaryTree<Type> s,const BinaryTree<Type> t){**

**return equal(s.m\_proot,t.m\_proot);**

**}**

**template<typename Type> void BinaryTree<Type>::InOrder(){**

**this->m\_proot->InOrder();**

**}**

**template<typename Type> void BinaryTree<Type>::PreOrder(){**

**this->m\_proot->PreOrder();**

**}**

**template<typename Type> void BinaryTree<Type>::PostOrder(){**

**this->m\_proot->PostOrder();**

**}**

**template<typename Type> int BinaryTree<Type>::Size(){**

**return this->m\_proot->Size();**

**}**

**template<typename Type> int BinaryTree<Type>::Height(){**

**return this->m\_proot->Height();**

**}**

### Test.cpp

**#include <iostream>**

**using namespace std;**

**#include "BinaryTree.h"**

**int main(){**

**BinaryTree<int> tree(-1);**

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

**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;**

**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;**

**}**

## 12、线索二叉树

### ThreadNode.h

**template<typename Type> class ThreadTree;**

**template<typename Type> class ThreadInorderIterator;**

**template<typename Type> class ThreadNode{**

**public:**

**friend class ThreadTree<Type>;**

**friend class ThreadInorderIterator<Type>;**

**ThreadNode():m\_nleftthread(1),m\_nrightthread(1){**

**m\_pleft=this;**

**m\_pright=this;**

**}**

**ThreadNode(const Type item):m\_data(item),m\_pleft(NULL),m\_pright(NULL)**

**,m\_nleftthread(0),m\_nrightthread(0){}**

**private:**

**int m\_nleftthread,m\_nrightthread;**

**ThreadNode<Type> \*m\_pleft,\*m\_pright;**

**Type m\_data;**

**};**

### ThreadTree.h

**#include "ThreadNode.h"**

**template<typename Type> class ThreadInorderIterator;**

**template<typename Type> class ThreadTree{**

**public:**

**friend class ThreadInorderIterator<Type>;**

**ThreadTree():m\_proot(new ThreadNode<Type>()){}**

### ThreadInorderIterator.h

**#include "ThreadTree.h"**

**template<typename Type> class ThreadInorderIterator{**

**public:**

**ThreadInorderIterator(ThreadTree<Type> &tree):m\_ptree(tree),m\_pcurrent(tree.m\_proot){**

**//InThread(m\_ptree.m\_proot->m\_pleft,m\_ptree.m\_proot);**

**}**

**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> \*current);**

**private:**

**ThreadTree<Type> &m\_ptree;**

**ThreadNode<Type> \*m\_pcurrent;**

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

**};**

**template<typename Type> void ThreadInorderIterator<Type>::InThread(**

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

**if(current!=m\_ptree.m\_proot){**

**InThread(current->m\_pleft,pre);**

**if(current->m\_pleft==NULL){**

**current->m\_pleft=pre;**

**current->m\_nleftthread=1;**

**}**

**if(pre->m\_pright==NULL){**

**pre->m\_pright=current;**

**pre->m\_nrightthread=1;**

**}**

**pre=current;**

**InThread(current->m\_pright,pre);**

**}**

**}**

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

**while(m\_pcurrent->m\_nleftthread==0){**

**m\_pcurrent=m\_pcurrent->m\_pleft;**

**}**

**return m\_pcurrent;**

**}**

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

**ThreadNode<Type> \*pmove=m\_pcurrent->m\_pleft;**

**if(0==m\_pcurrent->m\_nleftthread){**

**while(0==pmove->m\_nrightthread){**

**pmove=pmove->m\_pright;**

**}**

**}**

**m\_pcurrent=pmove;**

**if(m\_pcurrent==m\_ptree.m\_proot){**

**return NULL;**

**}**

**return m\_pcurrent;**

**}**

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

**ThreadNode<Type> \*pmove=m\_pcurrent->m\_pright;**

**if(0==m\_pcurrent->m\_nrightthread){**

**while(0==pmove->m\_nleftthread){**

**pmove=pmove->m\_pleft;**

**}**

**}**

**m\_pcurrent=pmove;**

**if(m\_pcurrent==m\_ptree.m\_proot){**

**return NULL;**

**}**

**return m\_pcurrent;**

**}**

**template<typename Type> void ThreadInorderIterator<Type>::InOrder(){**

**ThreadNode<Type> \*pmove=m\_ptree.m\_proot;**

**while(pmove->m\_pleft!=m\_ptree.m\_proot){**

**pmove=pmove->m\_pleft;**

**}**

**m\_pcurrent=pmove;**

**cout<<"root";**

**while(pmove!=m\_ptree.m\_proot&&pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=this->Next();**

**}**

**cout<<"--->end";**

**}**

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

**left->m\_pleft=m\_pcurrent->m\_pleft;**

**left->m\_nleftthread=m\_pcurrent->m\_nleftthread;**

**left->m\_pright=m\_pcurrent;**

**left->m\_nrightthread=1;**

**m\_pcurrent->m\_pleft=left;**

**m\_pcurrent->m\_nleftthread=0;**

**if(0==left->m\_nleftthread){**

**m\_pcurrent=left->m\_pleft;**

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

**temp->m\_pright=left;**

**}**

**m\_pcurrent=left;**

**}**

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

**right->m\_pright=m\_pcurrent->m\_pright;**

**right->m\_nrightthread=m\_pcurrent->m\_nrightthread;**

**right->m\_pleft=m\_pcurrent;**

**right->m\_nleftthread=1;**

**m\_pcurrent->m\_pright=right;**

**m\_pcurrent->m\_nrightthread=0;**

**if(0==right->m\_nrightthread){**

**m\_pcurrent=right->m\_pright;**

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

**temp->m\_pleft=right;**

**}**

**m\_pcurrent=right;**

**}**

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

**ThreadNode<Type> \*current){**

**ThreadNode<Type> \*pmove=current;**

**while(0==pmove->m\_nleftthread){**

**pmove=pmove->m\_pleft;**

**}**

**pmove=pmove->m\_pleft;**

**if(pmove==m\_ptree.m\_proot){**

**if(pmove->m\_pleft==current){**

**return NULL;**

**}**

**}**

**if(pmove->m\_pright==current){**

**return pmove;**

**}**

**pmove=pmove->m\_pright;**

**while(pmove->m\_pleft!=current){**

**pmove=pmove->m\_pleft;**

**}**

**return pmove;**

**}**

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

**if(start->m\_nleftthread&&start->m\_nrightthread){**

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

**cout<<" ";**

**}**

**if(n>=0){**

**cout<<start->m\_data<<"--->"<<endl;**

**}**

**return;**

**}**

**if(start->m\_nrightthread==0){**

**Print(start->m\_pright,n+1);**

**}**

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

**cout<<" ";**

**}**

**if(n>=0){**

**cout<<start->m\_data<<"--->"<<endl;**

**}**

**if(start->m\_nleftthread==0){**

**Print(start->m\_pleft,n+1);**

**}**

**}**

**template<typename Type> void ThreadInorderIterator<Type>::Print(){**

**Print(m\_ptree.m\_proot->m\_pleft);**

**}**

### 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;**

**}**

**private:**

**ThreadNode<Type> \*m\_proot;**

**};**

## 13、堆

### MinHeap.h

**template<typename Type> class MinHeap{**

**public:**

**MinHeap(int size):m\_nMaxSize(size > defaultsize ? size : defaultsize)**

**,m\_pheap(new Type[m\_nMaxSize]),m\_ncurrentsize(0){}**

**MinHeap(Type heap[],int n); //initialize heap by a array**

**~MinHeap(){**

**delete[] m\_pheap;**

**}**

**public:**

**bool Insert(const Type item); //insert element**

**bool Delete(const Type item); //delete element**

**bool IsEmpty() const{**

**return m\_ncurrentsize == 0;**

**}**

**bool IsFull() const{**

**reutrn m\_ncurrentsize == m\_nMaxSize;**

**}**

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

**private:**

**//adjust the elements of the child tree with the root of start from top to bottom**

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

**private:**

**static const int defaultsize = 100;**

**const int m\_nMaxSize;**

**Type \*m\_pheap;**

**int m\_ncurrentsize;**

**};**

**template<typename Type> void MinHeap<Type>::Adjust(const int start, const int end){**

**int i = start,j = i\*2+1; //get the position of the child of i**

**Type temp=m\_pheap[i];**

**while(j <= end){**

**if(j<end && m\_pheap[j]>m\_pheap[j+1]){ //left>right**

**j++;**

**}**

**if(temp <= m\_pheap[j]){ //adjust over**

**break;**

**}**

**else{ //change the parent and the child, then adjust the child**

**m\_pheap[i] = m\_pheap[j];**

**i = j;**

**j = 2\*i+1;**

**}**

**}**

**m\_pheap[i] = temp;**

**}**

**template<typename Type> MinHeap<Type>::MinHeap(Type heap[], int n):m\_nMaxSize(**

**n > defaultsize ? n : defaultsize){**

**m\_pheap = new Type[m\_nMaxSize];**

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

**m\_pheap[i] = heap[i];**

**}**

**m\_ncurrentsize = n;**

**int pos=(n-2)/2; //Find the last child tree which has more than one element;**

**while(pos>=0){**

**Adjust(pos, n-1);**

**pos--;**

**}**

**}**

**template<typename Type> bool MinHeap<Type>::Insert(const Type item){**

**if(m\_ncurrentsize == m\_nMaxSize){**

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

**return 0;**

**}**

**m\_pheap[m\_ncurrentsize] = item;**

**int j = m\_ncurrentsize, i = (j-1)/2; //get the position of the parent of j**

**Type temp = m\_pheap[j];**

**while(j > 0){ //adjust from bottom to top**

**if(m\_pheap[i] <= temp){**

**break;**

**}**

**else{**

**m\_pheap[j] = m\_pheap[i];**

**j = i;**

**i = (j-1)/2;**

**}**

**}**

**m\_pheap[j] = temp;**

**m\_ncurrentsize++;**

**return 1;**

**}**

**template<typename Type> bool MinHeap<Type>::Delete(const Type item){**

**if(0 == m\_ncurrentsize){**

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

**return 0;**

**}**

**for(int i=0; i<m\_ncurrentsize; i++){**

**if(m\_pheap[i] == item){**

**m\_pheap[i] = m\_pheap[m\_ncurrentsize-1]; //filled with the last element**

**Adjust(i,m\_ncurrentsize-2); //adjust the tree with start of i**

**m\_ncurrentsize--;**

**i=0;**

**}**

**}**

**return 1;**

**}**

**template<typename Type> void MinHeap<Type>::Print(const int start, int n){**

**if(start >= m\_ncurrentsize){**

**return;**

**}**

**Print(start\*2+2, n+1); //print the right child tree**

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

**cout<<" ";**

**}**

**cout<< m\_pheap[start] << "--->" << endl;**

**Print(start\*2+1, n+1); //print the left child tree**

**}**

### 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<<endl<<endl;**

**heap.Insert(20);**

**heap.Print();**

**cout<<endl<<endl<<endl;**

**heap.Delete(20);**

**heap.Print();**

**cout<<endl<<endl<<endl;**

**return 0;**

**}**

## 14、哈夫曼树

### BinTreeNode.h

**template<typename Type> class BinaryTree;**

**template<typename Type> void Huffman(Type \*, int, BinaryTree<Type> &);**

**template<typename Type> class BinTreeNode{**

**public:**

**friend class BinaryTree<Type>;**

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

**BinTreeNode():m\_pleft(NULL),m\_pright(NULL){}**

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

**:m\_data(item),m\_pleft(left),m\_pright(right){}**

**void Destroy(){ //destroy the tree with the root of the node**

**if(this!=NULL){**

**this->m\_pleft->Destroy();**

**this->m\_pright->Destroy();**

**delete this;**

**}**

**}**

**Type GetData(){**

**return m\_data;**

**}**

**BinTreeNode<Type> \*Copy(const BinTreeNode<Type> \*copy); //copy the node**

**private:**

**BinTreeNode<Type> \*m\_pleft,\*m\_pright;**

**Type m\_data;**

**};**

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

**if(copy==NULL){**

**return NULL;**

**}**

**BinTreeNode<Type> \*temp=new BinTreeNode<Type>(copy->m\_data);**

**temp->m\_pleft=Copy(copy->m\_pleft);**

**temp->m\_pright=Copy(copy->m\_pright);**

**return temp;**

**}**

### BinaryTree.h

**#include "BinTreeNode.h"**

**template<typename Type> void Huffman(Type \*, int, BinaryTree<Type> &);**

**template<typename Type> class BinaryTree{**

**public:**

**BinaryTree(BinaryTree<Type> &bt1, BinaryTree<Type> &bt2){**

**m\_proot = new BinTreeNode<Type>(bt1.m\_proot->m\_data**

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

**}**

**BinaryTree(Type item){**

**m\_proot = new BinTreeNode<Type>(item);**

**}**

**BinaryTree(const BinaryTree<Type> &copy){**

**this->m\_proot = copy.m\_proot;**

**}**

**BinaryTree(){**

**m\_proot = NULL;**

**}**

**void Destroy(){**

**m\_proot->Destroy();**

**}**

**~BinaryTree(){**

**// m\_proot->Destroy();**

**}**

**BinaryTree<Type>& operator=(BinaryTree<Type> copy); //evaluate node**

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

**friend bool operator < <Type>(BinaryTree<Type> &l, BinaryTree<Type> & r);**

**friend bool operator > <Type>(BinaryTree<Type> &l, BinaryTree<Type> & r);**

**friend bool operator <= <Type>(BinaryTree<Type> &l, BinaryTree<Type> & r);**

**friend ostream& operator<< <Type>(ostream& ,BinaryTree<Type>&); //output the data**

**private:**

**BinTreeNode<Type> \*m\_proot;**

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

**};**

**template<typename Type> bool operator <(BinaryTree<Type> &l, BinaryTree<Type> &r){**

**return l.m\_proot->GetData() < r.m\_proot->GetData();**

**}**

**template<typename Type> bool operator >(BinaryTree<Type> &l, BinaryTree<Type> &r){**

**return l.m\_proot->GetData() > r.m\_proot->GetData();**

**}**

**template<typename Type> bool operator <=(BinaryTree<Type> &l, BinaryTree<Type> &r){**

**return l.m\_proot->GetData() <= r.m\_proot->GetData();**

**}**

**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->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**

**}**

**template<typename Type> ostream& operator<<(ostream& os,BinaryTree<Type>& out){**

**out.Print(out.m\_proot);**

**return os;**

**}**

**template<typename Type> BinaryTree<Type>& BinaryTree<Type>::operator=(BinaryTree<Type> copy){**

**m\_proot=m\_proot->Copy(copy.m\_proot);**

**return \*this;**

**}**

### MinHeap.h

**template<typename Type> class MinHeap{**

**public:**

**MinHeap(Type heap[],int n); //initialize heap by a array**

**~MinHeap(){**

**delete[] m\_pheap;**

**}**

**public:**

**bool Insert(const Type item);**

**bool DeleteMin(Type &first);**

**private:**

**void Adjust(const int start, const int end); //adjust the elements from start to end**

**private:**

**const int m\_nMaxSize;**

**Type \*m\_pheap;**

**int m\_ncurrentsize;**

**};**

**template<typename Type> void MinHeap<Type>::Adjust(const int start, const int end){**

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

**Type temp=m\_pheap[i];**

**while(j <= end){**

**if(j<end && m\_pheap[j]>m\_pheap[j+1]){**

**j++;**

**}**

**if(temp <= m\_pheap[j]){**

**break;**

**}**

**else{**

**m\_pheap[i] = m\_pheap[j];**

**i = j;**

**j = 2\*i+1;**

**}**

**}**

**m\_pheap[i] = temp;**

**}**

**template<typename Type> MinHeap<Type>::MinHeap(Type heap[], int n):m\_nMaxSize(n){**

**m\_pheap = new Type[m\_nMaxSize];**

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

**m\_pheap[i] = heap[i];**

**}**

**m\_ncurrentsize = n;**

**int pos=(n-2)/2; //Find the last tree which has more than one element;**

**while(pos>=0){**

**Adjust(pos, n-1);**

**pos--;**

**}**

**}**

**template<typename Type> bool MinHeap<Type>::DeleteMin(Type &first){**

**first = m\_pheap[0];**

**m\_pheap[0] = m\_pheap[m\_ncurrentsize-1];**

**m\_ncurrentsize--;**

**Adjust(0, m\_ncurrentsize-1);**

**return 1;**

**}**

**template<typename Type> bool MinHeap<Type>::Insert(const Type item){**

**if(m\_ncurrentsize == m\_nMaxSize){**

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

**return 0;**

**}**

**m\_pheap[m\_ncurrentsize] = item;**

**int j = m\_ncurrentsize, i = (j-1)/2;**

**Type temp = m\_pheap[j];**

**while(j > 0){**

**if(m\_pheap[i] <= temp){**

**break;**

**}**

**else{**

**m\_pheap[j] = m\_pheap[i];**

**j = i;**

**i = (j-1)/2;**

**}**

**}**

**m\_pheap[j] = temp;**

**m\_ncurrentsize++;**

**return 1;**

**}**

### Huffman.h

**#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>**

**using namespace std;**

**#include "Huffman.h"**

**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;**

**}**

## 15、树

### QueueNode.h

**template<typename Type> class LinkQueue;**

**template<typename Type> class QueueNode{**

**private:**

**friend class LinkQueue<Type>;**

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

**:m\_data(item),m\_pnext(next){}**

**private:**

**Type m\_data;**

**QueueNode<Type> \*m\_pnext;**

**};**

### LinkQueue.h

**#include "QueueNode.h"**

**template<typename Type> class LinkQueue{**

**public:**

**LinkQueue():m\_prear(NULL),m\_pfront(NULL){}**

**~LinkQueue(){**

**MakeEmpty();**

**}**

**void Append(const Type item);**

**Type Delete();**

**Type GetFront();**

**void MakeEmpty();**

**bool IsEmpty() const{**

**return m\_pfront==NULL;**

**}**

**void Print();**

**private:**

**QueueNode<Type> \*m\_prear,\*m\_pfront;**

**};**

**template<typename Type> void LinkQueue<Type>::MakeEmpty(){**

**QueueNode<Type> \*pdel;**

**while(m\_pfront){**

**pdel=m\_pfront;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**}**

**}**

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

**if(m\_pfront==NULL){**

**m\_pfront=m\_prear=new QueueNode<Type>(item);**

**}**

**else{**

**m\_prear=m\_prear->m\_pnext=new QueueNode<Type>(item);**

**}**

**}**

**template<typename Type> Type LinkQueue<Type>::Delete(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**QueueNode<Type> \*pdel=m\_pfront;**

**Type temp=m\_pfront->m\_data;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type LinkQueue<Type>::GetFront(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pfront->m\_data;**

**}**

**template<typename Type> void LinkQueue<Type>::Print(){**

**QueueNode<Type> \*pmove=m\_pfront;**

**cout<<"front";**

**while(pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->rear"<<endl<<endl<<endl;**

**}**

### TreeNode.h

**template<typename Type> class Tree;**

**template<typename Type> class TreeNode{**

**public:**

**friend class Tree<Type>;**

**private:**

**Type m\_data;**

**TreeNode<Type> \*m\_pfirst,\*m\_pnext;**

**TreeNode():m\_pfirst(NULL), m\_pnext(NULL){}**

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

**:m\_data(item), m\_pfirst(first), m\_pnext(next){}**

**};**

### Tree.h

**#include "TreeNode.h"**

**#include "LinkQueue.h"**

**template<typename Type> class Tree{**

**public:**

**Tree():m\_proot(NULL), m\_pcurrent(NULL){}**

**public:**

**TreeNode<Type> \*GetCurrent(){ //Get the current node**

**return m\_pcurrent;**

**}**

**void SetCurrent(TreeNode<Type> \*current){ //set the current node**

**m\_pcurrent = current;**

**}**

**bool Insert(Type item); //insert an new node to current node**

**void Remove(Type item); //delete the node whose data is equal to item**

**void Remove(TreeNode<Type> \*current); //delete the node**

**bool Find(Type item); //find the node whose data is equal to item**

**void PrintChild(TreeNode<Type> \*current); //print the child tree**

**TreeNode<Type> \*Parent(TreeNode<Type> \*current); //get the parent**

**void Print(); //print the tree**

**void PreOrder(TreeNode<Type> \*root); //ordering the tree by visiting the root first**

**void PostOrder(TreeNode<Type> \*root); //ordering the tree by visiting the root last**

**void LevelOrder(TreeNode<Type> \*root); //ordering the tree by level**

**void PreOrder();**

**void PostOrder();**

**void LevelOrder();**

**private:**

**TreeNode<Type> \*m\_proot,\*m\_pcurrent;**

**bool Find(TreeNode<Type> \*root, Type item);**

**void Remove(TreeNode<Type> \*root, Type item);**

**TreeNode<Type> \*Parent(TreeNode<Type> \*root, TreeNode<Type> \*current);**

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

**};**

**template<typename Type> bool Tree<Type>::Insert(Type item){**

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

**if (NULL == newnode){**

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

**exit(1);**

**}**

**if (NULL == m\_proot){**

**m\_proot = newnode;**

**m\_pcurrent = m\_proot;**

**return 1;**

**}**

**if (NULL == m\_pcurrent){**

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

**return 0;**

**}**

**if(NULL == m\_pcurrent->m\_pfirst){**

**m\_pcurrent->m\_pfirst = newnode;**

**m\_pcurrent = newnode;**

**return 1;**

**}**

**TreeNode<Type> \*pmove = m\_pcurrent->m\_pfirst;**

**while(pmove->m\_pnext){**

**pmove = pmove->m\_pnext;**

**}**

**pmove->m\_pnext = newnode;**

**m\_pcurrent = newnode;**

**return 1;**

**}**

**template<typename Type> void Tree<Type>::Remove(TreeNode<Type> \*current){**

**if(NULL == current){**

**return;**

**}**

**TreeNode<Type> \*temp = Parent(current);**

**if(NULL == temp){**

**TreeNode<Type> \*pmove = current->m\_pfirst;**

**if(NULL != pmove->m\_pfirst){**

**pmove=pmove->m\_pfirst;**

**while(pmove->m\_pnext){**

**pmove = pmove->m\_pnext;**

**}**

**pmove->m\_pnext = current->m\_pfirst->m\_pnext;**

**current->m\_pfirst->m\_pnext = NULL;**

**}**

**else{**

**pmove->m\_pfirst = pmove->m\_pnext;**

**}**

**m\_proot = current->m\_pfirst;**

**}**

**else{**

**if(temp->m\_pfirst == current){**

**TreeNode<Type> \*pmove = current->m\_pfirst;**

**if (pmove){**

**while (pmove->m\_pnext){**

**pmove = pmove->m\_pnext;**

**}**

**pmove->m\_pnext = current->m\_pnext;**

**}**

**else{**

**current->m\_pfirst = current->m\_pnext;**

**}**

**}**

**else{**

**TreeNode<Type> \*pmove = temp->m\_pfirst;**

**while(pmove->m\_pnext != current){**

**pmove = pmove->m\_pnext;**

**}**

**pmove->m\_pnext = current->m\_pnext;**

**while(pmove->m\_pnext){**

**pmove = pmove->m\_pnext;**

**}**

**pmove->m\_pnext = current->m\_pfirst;**

**}**

**}**

**delete current;**

**}**

**template<typename Type> void Tree<Type>::Remove(TreeNode<Type> \*root, Type item){**

**if(NULL == root){**

**return;**

**}**

**if(root->m\_pfirst){**

**TreeNode<Type> \*pmove=root->m\_pfirst;**

**while(pmove){**

**Remove(pmove, item);**

**pmove = pmove->m\_pnext;**

**}**

**}**

**if(root->m\_data == item){**

**Remove(root);**

**}**

**}**

**template<typename Type> void Tree<Type>::Remove(Type item){**

**return Remove(m\_proot, item);**

**}**

**template<typename Type> TreeNode<Type>\* Tree<Type>::Parent(**

**TreeNode<Type> \*root, TreeNode<Type> \*current){**

**if(NULL == root){**

**return NULL;**

**}**

**TreeNode<Type> \*pmove=root->m\_pfirst,\*temp;**

**if(NULL != pmove){**

**while(pmove){**

**if(pmove == current){**

**return root;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**}**

**pmove = root->m\_pfirst;**

**while(pmove){**

**temp = Parent(pmove, current);**

**if(temp){**

**return temp;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**return NULL;**

**}**

**template<typename Type> TreeNode<Type>\* Tree<Type>::Parent(TreeNode<Type> \*current){**

**return Parent(m\_proot,current);**

**}**

**template<typename Type> void Tree<Type>::PrintChild(TreeNode<Type> \*current){**

**TreeNode<Type> \*pmove = current->m\_pfirst;**

**cout<<"first";**

**if(NULL != pmove){**

**cout<<"--->"<<pmove->m\_data;**

**}**

**while(pmove->m\_pnext){**

**cout<<"--->"<<pmove->m\_data;**

**pmove = pmove->m\_pnext;**

**}**

**}**

**template<typename Type> bool Tree<Type>::Find(TreeNode<Type> \*root, Type item){**

**if (root->m\_data == item){**

**return 1;**

**}**

**if (NULL == root){**

**return 0;**

**}**

**TreeNode<Type> \*pmove=root->m\_pfirst;**

**if (NULL == pmove){**

**return 0;**

**}**

**while (pmove){**

**if (Find(pmove, item)){**

**return 1;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**return 0;**

**}**

**template<typename Type> bool Tree<Type>::Find(Type item){**

**return Find(m\_proot,item);**

**}**

**template<typename Type> void Tree<Type>::Print(TreeNode<Type> \*start, int n = 0){**

**if (NULL == start){**

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

**cout << " ";**

**}**

**cout << "NULL" << endl;**

**return;**

**}**

**TreeNode<Type> \*pmove = start->m\_pfirst;**

**Print(pmove, n+1);**

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

**cout << " ";**

**}**

**cout << start->m\_data << "--->" <<endl;**

**if (NULL == pmove){**

**return;**

**}**

**pmove = pmove->m\_pnext;**

**while (pmove){**

**Print(pmove, n+1);**

**pmove = pmove->m\_pnext;**

**}**

**}**

**template<typename Type> void Tree<Type>::Print(){**

**Print(m\_proot);**

**}**

**template<typename Type> void Tree<Type>::PreOrder(TreeNode<Type> \*root){**

**if (NULL == root){**

**return;**

**}**

**cout << root->m\_data;**

**TreeNode<Type> \*pmove = root->m\_pfirst;**

**while (pmove){**

**PreOrder(pmove);**

**pmove = pmove->m\_pnext;**

**}**

**}**

**template<typename Type> void Tree<Type>::PostOrder(TreeNode<Type> \*root){**

**if (NULL == root){**

**return;**

**}**

**TreeNode<Type> \*pmove = root->m\_pfirst;**

**while (pmove){**

**PostOrder(pmove);**

**pmove = pmove->m\_pnext;**

**}**

**cout << root->m\_data;**

**}**

**template<typename Type> void Tree<Type>::PreOrder(){**

**PreOrder(m\_proot);**

**}**

**template<typename Type> void Tree<Type>::PostOrder(){**

**PostOrder(m\_proot);**

**}**

**template<typename Type> void Tree<Type>::LevelOrder(TreeNode<Type> \*root){ //using queue**

**LinkQueue<TreeNode<Type> \*> queue;**

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

**if (root != NULL){**

**queue.Append(root);**

**while (!queue.IsEmpty()){**

**ptemp = queue.Delete();**

**cout << ptemp->m\_data;**

**pmove = ptemp->m\_pfirst;**

**while(pmove){**

**queue.Append(pmove);**

**pmove = pmove->m\_pnext;**

**}**

**}**

**}**

**}**

**template<typename Type> void Tree<Type>::LevelOrder(){**

**LevelOrder(m\_proot);**

**}**

### 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;**

**}**

## 16、B+树

### BTreeNode.h

**template<typename Type> class BTree;**

**template<typename Type> class BTreeNode{**

**public:**

**friend BTree<Type>;**

**BTreeNode(): m\_nMaxSize(0), m\_ptr(NULL), m\_pparent(NULL){}**

**BTreeNode(int size): m\_nsize(0), m\_nMaxSize(size), m\_pparent(NULL){**

**m\_pkey = new Type[size+1];**

**m\_ptr = new BTreeNode<Type> \*[size+1];**

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

**m\_ptr[i] = NULL;**

**m\_pkey[i] = this->m\_Infinity;**

**}**

**}**

**void Destroy(BTreeNode<Type> \*root);**

**~BTreeNode(){**

**if (m\_nMaxSize){**

**delete[] m\_pkey;**

**for (int i=0; i<=m\_nMaxSize; i++){**

**m\_ptr[i] = NULL;**

**}**

**}**

**}**

**bool IsFull(){**

**return m\_nsize == m\_nMaxSize;**

**}**

**Type GetKey(int i){**

**if (this){**

**return this->m\_pkey[i];**

**}**

**return -1;**

**}**

**private:**

**int m\_nsize;**

**int m\_nMaxSize; //the Max Size of key**

**Type \*m\_pkey;**

**BTreeNode<Type> \*m\_pparent;**

**BTreeNode<Type> \*\*m\_ptr;**

**static const Type m\_Infinity = 10000;**

**};**

**template<typename Type> struct Triple{**

**BTreeNode<Type> \*m\_pfind;**

**int m\_nfind;**

**bool m\_ntag;**

**};**

**template<typename Type> void BTreeNode<Type>::Destroy(BTreeNode<Type> \*root){**

**if (NULL == root){**

**return;**

**}**

**for (int i=0; i<root->m\_nsize; i++){**

**Destroy(root->m\_ptr[i]);**

**}**

**delete root;**

**}**

### BTree.h

**#include "BTreeNode.h"**

**template<typename Type> class BTree{**

**public:**

**BTree(int size): m\_nMaxSize(size), m\_proot(NULL){}**

**~BTree();**

**Triple<Type> Search(const Type item);**

**int Size();**

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

**bool Insert(const Type item); //insert item**

**bool Remove(const Type item); //delete item**

**void Print(); //print the BTree**

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

**private:**

**//insert the pright and item to pinsert in the nth place;**

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

**void PreMove(BTreeNode<Type> \*root, int n); //move ahead**

**//merge the child tree**

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

**//adjust with the parent and the left child tree**

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

**//adjust with the parent and the left child tree**

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

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

**private:**

**BTreeNode<Type> \*m\_proot;**

**const int m\_nMaxSize;**

**};**

**template<typename Type> BTree<Type>::~BTree(){**

**m\_proot->Destroy(m\_proot);**

**}**

**template<typename Type> Triple<Type> BTree<Type>::Search(const Type item){**

**Triple<Type> result;**

**BTreeNode<Type> \*pmove = m\_proot, \*parent = NULL;**

**int i = 0;**

**while (pmove){**

**i = -1;**

**while (item > pmove->m\_pkey[++i]); //find the suit position**

**if (pmove->m\_pkey[i] == item){**

**result.m\_pfind = pmove;**

**result.m\_nfind = i;**

**result.m\_ntag = 1;**

**return result;**

**}**

**parent = pmove;**

**pmove = pmove->m\_ptr[i]; //find in the child tree**

**}**

**result.m\_pfind = parent;**

**result.m\_nfind = i;**

**result.m\_ntag = 0;**

**return result;**

**}**

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

**pinsert->m\_nsize++;**

**for (int i=pinsert->m\_nsize; i>n; i--){**

**pinsert->m\_pkey[i] = pinsert->m\_pkey[i-1];**

**pinsert->m\_ptr[i+1] = pinsert->m\_ptr[i];**

**}**

**pinsert->m\_pkey[n] = item;**

**pinsert->m\_ptr[n+1] = pright;**

**if (pinsert->m\_ptr[n+1]){ //change the right child tree's parent**

**pinsert->m\_ptr[n+1]->m\_pparent = pinsert;**

**for (int i=0; i<=pinsert->m\_ptr[n+1]->m\_nsize; i++){**

**if (pinsert->m\_ptr[n+1]->m\_ptr[i]){**

**pinsert->m\_ptr[n+1]->m\_ptr[i]->m\_pparent = pinsert->m\_ptr[n+1];**

**}**

**}**

**}**

**}**

**template<typename Type> bool BTree<Type>::Insert(const Type item){**

**if (NULL == m\_proot){ //insert the first node**

**m\_proot = new BTreeNode<Type>(m\_nMaxSize);**

**m\_proot->m\_nsize = 1;**

**m\_proot->m\_pkey[1] = m\_proot->m\_pkey[0];**

**m\_proot->m\_pkey[0] = item;**

**m\_proot->m\_ptr[0] = m\_proot->m\_ptr[1] =NULL;**

**return 1;**

**}**

**Triple<Type> find = this->Search(item); //search the position**

**if (find.m\_ntag){**

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

**return 0;**

**}**

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

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

**Type key = item;**

**int n = find.m\_nfind;**

**while (1){**

**if (pinsert->m\_nsize < pinsert->m\_nMaxSize-1){ //There is some space**

**InsertKey(pinsert, n, key, pright);**

**return 1;**

**}**

**int m = (pinsert->m\_nsize + 1) / 2; //get the middle item**

**InsertKey(pinsert, n, key, pright); //insert first, then break up**

**newnode = new BTreeNode<Type>(this->m\_nMaxSize);//create the newnode for break up**

**//break up**

**for (int i=m+1; i<=pinsert->m\_nsize; i++){**

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

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

**pinsert->m\_pkey[i] = pinsert->m\_Infinity;**

**pinsert->m\_ptr[i] = NULL;**

**}**

**newnode->m\_nsize = pinsert->m\_nsize - m - 1;**

**pinsert->m\_nsize = m;**

**for (int i=0; i<=newnode->m\_nsize; i++){ //change the parent**

**if (newnode->m\_ptr[i]){**

**newnode->m\_ptr[i]->m\_pparent = newnode;**

**for (int j=0; j<=newnode->m\_ptr[i]->m\_nsize; j++){**

**if (newnode->m\_ptr[i]->m\_ptr[j]){**

**newnode->m\_ptr[i]->m\_ptr[j]->m\_pparent = newnode->m\_ptr[i];**

**}**

**}**

**}**

**}**

**for (int i=0; i<=pinsert->m\_nsize; i++){ //change the parent**

**if (pinsert->m\_ptr[i]){**

**pinsert->m\_ptr[i]->m\_pparent = pinsert;**

**for (int j=0; j<=pinsert->m\_nsize; j++){**

**if (pinsert->m\_ptr[i]->m\_ptr[j]){**

**pinsert->m\_ptr[i]->m\_ptr[j]->m\_pparent = pinsert->m\_ptr[i];**

**}**

**}**

**}**

**}**

**//break up over**

**key = pinsert->m\_pkey[m];**

**pright = newnode;**

**if (pinsert->m\_pparent){ //insert the key to the parent**

**pparent = pinsert->m\_pparent;**

**n = -1;**

**pparent->m\_pkey[pparent->m\_nsize] = pparent->m\_Infinity;**

**while (key > pparent->m\_pkey[++n]);**

**newnode->m\_pparent = pinsert->m\_pparent;**

**pinsert = pparent;**

**}**

**else { //create new root**

**m\_proot = new BTreeNode<Type>(this->m\_nMaxSize);**

**m\_proot->m\_nsize = 1;**

**m\_proot->m\_pkey[1] = m\_proot->m\_pkey[0];**

**m\_proot->m\_pkey[0] = key;**

**m\_proot->m\_ptr[0] = pinsert;**

**m\_proot->m\_ptr[1] = pright;**

**newnode->m\_pparent = pinsert->m\_pparent = m\_proot;**

**return 1;**

**}**

**}**

**}**

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

**root->m\_pkey[root->m\_nsize] = root->m\_Infinity;**

**for (int i=n; i<root->m\_nsize; i++){**

**root->m\_pkey[i] = root->m\_pkey[i+1];**

**root->m\_ptr[i+1] = root->m\_ptr[i+2];**

**}**

**root->m\_nsize--;**

**}**

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

**pleft->m\_pkey[pleft->m\_nsize] = pparent->m\_pkey[n];**

**BTreeNode<Type> \*ptemp;**

**for (int i=0; i<=pright->m\_nsize; i++){ //merge the two child tree and the parent**

**pleft->m\_pkey[pleft->m\_nsize+i+1] = pright->m\_pkey[i];**

**pleft->m\_ptr[pleft->m\_nsize+i+1] = pright->m\_ptr[i];**

**ptemp = pleft->m\_ptr[pleft->m\_nsize+i+1];**

**if (ptemp){ //change thd right child tree's parent**

**ptemp->m\_pparent = pleft;**

**for (int j=0; j<=ptemp->m\_nsize; j++){**

**if (ptemp->m\_ptr[j]){**

**ptemp->m\_ptr[j]->m\_pparent = ptemp;**

**}**

**}**

**}**

**}**

**pleft->m\_nsize = pleft->m\_nsize + pright->m\_nsize + 1;**

**delete pright;**

**PreMove(pparent, n);**

**// this->Print();**

**}**

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

**BTreeNode<Type> \*pleft = pparent->m\_ptr[n-1], \*ptemp;**

**if (pleft->m\_nsize > min-1){**

**for (int i=pright->m\_nsize+1; i>0; i--){**

**pright->m\_pkey[i] = pright->m\_pkey[i-1];**

**pright->m\_ptr[i] = pright->m\_ptr[i-1];**

**}**

**pright->m\_pkey[0] = pparent->m\_pkey[n-1];**

**pright->m\_ptr[0] = pleft->m\_ptr[pleft->m\_nsize];**

**ptemp = pright->m\_ptr[0];**

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

**ptemp->m\_pparent = pright;**

**for (int i=0; i<ptemp->m\_nsize; i++){**

**if (ptemp->m\_ptr[i]){**

**ptemp->m\_ptr[i]->m\_pparent = ptemp;**

**}**

**}**

**}**

**pparent->m\_pkey[n-1] = pleft->m\_pkey[pleft->m\_nsize-1];**

**pleft->m\_pkey[pleft->m\_nsize] = pleft->m\_Infinity;**

**pleft->m\_nsize--;**

**pright->m\_nsize++;**

**}**

**else {**

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

**}**

**// this->Print();**

**}**

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

**BTreeNode<Type> \*pright = pparent->m\_ptr[1], \*ptemp;**

**if (pright && pright->m\_nsize > min-1){**

**pleft->m\_pkey[pleft->m\_nsize] = pparent->m\_pkey[0];**

**pparent->m\_pkey[0] = pright->m\_pkey[0];**

**pleft->m\_ptr[pleft->m\_nsize+1] = pright->m\_ptr[0];**

**ptemp = pleft->m\_ptr[pleft->m\_nsize+1];**

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

**ptemp->m\_pparent = pleft;**

**for (int i=0; i<ptemp->m\_nsize; i++){**

**if (ptemp->m\_ptr[i]){**

**ptemp->m\_ptr[i]->m\_pparent = ptemp;**

**}**

**}**

**}**

**pright->m\_ptr[0] = pright->m\_ptr[1];**

**pleft->m\_nsize++;**

**PreMove(pright,0);**

**}**

**else {**

**Merge(pleft, pparent, pright, 0);**

**}**

**}**

**template<typename Type> bool BTree<Type>::Remove(const Type item){**

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

**if (!result.m\_ntag){**

**return 0;**

**}**

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

**int n = result.m\_nfind;**

**pdel = result.m\_pfind;**

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

**pmin = pdel->m\_ptr[n+1];**

**pparent = pdel;**

**while (pmin != NULL){**

**pparent = pmin;**

**pmin = pmin->m\_ptr[0];**

**}**

**pdel->m\_pkey[n] = pparent->m\_pkey[0];**

**pdel = pparent;**

**n = 0;**

**}**

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

**int min = (this->m\_nMaxSize + 1) / 2;**

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

**n = 0;**

**pparent = pdel->m\_pparent;**

**if (NULL == pparent)**

**{**

**return 1;**

**}**

**while (n<= pparent->m\_nsize && pparent->m\_ptr[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 == m\_proot){**

**break;**

**}**

**}**

**if (!m\_proot->m\_nsize){ //the root is merged**

**pdel = m\_proot->m\_ptr[0];**

**delete m\_proot;**

**m\_proot = pdel;**

**m\_proot->m\_pparent = NULL;**

**for (int i=0; i<m\_proot->m\_nsize; i++){**

**if (m\_proot->m\_ptr[i]){**

**m\_proot->m\_ptr[i]->m\_pparent = m\_proot;**

**}**

**}**

**}**

**return 1;**

**}**

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

**if (NULL == start){**

**return;**

**}**

**if (start->m\_ptr[0]){**

**Print(start->m\_ptr[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->m\_nsize; i++){ //print the orther child tree**

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

**cout << " ";**

**}**

**cout << start->m\_pkey[i] << "--->" <<endl;**

**if (start->m\_ptr[i+1]){**

**Print(start->m\_ptr[i+1], n+1);**

**}**

**else {**

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

**cout << " ";**

**}**

**cout << "NULL" << endl;**

**}**

**}**

**}**

**template<typename Type> void BTree<Type>::Print(){**

**Print(m\_proot);**

**}**

**template<typename Type> int BTree<Type>::Size(BTreeNode<Type> \*root){**

**if (NULL == root){**

**return 0;**

**}**

**int size=root->m\_nsize;**

**for (int i=0; i<=root->m\_nsize; i++){**

**if (root->m\_ptr[i]){**

**size += this->Size(root->m\_ptr[i]);**

**}**

**}**

**return size;**

**}**

**template<typename Type> int BTree<Type>::Size(){**

**return this->Size(this->m\_proot);**

**}**

**template<typename Type> BTreeNode<Type>\* BTree<Type>::GetParent(const Type item){**

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

**return result.m\_pfind->m\_pparent;**

**}**

### 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 (int i=0; i<49; i++){**

**btree.Remove(init[i]);**

**btree.Print();**

**cout << endl << endl << endl;**

**}**

**return 0;**

**}**

## 17、图

### MinHeap.h

**template<typename Type> class MinHeap{**

**public:**

**MinHeap(Type heap[],int n); //initialize heap by a array**

**~MinHeap(){**

**delete[] m\_pheap;**

**}**

**public:**

**bool Insert(const Type item);**

**bool DeleteMin(Type &first);**

**private:**

**void Adjust(const int start, const int end); //adjust the elements from start to end**

**private:**

**const int m\_nMaxSize;**

**Type \*m\_pheap;**

**int m\_ncurrentsize;**

**};**

**template<typename Type> void MinHeap<Type>::Adjust(const int start, const int end){**

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

**Type temp=m\_pheap[i];**

**while(j <= end){**

**if(j<end && m\_pheap[j]>m\_pheap[j+1]){**

**j++;**

**}**

**if(temp <= m\_pheap[j]){**

**break;**

**}**

**else{**

**m\_pheap[i] = m\_pheap[j];**

**i = j;**

**j = 2\*i+1;**

**}**

**}**

**m\_pheap[i] = temp;**

**}**

**template<typename Type> MinHeap<Type>::MinHeap(Type heap[], int n):m\_nMaxSize(n){**

**m\_pheap = new Type[m\_nMaxSize];**

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

**m\_pheap[i] = heap[i];**

**}**

**m\_ncurrentsize = n;**

**int pos=(n-2)/2; //Find the last tree which has more than one element;**

**while(pos>=0){**

**Adjust(pos, n-1);**

**pos--;**

**}**

**}**

**template<typename Type> bool MinHeap<Type>::DeleteMin(Type &first){**

**first = m\_pheap[0];**

**m\_pheap[0] = m\_pheap[m\_ncurrentsize-1];**

**m\_ncurrentsize--;**

**Adjust(0, m\_ncurrentsize-1);**

**return 1;**

**}**

**template<typename Type> bool MinHeap<Type>::Insert(const Type item){**

**if(m\_ncurrentsize == m\_nMaxSize){**

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

**return 0;**

**}**

**m\_pheap[m\_ncurrentsize] = item;**

**int j = m\_ncurrentsize, i = (j-1)/2;**

**Type temp = m\_pheap[j];**

**while(j > 0){**

**if(m\_pheap[i] <= temp){**

**break;**

**}**

**else{**

**m\_pheap[j] = m\_pheap[i];**

**j = i;**

**i = (j-1)/2;**

**}**

**}**

**m\_pheap[j] = temp;**

**m\_ncurrentsize++;**

**return 1;**

**}**

### Edge.h

**template<typename DistType> struct Edge{**

**public:**

**Edge(int dest, DistType cost): m\_ndest(dest), m\_cost(cost), m\_pnext(NULL){}**

**public:**

**int m\_ndest;**

**DistType m\_cost;**

**Edge<DistType> \*m\_pnext;**

**};**

### Vertex.h

**#include "Edge.h"**

**template<typename NameType, typename DistType> struct Vertex{**

**public:**

**Vertex(): adj(NULL){}**

**NameType m\_data;**

**Edge<DistType> \*adj;**

**~Vertex();**

**};**

**template<typename NameType, typename DistType> Vertex<NameType, DistType>::~Vertex(){**

**Edge<DistType> \*pmove = adj;**

**while (pmove){**

**adj = pmove->m\_pnext;**

**delete pmove;**

**pmove = adj;**

**}**

**}**

### Graph.h

**#include "Vertex.h"**

**template<typename NameType, typename DistType> class Graph{**

**public:**

**Graph(int size = m\_nDefaultSize); //create the Graph with the most vertex of size**

**~Graph();**

**bool GraphEmpty() const{ //Is empty?**

**return 0 == m\_nnumvertex;**

**}**

**bool GraphFull() const{ //Is full?**

**return m\_nMaxNum == m\_nnumvertex;**

**}**

**int NumberOfVertex() const{ //get the number of vertex**

**return m\_nnumvertex;**

**}**

**int NumberOfEdge() const{ //get the number of edge**

**return m\_nnumedges;**

**}**

**NameType GetValue(int v); //get the value of the vth vertex**

**DistType GetWeight(int v1, int v2); //get the weight between v1 and v2**

**int GetFirst(int v); //get the first neighbor vertex of v**

**int GetNext(int v1, int v2);//get the next neighbor vertex of v1 behind v2**

**bool InsertVertex(const NameType vertex); //insert vertex with the name of vertex**

**bool Removevertex(int v); //remove the vth vertex**

**//insert the edge between v1 and v2**

**bool InsertEdge(int v1, int v2, DistType weight=m\_Infinity);**

**bool RemoveEdge(int v1, int v2); //delete the edge between v1 and v2**

**void Print(); //print the graph**

**Edge<DistType> \*GetMin(int v, int \*visited); //get the min weight of the neighbor vertex of v**

**void Prim(Graph<NameType, DistType> &graph); //get the minimize span tree**

**void DFS(int v, int \*visited); //depth first search**

**void DFS();**

**void Dijkstra(int v, DistType \*shotestpath); //get the min weight from v to other vertex**

**private:**

**Vertex<NameType, DistType> \*m\_pnodetable; //neighbor list**

**int m\_nnumvertex;**

**const int m\_nMaxNum;**

**static const int m\_nDefaultSize = 10; //the default maximize vertex**

**static const DistType m\_Infinity = 100000; //there is no edge**

**int m\_nnumedges;**

**int Getvertexpos(const NameType vertex); //get the vertex's position with the name of vertex**

**};**

**template<typename NameType, typename DistType> Graph<NameType, DistType>::Graph(int size)**

**: m\_nnumvertex(0), m\_nMaxNum(size), m\_nnumedges(0){**

**m\_pnodetable = new Vertex<NameType, DistType>[size];**

**}**

**template<typename NameType, typename DistType> Graph<NameType, DistType>::~Graph(){**

**Edge<DistType> \*pmove;**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**pmove = this->m\_pnodetable[i].adj;**

**if (pmove){**

**this->m\_pnodetable[i].adj = pmove->m\_pnext;**

**delete pmove;**

**pmove = this->m\_pnodetable[i].adj;**

**}**

**}**

**delete[] m\_pnodetable;**

**}**

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

**if (v<0 || v>=this->m\_nnumvertex){**

**return -1;**

**}**

**Edge<DistType> \*ptemp = this->m\_pnodetable[v].adj;**

**return m\_pnodetable[v].adj ? m\_pnodetable[v].adj->m\_ndest : -1;**

**}**

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

**if (-1 != v1){**

**Edge<DistType> \*pmove = this->m\_pnodetable[v1].adj;**

**while (NULL != pmove->m\_pnext){**

**if (pmove->m\_ndest==v2){**

**return pmove->m\_pnext->m\_ndest;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**}**

**return -1;**

**}**

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

**if (v<0 || v>=this->m\_nnumvertex){**

**cerr << "The vertex is not exsit" <<endl;**

**exit(1);**

**}**

**return m\_pnodetable[v].m\_data;**

**}**

**template<typename NameType, typename DistType> int Graph<NameType, DistType>::Getvertexpos(const NameType vertex){**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**if (vertex == m\_pnodetable[i].m\_data){**

**return i;**

**}**

**}**

**return -1;**

**}**

**template<typename NameType, typename DistType> DistType Graph<NameType, DistType>::GetWeight(int v1, int v2){**

**if (v1>=0 && v1<this->m\_nnumvertex && v2>=0 && v2<this->m\_nnumvertex){**

**if (v1 == v2){**

**return 0;**

**}**

**Edge<DistType> \*pmove = m\_pnodetable[v1].adj;**

**while (pmove){**

**if (pmove->m\_ndest == v2){**

**return pmove->m\_cost;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**}**

**return m\_Infinity;**

**}**

**template<typename NameType, typename DistType> bool Graph<NameType, DistType>::InsertEdge(int v1, int v2, DistType weight){**

**if (v1>=0 && v1<this->m\_nnumvertex && v2>=0 && v2<this->m\_nnumvertex){**

**Edge<DistType> \*pmove = m\_pnodetable[v1].adj;**

**if (NULL == pmove){ //the first neighbor**

**m\_pnodetable[v1].adj = new Edge<DistType>(v2, weight);**

**return 1;**

**}**

**while (pmove->m\_pnext){**

**if (pmove->m\_ndest == v2){**

**break;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**if (pmove->m\_ndest == v2){ //if the edge is exist, change the weight**

**pmove->m\_cost = weight;**

**return 1;**

**}**

**else{**

**pmove->m\_pnext = new Edge<DistType>(v2, weight);**

**return 1;**

**}**

**}**

**return 0;**

**}**

**template<typename NameType, typename DistType> bool Graph<NameType, DistType>::InsertVertex(const NameType vertex){**

**int i = this->Getvertexpos(vertex);**

**if (-1 != i){**

**this->m\_pnodetable[i].m\_data = vertex;**

**}**

**else{**

**if (!this->GraphFull()){**

**this->m\_pnodetable[this->m\_nnumvertex].m\_data = vertex;**

**this->m\_nnumvertex++;**

**}**

**else{**

**cerr << "The Graph is Full" <<endl;**

**return 0;**

**}**

**}**

**return 1;**

**}**

**template<typename NameType, typename DistType> bool Graph<NameType, DistType>::RemoveEdge(int v1, int v2){**

**if (v1>=0 && v1<this->m\_nnumvertex && v2>=0 && v2<this->m\_nnumvertex){**

**Edge<DistType> \*pmove = this->m\_pnodetable[v1].adj, \*pdel;**

**if (NULL == pmove){**

**cerr << "the edge is not exist!" <<endl;**

**return 0;**

**}**

**if (pmove->m\_ndest == v2){ //the first neighbor**

**this->m\_pnodetable[v1].adj = pmove->m\_pnext;**

**delete pmove;**

**return 1;**

**}**

**while (pmove->m\_pnext){**

**if (pmove->m\_pnext->m\_ndest == v2){**

**pdel = pmove->m\_pnext;**

**pmove->m\_pnext = pdel->m\_pnext;**

**delete pdel;**

**return 1;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**}**

**cerr << "the edge is not exist!" <<endl;**

**return 0;**

**}**

**template<typename NameType, typename DistType> bool Graph<NameType, DistType>::Removevertex(int v){**

**if (v<0 || v>=this->m\_nnumvertex){**

**cerr << "the vertex is not exist!" << endl;**

**return 0;**

**}**

**Edge<DistType> \*pmove, \*pdel;**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**pmove = this->m\_pnodetable[i].adj;**

**if (i != v){ //delete the edge point to v**

**if (NULL == pmove){**

**continue;**

**}**

**if (pmove->m\_ndest == v){**

**this->m\_pnodetable[i].adj = pmove->m\_pnext;**

**delete pmove;**

**continue;**

**}**

**else {**

**if (pmove->m\_ndest > v){ //the vertex more than v subtract 1**

**pmove->m\_ndest--;**

**}**

**}**

**while (pmove->m\_pnext){**

**if (pmove->m\_pnext->m\_ndest == v){**

**pdel = pmove->m\_pnext;**

**pmove->m\_pnext = pdel->m\_pnext;**

**delete pdel;**

**}**

**else {**

**if (pmove->m\_pnext->m\_ndest > v){**

**pmove->m\_pnext->m\_ndest--;**

**pmove = pmove->m\_pnext;**

**}**

**}**

**}**

**}**

**else { //delete the edge point from v**

**while (pmove){**

**this->m\_pnodetable[i].adj = pmove->m\_pnext;**

**delete pmove;**

**pmove = this->m\_pnodetable[i].adj;**

**}**

**}**

**}**

**this->m\_nnumvertex--;**

**for (int i=v; i<this->m\_nnumvertex; i++) //delete the vertex**

**{**

**this->m\_pnodetable[i].adj = this->m\_pnodetable[i+1].adj;**

**this->m\_pnodetable[i].m\_data = this->m\_pnodetable[i+1].m\_data;**

**}**

**this->m\_pnodetable[this->m\_nnumvertex].adj = NULL;**

**return 1;**

**}**

**template<typename NameType, typename DistType> void Graph<NameType, DistType>::Print(){**

**Edge<DistType> \*pmove;**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**cout << this->m\_pnodetable[i].m\_data << "--->";**

**pmove = this->m\_pnodetable[i].adj;**

**while (pmove){**

**cout << pmove->m\_cost << "--->" << this->m\_pnodetable[pmove->m\_ndest].m\_data << "--->";**

**pmove = pmove->m\_pnext;**

**}**

**cout << "NULL" << endl;**

**}**

**}**

**template<typename NameType, typename DistType> void Graph<NameType, DistType>::Prim(Graph<NameType, DistType> &graph){**

**int \*node = new int[this->m\_nnumvertex]; //using for store the vertex visited**

**int \*visited = new int[this->m\_nnumvertex];**

**int count = 0;**

**Edge<DistType> \*ptemp, \*ptemp2 = new Edge<DistType>(0, this->m\_Infinity), \*pmin;**

**int min;**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**graph.InsertVertex(this->m\_pnodetable[i].m\_data);**

**node[i] = 0;**

**visited[i] = 0;**

**}**

**visited[0] = 1;**

**while(++count < this->m\_nnumvertex){**

**pmin = ptemp2;**

**pmin->m\_cost = this->m\_Infinity;**

**//get the minimize weight between the vertex visited and the vertex which is not visited**

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

**ptemp = GetMin(node[i], visited);**

**if (NULL == ptemp){**

**continue;**

**}**

**if (pmin->m\_cost > ptemp->m\_cost){**

**pmin = ptemp;**

**min = node[i];**

**}**

**}**

**node[count] = pmin->m\_ndest;**

**visited[node[count]] = 1;**

**graph.InsertEdge(pmin->m\_ndest, min, pmin->m\_cost);**

**graph.InsertEdge(min, pmin->m\_ndest, pmin->m\_cost);**

**}**

**graph.DFS();**

**delete ptemp2;**

**delete[] node;**

**delete[] visited;**

**}**

**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);**

**}**

**}**

**template<typename NameType, typename DistType> void Graph<NameType, DistType>::DFS(){**

**int \*visited = new int[this->m\_nnumvertex];**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**visited[i] = 0;**

**}**

**cout << "head";**

**DFS(0, visited);**

**cout << "--->end";**

**}**

**template<typename NameType, typename DistType> Edge<DistType>\* Graph<NameType, DistType>::GetMin(int v, int \*visited){**

**Edge<DistType> \*pmove = this->m\_pnodetable[v].adj, \*ptemp = new Edge<DistType>(0, this->m\_Infinity), \*pmin = ptemp;**

**while (pmove){**

**if (!visited[pmove->m\_ndest] && pmin->m\_cost>pmove->m\_cost){**

**pmin = pmove;**

**}**

**pmove = pmove->m\_pnext;**

**}**

**if (pmin == ptemp){**

**delete ptemp;**

**return NULL;**

**}**

**delete ptemp;**

**return pmin;**

**}**

**template<typename NameType, typename DistType> void Graph<NameType, DistType>::Dijkstra(int v, DistType \*shotestpath){**

**int \*visited = new int[this->m\_nnumvertex];**

**int \*node = new int[this->m\_nnumvertex];**

**for (int i=0; i<this->m\_nnumvertex; i++){**

**visited[i] = 0;**

**node[i] = 0;**

**shotestpath[i] = this->GetWeight(v, i);**

**}**

**visited[v] = 1;**

**for (int i=1; i<this->m\_nnumvertex; i++){**

**DistType min = this->m\_Infinity;**

**int u=v;**

**for (int j=0; j<this->m\_nnumvertex; j++){ //get the minimize weight**

**if (!visited[j] && shotestpath[j]<min){**

**min = shotestpath[j];**

**u = j;**

**}**

**}**

**visited[u] = 1;**

**for (int w=0; w<this->m\_nnumvertex; w++){ //change the weight from v to other vertex**

**DistType weight = this->GetWeight(u, w);**

**if (!visited[w] && weight!=this->m\_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 (int 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 (int 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;**

**}**

## 18、排序

### Data.h

**template<typename Type> class Element{**

**public:**

**Type GetKey(){**

**return key;**

**}**

**void SetKey(Type item){**

**key = item;**

**}**

**public:**

**Element<Type>& operator =(Element<Type> copy){**

**key = copy.key;**

**return \*this;**

**}**

**bool operator ==(Element<Type> item){**

**return this->key == item.key;**

**}**

**bool operator !=(Element<Type> item){**

**return this->key != item.key;**

**}**

**bool operator <(Element<Type> item){**

**return this->key < item.key;**

**}**

**bool operator >(Element<Type> item){**

**return this->key > item.key;**

**}**

**bool operator >=(Element<Type> item){**

**return this->key >= item.key;**

**}**

**bool operator <=(Element<Type> item){**

**return this->key <= item.key;**

**}**

**private:**

**Type key;**

**};**

**template<typename Type> class Sort;**

**template<typename Type> class DataList{**

**public:**

**friend class Sort<Type>;**

**DataList(int size=m\_nDefaultSize): m\_nMaxSize(size), m\_ncurrentsize(0){**

**m\_pvector = new Element<Type>[size];**

**}**

**DataList(Type \*data, int size);**

**bool Insert(Type item);**

**~DataList(){**

**delete[] m\_pvector;**

**}**

**int Size(){**

**return this->m\_ncurrentsize;**

**}**

**void Swap(Element<Type> &left, Element<Type> &right){**

**Element<Type> temp = left;**

**left = right;**

**right = temp;**

**}**

**void Print();**

**private:**

**static const int m\_nDefaultSize = 10;**

**Element<Type> \*m\_pvector;**

**const int m\_nMaxSize;**

**int m\_ncurrentsize;**

**};**

**template<typename Type> DataList<Type>::DataList(Type \*data, int size)**

**: m\_nMaxSize(size > m\_nDefaultSize ? size : m\_nDefaultSize), m\_ncurrentsize(0){**

**this->m\_pvector = new Element<Type>[size];**

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

**this->m\_pvector[i].SetKey(data[i]);**

**}**

**this->m\_ncurrentsize += size;**

**}**

**template<typename Type> bool DataList<Type>::Insert(Type item){**

**if (this->m\_ncurrentsize == this->m\_nMaxSize){**

**cerr << "The list is full!" <<endl;**

**return 0;**

**}**

**this->m\_pvector[this->m\_ncurrentsize++].SetKey(item);**

**}**

**template<typename Type> void DataList<Type>::Print(){**

**cout << "The list is:";**

**for (int i=0; i<this->m\_ncurrentsize; i++){**

**cout << " " << this->m\_pvector[i].GetKey();**

**}**

**}**

### QueueNode.h

**#include "QueueNode.h"**

**template<typename Type> class LinkQueue{**

**public:**

**LinkQueue():m\_prear(NULL),m\_pfront(NULL){}**

**~LinkQueue(){**

**MakeEmpty();**

**}**

**void Append(const Type item);**

**Type Delete();**

**Type GetFront();**

**void MakeEmpty();**

**bool IsEmpty() const{**

**return m\_pfront==NULL;**

**}**

**void Print();**

**private:**

**QueueNode<Type> \*m\_prear,\*m\_pfront;**

**};**

**template<typename Type> void LinkQueue<Type>::MakeEmpty(){**

**QueueNode<Type> \*pdel;**

**while(m\_pfront){**

**pdel=m\_pfront;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**}**

**}**

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

**if(m\_pfront==NULL){**

**m\_pfront=m\_prear=new QueueNode<Type>(item);**

**}**

**else{**

**m\_prear=m\_prear->m\_pnext=new QueueNode<Type>(item);**

**}**

**}**

**template<typename Type> Type LinkQueue<Type>::Delete(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**QueueNode<Type> \*pdel=m\_pfront;**

**Type temp=m\_pfront->m\_data;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type LinkQueue<Type>::GetFront(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pfront->m\_data;**

**}**

**template<typename Type> void LinkQueue<Type>::Print(){**

**QueueNode<Type> \*pmove=m\_pfront;**

**cout<<"front";**

**while(pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->rear"<<endl<<endl<<endl;**

**}**

### LinkQueue.h

**#include "QueueNode.h"**

**template<typename Type> class LinkQueue{**

**public:**

**LinkQueue():m\_prear(NULL),m\_pfront(NULL){}**

**~LinkQueue(){**

**MakeEmpty();**

**}**

**void Append(const Type item);**

**Type Delete();**

**Type GetFront();**

**void MakeEmpty();**

**bool IsEmpty() const{**

**return m\_pfront==NULL;**

**}**

**void Print();**

**private:**

**QueueNode<Type> \*m\_prear,\*m\_pfront;**

**};**

**template<typename Type> void LinkQueue<Type>::MakeEmpty(){**

**QueueNode<Type> \*pdel;**

**while(m\_pfront){**

**pdel=m\_pfront;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**}**

**}**

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

**if(m\_pfront==NULL){**

**m\_pfront=m\_prear=new QueueNode<Type>(item);**

**}**

**else{**

**m\_prear=m\_prear->m\_pnext=new QueueNode<Type>(item);**

**}**

**}**

**template<typename Type> Type LinkQueue<Type>::Delete(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**QueueNode<Type> \*pdel=m\_pfront;**

**Type temp=m\_pfront->m\_data;**

**m\_pfront=m\_pfront->m\_pnext;**

**delete pdel;**

**return temp;**

**}**

**template<typename Type> Type LinkQueue<Type>::GetFront(){**

**if(IsEmpty()){**

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

**exit(1);**

**}**

**return m\_pfront->m\_data;**

**}**

**template<typename Type> void LinkQueue<Type>::Print(){**

**QueueNode<Type> \*pmove=m\_pfront;**

**cout<<"front";**

**while(pmove){**

**cout<<"--->"<<pmove->m\_data;**

**pmove=pmove->m\_pnext;**

**}**

**cout<<"--->rear"<<endl<<endl<<endl;**

**}**

### Sort.h

**#include "Data.h"**

**#include "LinkQueue.h"**

**template<typename Type> class Sort{**

**public:**

**void InsertSort(DataList<Type> &list, int n=-1);**

**void BinaryInsertSort(DataList<Type> &list, int n=-1);**

**void ShellSort(DataList<Type> &list, const int gap=-1);**

**void BubbleSort(DataList<Type> &list);**

**void QuickSort(DataList<Type> &list, int left=0, int right=-3);**

**void SelectSort(DataList<Type> &list);**

**void HeapSort(DataList<Type> &list);**

**void MergeSort(DataList<Type> &list);**

**void RadixSort(DataList<int> &list, int m, int d); //just use for integer!**

**private:**

**void BubbleSwap(DataList<Type> &list, const int n, int &flag);**

**void SelectChange(DataList<Type> &list, const int n);**

**void HeapAdjust(DataList<Type> &list, const int start, const int end);**

**void Merge(DataList<Type> &list, DataList<Type> &mergedlist, const int len);**

**void MergeDouble(DataList<Type> &list, DataList<Type> &mergedlist, const int start, const int part, const int end);**

**};**

**template<typename Type> void Sort<Type>::InsertSort(DataList<Type> &list, int n){**

**if (-1 == n){**

**for (int i=1; i<list.m\_ncurrentsize; i++){**

**InsertSort(list, i);**

**}**

**return;**

**}**

**Element<Type> temp = list.m\_pvector[n];**

**int i;**

**for (i=n; i>0; i--){**

**if (temp > list.m\_pvector[i-1]){**

**break;**

**}**

**else{**

**list.m\_pvector[i] = list.m\_pvector[i-1];**

**}**

**}**

**list.m\_pvector[i] = temp;**

**}**

**template<typename Type> void Sort<Type>::BinaryInsertSort(DataList<Type> &list, int n){**

**if (-1 == n){**

**for (int i=1; i<list.m\_ncurrentsize; i++){**

**BinaryInsertSort(list, i);**

**}**

**return;**

**}**

**Element<Type> temp = list.m\_pvector[n];**

**int left = 0, right = n-1;**

**while(left <= right){**

**int middle = (left + right) / 2;**

**if (temp < list.m\_pvector[middle]){**

**right = middle - 1;**

**}**

**else {**

**left = middle + 1;**

**}**

**}**

**for (int i=n-1; i>=left; i--){**

**list.m\_pvector[i+1] = list.m\_pvector[i];**

**}**

**list.m\_pvector[left] = temp;**

**}**

**template<typename Type> void Sort<Type>::ShellSort(DataList<Type> &list, const int gap){**

**if (-1 == gap){**

**int gap = list.m\_ncurrentsize / 2;**

**while (gap){**

**ShellSort(list, gap);**

**gap = (int)(gap / 2);**

**}**

**return;**

**}**

**for (int i=gap; i<list.m\_ncurrentsize; i++){**

**InsertSort(list, i);**

**}**

**}**

**template<typename Type> void Sort<Type>::BubbleSwap(DataList<Type> &list, const int n, int &flag){**

**flag = 0;**

**for (int i=list.m\_ncurrentsize-1; i>=n; i--){**

**if (list.m\_pvector[i-1] > list.m\_pvector[i]){**

**list.Swap(list.m\_pvector[i-1], list.m\_pvector[i]);**

**flag = 1;**

**}**

**}**

**}**

**template<typename Type> void Sort<Type>::BubbleSort(DataList<Type> &list){**

**int flag = 1, n = 0;**

**while (++n<list.m\_ncurrentsize && flag){**

**BubbleSwap(list, n, flag);**

**}**

**}**

**template<typename Type> void Sort<Type>::QuickSort(DataList<Type> &list, int left=0, int right=-1){**

**if (-3 == right){**

**right = list.m\_ncurrentsize - 1;**

**}**

**if (left < right){**

**int pivotpos = left;**

**Element<Type> pivot = list.m\_pvector[left];**

**for (int i=left+1; i<=right; i++){**

**if (list.m\_pvector[i]<pivot && ++pivotpos!=i){**

**list.Swap(list.m\_pvector[pivotpos], list.m\_pvector[i]);**

**}**

**list.Swap(list.m\_pvector[left], list.m\_pvector[pivotpos]);**

**}**

**QuickSort(list, left, pivotpos-1);**

**QuickSort(list, pivotpos+1, right);**

**}**

**}**

**template<typename Type> void Sort<Type>::SelectChange(DataList<Type> &list, const int n){**

**int j = n;**

**for (int i=n+1; i<list.m\_ncurrentsize; i++){**

**if (list.m\_pvector[i] < list.m\_pvector[j]){**

**j = i;**

**}**

**}**

**if (j != n){**

**list.Swap(list.m\_pvector[n], list.m\_pvector[j]);**

**}**

**}**

**template<typename Type> void Sort<Type>::SelectSort(DataList<Type> &list){**

**for (int i=0; i<list.m\_ncurrentsize-1; i++){**

**SelectChange(list, i);**

**}**

**}**

**template<typename Type> void Sort<Type>::HeapAdjust(DataList<Type> &list, const int start, const int end){**

**int current = start, child = 2 \* current + 1;**

**Element<Type> temp = list.m\_pvector[start];**

**while (child <= end){**

**if (child<end && list.m\_pvector[child]<list.m\_pvector[child+1]){**

**child++;**

**}**

**if (temp >= list.m\_pvector[child]){**

**break;**

**}**

**else {**

**list.m\_pvector[current] = list.m\_pvector[child];**

**current = child;**

**child = 2 \* current + 1;**

**}**

**}**

**list.m\_pvector[current] = temp;**

**}**

**template<typename Type> void Sort<Type>::HeapSort(DataList<Type> &list){**

**for (int i=(list.m\_ncurrentsize-2)/2; i>=0; i--){**

**HeapAdjust(list, i, list.m\_ncurrentsize-1);**

**}**

**for (int i=list.m\_ncurrentsize-1; i>=1; i--){**

**list.Swap(list.m\_pvector[0], list.m\_pvector[i]);**

**HeapAdjust(list, 0, i-1);**

**}**

**}**

**template<typename Type> void Sort<Type>::MergeDouble(DataList<Type> &list, DataList<Type> &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.m\_pvector[i] <= list.m\_pvector[j]){**

**mergedlist.m\_pvector[k++] = list.m\_pvector[i++];**

**}**

**else {**

**mergedlist.m\_pvector[k++] = list.m\_pvector[j++];**

**}**

**}**

**if (i <= part){**

**for (int m=i; m<=part && k<=end;){**

**mergedlist.m\_pvector[k++] = list.m\_pvector[m++];**

**}**

**}**

**else {**

**for (int m=j; m<=end && k<=end; m++){**

**mergedlist.m\_pvector[k++] = list.m\_pvector[m];**

**}**

**}**

**}**

**template<typename Type> void Sort<Type>::Merge(DataList<Type> &list, DataList<Type> &mergedlist, const int len){**

**int n = 0;**

**while (n+2\*len < list.m\_ncurrentsize){**

**MergeDouble(list, mergedlist, n, n+len-1, n+2\*len-1);**

**n += 2\*len;**

**}**

**if (n+len < list.m\_ncurrentsize){**

**MergeDouble(list, mergedlist, n, n+len-1, list.m\_ncurrentsize-1);**

**}**

**else {**

**for (int i=n; i<list.m\_ncurrentsize; i++){**

**mergedlist.m\_pvector[i] = list.m\_pvector[i];**

**}**

**}**

**}**

**template<typename Type> void Sort<Type>::MergeSort(DataList<Type> &list){**

**DataList<Type> temp(list.m\_nMaxSize);**

**temp.m\_ncurrentsize = list.m\_ncurrentsize;**

**int len = 1;**

**while (len < list.m\_ncurrentsize){**

**Merge(list, temp, len);**

**len \*= 2;**

**Merge(temp, list, len);**

**len \*= 2;**

**}**

**}**

**template<typename Type> void Sort<Type>::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.m\_ncurrentsize; j++){**

**int k = (list.m\_pvector[j].GetKey() / power) % d;**

**queue[k].Append(list.m\_pvector[j].GetKey());**

**}**

**for (int j=0,k=0; j<d; j++){**

**while (!queue[j].IsEmpty()){**

**list.m\_pvector[k++].SetKey(queue[j].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;**

**}**