目　录

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;

}
```

2、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;
}
```

3、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;
}
```

4、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;
```

```
}
```

5、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;

}
```

6、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;

}
```