1. 顺序表的验证

(1)输出最大最小数位置

在class SeqList中添加

{

Public:

Status Max(ElemType &e,int imax);

Status Min(ElemType &e,int imin);

}

函数成员：

template <class ElemType>

Status SeqList<ElemType>::Max(ElemType &e,int imax)

{

e=elems[0];

imax=1;

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

{

if(elems[i]>e)

{

e=elems[i];

imax=i+1;

}

}

return SUCCESS;

}

template <class ElemType>

Status SeqList<ElemType>::Min(ElemType &e,int imin)

{

e=elems[0];

imin=1;

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

{

if(elems[i]<e)

{

e=elems[i];

imin=i+1;

}

}

return SUCCESS;

}

(2)变成有序表

在class SeqList中添加

{

Public:

Status Sort();

}

函数成员：

template <class ElemType>

Status SeqList<ElemType>::Sort()

{

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

{

for(int j=0;j<length-1;j++)

{

ElemType n;

if(elems[j]>elems[j+1])

{

n=elems[j];

elems[j]=elems[j+1];

elems[j+1]=n;

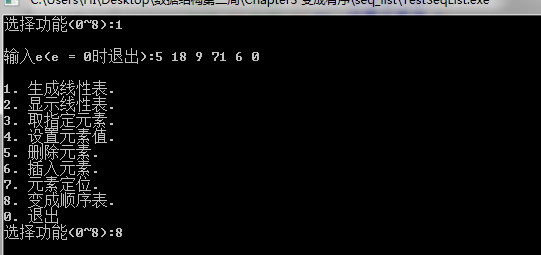
}

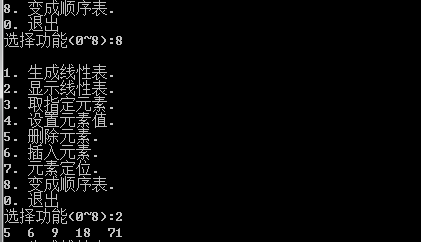
}

}

return SUCCESS;

}





1. 单链表的验证

（1）node改为student

**首先修改node类，改为student类，具体代码如下，为了方便修改程序，只在student下设定了学生学号和班级两个数据成员**

class student

{

public:

// 数据成员:

int id,classe; // 数据域

student \*next; // 指针域

// 构造函数:

student(); // 无参数的构造函数

student(int i, int c,student \*link = NULL); // 已知数数据元素值和指针建立结构

};

// 结点类的实现部分

student::student()

// 操作结果：构造指针域为空的结点

{

next = NULL;

}

student::student(int i, int c, student \*link)

// 操作结果：构造一个数据域为e和指针域为link的结点

{

id = i;

classe=c;

next = link;

}

**其次应当修改LinkList类下的各项成员以及成员函数，以满足适应student类，具体代码如下，主要修改点为：改变各个函数的参数个数，分别对应为student的id和classe这两个成员（原node类只有一个数据成员data），再修改各个函数**

class LinkList

{

public:

// 单链表的数据成员

student \*head; // 头结点指针

int length; // 单链表长度

public:

// 单链表的函数成员

LinkList(); // 无参数的构造函数

LinkList(student v[], int n); // 有参数的构造函数

virtual ~LinkList(); // 析构函数

int GetLength() const; // 求单链表长度

bool IsEmpty() const; // 判断单链表是否为空

void Clear(); // 将单链表清空

//void Traverse(void (\*Visit)(const student &)) const;// 遍历单链表

int LocateElem(const int &e) const; // 元素定位

Status GetElem(int position, int e,int c) const; // 求指定位置的元素

Status SetElem(int position, const int &e); // 设置指定位置的元素值

Status DeleteElem(int position, int &e); // 删除元素

Status InsertElem(int position, const int i,const int c); // 在制定位置插入元素

Status InsertElem(const int &i,const int &c); // 在表尾插入元素

LinkList(const LinkList &la); // 复制构造函数

LinkList &operator =(const LinkList &la); // 重载赋值运算

};

// 单链表类的实现部分

LinkList::LinkList()

// 操作结果：构造一个空链表

{

head = new student; // 构造头结点

assert(head); // 构造头结点失败，终止程序运行

length = 0; // 初始化单链表长度为0

}

LinkList::LinkList(student v[], int n)

// 操作结果：根据数组v中的元素构造单链表

{

student \*p;

p = head = new student; // 构造头结点

assert(head != 0); // 构造头结点失败，终止程序运行

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

p->next = new student(v[i].id,v[i].classe,NULL);

assert(p->next); // 构造元素结点失败，终止程序运行

p = p->next;

}

length = n; // 初始化单链表长度为n

}

LinkList::~LinkList()

// 操作结果：销毁单链表

{

Clear(); // 清空单链表

delete head; // 释放头结点所指空间

}

int LinkList::GetLength() const

// 操作结果：返回单链表的长度

{

return length;

}

bool LinkList::IsEmpty() const

// 操作结果：如单链表为空，则返回true，否则返回false

{

return head->next == NULL;

}

void LinkList::Clear()

// 操作结果：清空单链表,删除单链表中所有元素结点

{

student \*p = head->next;

while (p != NULL) {

head->next = p->next;

delete p;

p = head->next;

}

length = 0;

}

void LinkList::Traverse(void (\*Visit)(const int i)) const

// 操作结果：依次对单链表的每个元素调用函数(\*visit)访问

{

student \*p = head->next;

while (p != NULL) {

(\*Visit)(p->id); // 对单链表中每个元素调用函数(\*visit)访问

p = p->next;

}

int LinkList::LocateElem(const int &e) const

// 元素定位

{

student \*p = head->next;

int count = 1;

while (p != NULL && p->id != e) {

count++;

p = p->next;

}

return (p != NULL) ? count : 0;

}

Status LinkList::GetElem(int i, int e,int c) const

// 操作结果：当单链表存在第i个元素时，用e返回其值，函数返回ENTRY\_FOUND,

// 否则函数返回NOT\_PRESENT

{

if (i < 1 || i > length)

return RANGE\_ERROR;

else {

student \*p = head->next;

int count;

for (count = 1; count < i; count++)

p = p->next; // p指向第i个结点

e = p->id;

c= p->classe; // 用e返回第i个元素的值

return ENTRY\_FOUND;

}

}

Status LinkList::SetElem(int i, const int &e)

// 操作结果：将单链表的第i个位置的元素赋值为e,

// i的取值范围为1≤i≤length,

// i合法时函数返回SUCCESS,否则函数返回RANGE\_ERROR

{

if (i < 1 || i > length)

return RANGE\_ERROR;

else {

student \*p = head->next;

int count;

for (count = 1; count < i; count++)

p = p->next; // 取出指向第i个结点的指针

p->id = e; // 修改第i个元素的值为e

return SUCCESS;

}

}

Status LinkList::DeleteElem(int i, int &e)

// 操作结果：删除单链表的第i个位置的元素, 并用e返回其值,

// i的取值范围为1≤i≤length,

// i合法时函数返回SUCCESS,否则函数返回RANGE\_ERROR

{

if (i < 1 || i > length)

return RANGE\_ERROR; // i范围错

else {

student \*p = head, \*q;

int count;

for (count = 1; count < i; count++)

p = p->next; // p指向第i-1个结点

q = p->next; // q指向第i个结点

p->next = q->next; // 删除结点

e = q->id; // 用e返回被删结点元素值

length--; // 删除成功后元素个数减1

delete q; // 释放被删结点

return SUCCESS;

}

}

Status LinkList::InsertElem(int i, const int id,const int c)

// 操作结果：在单链表的第i个位置前插入元素e

// i的取值范围为1≤i≤length+1

// i合法时返回SUCCESS, 否则函数返回RANGE\_ERROR

{

if (i < 1 || i > length+1)

return RANGE\_ERROR;

else {

student \*p = head, \*q;

int count;

for (count = 1; count < i; count++)

p = p->next; // p指向第i-1个结点

q = new student(id,c,p->next); // 生成新结点q

assert(q); // 申请结点失败，终止程序运行

p->next = q; // 将q插入到链表中

length++; // 插入成功后，单链表长度加1

return SUCCESS;

}

}

Status LinkList::InsertElem(const int &i,const int &c)

// 操作结果：在单链表的表尾位置插入元素e

{

student \*p, \*q;

q = new student(i,c, NULL); // 生成新结点q

assert(q); // 申请结点失败，终止程序运行

for (p = head; p->next != NULL; p = p->next) ; // p指向表尾结点

p->next = q; // 在单链表的表尾位置插入新结点

length++; // 插入成功后，单链表长度加1

return SUCCESS;

}

LinkList::LinkList(const LinkList &la)

// 操作结果：复制构造函数，由单链表la构造新单链表

{

int laLength = la.GetLength(); // 取被复制单链表的长度

int e;

int c;

head = new student; // 构造头指针

assert(head); // 构造头指针失败，终止程序运行

length = 0; // 初始化元素个数

for (int i = 1; i <= laLength; i++) { // 复制数据元素

la.GetElem(i, e,c); // 取出第i个元素的值放在e中

InsertElem(e,c); // 将e插入到当前单链表的表尾

}

}

LinkList &LinkList::operator =(const LinkList &la)

// 操作结果：重载赋值运算符，将单链表la赋值给当前单链表

{

if (&la != this) {

int laLength = la.GetLength();// 取被赋值单链表的长度

int e;

int c;

Clear(); // 清空当前单链表

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

la.GetElem(i,e,c); // 取出第i个元素的值放在e中

InsertElem(e,c); // 将e插入到当前单链表的表尾

}

}

return \*this;

}

（2）逆序

在LinkList中添加

void reversal();

并在类外定义

template <class ElemType>

void LinkList<ElemType>::reversal()

{

Node<ElemType>\* head2=head->next;

int count=length;

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

{

Node<ElemType>\*p=head2;

for(int j=1;j<count-1;j++)

{

p=p->next;

}

if(count==length) head->next=p->next;

p->next->next=p;

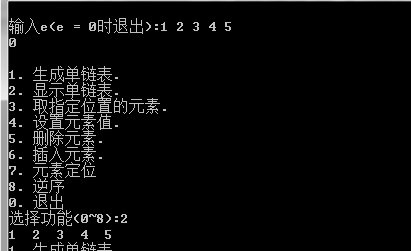
count--;

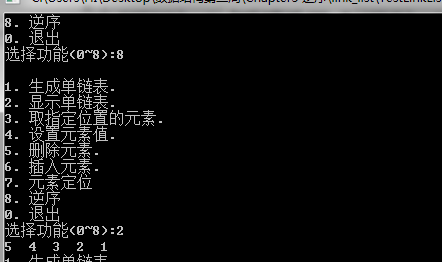
}

Node<ElemType>\* p=head2;

p->next=NULL;

}





1. 不带头结点的单循环列表

修改如下

template <class ElemType>

class LinkList

{

protected:

// 单链表的数据成员

Node<ElemType> \*head=NULL; // 头结点指针

int length; // 单链表长度

public:

// 单链表的函数成员

LinkList(); // 无参数的构造函数

LinkList(ElemType v[], int n); // 有参数的构造函数

virtual ~LinkList(); // 析构函数

int GetLength() const; // 求单链表长度

bool IsEmpty() const; // 判断单链表是否为空

void Clear(); // 将单链表清空

void Traverse(void (\*Visit)(const ElemType &)) const;// 遍历单链表

int LocateElem(const ElemType &e) const; // 元素定位

Status GetElem(int position, ElemType &e) const; // 求指定位置的元素

Status SetElem(int position, const ElemType &e); // 设置指定位置的元素值

Status DeleteElem(int position, ElemType &e); // 删除元素

Status InsertElem(int position, const ElemType &e); // 在制定位置插入元素

Status InsertElem(const ElemType &e); // 在表尾插入元素

LinkList(const LinkList<ElemType> &la); // 复制构造函数

LinkList<ElemType> &operator =(const LinkList<ElemType> &la); // 重载赋值运算

};

// 单链表类的实现部分

template <class ElemType>

LinkList<ElemType>::LinkList()

// 操作结果：构造一个空链表

{

length = 0; // 初始化单链表长度为0

}

template <class ElemType>

LinkList<ElemType>::LinkList(ElemType v[], int n)

// 操作结果：根据数组v中的元素构造单链表

{

Node<ElemType> \*p;

assert(head != 0); // 构造头结点失败，终止程序运行

head=p=new Node<ElemType>(v[0],NULL);

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

p->next = new Node<ElemType>(v[i],NULL);

assert(p->next); // 构造元素结点失败，终止程序运行

p = p->next;

}

length = n; // 初始化单链表长度为n

}

template <class ElemType>

LinkList<ElemType>::~LinkList()

// 操作结果：销毁单链表

{

Clear(); // 清空单链表

}

template <class ElemType>

int LinkList<ElemType>::GetLength() const

// 操作结果：返回单链表的长度

{

return length;

}

template <class ElemType>

bool LinkList<ElemType>::IsEmpty() const

// 操作结果：如单链表为空，则返回true，否则返回false

{

return head == NULL;

}

template <class ElemType>

void LinkList<ElemType>::Clear()

// 操作结果：清空单链表,删除单链表中所有元素结点

{

Node<ElemType> \*p = head;

if (p != NULL) {

head = p->next;

delete p;

p = head;

}

else {delete head;}

length = 0;

}

template <class ElemType>

void LinkList<ElemType>::Traverse(void (\*Visit)(const ElemType &)) const

// 操作结果：依次对单链表的每个元素调用函数(\*visit)访问

{

Node<ElemType> \*p = head;

while (p != NULL) {

(\*Visit)(p->data); // 对单链表中每个元素调用函数(\*visit)访问

p = p->next;

}

}

template <class ElemType>

int LinkList<ElemType>::LocateElem(const ElemType &e) const

// 元素定位

{

Node<ElemType> \*p = head;

int count = 1;

while (p != NULL && p->data != e) {

count++;

p = p->next;

}

return (p != NULL) ? count : 0;

}

template <class ElemType>

Status LinkList<ElemType>::GetElem(int i, ElemType &e) const

// 操作结果：当单链表存在第i个元素时，用e返回其值，函数返回ENTRY\_FOUND,

// 否则函数返回NOT\_PRESENT

{

if (i < 1 || i > length)

return RANGE\_ERROR;

else {

Node<ElemType> \*p = head;

int count;

for (count = 1; count < i; count++)

p = p->next; // p指向第i个结点

e = p->data; // 用e返回第i个元素的值

return ENTRY\_FOUND;

}

}

template <class ElemType>

Status LinkList<ElemType>::SetElem(int i, const ElemType &e)

// 操作结果：将单链表的第i个位置的元素赋值为e,

// i的取值范围为1≤i≤length,

// i合法时函数返回SUCCESS,否则函数返回RANGE\_ERROR

{

if (i < 1 || i > length)

return RANGE\_ERROR;

else {

Node<ElemType> \*p = head;

int count;

for (count = 1; count < i; count++)

p = p->next; // 取出指向第i个结点的指针

p->data = e; // 修改第i个元素的值为e

return SUCCESS;

}

}

template <class ElemType>

Status LinkList<ElemType>::DeleteElem(int i, ElemType &e)

// 操作结果：删除单链表的第i个位置的元素, 并用e返回其值,

// i的取值范围为1≤i≤length,

// i合法时函数返回SUCCESS,否则函数返回RANGE\_ERROR

{

if (i < 1 || i > length)

return RANGE\_ERROR; // i范围错

else {

Node<ElemType> \*p = head, \*q;

int count;

if(i==1){head=head->next;delete p;}

else

{

for (count = 1; count < i-1; count++) p = p->next; // p指向第i-1个结点

q = p->next; // q指向第i个结点

p->next = q->next; // 删除结点

e = q->data; // 用e返回被删结点元素值

length--; // 删除成功后元素个数减1

delete q; // 释放被删结点

return SUCCESS;

}

}

}

template <class ElemType>

Status LinkList<ElemType>::InsertElem(int i, const ElemType &e)

// 操作结果：在单链表的第i个位置前插入元素e

// i的取值范围为1≤i≤length+1

// i合法时返回SUCCESS, 否则函数返回RANGE\_ERROR

{

if (i < 1 || i > length+1)

return RANGE\_ERROR;

else {

Node<ElemType> \*p = head, \*q;

int count;

if(i=1){q = new Node<ElemType>(e,p->next);head=q;}

else

{

for (count = 1; count < i-1; count++)

p = p->next; // p指向第i-1个结点

q = new Node<ElemType>(e, p->next); // 生成新结点q

assert(q); // 申请结点失败，终止程序运行

p->next = q; // 将q插入到链表中

length++; // 插入成功后，单链表长度加1

return SUCCESS;

}

}

}

template <class ElemType>

Status LinkList<ElemType>::InsertElem(const ElemType &e)

// 操作结果：在单链表的表尾位置插入元素e

{

Node<ElemType> \*p, \*q;

q = new Node<ElemType>(e, NULL); // 生成新结点q

assert(q); // 申请结点失败，终止程序运行

for (p = head; p->next != NULL; p = p->next) ; // p指向表尾结点

p->next = q; // 在单链表的表尾位置插入新结点

length++; // 插入成功后，单链表长度加1

return SUCCESS;

}

template <class ElemType>

LinkList<ElemType>::LinkList(const LinkList<ElemType> &la)

// 操作结果：复制构造函数，由单链表la构造新单链表

{

int laLength = la.GetLength(); // 取被复制单链表的长度

ElemType e;// 构造头指针失败，终止程序运行

length = 0; // 初始化元素个数

for (int i = 1; i <= laLength; i++) { // 复制数据元素

la.GetElem(i, e); // 取出第i个元素的值放在e中

InsertElem(e); // 将e插入到当前单链表的表尾

}

}

template <class ElemType>

LinkList<ElemType> &LinkList<ElemType>::operator =(const LinkList<ElemType> &la)

// 操作结果：重载赋值运算符，将单链表la赋值给当前单链表

{

if (&la != this) {

int laLength = la.GetLength();// 取被赋值单链表的长度

ElemType e;

Clear(); // 清空当前单链表

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

la.GetElem(i, e); // 取出第i个元素的值放在e中

InsertElem(e); // 将e插入到当前单链表的表尾

}

}

return \*this;

}