计算机学院 高级程序语言设计 课程实验报告

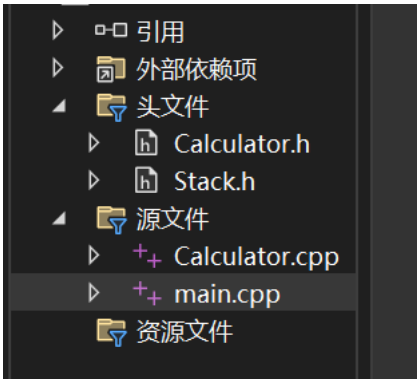
实验题目：线性群体类和算法		学号：202300130150
日期：2024.4.30	班级： 23.4	姓名： 王成意

实验目的：
了解并掌握线性群体类和算法。

实验步骤与内容：

1. 实践第 9 章 PPT，例 9-8,9-9，栈实现计算器。

框架：



具体实现：

1.

```
#ifndef CALCULATOR_H
#define CALCULATOR_H
#include "Stack.h"    // 包含栈类模板定义文件

class Calculator {    //计算器类
private:
    Stack<double> s;  // 操作数栈
    void enter(double num);    //将操作数num压入栈

    //连续将两个操作数弹出栈，放在opnd1和opnd2中
    bool getTwoOperands(double& opnd1, double& opnd2);

    void compute(char op);//执行由操作符op指定的运算
public:
    void run();       //运行计算器程序
    void clear();//清空操作数栈
};
#endif //CALCULATOR_H
```

2.

```
#pragma once
```

```
// Stack.h
#ifndef STACK_H
#define STACK_H
#include <cassert>

template <class T, int SIZE = 50>
class Stack
{
private:
    T list[SIZE];
    int top;

public:
    Stack();
    void push(const T& item);
    T pop();
    void clear();
    const T& peek() const;
    bool isEmpty() const;
    bool isFull() const;
};
// 模板的实现
template <class T, int SIZE>
Stack<T, SIZE>::Stack() : top(-1) {}
template <class T, int SIZE>
void Stack<T, SIZE>::push(const T& item)
{
    assert(!isFull());
    list[++top] = item;
}
template <class T, int SIZE>
T Stack<T, SIZE>::pop()
{
    assert(!isEmpty());
    return list[top--];
}
template <class T, int SIZE>
const T& Stack<T, SIZE>::peek() const
{
    assert(!isEmpty());
    return list[top]; // 返回栈顶元素
}
template <class T, int SIZE>
bool Stack<T, SIZE>::isEmpty() const
```

```
{
    return top == -1;
}
template <class T, int SIZE>
bool Stack<T, SIZE>::isFull() const
{
    return top == SIZE - 1;
}
template <class T, int SIZE>
void Stack<T, SIZE>::clear()
{
    top = -1;
}

#endif // STACK_H
```

3.
```
// Calculator.cpp
#include "Calculator.h"
#include <iostream>
#include <sstream>
#include <cmath>
using namespace std;

// 工具函数，用于将字符串转换为实数
inline double stringToDouble(const string& str)
{
    istringstream stream(str); // 字符串输入流
    double result;
    stream >> result;
    return result;
}

void Calculator::enter(double num)
{ // 将操作数num压入栈
    s.push(num);
}
bool Calculator::getTwoOperands(double& opnd1, double& opnd2)
{
    if (s.isEmpty())
    { // 检查栈是否空
        cerr << "Missing right operand!" << endl;
        return false;
    }
    opnd1 = s.pop(); // 将右操作数弹出栈
```

```
    if (s.isEmpty())
    { // 检查栈是否空
        cerr << "Missing left operand!" << endl;
        return false;
    }
    opnd2 = s.pop(); // 将左操作数弹出栈

    return true;
}
void Calculator::compute(char op)
{ // 执行运算
    double operand1, operand2;
    bool result = getTwoOperands(operand1, operand2);
    if (result)
    { // 如果成功，执行运算并将运算结果压入栈
        switch (op)
        {
        case '+':
            s.push(operand2 + operand1);
            break;
        case '-':
            s.push(operand2 - operand1);
            break;
        case '*':
            s.push(operand2 * operand1);
            break;
        case '/':
            if (operand1 == 0)
            { // 检查除数是否为0
                cerr << "Divided by 0!" << endl;
                s.clear(); // 除数为0时清空栈
            }
            else
                s.push(operand2 / operand1);
            break;
        case '^':
            s.push(pow(operand2, operand1));
            break;
        default:
            cerr << "Unrecognized operator!" << endl;
            break;
        }
        cout << "= " << s.peek() << " "; // 输出本次运算结果
    }
```

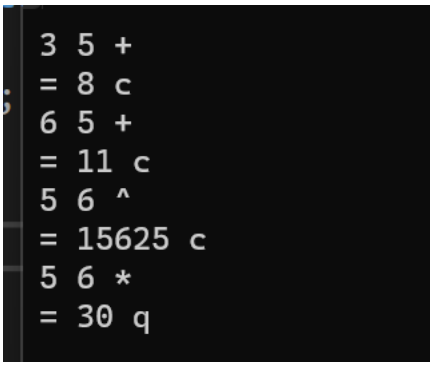
```
    else
        s.clear(); // 操作数不够，清空栈
}
void Calculator::run()
{ // 读入并处理后缀表达式
    string str;
    while (cin >> str, str != "q")
    {
        switch (str[0])
        {
        case 'c':
            s.clear();
            break;
        case '-':                // 遇'-'需判断是减号还是负号
            if (str.size() > 1) // 不是减号
                enter(stringToDouble(str));
            else
                compute(str[0]);
            break;
        case '+': // 遇到其它操作符时
        case '*':
        case '/':
        case '^':
            compute(str[0]);
            break;

        default: // 若读入的是操作数，转换为整型后压入栈
            enter(stringToDouble(str));
            break;
        }
    }
}
void Calculator::clear()
{ // 清空操作数栈
    s.clear();
}
```

4.

```
#include "Calculator.h"
int main()
{
    Calculator c;
    c.run();
    return 0;
}
```

结果：



，符合预期。

参照《C++语言程序设计》学生用书，完成以下内容：没有学生用书的同学可以参考文件夹里实验用书、课本的相关例题、习题部分的图片

具体实现：

1）

```
#ifndef NODE_H_
#define NODE_H_

template <class T>
class Node {
private:
    T data;
public:
    Node* next;
    Node();
    Node(const T& data, Node<T>* nt = 0);
    //  Node(T data, Node<T>* n = NULL);
    T& getData();
    Node<T>* deleteAfter();//删除本结点的后继结点，并返回其地址
};

template <class T>
Node<T>::Node() {
    data = 0;
    next = NULL;
}

template <class T>
Node<T>::Node(const T& d, Node<T>* nt) {
    data = d;
    next = nt;
}
```

```
template <class T>
T& Node<T>::getData() {
    return data;
}

template <class T>
Node<T>* Node<T>::deleteAfter()
{
    Node<T>* tempPtr = next;
    if (next == 0)return 0;
    next = tempPtr->next;
    return tempPtr;
}
       #endif //NODE_H_
    2）
#ifndef LINK_H_
#define LINK_H_

#include <iostream>
using namespace std;
#include "node.h"

/*
* 在LinkedList的设计中，采用了附加指针front和rear，即链表的结构为
front->a1->a2->...->rear
* 只有在析构函数中才删除这两个指针，其余的时间这两个指针都是存在的，其中的数据始终为0，不存储用户数据
*/
template <class T>
class LinkedList {
private:
    Node<T>* front, * rear; //表头和表尾指针
    Node<T>* prevPtr, * currPtr; //当前节点的前指针和当前指针。记录表当前遍历位置的指针，由插入和删除操作更新
    int size; //表中元素个数
    int position; //当前元素(*currPtr)在表中的位置序号。由函数reset使用
    //内部函数
    Node<T>* newNode(const T& item, Node<T>* ptrNext = NULL); //生成新节点，数据与为item，指针域为prtNext
    void freeNode(Node<T>* p); //释放节点

    void copy(const LinkedList<T>& L); //将链表L复制到当前表（假设当前表为空），被复制构造函数和operator=调用
```

```
public:  //外部接口
    LinkedList(); //构造函数
    LinkedList(const LinkedList<T>& L); //复制构造函数
    ~LinkedList(); //析构函数
    LinkedList<T>& operator = (const LinkedList<T>& L); //重载赋值运算符

    int getSize() const; //返回链表中元素个数
    bool isEmpty() const; //链表是否为空

    void reset(int pos = 0); //初始化游标的位置
    void Next(); //使游标移动到下一个节点
    bool endOfList() const; //游标是否移动到链尾
    int currentPosition() const; //返回游标当前的位置

    void insertFront(const T& item); //在表头节点front之前插入节点
    void insertRear(const T& item); //在链表尾节点rear之前插入节点
    void insertAt(const T& item); //在当前节点之前插入节点
    void insertAfter(const T& item); //在当前节点之后插入节点

    T deleteFront(); //删除头节点
    void deleteCurrent(); //删除当前节点

    T& data(); //返回对当前节点成员数据的引用
    const T& data() const; //返回对当前节点成员数据的常引用

    void clear(); //清空链表：释放所有节点的内存空间，被析构函数和operator=使用
    void show() {
        cout << "*front:" << front << endl;
        cout << " *rear:" << rear << endl;  //表头和表尾指针
        cout << "*prevPtr:" << prevPtr << " *currPtr:" << currPtr << endl;   //记录
表当前遍历位置的指针，由插入和删除操作更新
        cout << "size:" << size << "  position:" << position << endl;
    }
};

//生成新节点，数据与为item，指针域为prtNext。内部函数
template <class T>
Node<T>* LinkedList<T>::newNode(const T& item, Node<T>* ptrNext) {
    Node<T>* n = new Node<T>(item, ptrNext);
    return n;
}

//释放节点，内部函数
template <class T>
```

```
void LinkedList<T>::freeNode(Node<T>* p) {
    Node<T>* temp = front;
    while (temp->next != p) {  //从头节点后面的节点开始遍历查找指针p的节点
        temp = temp->next;
        if (temp == currPtr)   //
            position++;      //currPtr要动吗?
    }
    temp->next = p->next;   //删除节点操作 。若没找到?

    if (currPtr == p)       //若删除的是当前指针所指结点
        currPtr = currPtr->next;

    if (prevPtr == p) {    //若删除的是当前指针的前一个结点
        prevPtr = prevPtr->next;
        currPtr = currPtr->next;
    }
    delete p;
    size--;
    position--;
}

//将链表L复制到当前表（假设当前表为空），被复制构造函数和operator=调用 。内部函数
template <class T>
void LinkedList<T>::copy(const LinkedList<T>& L) {
    Node<T>* temp = L.front, * ptr = front;
    while (temp != L.rear) {
        ptr->next = new Node<T>(temp->getData, NULL); // temp->getData()
        ptr = ptr->next;
        temp = temp->next;
    }
    ptr->next = rear;
    size = L.getSize();
    position = L.currentPosition();
}

//构造函数
template <class T>
LinkedList<T>::LinkedList() {
    front = new Node<T>();  //指向新建节点，实际建立一个空的头node结点和尾节点
    rear = new Node<T>();
    front->next = rear;
    currPtr = rear;   //当前指针指向尾节点
    prevPtr = front;  //之前指针指向头节点
    size = 0; //不计算front和rear节点
```

```
    position = 0; //在front下一个元素视为0
}

//复制构造函数
template <class T>
LinkedList<T>::LinkedList(const LinkedList<T>& L) {
    clear();  //清空释放当前链表
    copy(L);  //复制链表
}

//析构函数
template <class T>
LinkedList<T>::~LinkedList() {
    LinkedList::clear();  //清空释放当前链表
    delete front;  //释放头指针所指节点
    delete rear;   //释放尾指针所指节点
}

//重载赋值运算符
template <class T>
LinkedList<T>& LinkedList<T>::operator = (const LinkedList<T>& L) {
    clear();   //清空释放当前链表
    copy(L);   //复制链表
}

//返回链表中元素个数
template <class T>
int LinkedList<T>::getSize() const {
    return size;
}

//链表是否为空
template <class T>
bool LinkedList<T>::isEmpty() const {
    return (size == 0);
}

//初始化游标的位置
template <class T>
void LinkedList<T>::reset(int pos) { //移动游标(当前指针currPtr)到pos所指的位置
    if (pos > size) {
        cout << "越界，无法访问" << endl;
        return;
    }
```

```
    int i = 0;
    prevPtr = front;
    currPtr = front->next;
    while (i < pos) {
        if (currPtr == rear) {
            cout << "越界，无法访问" << endl;
            return;
        }
        i++;
        currPtr = currPtr->next;
        prevPtr = prevPtr->next;
    }
    position = pos;
}

//使游标(当前指针currPtr)移动到下一个节点
template <class T>
void LinkedList<T>::Next() {
    prevPtr = currPtr;
    currPtr = currPtr->next;
}

//游标是否移动到链尾
template <class T>
bool LinkedList<T>::endOfList() const {
    return (currPtr == rear);
}

//返回游标当前的位置
template <class T>
int LinkedList<T>::currentPosition() const {
    return position;
}

//在表头插入节点,原头节点front之前插入 ，插入之后原头节点的值为空，并且新插入的头结点也不为空，值为item.
template <class T>
void LinkedList<T>::insertFront(const T& item) {
    Node<T>* n = new Node<T>(item, front);
    front = n;  //front指向新的节点
    size++;
    position++;
}
```

```
//在链表尾节点之前插入节点
template <class T>
void LinkedList<T>::insertRear(const T& item) {
    Node<T>* temp = front;
    while (temp->next != rear)
        temp = temp->next;    //temp移至rear之前
    Node<T>* n = new Node<T>(item, rear);
    temp->next = n;
    size++;
}

//在当前节点之前插入节点
template <class T>
void LinkedList<T>::insertAt(const T& item) {
    Node<T>* temp = new Node<T>(item, currPtr);
    prevPtr->next = temp;
    prevPtr = temp;
    size++;
    position++;
}

//在当前节点之后插入节点
template <class T>
void LinkedList<T>::insertAfter(const T& item) {
    Node<T>* temp = new Node<T>(item, NULL);
    temp->next = currPtr->next;
    currPtr->next = temp;
    size++;
}

//删除头节点，实质是删除front->next，并返回被删头节点的值
template <class T>
T LinkedList<T>::deleteFront() {
    if (front->next == rear) {
        cout << "没有节点可以删除" << endl;
    }
    if (prevPtr == front->next) { //如果当前节点的前节点是被删的头结点front->next
        prevPtr = prevPtr->next;  //不应该移动？ 应该判断currPtr== front->next
        currPtr = currPtr->next;  //如果position不变，当前位置要下移一位，但下面的
position-- ，矛盾
    }
    Node<T>* temp = front->next; // 被删结点处理
    T d = temp->getData();
    front->next = temp->next;   //修改头指针
```

```
    delete temp;
    size--;
    if (front->next != rear)
        position--;    //被删节点在当前节点之前。
    return d;
}

//删除当前节点
template <class T>
void LinkedList<T>::deleteCurrent() {
    Node<T>* tempPtr = currPtr;
    prevPtr->deleteAfter();
    delete currPtr;
    currPtr = prevPtr;
    size--;
}

//返回对当前节点成员数据的引用
template <class T>
T& LinkedList<T>::data() {
    return currPtr->getData();
}

//返回对当前节点成员数据的常引用
template <class T>
const T& LinkedList<T>::data() const {
    return currPtr->getData();
}

//清空链表内容：释放所有节点的内存空间，被析构函数和operator=使用
template <class T>
void LinkedList<T>::clear() {
    Node<T>* temp;
    while (front->next != rear) {
        temp = front->next;
        front->next = temp->next;
        delete temp;
    }
    size = 0;
    position = 0;
}

#endif //LINK_H_
```

2. 实现第 9 章实验 9，实验任务（1），利用课本上的例 9-7 作为应用程序。

测试结果（已修改 deleteCurrent）：

```
3 6 5 7 5 2 4 5 9 10
List:3 6 5 7 5 2 4 5 9 10
number that need to be deleted:5
List:3 6 7 2 4 9 10
D:\C++ programs\project\LinkList\x64\Debug\LinkList.exe (进程 3344)已退出，代码为 0。
按任意键关闭此窗口. . .
```

太坑人了!!

更改部分：

```
//删除当前节点
template <class T>
void LinkedList<T>::deleteCurrent() {
    Node<T>* tempPtr = currPtr;
    prevPtr->deleteAfter();
    delete currPtr;
    currPtr = prevPtr;
    size --;
}
```

```
    return data;
}

template <class T>
Node<T>* Node<T>::deleteAfter()
{
    Node<T>* tempPtr = next;
    if (next == 0)return 0;
    next = tempPtr->next;
    return tempPtr;
}
#endif //NODE_H_
```

3. 实现第 9 章实验 9，实验任务（2），利用本实验给出的 link.h 文件，实现例 9-6 的链表类。完成 lab9_2.cpp

问 link.h 的实现代码中有 bug 吗？如何修改?

有 bug，分析如 2. 所示。

更改前：

```
init:
*front:000001DB40443A00
 *rear:000001DB40443050
*prevPtr:000001DB40443A00 *currPtr:000001DB40443050
size:0  position:0
insertFront Bug
List: 0  7
insertAfter Bug
List: 0  7
deleteCurrent Bug
List: 0  7
clear Bug
7
-572662307
```

更改后：

```
init:
*front:00000273027D3340
 *rear:00000273027D3750
*prevPtr:00000273027D3340 *currPtr:00000273027D3750
size:0  position:0
insertFront Bug
List: 0  7
insertAfter Bug
List: 0  7
deleteCurrent Bug
List: 0  7  8
D:\C++ programs\project\LinkList\x64\Debug\LinkList.exe (进程 26376)已退出，代码为 -1073741819。
按任意键关闭此窗口. . .
```

还是有问题，发现是代码把 rear 删了：

```
cout<<"deleteCurrent Bug"<<endl;
list.deleteCurrent();   //导致尾指针rear节点被提前删除
disp(list);

/*cout<<"deleteFront Bug"<<endl;
list.reset(0);
cout<<list.data()<<endl;
list.deleteFront();
cout<<list.data()<<endl;  //currPtr未改
*/
```

再次更改：

```
init:
*front:000001C9036D54E0
 *rear:000001C9036D5530
*prevPtr:000001C9036D54E0 *currPtr:000001C9036D5530
size:0  position:0
insertFront Bug
List: 0  7
insertAfter Bug
List: 0  7
deleteFront Bug
0
-572662307
clear Bug
0
0

D:\C++ programs\project\LinkList\x64\Debug\LinkList.exe (进程 392)已退出，代码为 0。
按任意键关闭此窗口. . .
```

4. 实现第 9 章实验 9，实验任务（3），用链表实现队列，完成 lab9_3.cpp。我们在用链表实现队列的时候，除了保存链表的头节点之外，还需要保存最后的尾节点来实现更快的 push 和 pop 操作。那队列头对应链表头还是链表尾更好？请按照你的想法用链表实现一个队列。

我认为队列头对应链表头更好。

实现：

```
//Queue.h
#ifndef QUEUE_H
#define QUEUE_H
#include <cassert>
//类模板的定义
template <class T, int SIZE = 50>
class Queue {
private:
    int front, rear, count;    //队头指针、队尾指针、元素个数
    T list[SIZE];//队列元素数组
public:
    Queue();           //构造函数，初始化队头指针、队尾指针、元素个数
    void insert(const T& item);    //新元素入队
    T remove();  //元素出队
    void clear();//清空队列
    const T& getFront() const; //访问队首元素
    //测试队列状态
    int getLength() const;//求队列长度
    bool isEmpty() const;//判队队列空否
    bool isFull() const;//判断队列满否
};
//构造函数，初始化队头指针、队尾指针、元素个数
template <class T, int SIZE>
Queue<T, SIZE>::Queue() : front(0), rear(0), count(0) { }

template <class T, int SIZE>
void Queue<T, SIZE>::insert(const T& item) {//向队尾插入元素
```

```
    assert(count != SIZE);
    count++; //元素个数增1
    list[rear] = item;    //向队尾插入元素
    rear = (rear + 1) % SIZE; //队尾指针增1，用取余运算实现循环队列
}
template <class T, int SIZE>
T Queue<T, SIZE>::remove() {
    assert(count != 0);
    int temp = front; //记录下原先的队首指针
    count--;          //元素个数自减
    front = (front + 1) % SIZE;//队首指针增1。取余以实现循环队列
    return list[temp];    //返回首元素值
}
template <class T, int SIZE>
const T& Queue<T, SIZE>::getFront() const {
    return list[front];
}
template <class T, int SIZE>
int Queue<T, SIZE>::getLength() const { //返回队列元素个数
    return count;
}
template <class T, int SIZE>
bool Queue<T, SIZE>::isEmpty() const {  //测试队空否
    return count == 0;
}
template <class T, int SIZE>
bool Queue<T, SIZE>::isFull() const {   //测试队满否
    return count == SIZE;
}
template <class T, int SIZE>
void Queue<T, SIZE>::clear() { //清空队列
    count = 0;
    front = 0;
    rear = 0;
}
#endif  //QUEUE_H
```

6. 练习习题 9-4，利用 DNode.h，编写使用双向链表的程序，其中实现左插入、右插入，删除，显示等链表的功能。

实现：

1）

```
#ifndef LINKLIST_H
#define LINKLIST_H
#include<iostream>
#include<cstdio>
```

```
#include<cassert>
using namespace std;
template<class T>
class dnode {
public:
    dnode<T>* last, * next;
    T data;
};

template<class T>
class linkList {
private:
    dnode<T>* head, * tail;
    dnode<T>* now;
public:
    linkList();
    ~linkList();
    linkList(const int& n);
    void deletenode(const dnode<T>* p);
    void linsert(dnode<T>* p, const T& x);
    void rinsert(dnode<T>* p, const T& x);
    void show();
    int getLen();
    dnode<T>* end();
    dnode<T>* start();
    dnode<T>* getHead();
};

template<class T>
linkList<T>::linkList()
{
    this->head = new dnode<T>();
    this->head->last = 0;
    this->head->next = 0;
    this->head->data = 0;
    this->now = head;
    this->tail = head;
}

template<class T>
linkList<T>::~linkList()
{
    dnode<T>* tmp;
    while (head->next != 0)
```

```
    {
        tmp = head->next;
        delete head;
        head = tmp;
    }
    delete head;
}

template<class T>
linkList<T>::linkList(const int& n)
{
    assert(n > 0);
    this->head = new dnode<T>();
    this->now = head;
    this->head->last = 0;
    this->head->next = 0;
    this->head->data = 0;
    dnode<T>* newnode = 0, * tmpnode = this->head;
    printf("输入%d个元素:\n", n);
    for (int i = 1; i <= n; ++i)
    {
        newnode = new dnode<T>();
        std::cin >> newnode->data;
        newnode->next = 0;
        newnode->last = tmpnode;
        tmpnode->next = newnode;
        tmpnode = newnode;
    }
    this->tail = newnode;
}

template<class T>
inline void linkList<T>::deletenode(const dnode<T>* p)
{
    assert(p != head);
    if (p == tail)
    {
        tail = tail->last;
    }
    if (p->last != 0)
    {
        p->last->next = p->next;
    }
    if (p->next != 0)
```

```
    {
        p->next->last = p->last;
    }
    delete p;
}

template<class T>
int linkList<T>::getLen()
{
    int len = 0;
    this->now = this->head;
    while (this->now->next)
    {
        now = now->next;
        len++;
    }
    return len;
}

template<class T>
inline void linkList<T>::linsert(dnode<T>* p, const T& x)
{
    assert(p != head);
    dnode<T>* q = new dnode<T>();
    q->data = x;
    q->next = p;
    q->last = p->last;
    p->last->next = q;
    p->last = q;
}

template<class T>
inline void linkList<T>::rinsert(dnode<T>* p, const T& x)
{
    dnode<T>* q = new dnode<T>();
    q->data = x;
    q->last = p;
    q->next = p->next;
    if (p != tail)
    {
        p->next->last = q;
        p->next = q;
    }
    else
```

```
    {
        p->next = q;
        tail = q;
    }
}

template<class T>
inline void linkList<T>::show()
{
    this->now = head->next;
    while (now->next)
    {
        std::cout << now->data << " ";
        now = now->next;
    }

    std::cout << now->data << "\n";
}

template<class T>
inline dnode<T>* linkList<T>::end()
{
    return this->tail;
}
template<class T>
inline dnode<T>* linkList<T>::start()
{
    return head->next;
}
template<class T>
inline dnode<T>* linkList<T>::getHead()
{
    return head;
}
#endif // !LINKLIST_H
```

2)

```
#pragma once
#ifndef ORDERLIST_H
#define ORDERLIST_H
#include"linkList.h"
using namespace std;
template<class T>
class orderList
```

```
{
private:
    linkList<T> list;
public:
    void orderinsert(const T& x);
    void show();
    dnode<T>* getStart();
    dnode<T>* getEnd();
};

template<class T>
void orderList<T>::orderinsert(const T& x)
{
    if (list.getLen() == 0)
    {
        list.rinsert(list.getHead(), x);
        return;
    }
    for (dnode<T>* i = list.start(); i != 0; i = i->next)
    {
        if (x <= i->data)
        {
            list.linsert(i, x);
            return;
        }
    }
    list.rinsert(list.end(), x);
    return;
}

template<class T>
void orderList<T>::show()
{
    for (dnode<T>* i = list.start(); i != 0; i = i->next)
    {
        cout << i->data << " ";
    }
    cout << endl;
}

template<class T>
dnode<T>* orderList<T>::getStart()
{
    return this->list.start();
```

```
}

template<class T>
dnode<T>* orderList<T>::getEnd()
{
    return this->list.end();
}
#endif
```

3）

```
#include<cstdio>
#include"orderList.h"
using namespace std;
int main()
{
    linkList<int> s;
    s.rinsert(s.getHead(), 5);
    s.show();
    s.linsert(s.end(), 6);
    s.show();
    s.rinsert(s.end(), 7);
    s.show();
    s.rinsert(s.start(), 2);
    s.show();
    s.deletenode(s.end()->last);
    s.show();
    s.deletenode(s.end());
    s.show();
    return 0;
}
```

运行结果：

符合预期。

7. 练习习题 9-10，直接插入排序。

代码：

1）

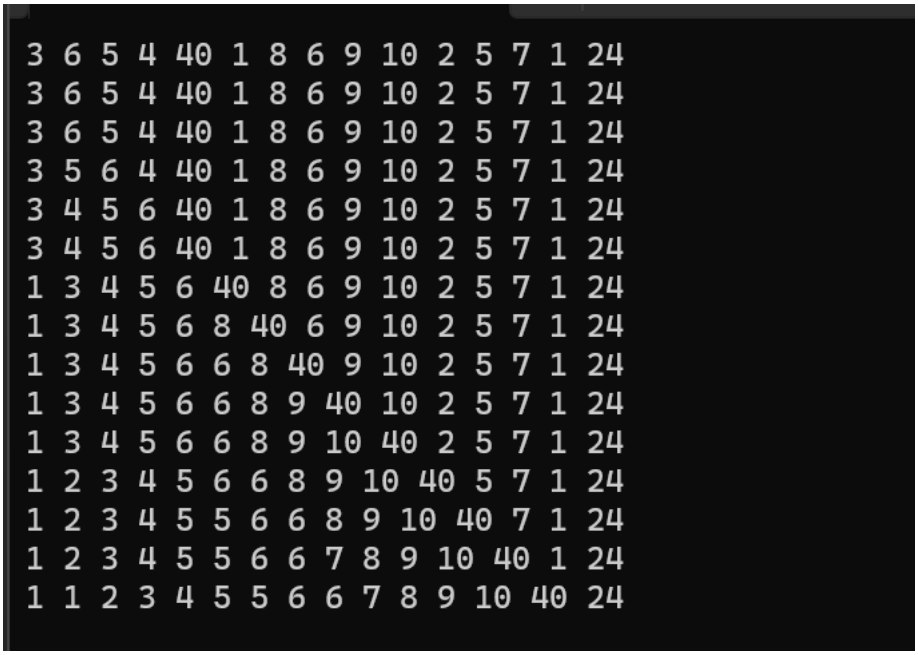
```
template <class T>
```

```
void insertionSort(T a[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        T temp = a[i];
        while (j > 0 && temp < a[j - 1])
        {
            a[j] = a[j - 1];
            j--;
        }
        a[j] = temp;
    }
}
```

2）

```
#include"sort.h"
#include<cstdio>
using namespace std;
int main()
{
    int a[] = { 3,6,5,4,40,1,8,6,9,10,2,5,7,1,24};
    for (int i = 0; i < 15; ++i)
    {
        sort(a, i);
        for (int j = 0; j < 15; ++j)
        {
            printf("%d ", a[j]);
        }
        printf("\n");
    }
    return 0;
}
```

运行结果：



8. 练习习题 9-12，选择排序。

代码：

1）

```
template <class T>
void mySwap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}

template <class T>
void sort(T a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int leastIndex = i;
        for (int j = i + 1; j < n; j++)
            if (a[j] < a[leastIndex])
                leastIndex = j;
        mySwap(a[i], a[leastIndex]);
    }
}
```

2）

```
#include"sort.h"
#include<cstdio>
using namespace std;
int main()
{
    int a[] = { 3,6,5,4,40,1,8,6,9,10,2,5,7,1,24};
    for (int i = 0; i < 15; ++i)
```

```
    {
        sort(a, i);
        for (int j = 0; j < 15; ++j)
        {
            printf("%d ", a[j]);
        }
        printf("\n");
    }
    return 0;
}
```

结果：

```
3 6 5 4 40 1 8 6 9 10 2 5 7 1 24
3 6 5 4 40 1 8 6 9 10 2 5 7 1 24
3 6 5 4 40 1 8 6 9 10 2 5 7 1 24
3 5 6 4 40 1 8 6 9 10 2 5 7 1 24
3 4 5 6 40 1 8 6 9 10 2 5 7 1 24
3 4 5 6 40 1 8 6 9 10 2 5 7 1 24
1 3 4 5 6 40 8 6 9 10 2 5 7 1 24
1 3 4 5 6 8 40 6 9 10 2 5 7 1 24
1 3 4 5 6 6 8 40 9 10 2 5 7 1 24
1 3 4 5 6 6 8 9 40 10 2 5 7 1 24
1 3 4 5 6 6 8 9 10 40 2 5 7 1 24
1 2 3 4 5 6 6 8 9 10 40 5 7 1 24
1 2 3 4 5 5 6 6 8 9 10 40 7 1 24
1 2 3 4 5 5 6 6 7 8 9 10 40 1 24
1 1 2 3 4 5 5 6 6 7 8 9 10 40 24
```

9. 练习习题 9-14，起泡排序。

代码：

1）

```
template <class T>
void mySwap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}
template <class T>
void sort(T a[], int n)
{
    int i = n - 1;
    while (i > 0) {
        int lastExchangeIndex = 0;
        for (int j = 0; j < i; j++)
            if (a[j + 1] < a[j]) {
```

```
                mySwap(a[j], a[j + 1]);
                lastExchangeIndex = j;
            }
        i = lastExchangeIndex;
    }
}
```

2)

```
#include"sort.h"
#include<cstdio>
using namespace std;
int main()
{
    int a[] = { 3,6,5,4,40,1,8,6,9,10,2,5,7,1,24};
    for (int i = 0; i < 15; ++i)
    {
        sort(a, i);
        for (int j = 0; j < 15; ++j)
        {
            printf("%d ", a[j]);
        }
        printf("\n");
    }
    return 0;
}
```

结果：

```
3 6 5 4 40 1 8 6 9 10 2 5 7 1 24
3 6 5 4 40 1 8 6 9 10 2 5 7 1 24
3 6 5 4 40 1 8 6 9 10 2 5 7 1 24
3 5 6 4 40 1 8 6 9 10 2 5 7 1 24
3 4 5 6 40 1 8 6 9 10 2 5 7 1 24
3 4 5 6 40 1 8 6 9 10 2 5 7 1 24
1 3 4 5 6 40 8 6 9 10 2 5 7 1 24
1 3 4 5 6 8 40 6 9 10 2 5 7 1 24
1 3 4 5 6 6 8 40 9 10 2 5 7 1 24
1 3 4 5 6 6 8 9 40 10 2 5 7 1 24
1 3 4 5 6 6 8 9 10 40 2 5 7 1 24
1 2 3 4 5 6 6 8 9 10 40 5 7 1 24
1 2 3 4 5 5 6 6 8 9 10 40 7 1 24
1 2 3 4 5 5 6 6 7 8 9 10 40 1 24
1 1 2 3 4 5 5 6 6 7 8 9 10 40 24
```

10. 练习习题 9-19，折半查找。

代码：

1)

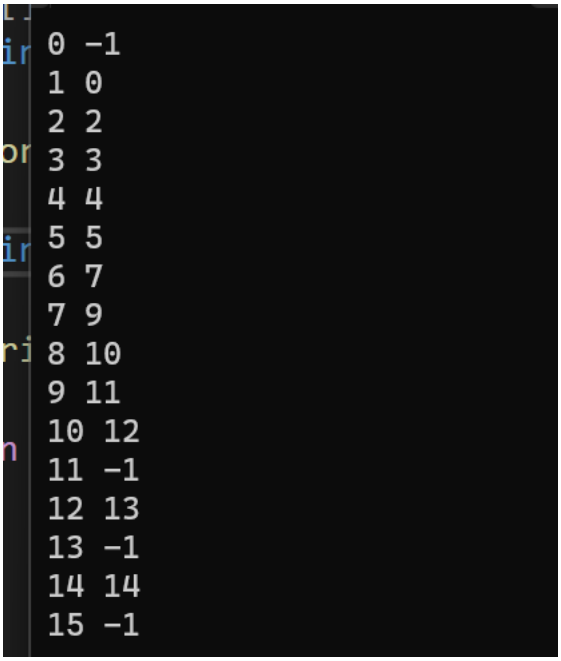
```
template <class T>
void mySwap(T& x, T& y) {
    T temp = x;
    x = y;
    y = temp;
}
template <class T>
void sort(T a[], int n)
{
    int i = n - 1;
    while (i > 0) {
        int lastExchangeIndex = 0;
        for (int j = 0; j < i; j++)
            if (a[j + 1] < a[j]) {
                mySwap(a[j], a[j + 1]);
                lastExchangeIndex = j;
            }
        i = lastExchangeIndex;
    }
}
template <class T>
int seqSearch(const T list[], int n, const T& key) {
    for (int i = 0; i < n; i++)
        if (list[i] == key)
            return i;
    return -1;
}
```

2)

```
#include"sort.h"
#include<cstdio>
using namespace std;
int main()
{
    int a[] = { 3,6,5,4,12,1,8,6,9,10,2,5,7,1,14};
    for (int i = 0; i < 15; ++i)
    {
        sort(a, i);
    }
    for (int i = 0; i < 16; ++i)
    {
        printf("%d %d\n", i,seqSearch(a,15,i));
    }
```

```
    return 0;
}
```

结果：



结论分析与体会：

非常好的实验，让我进一步认识了指针和线性数据结构以及一些排序，选择算法。