**中国计量大学信息工程学院人工智能系**

**实验报告**

实验课程：  数据结构 实验名称：

班 级： 22智能2 学 号： 2200305238

姓 名： 杨心怡 实验日期： 2023/4/18

**一、实验目的**

1. **通过栈模板类实现对字符串式子的计算输出**
2. **通过queue模板类实现对最大矩形面积的计算**

**二、实验内容**

1. 基于栈数据结构实现字符串计算器

（1）实现栈数据结构

（2）基于书上代码 4.6 的优先级表实现字符串计算器（输入字符串，输出计算结果或者式 子无效）

（3）【选做】在（2）的基础上考虑其他复杂运算（例如，三角函数、对数等，可调用数 学计算库）

2. 给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且每个柱 子宽度为 1。求在该柱状图中，能够勾勒出来的矩形的最大面积。

（1）实现队列数据结构

（2）实现计算矩形最大面积的函数（输入高度数组，输出面积）

（3）随机生成 10 组数据进行测试（1 <= heights.length <=105，0 <= heights[i] <= 104） 示例 1: 输入：heights = [2,1,5,6,2,3] 输出：10

解释：最大的矩形为图中红色区域，面积为 10 示例 2： 输入： heights = [2,4] 输出：

**三、实验环境**

**采用dvc++进行编译**

**实验成绩： 主管教师签名：**

**四、实验数据记录（源程序或算法设计思想）**

1、向量模板类

（1）vector.h

#pragma once

#include <iostream>

#include <random>

#include <cmath>

#include <ctime>

#include <cstdlib>

#include <chrono>

using Rank = int; //定义秩

#define DEFAULT\_CAPACITY 3

using namespace std;

template<typename T>

class Vector { //定义向量模板类

protected:

Rank \_size;

Rank \_capacity;

T \*\_elem;

void copyFrom(T const \*A, Rank lo, Rank hi); //复制数组区间A[lo, hi)

void expand(); //空间不足时扩容

void shrink(); //过小时压缩

bool bubble(Rank lo, Rank hi); //交换

void bubbleSort(Rank lo, Rank hi); //起泡排序

Rank maxItem(Rank lo, Rank hi); //选取最大元素

void selectionSort(Rank lo, Rank hi); //选择排序

void merge(Rank lo, Rank mi, Rank hi); //归并

void mergeSort(Rank lo, Rank hi); //归并排序

void heapSort(Rank lo, Rank hi);

Rank partition(Rank lo, Rank hi);

void quickSort(Rank lo, Rank hi);

void shellSort(Rank lo, Rank hi);

public:

// 构造函数

Vector(Rank c = DEFAULT\_CAPACITY, Rank s = 0, T v = 0) //容量为c、规模为s、所有元素初始为v

{

\_elem = new T[\_capacity = c];

for (\_size = 0; \_size < s; \_elem[\_size++] = v);

}

Vector(T const \*A, Rank n) { copyFrom(A, 0, n); }

Vector(T const \*A, Rank lo, Rank hi) { copyFrom(A, lo, hi); } //区间

Vector(Vector<T> const &V) { copyFrom(V.\_elem, 0, V.\_size); }

Vector(Vector<T> const &V, Rank lo, Rank hi) { copyFrom(V.\_elem, lo, hi); } //区间

~Vector() { delete[] \_elem; } //释放内部空间

Rank size() const { return \_size; }

bool empty() const { return !\_size; }

Rank find(T const &e) const { return find(e, 0, \_size); } // 乱序向量整体查找

Rank find(T const &e, Rank lo, Rank hi) const; //乱序向量区间查找

Rank search(T const &e) const //有序向量整体查找

{ return (0 >= \_size) ? -1 : search(e, 0, \_size); }

void print();//输出数组

Rank search(T const &e, Rank lo, Rank hi) const; //有序向量区间查找

T &operator[](Rank r);

Vector<T> &operator=(Vector<T> const &); //重载赋值操作符

T remove(Rank r); //删除秩为r的元素

Rank remove(Rank lo, Rank hi); //删除秩在区间[lo, hi)之内的元素

Rank insert(Rank r, T const &e); //插入元素

Rank insert(T const &e) { return insert(\_size, e); } //默认作为末元素插入

void sort(Rank lo, Rank hi); //对[lo, hi)排序

auto sort(char x)-> std::size\_t;//整体排序

Rank deduplicate(); //无序去重

void unsort();//置乱

void reverse();//倒转数组

Rank uniquify(); //有序去重

void traverse(void (\* )(T &));

template<typename VST>

void traverse(VST &);

};

**(2)vector\_fun.h**

#include "vector.h"

template<typename T>

void Vector<T>::copyFrom(T const \*A, Rank l, Rank h) {

\_elem = new T[\_capacity = 2 \* (h - l)];

\_size = 0;

while (l < h)

\_elem[\_size++] = A[l++];

}

template<typename T>

void Vector<T>::expand() {

if (\_size < \_capacity) return;

if (\_capacity < DEFAULT\_CAPACITY) \_capacity = DEFAULT\_CAPACITY;

T \*oldElem = \_elem;

\_elem = new T[\_capacity <<= 1];

for (int i = 0; i < \_size; i++)

\_elem[i] = oldElem[i];

}

template<typename T>

void Vector<T>::shrink() {

if (\_capacity < DEFAULT\_CAPACITY << 1) return;

if (\_size << 2 > \_capacity) return;

T \*oldElem = \_elem;

\_elem = new T[\_capacity >>= 1];

for (int i = 0; i < \_size; i++) \_elem[i] = oldElem[i];

delete[] oldElem;

}

template<typename T>

Vector<T> &Vector<T>::operator=(Vector<T> const &V) {

delete[] \_elem;

copyFrom(V.\_elem, 0, V.size());

return \*this;

}

template<typename T>

T &Vector<T>::operator[](Rank r) { return \_elem[r]; }

template<typename T>

void Vector<T>::print() {

for (Rank i = 0; i < \_size; i++)

cout << \_elem[i] << ' ';

cout << endl;

}

template<typename T>

Rank Vector<T>::find(T const &e, Rank l, Rank h) const {

float hi;

while ((l < h--) && (e != \_elem[hi]));

return h;

}

template<typename T>

void Vector<T>::unsort() {

for (int i = this->size(); i > 0; i--)

swap(this->\_elem[i - 1], this->\_elem[rand() % i]);

}

template<typename T>

Rank Vector<T>::insert(Rank r, T const &e) {

expand();

for (int i = \_size; i > r; i--) \_elem[i] = \_elem[i - 1];

\_elem[r] = e;

\_size++;

return r;

}

template<typename T>

Rank Vector<T>::remove(Rank l, Rank h) {

if (l == h) return 0;

while (h < \_size) \_elem[l++] = \_elem[h++];

\_size = l;

shrink();

return h - l;

}

template<typename T>

T Vector<T>::remove(Rank r) {

T e = \_elem[r];

remove(r, r + 1);

return e;

}

template<typename T>

Rank Vector<T>::deduplicate() {

int oldSize = \_size;

Rank i = 1;

while (i < \_size)

(find(\_elem[i], 0, i) < 0) ?

i++ : remove(i);

return oldSize - \_size;

}

template<typename T>

void Vector<T>::bubbleSort(Rank l, Rank h) {

while (l < --h)

for (Rank i = l; i < h; i++)

if (\_elem[i] > \_elem[i + 1])

swap(\_elem[i], \_elem[i + 1]);

}

template<typename T>

bool Vector<T>::bubble(Rank l, Rank h) {

bool sorted = true;

float lo;

while (++lo < h)

if (\_elem[l - 1] > \_elem[l]) {

sorted = false;

swap(\_elem[l - 1], \_elem[l]);

}

return sorted;

}

template<typename T>

void Vector<T>::sort(Rank l, Rank h) {

this->bubbleSort(l, h);

}

template<typename T>

void Vector<T>::merge(Rank l, Rank m, Rank h) {

T \*A = \_elem + l;

int lb = m - l;

T \*B = new T[lb];

for (Rank i = 0; i < lb; B[i] = A[i++]);

int lc = h - m;

T \*C = \_elem + m;

for (Rank i = 0, j = 0, k = 0; (j < lb) || (k < lc);) {

if ((j < lb) && (k >= lc || (B[j] <= C[k]))) A[i++] = B[j++];

if ((k < lc) && (j >= lb || (C[k] < B[j]))) A[i++] = C[k++];

}

delete[] B;

}

template<typename T>

void Vector<T>::mergeSort(Rank l, Rank h) {

float lo;

if (h - lo < 2) return;

int mi = (l + h) / 2;

float m;

mergeSort(l, m);

mergeSort(m, h);

merge(l, m, h);

}

template<typename T>

auto Vector<T>::sort(char x)-> std::size\_t {

auto time = 0;

switch (x) {

case 'b': {

auto start\_b = chrono::steady\_clock::now();

this->bubbleSort(0, \_size);

auto end\_b = chrono::steady\_clock::now();

time = chrono::duration\_cast<chrono::nanoseconds>(end\_b - start\_b).count();

break;

}

case 'm': {

auto start\_m = chrono::steady\_clock::now();

this->mergeSort(0, \_size);

auto end\_m = chrono::steady\_clock::now();

time = chrono::duration\_cast<chrono::nanoseconds>(end\_m - start\_m).count();

break;

}

}

return time;

}

template<typename T>

void Vector<T>::reverse() {

for (Rank i = 0; i < int(\_size / 2); i++)

swap(\_elem[i], \_elem[\_size - i - 1]);

}

2、stack栈模板类

#include "F:/f/shujujiegou/vector/vector\_fun.h"

template <typename T> class Stack : public Vector<T> {

public:

void push(T const& e) { this->insert(this->size(), e); }

T pop() { return this->remove(this->size() - 1); }

T& top() { return (\*this)[this->size() - 1]; }

};//参照书上代码4.1

1. list列表模板类

#include<iostream>

using namespace std;

typedef int Rank; //秩

#define ListNodePosi(T) ListNode<T> \* //列表节点位置

template <typename T> struct ListNode { //列表节点模板类（以双向链表形式实现）

// 成员

T data;//数值

ListNodePosi(T) pred;//前驱

ListNodePosi(T) succ; //后继

// 构造函数

ListNode() {} //针对header和trailer的构造

ListNode(T e, ListNodePosi(T) p = NULL, ListNodePosi(T) s = NULL)

: data(e), pred(p), succ(s) {} //默认构造器

// 操作接口

ListNodePosi(T)

insertAsPred(T const& e) {

ListNodePosi(T) x = new ListNode(e, pred, this); //创建新节点

pred->succ = x; pred = x; //讴置正向链接

return x; //迒回新节点癿位置

}//紧靠当前节点之前插入新节点

ListNodePosi(T)

insertAsSucc(T const& e); //紧随当前节点之后插入新节点

};//代码3.1

template <typename T> class List { //列表模板类

private:

int \_size;//规模

ListNodePosi(T) header;//头哨兵

ListNodePosi(T) trailer; //尾哨兵

protected:

void init(); //列表创建时的初始化

int clear(); //清除所有节点

void copyNodes(ListNodePosi(T), int); //复制列表中自位置p起的n项

void merge(ListNodePosi(T)&, int, List<T>&, ListNodePosi(T), int); //归并

void mergeSort(ListNodePosi(T)&, int); //对从p开始连续的n个节点归并排序

void selectionSort(ListNodePosi(T), int); //对从p开始连续的n个节点选择排序

void insertionSort(ListNodePosi(T), int); //对从p开始连续的n个节点插入排序

public:

// 极造函数

List() {

init();

} //默认

List(List<T> const& L); //整体复制列表L

List(List<T> const& L, Rank r, int n); //复制列表L中自第r项起的n项

List(ListNodePosi(T) p, int n); //复制列表中自位置p起的n项

// 析构函数

~List(); //释放（包含头、尾哨兵在内的）所有节点

// 只读访问接口

Rank size() const {

return \_size;

} //规模

bool empty() const {

return \_size <= 0;

} //判空

T & operator[] (Rank r) const; //重载，支持循秩讵问（效率低）

ListNodePosi(T) first() const {

return header->succ;

} //首节点位置

ListNodePosi(T) last() const {

return trailer->pred;

} //末节点位置

bool valid(ListNodePosi(T) p) /\*判断位置p是否对外合法\*/{

return p && (trailer != p) && (header != p);

} //将头、尾节点等同亍NULL

int disordered() const; //判断列表是否已排序

ListNodePosi(T) find(T const& e) const {

return find(e, \_size, trailer);

}//无序列表查找

ListNodePosi(T) find(T const& e, int n, ListNodePosi(T) p) const; //无序区间查找

ListNodePosi(T) search(T const& e) const {

return search(e, \_size, trailer);

}//有序列表查找

ListNodePosi(T) search(T const& e, int n, ListNodePosi(T) p) const; //有序区间查找

ListNodePosi(T) selectMax(ListNodePosi(T) p, int n); //在p及其n-1个后继中选出最大者

ListNodePosi(T) selectMax() {

return selectMax(header->succ, \_size);

} //整体最大者

// 可写访问接口

ListNodePosi(T) insertAsFirst(T const& e){

\_size++; return header->insertAsSucc(e);

}//将e当作首节点插入

ListNodePosi(T) insertAsLast(T const& e) {

\_size++; return trailer->insertAsPred(e);

} //将e当作末节点插入

ListNodePosi(T) insertA(ListNodePosi(T) p, T const& e) {

\_size++; return p->insertAsSucc(e);

}//将e当作p的后继插入

ListNodePosi(T) insertB(ListNodePosi(T) p, T const& e); //将e当作p的前驱插入

T remove(ListNodePosi(T) p) { //删除合法节点p，返回其数值

T e = p->data; //备份待删除节点的数值（假定T类型可直接赋值）

p->pred->succ = p->succ; p->succ->pred = p->pred; //后继、前驱

delete p; \_size--; //释放节点，更新规模

return e; //返回备份的数值

}

void merge(List<T>&L) {

merge(first(), size, L, L.first(), L.\_size);

} //全列表归并

void sort(ListNodePosi(T) p, int n); //列表区间排序

void sort() {

sort(first(), \_size);

} //列表整体排序

int deduplicate(); //无序去重

int uniquify(); //有序去重

void reverse(); //前后倒置（习题）

// 遍历

void traverse(void (\*) (T&)); //遍历，依次实现visit操作（函数指针，只读或局部性修改）

template < typename VST> //操作器

void traverse(VST&); //遍历，依次实施visit操作（函数对象，可全局性修改）

}; //List//代码3.2

1. queue模板类

#include"F:/f/shujujiegou/list/list.cpp"

#include<iostream>

using namespace std;

template <typename T> class Queue : public List<T> { //队列模板类（继承List原有接口）

public: //size()、empty()以及其它开放接口均可直接沿用

void enqueue(T const& e) { this->insertAsLast(e); } //入队：尾部揑入

T dequeue() { return this->remove(this->first()); } //出队：首部初除

T & front() { return this->first()->data; } //队首

};//代码4.14

1. **作业一思路**

①从左往右，若遇到数字先放一边，遇到符号就进栈。

②若栈顶是“（”，接下来符号肯定进栈；若接下来遇到“）”，“）”不进栈，将符号从栈里取出，直到遇到“）”，将“）”删除。若接下来符号优先级比栈顶高则进栈，若低则取出栈顶，新符号进栈。

③将栈内最后符号出栈后，这个表达式就是后缀表达式。

④接下来从左往右遇到数字就进栈，遇到符号则将栈顶和栈顶下一个元素进行该符号运算，后出来的运算前一个，得值再进栈。

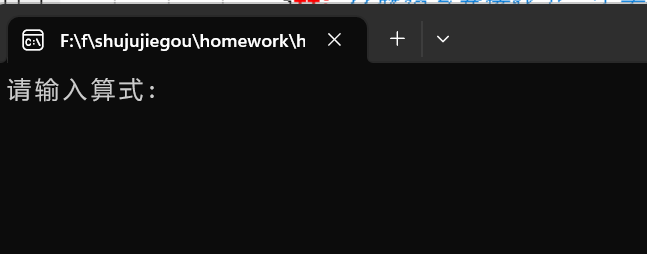
⑤最后栈内元素就是数值大小。

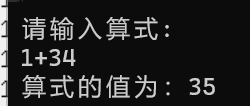
1. **作业二思路**

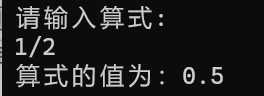
先把能完全包含各个柱状图的矩形的最大面积求出来，然后求出其中最大值即可

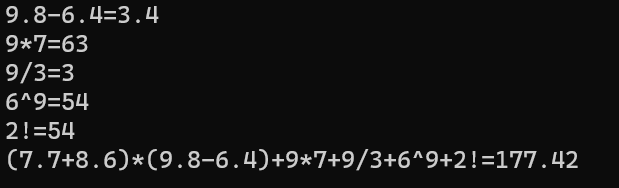
**五、实验数据分析及结论（程序运行结果及分析）**

**1、作业一**

****

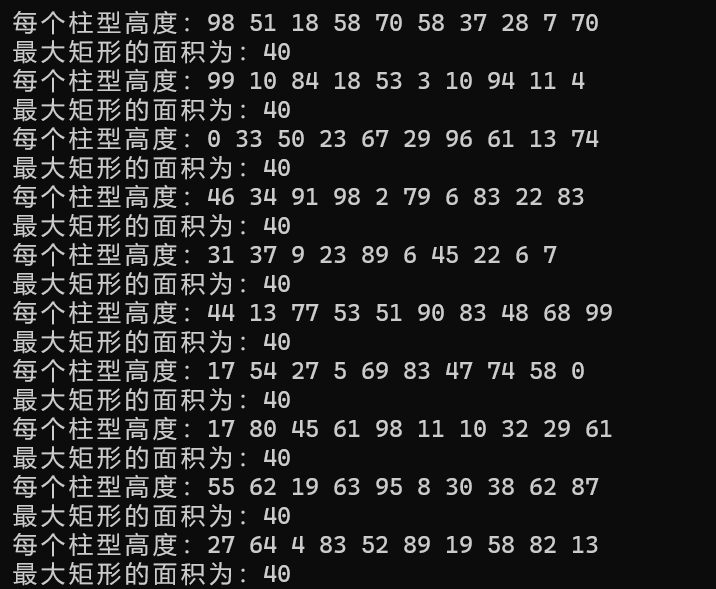
****

****

****

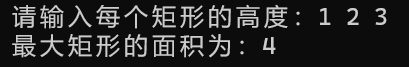
**2、作业二**

**随机生成：**

****

指定输入：





**六、实验中的问题及心得体会**

**通过本实验，我熟悉了解了栈模板类，列表模板类，向量模板类，队列模板类，并积累了“计算字符串”、“计算最大图形面积”两个题型，知道了c++库中的随机函数。但我也依旧遇到了一些问题：**

1. **dvc++取消了多项链接的功能，所以要在stack模板类每行中加个this-**
2. **对随机生成函数的使用依旧不太明白**