面向对象程序设计库函数编程作业报告

小组成员：170319-2017040334-孟铃翔

1. 容器类模板
2. 线性表Vector
3. 模板设计和功能介绍：

私有成员：

缓冲区元素个数s, 缓冲区大小c, 缓冲区动态数组对应的指针buffer

公有成员：

* 构造函数

1. vector(): 初始化数组的缓冲区大小为0，元素个数为0。
2. vector(int n): 初始化定义数组的大小和元素个数为n。同时为缓冲区申请内存空间。
3. vector(int n, T elem): 初始化定义数组的大小和元素个数都为n。同时将缓冲区中每个元素都初始化为指定元素。

* 拷贝构造函数

vector(vector<T> &c2)：接受的参数为一个已存在的vector, 初始化当前vector的缓冲区大小与元素个数与c2相同，同时将已存在的vector中的元素复制到缓冲区。

* 析构函数

~vector()：删除为缓冲区申请的内存空间，并使得缓冲区大小和元素个数为0。

* 其他包含vector属性的函数

1. int size()：返回缓冲区元素的个数
2. int capacity(): 返回缓冲区的大小
3. int max\_size(): 与capacity()作用类似
4. bool empty(): 返回当前vector是否为空，即缓冲区元素个数是否为0.
5. T front(): 返回缓冲区的第一个元素
6. T back(): 返回缓冲区中的最后一个元素
7. T at(int index): 接受参数为一个下标，返回缓冲区中对应下标的元素

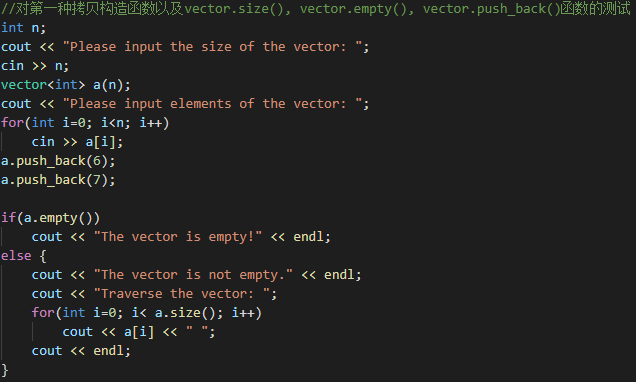
* 其他操作

1. void reserve(int n): 重新规定缓冲区大小。接受参数为一个缓冲区大小。当n大于capacity是，为缓冲区重新申请大小为n的空间。不改变当前元素个数及当前元素。
2. void resize(int n): 重新规定缓冲区元素的个数。接收参数为新的缓冲区元素个数n。当n小于当前capacity时，重定义size = n；当n大于当前capacity时，调用reserve(c + c / 2)函数重新定义缓冲区大小。c + c / 2参考了stl源码。
3. void resize(int n, T elem): 类似于resize(int n)函数。多出了形参，指定的元素elem。但是当n大于当前size小于capacity时，需要用指定的elem填充多出来的缓冲区元素。
4. void assign(int n, T elem): 重新对vector进行赋值。先对vector执行resize(n)操作，再用指定数目的elem元素替代之前缓冲区的元素。
5. void insert(int index, T element): 在指定下标index的元素前插入指定的元素element。先对缓冲区执行resize(n, elem)操作，重定义缓冲区大小，再执行插入操作。
6. void insert(int index, int n, T elem): 在指定index下标前插入n个指定的elem元素。同样先重新定义缓冲区，再进行插入操作。
7. void push\_back(T elem): 将指定的elem元素插入缓冲区的最后。先将size+1，再重定义缓冲区
8. void pop\_back(): 将缓冲区最后一个元素销毁，并将size-1。
9. void clear(): 清除整个缓冲区，并定义s和c均为0。

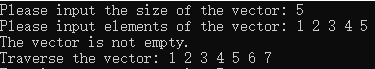
* 运算符重载

1. 对==、!=、<=、<、>=、>等运算符进行了重载。用来判断两个vector之间的关系。
2. 对[]运算符进行了重载，返回缓冲区对应下标的元素的引用。
3. 应用实践及测试

测试一：



运行结果：



对vector.size(), vector(int n), vector.empty(), vector.push\_back()以及下标运算符重载的测试。先向vector中输入1 2 3 4 5，之后将6和7放到vector末端。判断vector不为空，故输出1 2 3 4 5 6 7.测试成功。由于vector.push\_back()函数间接调用了vector.resize()函数，同时vector.resize()函数间接调用了vector.reserve()函数，故resize()与reserve()函数也测试成功。

测试二：

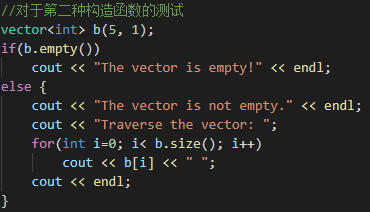


运行结果：



对vector.front(), vector.back()函数进行测试，可以看到运行结果判断当前的size是7，首元素是1，尾元素是7。测试正确。

测试三：

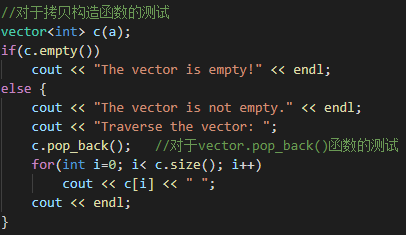


运行结果：



对构造函数vector(int n, T elem)进行测试。初始化vector含有5个1，判断vector非空并输出vector。测试成功。

测试四：



运行结果：



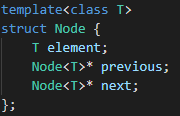
对拷贝构造函数进行测试。vector c拷贝构造vector a，判断c非空，弹出最后一个元素并遍历，输出1 2 3 4 5 6序列。测试成功。

1. 链表list
2. 模板设计和功能介绍

私有成员：

双向链表的头结点及尾结点。链表的元素个数s。链表元素的最大个数msize。

结点定义如下：



含有自定义类型的数据、上一个结点指针previous以及下一个结点指针next。

公有成员：

* 构造函数：

1. list(): 默认构造函数，为head结点以及tail结点申请内存空间。同时将head的下个结点指向tail，tail的上个结点指向head。定义元素个数s为0，且msize=100。
2. list(int n): 生成一个大小为n的结点。n个元素都被初始化为默认变量0。
3. list(int n, T e): 同上个构造函数类似。接受参数为元素个数n及指定元素e。对于新建的每个结点，用指定元素e进行初始化。

* 析构函数：

~list(): 清除所有结点。并还原元素个数s为0。

* 其他操作：

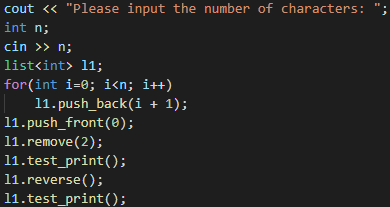
1. void insert(int position, const T& val): 插入函数。接受参数为position，代表将新的元素插到position对应元素之前；val代表被插入元素的值。
2. void erase(int position): 删除对应位置的元素。接收参数position代表将要被删除元素的位置。
3. void push\_front(T e): 在链表头部添加一个结点。接受参数e代表将要被插入链表的元素。
4. void push\_back(T e): 在链表尾部添加一个结点。接收参数e代表将要被插入链表的元素。
5. void pop\_front(): 删掉链表头部的元素。
6. void pop\_back(): 删掉链表尾部的元素。
7. void clear(): 清空链表中所有的元素。初始化链表内部元素个数为0。
8. void reverse(): 是链表按照初始元素的顺序倒序。
9. void assign(int n, T e): 将e的n个拷贝赋值插入到当前链表的头部。
10. void remove(T e): 将特定元素对应的结点从链表中删除。接受参数e为要删除的元素。

* 链表的属性：

1. Int size(): 返回当前链表中元素的个数
2. Bool empty(): 判断当前链表是否为空。
3. T front(): 返回链表的第一个元素
4. T back(): 返回链表尾部最后一个元素。

(2). 应用实践及测试

代码截图：



运行结果：



解释：

对list.push\_back(), list.push\_front(), list.remove(), list.reverse()函数进行测试。输入个数为5，则依次将1 2 3 4 5插入到链表。之后将0插入到链表头部，再将元素2对应的结点移除。为了测试方便，在class定义时定义了test\_print()函数用来遍历整个链表。可以看出输出结果0 1 3 4 5符合操作。对链表进行reverse操作实现链表的逆序。故输出5 4 3 1 0。测试正确。

1. 栈stack

(1). 模板设计和功能介绍

私有成员：

缓冲区的最大容量capacity，栈的元素个数number，缓冲区数组buffer

公有成员：

* 构造函数

Stack(int c=100): 初始化栈，并初始化元素个数为0。接受参数为栈的最大容量capacity。默认capacity为100。

* 栈的属性

bool empty(): 判断栈是否为空。

int size(): 返回栈内部元素的个数。

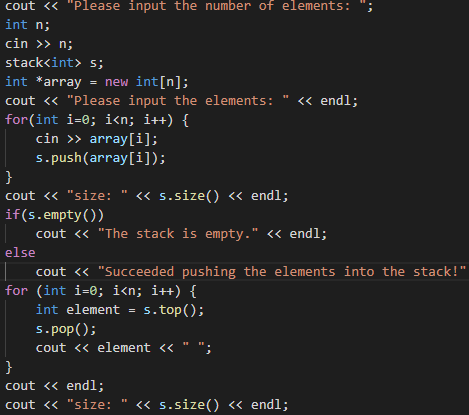
T top(): 返回当前栈顶的元素。

* 对于栈的操作

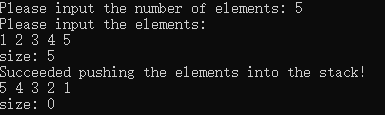
void push(T element): 将给定元素压栈。接受参数为要被压入栈内部的元素。同时将栈内元素个数+1.

void pop(): 将栈顶元素出栈。并将栈内元素个数减一。

(2). 应用实践及测试



运行结果：



该程序对stack.push(), stack.size(), stack.empty(), s.top(), s.pop()函数都进行了测试。先将5个元素1 2 3 4 5依次压入栈内。程序输出栈内元素的个数为5。之后判断栈非空，输出了成功插入的信息。最后将各个元素依次出栈，输出5 4 3 2 1。输出栈内元素个数为0。测试成功。

1. 队列queue

(1). 模板设计和功能介绍

私有成员：

缓冲区的最大容量capacity，队列的元素个数number，缓冲区数组buffer

公有成员：

* 构造函数

queue(int c=100): 初始化队列，并初始化元素个数为0。接受参数为队列的最大容量capacity。默认capacity为100。

* 队列的属性

bool empty(): 判断队列是否为空。

int size(): 返回队列内部元素的个数。

T front(): 返回当前队列最前边的元素。

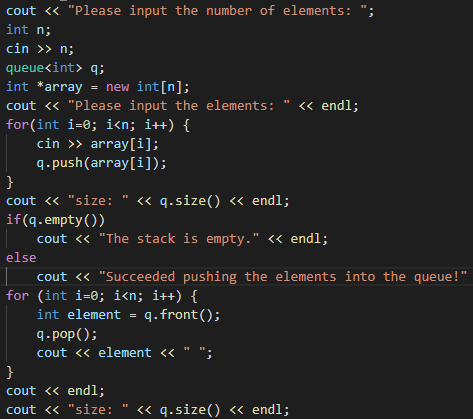
* 对于队列的操作

void push(T element): 将给定元素放入队列尾部。接受参数为要被放入队列内部的元素。同时将队列内元素个数加一。

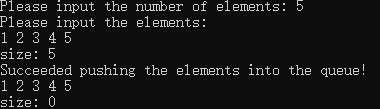
void pop(): 将队列最前边元素弹出。并将队列内元素个数减一。

(2). 应用实践及测试

测试代码：



运行结果：



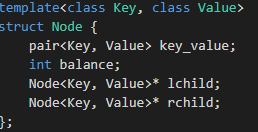
该程序对queue.push(), queue.size(), queue.empty(), queue.top(), queue.pop()函数都进行了测试。先将5个元素1 2 3 4 5依次放入队列。程序输出队列内元素的个数为5。之后判断队列非空，输出了成功插入的信息。最后将各个元素依次弹出队列，输出5 4 3 2 1。输出队列内元素个数为0。测试成功。

1. 映射表map

(1). 模板设计和功能介绍

私有成员：

Avl\_tree，为平衡二叉树，是map的主要的数据结构。平衡二叉树的结点构造如下所示。



以及map中结点的个数number。

* int RRotate(Node\* &p): 对失衡的平衡二叉树结点进行右旋调整。调整成功返回0。
* int LRotate(Node\* &p): 对失衡的平衡二叉树结点进行左旋调整。调整成功则返回0。
* int LLAdjust(Node\* &p): 当失衡结点为LL型时，对其进行LL类型的调整。调整成功则返回0。
* int RRAdjust(Node\* &p): 当失衡结点为RR型时，对其进行RR类型的调整。调整成功则返回0。
* int LRAdjust(Node\* &p): 当失衡结点为LR型时，对其进行LR类型的调整。调整成功则返回0。
* int RLAdjust(Node\* &p): 当失衡结点为RL型时，对其进行RL类型的调整。调整成功则返回0。
* int DestroyTree(Node\* &p): 递归销毁整个平衡二叉树。

公有成员：

* 构造函数

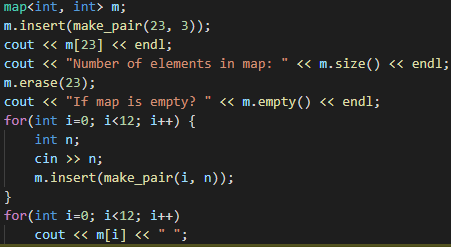
map(): 无参构造函数。初始化map内元素个数为0。平衡二叉树初始为空。

* 析构函数

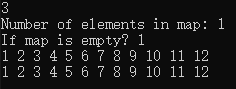
~map(): 调用clear()函数，清理掉整个平衡二叉树。

* Node\* Find(key aKey): 根据给定的键寻找对应的结点。返回值为该键对应的结点。
* int insert(pair<Key, Value> e): 传入一个pair类型的键值对。并将该键值对插入到平衡二叉树中。插入成功则返回值为0。否则返回1。
* int erase(Key aKey): 传入某个键。在平衡二叉树中删除该键对应的结点。删除成功则返回0。
* int size(): 返回当前map中元素的个数。
* bool empty(): 返回当前map是否为空。
* void clear(): 调用私有成员函数DestroyTree()函数清空整个二叉树。并将map中元素个数变为0。
* Key lower\_bound(Key aKey): 给定一个键，找到map中小于等于当前键的最大的键。
* Key upper\_bound(Key aKey): 给定一个键，找到map中大于当前键的最小的键。
* Value& operator[](Key aKey): 对map中的下标运算符进行重载以实现类似数组的操作。接受参数为一个键。若该键存在于map中，返回对应的值的引用。否则将该键插入map，默认值为0。

(2). 应用实践及测试



运行结果：



开始时将键值对<23, 3>插入。Map.insert()测试成功。

输出map中有1个键值对。Map.size()测试成功。

之后将键23对应的结点删除。显示map为空。Map.empty()测试成功。

之后分别插入12个键值对，输出他们的值与输入相同。Map下标运算符重载测试成功。

1. 函数模板
2. 排序类函数模板

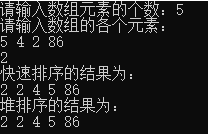
(1). 快速排序：

* Int Partition(T\* l, int s, int t): 对数组的某个序列进行划分。接受参数l为一个数组。S为起始排列下标，t为终止排列下标。返回值为乱序序列第一个值在排序后的位置下标。
* Int Qsort(T \*a, int s, int t): 对数组的某个序列进行快速排序。接受参数a为一个数组，s为起始排序下标，t为终止排序下标。之后调用Partition函数求得乱序序列第一个元素划分后的下标。对其左边进行递归Qsort(), 右边进行递归Qsort()，得到最后的排序序列。
* Int Quicksort(T \*a, int n): 对某个数组全部进行排序。传入参数a为数组，n为数组元素个数。并调用Qsort(a, 0, n-1)进行快速排序得到结果。排序结束得到返回值0.

(2). 堆排序

* Int Heapadjust(T \*a, int n, int s): 堆调整算法。S要进行操作的数组，n为堆的个数，s为当前元素对应的下标。
* int createHeap(T\* a, int n): 建立大顶堆。a为将要进行排序操作的数组，n为数组的个数。
* int HeapSort(T\* a, int n): a为要进行排序操作的数组，n为数组的大小。首先建堆，之后逐步进行堆调整，得到堆排序结果。返回值为0。

(3). 应用测试



代码逻辑简单，发现排序正确，测试成功。

1. 查找类模板

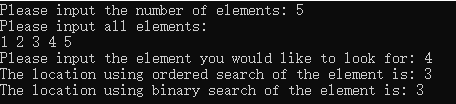
(1). 顺序查找

* int InOrderSearch(T \*a, T e, int n): 在数组中依次遍历查找对应元素是否存在。a为被查找的数组，e为需要查找的元素，n为数组中元素的个数。存在则返回对应下标。不存在则返回-1。

(2). 二分查找

* int BiSearch(T \*a, T e, int s, int t): 对某个数组某一部分进行二分查找。a为要进行操作的数组，e为要被查找的元素。S为数组元素的起始下标，t为数组元素的终止下标。
* int BinarySearch(T \*a, T e, int n): 对整个数组进行二分查找。a为被操作的数组，e为需要寻找的元素。n为数组中元素的个数。

(3). 应用实例测试

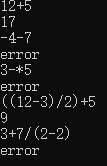


输入一个序列，两种查找算法找到的下标相同且正确。测试成功。

1. 应用实例
2. 四则表达式的计算

该程序需要用到栈的思想，运用了自定义的栈模板库。

测试实例：



运行结果符合题目要求，测试正确。

1. 图的广度优先遍历

说明：图的广度优先遍历需要用到队列作为遍历是存储的容器。本题使用自定义的队列模板进行测试。

输入：

输入的第一行包含一个正整数n，表示图中共有n个顶点。其中n不超过50。 以后的n行中每行有n个用空格隔开的整数0或1，对于第i行的第j个0或1，1表示第i个顶点和第j个顶点有直接连接，0表示没有直接连接。当i和j相等的时候，保证对应的整数为0。 输入保证邻接矩阵为对称矩阵，即输入的图一定是无向图。

输出：

只有一行，包含n个整数，表示按照题目描述中的广度优先遍历算法遍历整个图的访问顶点顺序。每个整数后输出一个空格，并请注意行尾输出换行。

运行结果：



发现输出结果符合题意，测试正确。

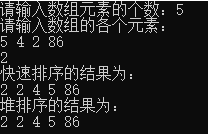
1. 排序算法

说明：排序算法实例使用之前自定义的排序函数模板。

输入：第一行为n，代表输入n个数。第二行为这n个数。

输出：对输入的n个数排序后的结果。

测试结果：



输出结果排序正确，测试成功。

1. 个人总结

在整个完成作业的过程中，自己态度端正，投入了不少时间，查阅了许多资料，结合课堂所学知识，完成了大部分模板库的功能，且实现结果正确。

同时在做作业的过程中自己也对面向对象程序设计的思想有了更加深刻和生动的理解，也复习到了许多遗忘的数据结构知识，感觉收益匪浅。

个人评价：A。