**2022212153 陈祥烨 计科22-2班 第四章作业**

第一题

由于链表采用的是动态结构，是从系统中申请的存储空间。当栈变量本身从逻辑上应当因失效而不用，即应当释放时，动态变量不会自动释放，因而造成动态变量的存储空间的不断的无谓的浪费。为此增设析构函数，以便自动释放占用的空间。

第二题

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct node

{

T data; //数据域

node<T>\* next; //指针域

};

template<class T>

class Queue

{

public:

Queue(); //默认构造

Queue(Queue<T>& queue); //拷贝构造

~Queue(); //析构

bool empty() const; //队空

bool get\_front(T& x)const; //获取首元素

bool append(const T x); //入队

bool serve(); //出队

private:

int count; //总量

node<T>\* front, \* rear; //头指针，尾指针

};

template<class T>

Queue<T>::Queue()

{

front = new node<T>; //头指针指向头结点，头结点无数据域

rear = front; //尾指针接到头指针上

front->next = NULL; //头指针质置空

count = 0;

}

template<class T>

Queue<T>::Queue(Queue<T>& queue)

{

front = new node<T>; //头指针指向头结点，头结点无数据域

rear = front; //尾指针接到头指针上

front->next = NULL; //头指针质置空

count = 0;

//深拷贝

node<T>\* ptr = queue.front;

while (ptr->next != NULL)

{

append(const T ptr->next->data);

ptr = ptr->next;

}

}

template<class T>

bool Queue<T>::empty()const

{

return rear == front;

}

template<class T>

bool Queue<T>::get\_front(T& x)const

{

if (empty())

return false;

x = front->next->data;

return true;

}

template<class T>

bool Queue<T>::append(const T x)

{

node<T>\* s = new node<T>; //创建新结点

s->data = x; //存入数据

s->next = NULL; //next置空

rear->next = s; //将新结点连接到尾结点的next

rear = s; //更新尾结点

count++; //更新总数量

return true;

}

template<class T>

bool Queue<T>::serve() //出队

{

node<T>\* temp;

if (empty())

return false;

temp = front->next; //暂存首结点

front->next = temp->next; //头指针连接第二个结点

delete temp; //删除头结点

count--; //更新总数量

if (front->next == NULL) //如果队空，则尾结点连接头结点

rear = front;

return true;

}

template<class T>

Queue<T>::~Queue()

{

while (!empty())serve();

delete front;

}

第三题

bool invert(); //倒置

template<class T>

bool List<T>::invert() //倒置

{

if\* (count == 0 || count == 1)

return false;

Node<T>\* getptr, //用于遍历原队列;

\* tempptr, //取得getptr;

\* headptr; //新顺序的头指针;

getptr = head->next; //旧顺序的头结点处变新顺序的尾结点

tempptr = getptr;

getptr = getptr->next;

tempptr->next = NULL;

headptr = tempptr;

rear = headptr; //更新尾指针

for (; getptr != NULL;) //倒置

{

tempptr = getptr;

getptr = getptr->next;

tempptr->next = headptr;

headptr = tempptr;

}

head->next = headptr; //头指针接到新顺序的第一个结点

return true;

}

第四题

源.cpp

#include"List.hpp"

#include"Stack.hpp"

int main()

{

List<int> list1;

List<int> list2;

Stack<int> stack;

List<int> list;

//初始化链表一

for (int i = 0; i < 10; i += 2)

list1.append(i);

cout << "list1: ";

for (int x = 0, int i = 1; i <= list1.length(); i++)

{

list1.get\_element(i, x);

cout << x << " ";

}

cout << endl;

//初始化链表二

for (int i = 1; i < 10; i += 2)

list2.append(i);

cout << "list2: ";

for (int x = 0, int i = 1; i <= list2.length(); i++)

{

list1.get\_element(i, x);

cout << x << " ";

}

cout << endl;

//合并链表一和链表二

int num1 = 1, num2 = 1, x1, x2, y;

while (num1 > list1.length() || num2 > list2.length())

{

list1.get\_element(num1, x1);

list2.get\_element(num2, x2);

y = (x1 > x2) ? x1 : x2;

(x1 > x2) ? (num1++) : (num2++);

stack.push(y);

}

//一格队列输出完成后，将另一个队列剩余的元素放入栈中

if (num1 > list1.length())

{

for (; num2 <= list2.length(); num2++)

{

list2.get\_element(num2, x2);

y = x2;

stack.push(y);

}

}

else

{

for (; num1 <= list1.length(); num1++)

{

list1.get\_element(num1, x1);

y = x1;

stack.push(y);

}

}

//放入新链表并输出

int temp;

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

{

stack.get\_top(temp);

list.append(temp);

}

cout << "list: ";

for (int x = 0, int i = 1; i <= list.length(); i++)

{

list1.get\_element(i, x);

cout << x << " ";

}

cout << endl;

return 0;

}

Stack.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct node

{

T data;

node<T>\* next;

};

template<class T>

class Stack //链栈

{

public:

Stack(); //默认构造函数

~Stack(); //析构函数

bool empty(); //判断栈空

bool get\_top(T& x); //取栈顶元素

bool push(const T x); //入栈

bool pop(); //出栈

int length(); //得到总数

private:

int count; //总数

node<T>\* top; //栈顶指针

};

template<class T>

Stack<T>::Stack()

{

count = 0;

top = NULL;

}

template<class T>

bool Stack<T>::empty()

{

return count == 0; //return top == NULL

}

template<class T>

bool Stack<T>::get\_top(T& x)

{

if (empty())

return false;

x = top->data;

return true;

}

template<class T>

bool Stack<T>::push(const T x)

{

node<T>\* s = new node<T>;

s->data = x;

s->next = top;

top = s;

count++;

return true;

}

template<class T>

bool Stack<T>::pop()

{

if (empty())

return false;

node<T>\* temp = top;

top = top->next;

delete temp;

count--;

return true;

}

template<class T>

Stack<T>::~Stack()

{

while (empty())

pop();

}

template<class T>

int Stack<T>::length()

{

return count;

}

List.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct Node

{

T data;

Node<T>\* next;

};

template<class T>

class List //单链表

{

public:

List(); //默认构造函数

List(List<T>& list); //拷贝构造函数

int length()const; //求长度

~List(); //析构函数

bool get\_element(const int i, T& x)const; //按序号取元素

Node<T>\* locate(const T x)const; //搜索元素

bool insert(const int i, const T x); //插入元素(第i个之前，原本的第i个元素变成第i+1个)

bool delete\_end(); //删除尾元素

Node<T>\* get\_head() { return head; } //读取链表表头指针

T get\_rear\_element() //读取链表表尾指针

{

return rear->data;

}

void copy(List A); //A链表复制到B链表

bool append(const T x); //尾增

bool isequal(List<T> A); //判断链表是否相等

private:

int count;

Node<T>\* head;

Node<T>\* rear;

};

template<class T>

List<T>::List()

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = 0;

}

template<class T>

List<T>::List(List<T>& list)

{

head = new Node<T>;

head->next = NULL;

rear = head;

count = 0;

Node<T>\* PA = list.get\_head()->next;

Node<T>\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node<T>\* s = new Node<T>; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

rear = s;

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

template<class T>

List<T>::~List()

{

while (head != rear)

{

Node<T>\* temp = head;

head = head->next;

delete temp;

}

delete head;

}

template<class T>

int List<T>::length() const

{

return count;

}

template<class T>

bool List<T>::get\_element(const int i, T& x)const

{

Node<T>\* p = head->next;

int j = 1;

while (p != NULL && j != i) //不是目标结点或不空，就继续搜索

{

p = p->next;

j++;

}

if (p == NULL)return false;

x = p->data;

return true;

}

template<class T>

Node<T>\* List<T>::locate(const T x)const

{

Node<T>\* p = head->next;

while (p != NULL)

{

if (p->data == x)return p;

else p = p->next;

}

return NULL;

}

template<class T>

bool List<T>::insert(const int i, const T x)

{

Node<T>\* p = head;

int j = 0;

for (; j != i - 1 && p != NULL; j++)

p = p->next;

if (i < 1 || i>count + 1) //等价与if(p == NULL)

return false;

Node<T>\* s = new Node<T>; //产生新结点

s->data = x;

s->next = p->next;

p->next = s;

count++;

return true;

}

template<class T>

bool List<T>::delete\_end()

{

if (count == 0)

return false;

Node<T>\* p = head;

while (p->next->next != NULL)

p = p->next;

rear = p;

delete p->next;

p->next = NULL;

return true;

}

template<class T>

void List<T>::copy(List<T> A)

{

Node<T>\* PA = A.get\_head()->next;

Node<T>\* PB = head; //设置当前表，即B表尾指针

while (PA != NULL) //在A表中还有元素时

{

Node<T>\* s = new Node<T>; //产生结点

s->data = PA->data; //复制结点的值

PB->next = s; // 插入复制的结点到表尾

PB = s; //重新指示表尾，以便下一个元素插入

count++; //继续复制A表的下一个元素

PA = PA->next;

PB->next = NULL; //将B表尾结点的后继指针置为空

}

}

template<class T>

bool List<T>::append(const T x)

{

Node<T>\* s = new Node<T>;

s->data = x;

s->next = NULL;

rear->next = s;

rear = s;

count++;

return true;

}

template<class T>

bool List<T>::isequal(List<T> A)

{

if (count != A.count)

return false;

else

{

Node<T>\* PA = head->next;

Node<T>\* PB = A.head->next;

while (PA != NULL)

{

if (PA->data != PB->data)

return false;

PA = PA->next;

PB = PB->next;

}

return true;

}

}

第五题

Stack.hpp

#pragma once

#include<iostream>

using namespace std;

template<class T>

struct node

{

T data;

node<T>\* next;

};

template<class T>

class Stack

{

public:

Stack(); //默认构造函数

~Stack(); //析构函数

bool empty(); //判断栈空

bool get\_top(T& x); //取栈顶元素

bool push(const T x); //入栈

bool pop(); //出栈

int length()const; //长度

bool isequal(Stack<T> s); //是否相等

node<T>\* getptop(){ return top; }//取栈顶结点指针

private:

int count; //总数

node<T>\* top; //栈顶指针

};

template<class T>

Stack<T>::Stack()

{

count = 0;

top = NULL;

}

template<class T>

bool Stack<T>::empty()

{

return count == 0; //return top == NULL

}

template<class T>

bool Stack<T>::get\_top(T& x)

{

if (empty())

return false;

x = top->data;

return true;

}

template<class T>

bool Stack<T>::push(const T x)

{

node<T>\* s = new node<T>;

s->data = x;

s->next = top;

top = s;

count++;

return true;

}

template<class T>

bool Stack<T>::pop()

{

if (empty())

return false;

node<T>\* temp = top;

top = top->next;

delete temp;

count--;

return true;

}

template<class T>

Stack<T>::~Stack()

{

while (empty())

pop();

}

template<class T>

int Stack<T>::length()const //长度

{

return count;

}

template<class T>

bool Stack<T>::isequal(Stack<T> s) //是否相等

{

if (count != s.count)

return false;

else

{

node<T>\* PA, \* PB;

PA = top;

PB = s.top;

if (count == 0)

return true;

else

{

while (PA != NULL)

{

if (PA->data != PB->data)

return false;

PA = PA->next;

PB = PB->next;

}

return true;

}

}

}

源.cpp

#include"Stack.hpp"

int main()

{

Stack<char> s1;

Stack<char> s2;

Stack<char> temp1;

Stack<char> temp2;

cout << "s1:";

for (int i = 0; i < 6; i++)

{

s1.push(char('a' + i));

cout << char('a' + i) << " ";

}

cout << endl;

s2.push('a');

s2.push('b');

s2.push('c');

s2.push('b');

s2.push('a');

cout << "s2: a b c b a" << endl;

cout << "temp1: ";

node<char>\* PA = s1.getptop();

for (char a, int i = 0; i < s1.length(); i++, PA = PA->next)

{

a = PA->data;

temp1.push(a);

cout << a << " ";

}

cout << endl;

if (s1.isequal(temp1))

cout << "s1 is." << endl;

else

cout << "s1 isn't." << endl;

cout << "temp2:";

node<char>\* PB = s2.getptop();

for (char a, int i = 0; i < s2.length(); i++, PB = PB->next)

{

a = PB->data;

temp2.push(a);

cout << a << " ";

}

cout << endl;

if (s2.isequal(temp2))

cout << "s2 is." << endl;

else

cout << "s2 isn't." << endl;

return 0;

}