第三次实验报告

实验题目

1. 利用数组实现双端栈（两栈共享空间）

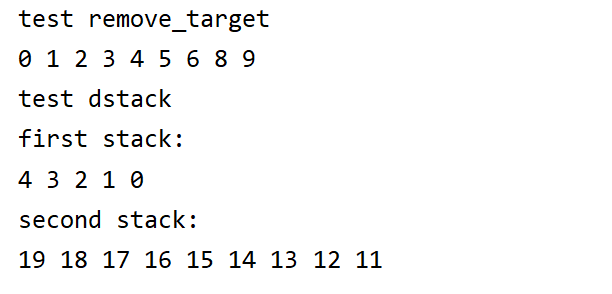
要求：给出双端栈的设计、实现，并验证所设计双端栈的正确性。

1. 设计一个函数实现在队列中删除第一个出现的target元素，并验证其正确性。

实验过程

1. 首先，为了以后数据结构课程和实验的方便，准备了两个庞大的对象node和dnode以及他们的辅助类，用于对单向链表和双向链表提供较为便捷的操作，这两个类在以后写生成树、约瑟夫环的问题时也是可以使用的。
2. i\_queue主要是实现队列容器的接口，list\_queue时队列的一种实现。
3. i\_stack主要是实现双端栈容器的接口，array\_stack,dqueue\_stack,list\_stack是他们的实现。
4. remove\_target使用了辅助队列，但由于其底层容器是单链表，所以并没有用到很多的辅助空间，这样操作也更加容易一些。主要是因为在remove\_target中，队列内部是不可见的，将队首元素插入尾部使用循环法删除元素是不可取的。

效果截图



实验代码

实验代码包括一下几个文件

main.cpp *# 程序入口*

node.h *# 单向链表结构和其辅助类*

dnode.h *# 双向链表结构和其辅助类*

i\_dstack.h *# 双向栈，包括三种实现模式(数组，双端队列，双向链表(自用))*

i\_queue.h *# 队列及其实现模式(单向链表)*

callbacks.h *# 回调函数，可以忽略*

main.cpp

main.cpp中,test\_remove\_target测试第2个小题，test\_dqueue测试第1个小题。

#include <iostream>

#include "i\_queue.h"

#include "i\_dstack.h"

using namespace std;

template <typename T>

void remove\_target(i\_queue<T>& queue, T target){

i\_queue<T>\* other = new list\_queue<T>();

bool find = false;

while(!queue.empty()){

auto front = queue.front();

if((!find && target!=front) || find){

other->push(front);

} else {

find = true;

}

queue.pop();

}

while(!other->empty()){

queue.push(other->front());

other->pop();

}

delete other;

}

template <typename T>

void read\_queue(i\_queue<T>& queue){

while(!queue.empty()){

cout << queue.front() << " " ;

queue.pop();

}

cout << endl;

}

void test\_remove\_target(){

cout << "test remove\_target" << endl;

i\_queue<int>\* queue = new list\_queue<int>();

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

queue->push(i);

}

remove\_target(\*queue,7);

read\_queue(\*queue);

delete queue;

}

void test\_dstack(){

cout << "test dstack" << endl;

i\_dstack<int>\* dstack = new array\_stack<int>();

for(int i = 0;i< 5 ;++i){

dstack->push(i);

}

for (int i = 11; i < 20; ++i) {

dstack->push(i,dstack\_direct::second);

}

cout << "first stack:" << endl;

while(!dstack->empty()){

cout << dstack->top() << " ";

dstack->pop();

}

cout << endl;

cout << "second stack:" << endl;

while(!dstack->empty(dstack\_direct::second)){

cout << dstack->top(dstack\_direct::second) << " ";

dstack->pop(dstack\_direct::second);

}

cout << endl;

}

int main(){

test\_remove\_target();

test\_dstack();

return 0;

}

node.h

*// ---- Foundation ----*

*// Created by cht.*

*// #relates*

*// node: 节点*

*// node\_fac: node生成器*

*// #endrelates*

#pragma once

#include<initializer\_list>

#include<stdexcept>

#include"callbacks.h"

using namespace std;

template <typename T>

class node {

public:

explicit node(T value, node<T>\* next = nullptr):value(value), next(next){}

T value;

node<T>\* next;

private:

};

class node\_fac{

public:

template <typename T>

static node<T>\* create(initializer\_list<T> data){

auto head = new node<T>(\*data.begin());

auto c = head;

for(auto p = data.begin() + 1; p < data.end(); ++p){

c = append(c,\*p);

}

return head;

}

template <typename T, typename T2>

static node<T>\* create(T2 begin, T2 end){

auto head = new node<T>(\*begin);

auto c = head;

for(auto p = begin+1;p< end;++p){

c = append(c,\*p);

}

return head;

}

template <typename T>

static node<T>\* append(node<T>\* pre, node<T>\* next){

pre->next = next;

return next;

}

template <typename T>

static node<T>\* append(node<T>\* pre, T value){

auto c = new node<T>(value);

return append(pre,c);

}

template <typename T>

static void insert(node<T>\* pre, node<T>\* next, node<T>\* insert){

pre->next = insert;

insert->next = next;

}

template <typename T>

static void insert(node<T>\* pre,node<T>\* next, T value){

auto c = new node<T>(value);

insert(pre,next,c);

}

template <typename T>

static void remove\_next(node<T>\* pre){

node<T>\* to\_remove = pre->next;

if(to\_remove == nullptr){

throw logic\_error("the removed node is nullptr");

}

node<T>\* next = to\_remove->next;

pre->next = next;

delete to\_remove;

}

template <typename T>

static node<T>\* offset(node<T>\* n,int length){

while(length > 0){

if(n == nullptr){

throw overflow\_error("node is nullptr in loop");

}

n=n->next;

--length;

}

return n;

}

template <typename T>

static node<T>\* remove\_where(node<T>\* head, node<T>\* to\_remove){

if(head == nullptr || to\_remove == nullptr){

throw logic\_error("node head/to\_remove is nullptr");

} else if(head == to\_remove){

auto c = head->next;

delete head;

return c;

} else {

auto c = head;

while(c!= nullptr){

auto p1 = c;

auto p2 = c->next;

if(p2 == to\_remove){

p1->next = p2->next;

delete p2;

return head;

}

c = c->next;

}

}

}

template <typename T>

static node<T>\* remove\_at(node<T>\* head, int index){

if(index < 0){

throw range\_error("index must be a optimistic number or zero.");

} else if(index == 0){

return remove\_where(head,head);

}

auto c = head;

while(index > 1){

if(c== nullptr){

throw overflow\_error("the removed node is nullptr");

}

c=c->next;

--index;

}

auto p1 = c->next;

auto p2 = p1->next;

c->next = p2;

delete p1;

return head;

}

template <typename T>

static void for\_any(node<T>\* head, action\_callback<T>& callback){

while(head!=nullptr){

callback.call(head->value);

head = head->next;

}

}

template <typename T>

static void destroy(node<T>\* head){

node<T>\* curr = head;

node<T>\* temp = nullptr;

while(curr != nullptr){

temp = curr;

curr = curr->next;

delete temp;

}

}

private:

};

dnode.h

*// ---- Foundation ----*

*// Created by cht.*

*// #relates*

*// dnode: 双端节点*

*// dnode\_fac: dnode工厂类*

*// #endrelates*

#pragma once

#include <stdexcept>

#include "callbacks.h"

using namespace std;

template <typename T>

class dnode{

public:

explicit dnode(T value, dnode<T>\* pre = nullptr, dnode<T>\* next = nullptr): value(value), pre(pre), next(next){

}

T value;

dnode<T>\* pre;

dnode<T>\* next;

private:

};

class dnode\_fac{

public:

template <typename T>

static dnode<T>\* create(initializer\_list<T> data){

auto first = new dnode<T>(\*data.begin());

auto c= first;

for(auto p = data.begin()+1; p != data.end(); ++p){

c= append\_back(c,\*p);

}

return first;

}

template <typename T>

static dnode<T>\* append\_back(dnode<T>\* pre, dnode<T>\* next){

pre->next = next;

next->pre = pre;

return next;

}

template <typename T>

static dnode<T>\* append\_pre(dnode<T>\* next, dnode<T>\* pre){

next->pre = pre;

pre->next = next;

return pre;

}

template <typename T>

static dnode<T>\* append\_back(dnode<T>\* pre, T value){

auto c = new dnode<T>(value);

return append\_back(pre,c);

}

template <typename T>

static dnode<T>\* append\_pre(dnode<T>\* next, T value){

auto c = new dnode<T>(value);

return append\_pre(next,c);

}

template <typename T>

static dnode<T>\* for\_any(dnode<T>\* first,action\_callback<T>& callback){

while(first!=nullptr){

callback.call(first->value);

first= first->next;

}

}

template<typename T>

static dnode<T>\* for\_any\_flag(dnode<T>\* first,action\_callback<T>& callback){

auto curr = first;

while(curr!= nullptr){

callback.call(curr->value);

curr = curr->next;

if(curr == first){

break;

}

}

}

template <typename T>

static bool unique(dnode<T>\* head){

return head->next == nullptr || head->next == head;

}

template<typename T>

static void join(dnode<T>\* pre, dnode<T>\* next){

pre->next = next;

next->pre = pre;

}

template <typename T>

static dnode<T>\* remove\_front(dnode<T>\* front){

auto c = front;

auto p = front->next;

if(p != nullptr){

p->pre = nullptr;

}

delete c;

return p;

}

template <typename T>

static dnode<T>\* remove\_back(dnode<T>\* back){

auto c= back;

auto p = back->pre;

if(p != nullptr){

p->next = nullptr;

}

delete c;

return p;

}

template <typename T>

static dnode<T>\* remove\_self\_return\_pre(dnode<T>\* self){

auto prev\_node = self->pre;

if(prev\_node == nullptr){

throw logic\_error("the prev\_node of self is nullptr");

}

auto next\_node = self->next;

prev\_node->next = next\_node;

if(next\_node != nullptr){

next\_node->pre = prev\_node;

}

delete self;

return prev\_node;

}

template <typename T>

static dnode<T>\* remove\_self\_return\_next(dnode<T>\* self){

auto prev\_node = self->pre;

auto next\_node = self->next;

if (next\_node == nullptr){

throw logic\_error("the next\_node of self is nullptr");

}

if(prev\_node != nullptr){

prev\_node->next = next\_node;

}

next\_node->pre = prev\_node;

delete self;

return next\_node;

}

template <typename T>

static void remove(dnode<T>\* self){

auto prev\_node = self->pre;

auto next\_node = self->next;

if(prev\_node != nullptr){

prev\_node -> next = next\_node;

}

if(next\_node != nullptr){

next\_node->pre = prev\_node;

}

delete self;

}

template <typename T>

static dnode<T>\* offset(dnode<T>\* n,int offset){

if(offset == 0){

} else if (offset > 0){

while(offset > 0){

n = n->next;

--offset;

}

} else {

while(offset < 0){

n = n ->pre;

++offset;

}

}

return n;

}

template <typename T>

static void destroy(dnode<T>\* head){

dnode<T>\* curr = head;

dnode<T>\* temp = nullptr;

while(curr!= nullptr){

temp = curr;

curr = curr->next;

if(curr == head){

break;

}

delete temp;

}

}

};

i\_dstack.h

i\_dstack包含了三种实现方法:array\_dstack,dqueue\_dstack,list\_dstack。三种实现都经过了测试。

*// ---- 上级题目3 ----*

*// Created by cht.*

*// 题目1: 使用数组实现双端栈*

*// 题目2: 使用deque实现双端栈*

*// 题目3: 使用链表实现双端栈*

*// #relates*

*// dstack\_direct: 表明调用的是哪一个栈*

*// i\_dstack: 双端栈*

*// array\_stack: 数组双端栈*

*// deque\_stack: 队列双端栈*

*// list\_stack: 双向链表栈*

*// #endrelates*

#pragma once

#include <stdexcept>

#include <deque>

#include "callbacks.h"

#include "dnode.h"

using namespace std;

*/\*\**

*\* 标识是第一个栈还是第二个栈。*

*\*/*

enum class dstack\_direct{

first = 1,

second = 2

};

template <typename T>

class i\_dstack{

public:

virtual ~i\_dstack()= default;

bool empty(dstack\_direct direct = dstack\_direct::first){

if(direct == dstack\_direct::first){

return empty\_first();

} else {

return empty\_second();

}

}

void push(T value, dstack\_direct direct = dstack\_direct::first){

if(direct == dstack\_direct::first){

push\_first(value);

} else {

push\_second(value);

}

}

void pop(dstack\_direct direct = dstack\_direct::first){

if(direct == dstack\_direct::first) {

pop\_first();

} else {

pop\_second();

}

}

T top(dstack\_direct direct = dstack\_direct::first){

if(direct == dstack\_direct::first) {

return peak\_first();

} else {

return peak\_second();

}

}

virtual bool empty\_first() const = 0;

virtual bool empty\_second() const = 0;

virtual void push\_first(T value) = 0;

virtual void push\_second(T value) = 0;

virtual void pop\_first() = 0;

virtual void pop\_second() = 0;

virtual T peak\_first() const = 0;

virtual T peak\_second() const = 0;

*//apply functions*

void show\_front(action\_callback<T>& callback){

while(!empty\_first()){

callback.call(peak\_first());

pop\_first();

}

}

void show\_back(action\_callback<T>& callback){

while(!empty\_second()){

callback.call(peak\_second());

pop\_second();

}

}

};

class limit\_container{

public:

virtual bool is\_full() const = 0;

};

template <typename T>

class array\_stack: public limit\_container,public i\_dstack<T>{

public:

~array\_stack() override {

delete container;

}

explicit array\_stack(int capacity = 100){

if(capacity <= 0){

throw logic\_error("capacity is too small");

}

container = new T[capacity];

this->capacity = capacity;

this->p\_front = 0;

this->p\_end = capacity - 1;

}

bool is\_full() const override {

return p\_front > p\_end;

}

bool empty\_first() const override {

return p\_front <= 0;

}

bool empty\_second() const override {

return p\_end >= capacity -1;

}

void push\_first(T value) override {

if(is\_full()){

throw logic\_error("array is full!");

}

container[p\_front++] = value;

}

void push\_second(T value) override {

if(is\_full()){

throw logic\_error("array is full!");

}

container[p\_end--] = value;

}

void pop\_first() override {

if(empty\_first()){

throw logic\_error("stack first is empty!");

}

--p\_front;

}

void pop\_second() override {

if(empty\_second()){

throw logic\_error("stack second is empty!");

}

++p\_end;

}

T peak\_first() const override {

if(empty\_first()){

throw logic\_error("stack first is empty!");

}

return container[p\_front-1];

}

T peak\_second() const override {

if (empty\_second()) {

throw logic\_error("stack second is empty");

}

return container[p\_end + 1];

}

private:

T\* container = nullptr;

int capacity = 0;

int p\_front = 0;

int p\_end = 0;

};

template <typename T>

class deque\_stack: public i\_dstack<T> {

public:

~deque\_stack() override = default;

bool empty\_first() const override {

return size\_first == 0;

}

bool empty\_second() const override {

return size\_second == 0;

}

void push\_first(T value) override {

container.push\_front(value);

++size\_first;

}

void push\_second(T value) override {

container.push\_back(value);

++size\_second;

}

void pop\_first() override {

if(empty\_first()){

throw logic\_error("stack first is empty! on pop");

}

container.pop\_front();

--size\_first;

}

void pop\_second() override {

if(empty\_second()){

throw logic\_error("stack second is empty! on pop");

}

container.pop\_back();

--size\_second;

}

T peak\_first() const override {

if(empty\_first()){

throw logic\_error("stack first is empty! on peak");

}

return \*container.begin();

}

T peak\_second() const override {

if(empty\_second()){

throw logic\_error("stack second is empty! on peak");

}

return \*container.end();

}

private:

deque<T> container;

int size\_first = 0;

int size\_second = 0;

};

template <typename T>

class list\_stack: public i\_dstack<T>{

public:

~list\_stack() override {

dnode\_fac::destroy(first);

}

bool empty\_first() const override {

return first\_bottom == nullptr;

}

bool empty\_second() const override {

return second\_bottom == nullptr;

}

void push\_first(T value) override {

if (first\_bottom == nullptr) {

if (second\_bottom != nullptr) {

first\_bottom = dnode\_fac::append\_pre(second\_bottom,value);

} else {

first\_bottom = new dnode<T>(value);

}

first = first\_bottom;

} else {

first = dnode\_fac::append\_pre(first,value);

}

}

void push\_second(T value) override {

if (second\_bottom == nullptr) {

if (first\_bottom != nullptr) {

second\_bottom = dnode\_fac::append\_back(first\_bottom, value);

} else {

second\_bottom = new dnode<T>(value);

}

second = second\_bottom;

} else {

second = dnode\_fac::append\_back(second,value);

}

}

T peak\_first() const override {

if (first == nullptr) {

throw range\_error("stack first is empty! on peak");

}

return first->value;

}

T peak\_second() const override {

if (second == nullptr) {

throw range\_error("stack second is empty! on peak");

}

return second->value;

}

void pop\_first() override {

if (first == nullptr) {

throw range\_error("stack first is empty! on pop");

}

auto c = first;

first = dnode\_fac::remove\_front(first);

if(first == nullptr){

first\_bottom = nullptr;

}

}

void pop\_second() override {

if (second == nullptr) {

throw range\_error("stack first is empty on pop");

}

auto c = second;

second = dnode\_fac::remove\_back(second);

if(second == nullptr){

second\_bottom = nullptr;

}

}

private:

dnode<T>\* first = nullptr;

dnode<T>\* second = nullptr;

dnode<T>\* first\_bottom = nullptr;

dnode<T>\* second\_bottom = nullptr;

};

i\_queue.h

i\_queue目前包含list\_queue的实现。

*// ---- 上级题目3 ----*

*// Created by cht.*

*// 题目1: 使用数组实现双端栈*

*// 题目2: 使用deque实现双端栈*

*// 题目3: 使用链表实现双端栈*

*// #relates*

*// i\_queue: 队列容器*

*// list\_queue: 链表队列*

*// #endrelates*

#include "node.h"

#include <stdexcept>

#pragma once

template <typename T>

class i\_queue{

public:

virtual ~i\_queue()= default;

virtual void push(T value) = 0;

virtual T front() const = 0;

virtual void pop() = 0;

virtual bool empty() const = 0;

};

template <typename T>

class list\_queue: public i\_queue<T>{

public:

list\_queue(): head(nullptr), tail(nullptr){

}

~list\_queue(){

node\_fac::destroy(head);

}

void push(T value) override {

if(head == nullptr){

head = new node<T>(value);

tail = head;

} else {

tail = node\_fac::append(tail,value);

}

}

T front() const override {

if(empty()){

throw range\_error("the queue is empty");

} else {

return head->value;

}

}

void pop() override {

if(empty()){

throw range\_error("the queue is empty");

} else {

head = node\_fac::remove\_where(head,head);

}

}

bool empty() const override {

return head == nullptr;

}

private:

node<T>\* head = nullptr;

node<T>\* tail = nullptr;

};

callbacks.h

回调接口，用于测试用的。

*// -- Foundation ---*

*// 回调表达式*

*// created be cht*

*// #relate*

*// #endrelate*

#pragma once

#include <iostream>

#include "flags.h"

using namespace std;

typedef void (\*action)();

class callback {

};

template <typename T>

class action\_callback: virtual public callback{

public:

virtual void call(T arg) = 0;

};

template <typename T, typename Tout>

class func1\_callback:virtual public callback{

public:

virtual Tout call(T arg) = 0;

};

template <typename T1, typename T2,typename Tout>

class func2\_callback:virtual public callback{

public:

virtual Tout call(T1 arg1, T2 arg2) = 0;

};

*//frequent usages.*

template <typename T>

class comparer\_callback:virtual public func2\_callback<T,T,bool>{

};

template <typename T>

class filter\_callback:virtual public func1\_callback<T,bool>{

};

*//defaults.*

template <typename T>

class print\_callback:virtual public action\_callback<T>{

public:

void call(T arg) override {

cout << arg << endl;

}

};

template <typename T>

class print\_space\_callback:virtual public action\_callback<T>{

public:

void call(T arg) override {

cout << arg << " ";

}

};