МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №8**

по курсу “Объектно-ориентированное программирование”

I семестр, 2021/22 учебный год

Студент: *Пономарев Никита Владимирович, группа М8О-207Б-20*

Преподаватель: *Дорохов Евгений Павлович, каф. 806*

**Задание:**

Используя структуру данных, разработанную для лабораторной работы №7, спроектировать и разработать аллокатор памяти для динамической структуры данных. Целью построения аллокатора является минимизация вызова операции malloc.

Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Аллокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианту задания).

Для вызова аллокатора должны быть переопределены операторы new и delete у классов-фигур.

**Вариант №19:**

* + Фигура: Квадрат (Square)
  + Контейнер первого уровня: Связный список (TLinkedList)
  + Контейнер второго уровня: Приоритетная очередь (TPriorityQueue)

**Описание программы:**

**Вариант №19:**

* + Фигура: Прямоугольник(Rectangle)
  + Контейнер первого уровня: Н-дерево (TnaryTree)
  + Контейнер второго уровня: Вектор(TМector)

**Описание программы:**

Исходный код разделён на 9 файлов:

1. point.h – описание класса точки
2. point.cpp – реализация класса точки
3. rectangle.h – описание класса квадрата
4. rectangle.cpp – реализация класса квадрата
5. TNaryTree\_item.h – описание элемента н-дерева с шаблонами
6. TNaryTree\_item.cpp – реализация элемента н-дерева с шаблонами
7. TNaryTree.h – описание н-дерева с шаблонами
8. TNaryTree.cpp – реализация н-дерева с шаблонами
9. Iterator.h – описание и реализация итератора н-дерева с шаблонами
10. TAllocatorBlock.h: реализация класса алокатора н-дерева
11. TVector.h: реализация класса шаблонного вектора для использования в аллокаторе
12. TVector\_item.cpp: реализация класса элемента шаблонного вектора для использования в аллокаторе
13. main.cpp – основная программа (для тестирования)

**Дневник отладки:**

При выполнении работы ошибок выявлено не было.

**Вывод:**  
 В процессе выполнения работы я на практике познакомился с понятием аллокатора. Так как во многих структурах данных используются аллокаторы, то это очень важная тема, которую должен знать каждый программист на С++. Написание собственноручного итератора помогает реализовать собственную логику выделения памяти, которая может быть более оправданной в некоторых ситуациях, чем стандартный аллокатор, как для самописных, так и для стандартных структур данных.

**Исходный код:**

**point.h:**

#ifndef POINT\_H

#define POINT\_H

#include <iostream>

class Point {

public:

Point();

Point(std::istream &is);

Point(double x, double y);

double dist(Point& other);

double X();

double Y();

friend std::istream& operator>>(std::istream& is, Point& p);

friend std::ostream& operator<<(std::ostream& os, const Point& p);

private:

double x\_;

double y\_;

};

#endif // POINT\_H

**point.cpp:**

#include "point.h"

#include <cmath>

Point::Point() : x\_(0.0), y\_(0.0) {}

Point::Point(double x, double y) : x\_(x), y\_(y) {}

Point::Point(std::istream &is) {

is >> x\_ >> y\_;

}

double Point::dist(Point& other) {

double dx = (other.x\_ - x\_);

double dy = (other.y\_ - y\_);

return std::sqrt(dx\*dx + dy\*dy);

}

double Point::X(){

return x\_;

};

double Point::Y(){

return y\_;

};

std::istream& operator>>(std::istream& is, Point& p) {

is >> p.x\_ >> p.y\_;

return is;

}

std::ostream& operator<<(std::ostream& os, const Point& p) {

os << "(" << p.x\_ << ", " << p.y\_ << ")";

return os;

}

**rectangle.h:**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include "figure.h"

class Rectangle: Figure {

public:

size\_t VertexesNumber();

double Area();

void Print(std::ostream& os);

Rectangle();

Rectangle(Point a\_, Point b\_, Point c\_, Point d\_);

Rectangle(std::istream& is);

friend std::istream &operator>>(std::istream &is, Rectangle &figure);

friend std::ostream &operator<<(std::ostream &os, const Rectangle &figure);

private:

Point a;

Point b;

Point c;

Point d;

};

#endif

**rectangle.cpp:**

#include "point.h"

#include "rectangle.h"

double Rectangle::Area(){

return a.dist(b) \* b.dist(c);

}

void Rectangle::Print(std::ostream& os){

os << a << " " << b << " " << c << " " << d << "**\n**";

}

size\_t Rectangle::VertexesNumber(){

return (size\_t)(4);

}

Rectangle::Rectangle() : a(Point()), b(Point()), c(Point()), d(Point()){

}

Rectangle::Rectangle(Point a\_, Point b\_, Point c\_, Point d\_):

a(a\_), b(b\_), c(c\_), d(d\_){

}

Rectangle::Rectangle(std::istream& is){

is >> a >> b >> c >> d;

}

std::istream &operator>>(std::istream &is, Rectangle &figure){

is >> figure.a >> figure.b >> figure.c >> figure.d;

return is;

}

std::ostream &operator<<(std::ostream &os, const Rectangle &figure){

os << "Rectangle: " << figure.a << " " << figure.b << " " << figure.c << " " << figure.d << std::endl;

return os;

}

**TNaryTree\_item.h:**

#ifndef TNARYTREE\_ITEM\_H

#define TNARYTREE\_ITEM\_H

#include <memory>

template <class T>

class Item {

public:

Item(T a);

Item(std::shared\_ptr<Item<T>> a);

Item();

void Set(T a);

void Set\_bro(std::shared\_ptr<Item<T>> bro\_);

void Set\_son(std::shared\_ptr<Item<T>> son\_);

Item Get\_data();

std::shared\_ptr<Item<T>> Get\_bro();

std::shared\_ptr<Item<T>> Get\_son();

void Print(std::ostream &os);

double Area();

~Item();

template<class A>

friend std::ostream &operator<<(std::ostream &os, const Item<A> &obj);

private:

std::shared\_ptr<Item<T>> bro = nullptr;

std::shared\_ptr<Item<T>> son = nullptr;

T data;

};

#endif

**TNaryTree\_item.cpp:**

#include "TNaryTree\_item.h"

#include <iostream>

template <class T>

Item<T>::Item() {

data = T();

}

template <class T>

Item<T>::Item(T a){

data = a;

}

template <class T>

void Item<T>::Set(T a){

data = a;

}

template <class T>

Item<T> Item<T>::Get\_data(){

return data;

}

template <class T>

std::shared\_ptr<Item<T>> Item<T>::Get\_bro(){

return bro;

}

template <class T>

std::shared\_ptr<Item<T>> Item<T>::Get\_son(){

return son;

}

template <class T>

Item<T>::Item(std::shared\_ptr<Item<T>> a){

bro = a->bro;

son = a->son;

data = a->data;

}

template <class T>

void Item<T>::Print(std::ostream &os){

os << data.Area();

}

template <class T>

void Item<T>::Set\_bro(std::shared\_ptr<Item<T>> bro\_){

bro = bro\_;

}

template <class T>

void Item<T>::Set\_son(std::shared\_ptr<Item<T>> son\_){

son = son\_;

}

template <class T>

double Item<T>::Area(){

return data.Area();

}

template <class T>

std::ostream &operator<<(std::ostream &os, const Item<T> &obj)

{

os << "Item: " << obj.data << std::endl;

return os;

}

template <class T>

Item<T>::~Item() {};

#include "rectangle.h"

template class Item<Rectangle>;

template std::ostream& operator<<(std::ostream& os, const Item<Rectangle> &obj);

**TNaryTree.h:**

#ifndef TNARYTREE\_H

#define TNARYTREE\_H

#include "TnaryTree\_item.h"

#include "Iterator.h"

template <class T>

class TNaryTree

{

public:

// Инициализация дерева с указанием размера

TNaryTree(int n);

// Полное копирование дерева

TNaryTree(const TNaryTree<T>& other);

// Добавление или обновление вершины в дереве согласно заданному пути.

// Путь задается строкой вида: "cbccbccc",

// где 'c' - старший ребенок, 'b' - младший брат

// последний символ строки - вершина, которую нужно добавить или обновить.

// Пустой путь "" означает добавление/обновление корня дерева.

// Если какой-то вершины в tree\_path не существует,

// то функция должна бросить исключение std::invalid\_argument

// Если вершину нельзя добавить из за переполнения,

// то функция должна бросить исключение std::out\_of\_range

void Update(T &&polygon, std::string &&tree\_path = "");

// Удаление поддерева

void Clear(std::string &&tree\_path = "");

// Проверка наличия в дереве вершин

bool Empty();

// Подсчет суммарной площади поддерева

double Area(std::string &&tree\_path);

int size();

// Вывод дерева в формате вложенных списков, где каждый вложенный список является:

// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры

template <class A>

friend std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree);

virtual ~TNaryTree();

private:

int curr\_number;

int max\_number;

std::shared\_ptr<Item<T>> root;

};

#endif

**TNaryTree.cpp:**

#include "TNaryTree.h"

#include <string>

#include <memory>

#include <stdexcept>

#include <iostream>

template <class T>

TNaryTree<T>::TNaryTree(int n) {

max\_number = n;

curr\_number = 0;

root = nullptr;

};

template <class T>

bool TNaryTree<T>::Empty() {

return curr\_number ? 0 : 1;

}

template <class T>

void TNaryTree<T>::Update(T &&polygon, std::string &&tree\_path){

if(tree\_path != "" && curr\_number == 0){

throw std::invalid\_argument("Error, there is not a root value**\n**");

return;

} else if(tree\_path == "" && curr\_number == 0){

std::shared\_ptr<Item<T>> q(new Item<T>(polygon));

root = q;

curr\_number++;

} else if(curr\_number + 1 > max\_number){

throw std::out\_of\_range("Current number of elements equals maximal number of elements in tree**\n**");

return;

} else {

std::shared\_ptr<Item<T>> tmp = root;

for(size\_t i = 0; i < tree\_path.length() - 1; i++) {

if(tree\_path[i] == 'b'){

std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

tmp = q;

} else if(tree\_path[i] == 'c'){

std::shared\_ptr<Item<T>> q = (\*tmp).Get\_son();

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

tmp = q;

} else {

throw std::invalid\_argument("Error in path**\n**");

return;

}

}

std::shared\_ptr<Item<T>> item(new Item<T>(polygon));

if(tree\_path.back() == 'b'){

*/\*std::shared\_ptr<Item> p = (\*tmp).Get\_bro();*

*p = item;\*/*

(\*tmp).Set\_bro(item);

curr\_number++;

} else if(tree\_path.back() == 'c'){

*/\*std::shared\_ptr<Item> p = (\*tmp).Get\_son();*

*p = item;\*/*

(\*tmp).Set\_son(item);

curr\_number++;

} else {

throw std::invalid\_argument("Error in path**\n**");

return;

}

}

}

template <class T>

std::shared\_ptr<Item<T>> copy(std::shared\_ptr<Item<T>> root){

if(!root){

return nullptr;

}

std::shared\_ptr<Item<T>> root\_copy(new Item<T>(root));

(\*root\_copy).Set\_bro(copy((\*root).Get\_bro()));

(\*root\_copy).Set\_son(copy((\*root).Get\_son()));

return root\_copy;

}

template <class T>

TNaryTree<T>::TNaryTree(const TNaryTree<T>& other){

curr\_number = 0;

max\_number = other.max\_number;

root = copy(other.root);

curr\_number = other.curr\_number;

;}

template <class T>

int TNaryTree<T>::size(){

return curr\_number;

}

template <class T>

int clear(std::shared\_ptr<Item<T>> node) {

if (!node) {

return 0;

}

int temp\_res = clear((\*node).Get\_bro()) + clear((\*node).Get\_son()) + 1;

return temp\_res;

}

template <class T>

void TNaryTree<T>::Clear(std::string &&tree\_path){

std::shared\_ptr<Item<T>> prev\_tmp = nullptr;

std::shared\_ptr<Item<T>> tmp;

tmp = root;

if (tree\_path.empty()) {

clear(root);

curr\_number = 0;

root = nullptr;

return;

}

for(size\_t i = 0; i < tree\_path.length(); i++) {

if(tree\_path[i] == 'b'){

std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

prev\_tmp = tmp;

tmp = q;

} else if(tree\_path[i] == 'c'){

std::shared\_ptr<Item<T>> q((\*tmp).Get\_son());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return;

}

prev\_tmp = tmp;

tmp = q;

} else {

throw std::invalid\_argument("Error in path**\n**");

return;

}

}

if (tmp == (\*prev\_tmp).Get\_son()) {

(\*prev\_tmp).Set\_son(nullptr);

} else {

(\*prev\_tmp).Set\_bro(nullptr);

}

curr\_number -= clear(tmp);

}

template <class T>

double area(std::shared\_ptr<Item<T>> node){

if(!node){

return 0;

}

return node->Area() + area((\*node).Get\_bro()) + area((\*node).Get\_son());

}

template <class T>

double TNaryTree<T>::Area(std::string &&tree\_path){

std::shared\_ptr<Item<T>> tmp;

tmp = root;

for(size\_t i = 0; i < tree\_path.length(); i++) {

if(tree\_path[i] == 'b'){

std::shared\_ptr<Item<T>> q((\*tmp).Get\_bro());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return -1;

}

tmp = q;

} else if(tree\_path[i] == 'c'){

std::shared\_ptr<Item<T>> q((\*tmp).Get\_son());

if(q == nullptr){

throw std::invalid\_argument("Path does not exist**\n**");

return -1;

}

tmp = q;

} else {

throw std::invalid\_argument("Error in path**\n**");

return -1;

}

}

return area(tmp);

}

// Вывод дерева в формате вложенных списков, где каждый вложенный список является:

// "S0: [S1: [S3, S4: [S5, S6]], S2]", где Si - площадь фигуры

template <class T>

void print(std::ostream& os, std::shared\_ptr<Item<T>> node){

if(!node){

return;

}

if((\*node).Get\_son()){

//os << <<node->pentagon.GetArea() << : ]" <<

os << node->Area() << ": [";

print(os, (\*node).Get\_son());

os << "]";

if((\*node).Get\_bro()){

os << ", ";

print(os, (\*node).Get\_bro());

}

} else if ((\*node).Get\_bro()) {

os << node->Area() << ", ";

print(os, (\*node).Get\_bro());

if((\*node).Get\_son()){

os << ": [";

print(os, (\*node).Get\_son());

os << "]";

}

}

else {

os << node->Area();

}

}

template <class T>

std::ostream& operator<<(std::ostream& os, const TNaryTree<T>& tree){

print(os, tree.root);

os << "**\n**";

return os;

}

template <class T>

TNaryTree<T>::~TNaryTree(){

Clear();

};

#include "rectangle.h"

template class TNaryTree<Rectangle>;

template std::ostream& operator<<(std::ostream& os, const TNaryTree<Rectangle>& stack);

**Iterator.h:**

#ifndef ITERATOR\_H

#define ITERATOR\_H

#include "TNaryTree.h"

#include <iostream>

template <class node, class T>

class Iterator{

public:

Iterator(std::shared\_ptr<node> n){

it = n;

it\_prev = nullptr;

}

Iterator& operator=(const Iterator& it\_){

it = it\_.it;

return \*this;

}

bool operator==(const Iterator& it\_){

return it == it\_.it;

}

bool operator!=(const Iterator& it\_){

return !(it == it\_.it);

}

node operator\*(){

return it->Get\_data();

}

Iterator& operator++ (){

if(it->Get\_son() != nullptr && it\_prev != it->Get\_bro()){

it\_prev = it;

it = it->Get\_son();

} else if (it->Get\_bro() != nullptr && it\_prev != it->Get\_bro()){

it\_prev = it;

it = it->Get\_bro();

} else {

while(it->Get\_bro() == nullptr || (it->Get\_bro() == it\_prev)){

it\_prev = it;

if(it->Get\_older() == nullptr){

return \*this;

}

it = it->Get\_older();

}

if(it->Get\_bro() == nullptr && it->Get\_son() == it\_prev){

it\_prev = it;

it = it->Get\_older();

if(it->Get\_older() == nullptr){

return \*this;

}

if(it->Get\_bro() == nullptr){

while(it->Get\_bro() == nullptr){

it\_prev = it;

it = it->Get\_older();

}

it\_prev = it;

it = it->Get\_bro();

} else {

it = it->Get\_bro();

}

} else {

it = it->Get\_bro();

}

}

return \*this;

}

Iterator& operator++ (int){

Iterator tmp(\*it);

++(\*this);

return it;

}

Iterator& operator-- (){

it = it.Get\_older();

return \*this;

}

Iterator& operator-- (int){

Iterator tmp(\*it);

--(\*this);

return it;

}

public:

std::shared\_ptr<node> it\_prev;

std::shared\_ptr<node> it;

};

#endif

**Tvector.h:**

#ifndef VECTOR\_H

#define VECTOR\_H

#include "TVector\_item.h"

#include <memory>

#include <iostream>

template<typename T>

class Vector{

public:

Vector(): length(0), head(nullptr) {};

void push\_back(T t){

if(length == 0){

head = std::make\_shared<Vector\_item<T>>(new(Vector\_item<T>));

} else {

Vector\_item<T> tmp;

while(tmp.Get\_next() != nullptr){

tmp = \*tmp.Get\_next();

}

}

++length;

}

int size() const {

return length;

}

void erase(int ind){

while(ind--){

head = head->Get\_next();

}

if(length > 0){

--length;

} else {

std::cout << "Error in delete element from vector!**\n**";

}

}

std::shared\_ptr<Vector\_item<T>> Get\_first(){

return head;

}

private:

std::shared\_ptr<Vector\_item<T>> head;

int length;

};

#endif

**Tvector\_item.h:**

#ifndef VECTOR\_ITEM\_H

#define VECTOR\_ITEM\_H

#include <memory>

template<typename T>

class Vector\_item {

public:

Vector\_item(): data(0) {};

Vector\_item(T t): data(t){};

std::shared\_ptr<Vector\_item<T>> Get\_next(){

return next;

};

void Set\_next(std::shared\_ptr<Vector\_item<T>> next\_){

next = next\_;

};

T Get\_data(){

return data;

}

private:

std::shared\_ptr<Vector\_item<T>> next = nullptr;

T data;

};

#endif

TallocatorBlock.cpp:

#ifndef TALLOCATORBLOCK\_H

#define TALLOCATORBLOCK\_H

#include "TVector.h"

#include <memory>

class TAllocatorBlock {

public:

TAllocatorBlock(const size\_t& size, const size\_t count){

this->size = size;

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

unused\_blocks.push\_back(malloc(size));

}

}

void\* Allocate(const size\_t& size){

if(size != this->size){

std::cout << "Error during allocation**\n**";

}

if(unused\_blocks.size()){

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

unused\_blocks.push\_back(malloc(size));

}

}

void\* tmp = unused\_blocks.Get\_first()->Get\_data();

used\_blocks.push\_back(unused\_blocks.Get\_first()->Get\_data());

unused\_blocks.erase(0);

return tmp;

}

void Deallocate(void\* ptr){

unused\_blocks.push\_back(ptr);

}

~TAllocatorBlock(){

while(used\_blocks.size()){

try{

free(used\_blocks.Get\_first()->Get\_data());

used\_blocks.erase(0);

} catch(...){

used\_blocks.erase(0);

}

}

while(unused\_blocks.size()){

try{

free(unused\_blocks.Get\_first()->Get\_data());

unused\_blocks.erase(0);

} catch(...){

unused\_blocks.erase(0);

}

}

}

private:

size\_t size;

Vector<void\*> used\_blocks;

Vector<void\*> unused\_blocks;

};

#endif

**main.cpp:**

#include "TNaryTree.h"

#include "rectangle.h"

#include "TAllocatorBlock.h"

int main(void)

{

TNaryTree<Rectangle> t(5);

t.Update(Rectangle(Point(0, 0), Point(1, 0),Point(1, 2), Point(0, 2)), "");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "b");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bb");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "bbc");

t.Update(Rectangle(Point(0, 0), Point(4, 0),Point(4, 1), Point(0, 1)), "c");

std::cout << t.size() << "**\n**";

std::cout << t.Area("") << "**\n**";

std::cout << t.size() << "**\n**";

std::cout << t;

TNaryTree<Rectangle> q(t);

t.Clear();

std::cout << q.size() << " " << q.Area("") << "**\n**";

std::cout << q;

return 0;

}

**Результат работы:**

5

17

5

5 17

1: [4], 4, 4: [4]

1: [4], 4, 4: [4]

0