

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ  
ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ  
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №8**  
по курсу объектно-ориентированное программирование I семестр, 2021/22  
уч. год

Студент: Сафонникова Анна Романовна, группа М8О-207Б-20

Преподаватель: Дорохов Евгений Павлович

## Условие

Вариант 22: N-Дерево.

Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – 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.

Алокатор 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 оператор `new` и `delete` у классов-фигур.

Нельзя использовать:

- Стандартные контейнеры `std`.

Программа должна позволять:

- ▷ Вводить произвольное количество фигур и добавлять их в контейнер.
- ▷ Распечатывать содержимое контейнера.
- ▷ Удалять фигуры из контейнера.

## Описание программы

Исходный код лежит в следующих файлах:

1. `main.cpp`: тестирование кода
2. `figure.h`: родительский класс интерфейс для фигур(любых)
3. `pentagon.h`: описание класса 5-угольника, наследующегося от `figures`
4. `hexagon.h`: описание класса 6-угольника, наследующегося от `figures`
5. `octagon.h`: описание класса 8-угольника, наследующегося от `figures`
6. `pentagon.cpp`: реализация класса треугольника, наследующегося от `figures`
7. `hexagon.cpp`: реализация класса прямоугольника, наследующегося от `figures`
8. `octagon.cpp`: реализация класса квадрата, наследующегося от `figures`
9. `TNaryTree.h`: структура общего дерева
10. `Item.h`: структура элемента дерева
11. `TNaryTree.cpp`: реализация общего дерева

12. Item.cpp: реализация элемента дерева
13. Iterator.h: описание и реализация итераторов
14. TAllocationBlock.h: описание аллокатора
15. TAllocationBlock.cpp: реализация аллокатора
16. TStack.h: описание стека
17. TStack.cpp: реализация стека
18. TStackItem.h: описание итератора
19. TStackItem.cpp: реализация итератора

## **Дневник отладки**

Ошибок не было.

## **Недочёты**

Недочётов не заметила.

## **Вывод**

В данной лабораторной работе я реализовала аллокатор класса. Благодаря аллокаторам мы можем придумать свои правила для выделения памяти и снизить количество вызовов операции malloc.

## Исходный код

### main.cpp

```
#include <cstdlib>
#include <iostream>
#include <memory>
#include <cstdint>

#include "Pentagon.h"
#include "Hexagon.h"
#include "Octagon.h"
#include "Item.h"
#include "TNaryTree.h"
#include "Figure.h"

void TestTree() {
    TNaryTree<Figure> tree;

    tree.Insert(std::make_shared<Pentagon>(5),1,1);
    tree.Insert(std::make_shared<Pentagon>(4),1,2);
    tree.Insert(std::make_shared<Hexagon>(3),1,3);
    tree.Insert(std::make_shared<Octagon>(2),2,4);

    for (auto i : tree) {
        i->Print();
        std::cout << std::endl;
    }

    tree.remove(1);
}

void TestAllocationBlock()
{
    TAllocationBlock allocator(sizeof(int),10);

    int *a1=nullptr;
    int *a2=nullptr;
    int *a3=nullptr;
    int *a4=nullptr;
    int *a5=nullptr;

    a1 = (int*)allocator.Allocate();
```

```

*a1 =1;
std::cout << "a1 pointer value:" << *a1 << std::endl;
a2 = (int*)allocator.Allocate();
*a2 =2;
std::cout << "a2 pointer value:" << *a2 << std::endl;
a3 = (int*)allocator.Allocate();
*a3 =3;
std::cout << "a3 pointer value:" << *a3 << std::endl;

allocator.Deallocate(a1);
allocator.Deallocate(a3);

a4 = (int*)allocator.Allocate();
*a4 =4;
std::cout << "a4 pointer value:" << *a4 << std::endl;
a5 = (int*)allocator.Allocate();
*a5 =5;
std::cout << "a5 pointer value:" << *a5 << std::endl;
std::cout << "a1 pointer value:" << *a1 << std::endl;
std::cout << "a2 pointer value:" << *a2 << std::endl;
std::cout << "a3 pointer value:" << *a3 << std::endl;

allocator.Deallocate(a2);
allocator.Deallocate(a4);
allocator.Deallocate(a5);
}

void help(){
    std::cout << "Press 1 to get help" << std::endl;
    std::cout << "Press 2 to add figure in tree" << std::endl;
    std::cout << "Press 3 to get a number of nodes on level" << std::endl;
    std::cout << "Press 4 to print tree" << std::endl;
    std::cout << "Press 5 to delete figure from tree" << std::endl;
    std::cout << "Press 6 to get demo iterator" << std::endl;
    std::cout << "Press 0 to exit" << std::endl;
}

int main(int argc, char** argv) {

    TestAllocationBlock();
    TestTree();
    system("pause");
}

```

```
        return 0;
    }
```

## figure.h

```
#ifndef OOP_LAB4_FIGURE_H
#define OOP_LAB4_FIGURE_H

#include <stdlib.h>
#include <iostream>
class Figure {
public:
    virtual size_t GetSide() = 0;
    virtual double Square() = 0;
    virtual void Print() = 0;
    virtual ~Figure(){};
};

#endif //OOP_LAB4_FIGURE_H
```

## Hexagon.cpp

```
#include "Hexagon.h"
#include <iostream>
#include <cmath>

Hexagon::Hexagon() :Hexagon(0) {

}

Hexagon::Hexagon(size_t i) : side_a(i){
    std::cout << "Hexagon created: " << side_a << std::endl;
}

double Hexagon::Square() {
    return 1.5 * pow(side_a,2) * sqrt(3);
}

void Hexagon::Print() {
    std::cout << "Hexagon with side = " << side_a;
}
```

```

size_t Hexagon::GetSide(){
    return side_a;
}

Hexagon::~Hexagon() {}

Hexagon::Hexagon(std::istream &is) {
    while (true) {
        is >> *this;
        if (std::cin.peek() == '\n') {
            std::cin.get();
            break;
        }
        else {
            std::cerr << "Error: insert data" << "\n";
            std::cin.clear();
            while (std::cin.get() != '\n') {}
        }
    }
}

Hexagon &Hexagon::operator=(const Hexagon &obj) {
    if (this == &obj) return *this;
    side_a = obj.side_a;
    return *this;
}

bool Hexagon::operator==(const Hexagon &obj) const {
    return side_a == obj.side_a;
}

bool Hexagon::operator!=(const Hexagon &obj) const {
    return side_a != obj.side_a;
}

std::ostream &operator<<(std::ostream &os, const Hexagon &obj) {
    os << "Hexagon with side: " << obj.side_a << "\n";
    return os;
}

```

```

std::istream &operator>>(std::istream &is, Hexagon &obj) {
    std::cout << "Insert side of Hexagon: \n";
    is >> obj.side_a;
    return is;
}

```

## Hexagon.h

```

#ifndef OOP_LAB4_HEXAGON_H
#define OOP_LAB4_HEXAGON_H

#include <iostream>
#include <cstdlib>
#include <stdint>
#include "Figure.h"

class Hexagon : public Figure{
public:
    Hexagon();
    Hexagon(std::istream &is);
    Hexagon(size_t i);
    size_t GetSide() override;
    double Square() override;
    void Print() override;

    bool operator==(const Hexagon &obj) const;
    bool operator!=(const Hexagon &obj) const;
    Hexagon &operator=(const Hexagon &obj);

    friend std::ostream &operator<<(std::ostream &os, const Hexagon &obj);
    friend std::istream &operator>>(std::istream &is, Hexagon &obj);

    virtual ~Hexagon();

private:
    size_t side_a;
};

#endif //OOP_LAB4_HEXAGON_H

```

## Item.cpp



```

#include <iostream>
#include <cstdlib>

#include "Item.h"

template <class T>
TTreeItem<T>::TTreeItem(const std::shared_ptr<T> &figure, size_t key) {
    this->figure = figure;
    this->son = nullptr;
    this->brother = nullptr;
    this->key = key;

    std::cout << "Tree item: created" << std::endl;
}

template <class T>
std::shared_ptr<TTreeItem<T>> TTreeItem<T>::GetParent() {
    return this->parent;
}

template <class T>
std::shared_ptr<T> TTreeItem<T>::GetFigure() const {
    return this->figure;
}

template <class T>
std::shared_ptr<TTreeItem<T>> TTreeItem<T>::GetSon() {
    return this->son;
}

template <class T>
size_t TTreeItem<T>::GetKey() const {
    return key;
}

template <class T>
void TTreeItem<T>::SetKey(size_t key) {
    this->key = key;
}

template <class T>
void TTreeItem<T>::SetSon(std::shared_ptr<TTreeItem<T>> son) {

```

```

        this->son = son;
    }

template <class T>
void TTreeItem<T>::SetBrother(std::shared_ptr<TTreeItem<T>> brother) {
    this->brother = brother;
}

template <class T>
void TTreeItem<T>::SetParent(std::shared_ptr<TTreeItem<T>> parent){
    this->parent = parent;
}

template <class T>
std::shared_ptr<TTreeItem<T>> TTreeItem<T>::GetBrother() {
    return this->brother;
}

template <class T>
std::shared_ptr<TTreeItem<T>> TTreeItem<T>::GetNext()
{
    std::shared_ptr<TTreeItem<T>> tmp;
    if (this->GetSon()) {
        return GetSon();
    }
    else if (this->GetBrother()) {
        return GetBrother();
    } else if (this->GetParent()) {
        tmp = this->GetParent();

        while (tmp && !(tmp->GetBrother())) {
            tmp = tmp->GetParent();
        }
        if (!tmp)
            return nullptr;
        tmp = tmp->GetBrother();
        return tmp;
    }
    return nullptr;
}

template <class T>

```

```

TTreeItem<T>::~~TTreeItem() {}

template <class T> TAllocationBlock
    TTreeItem<T>::treeItemAllocator(sizeof(TTreeItem<T>), 100);

template<class T>
void * TTreeItem<T>::operator new(size_t size)
{
    return treeItemAllocator.Allocate();
}

template<class T>
void TTreeItem<T>::operator delete(void * p)
{
    treeItemAllocator.Deallocate(p);
}

template <class A>
std::ostream& operator<<(std::ostream &os, const TTreeItem<A> &obj) {
    os << "[" << obj.GetFigure()->GetSide() << "]";
    return os;
}

template class TTreeItem<Figure>;
template std::ostream& operator<<(std::ostream& os, const TTreeItem<Figure>& obj);

```

## Item.h

```

#ifndef OOP_LAB4_ITEM_H
#define OOP_LAB4_ITEM_H

#include <stdio.h>
#include <memory>
#include "Figure.h"
#include "TAllocationBlock.h"

template <class T>
class TTreeItem {
public:
    TTreeItem(const std::shared_ptr<T> &figure, size_t key);

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

```

```

std::shared_ptr<TTreeItem<T>> GetParent();
std::shared_ptr<TTreeItem<T>> GetSon();
std::shared_ptr<TTreeItem<T>> GetBrother();
std::shared_ptr<T> GetFigure() const;

size_t GetKey() const;
void SetKey(size_t key);
void SetParent(std::shared_ptr<TTreeItem<T>> parent);
void SetSon(std::shared_ptr<TTreeItem<T>> son);
void SetBrother(std::shared_ptr<TTreeItem<T>> brother);

std::shared_ptr<TTreeItem<T>> GetNext();

void * operator new (size_t size);
void operator delete(void *p);

~TTreeItem();

private:
    std::shared_ptr<T> figure;
    std::shared_ptr<TTreeItem<T>> parent;
    std::shared_ptr<TTreeItem<T>> son;
    std::shared_ptr<TTreeItem<T>> brother;
    size_t key;

    static TAllocationBlock treeItemAllocator;
};

#endif //OOP_LAB4_ITEM_H

```

## Iterator.h

```

#ifndef OOP_LAB4_ITERATOR_H
#define OOP_LAB4_ITERATOR_H

#include <memory>
#include <iostream>

template <class N, class T>
class TIterator
{
public:

```

```

TIterator(std::shared_ptr<N> n) {
    cur = n;
}

std::shared_ptr<T> operator* () {
    return cur->GetFigure();
}

std::shared_ptr<T> operator-> () {
    return cur->GetFigure();
}

void operator++() {
    cur = cur->GetNext();
}

TIterator const operator++ (int) {
    TIterator cur(*this);
    ++(*this);
    return cur;
}

bool operator== (const TIterator &it) {
    return (cur == it.cur);
}

bool operator!= (const TIterator &it) {
    return (cur != it.cur);
}

private:
    std::shared_ptr<N> cur;
};

```

```

#endif //OOP_LAB4_ITERATOR_H

```

## Octagon.cpp

```

#include "Octagon.h"
#include <istream>
#include <cmath>

```

```

Octagon::Octagon() :Octagon(0) {

```

```

}

Octagon::Octagon(size_t i) : side_a(i){
    std::cout << "Octagon created: " << side_a << std::endl;
}

double Octagon::Square() {
    return 1.5 * pow(side_a,2) * sqrt(3);
}

void Octagon::Print() {
    std::cout << "Octagon with side = " << side_a;
}

size_t Octagon::GetSide(){
    return side_a;
}

Octagon::~~Octagon() {}

Octagon::Octagon(std::istream &is) {
    while (true) {
        is >> *this;
        if (std::cin.peek() == '\n') {
            std::cin.get();
            break;
        }
        else {
            std::cerr << "Error: insert data" << "\n";
            std::cin.clear();
            while (std::cin.get() != '\n') {}
        }
    }
}

Octagon &Octagon::operator=(const Octagon &obj) {
    if (this == &obj) return *this;
    side_a = obj.side_a;
    return *this;
}

```

```

bool Octagon::operator==(const Octagon &obj) const {
    return side_a == obj.side_a;
}

bool Octagon::operator!=(const Octagon &obj) const {
    return side_a != obj.side_a;
}

std::ostream &operator<<(std::ostream &os, const Octagon &obj) {
    os << "Octagon with side: " << obj.side_a << "\n";
    return os;
}

std::istream &operator>>(std::istream &is, Octagon &obj) {
    std::cout << "Insert side of Octagon: \n";
    is >> obj.side_a;
    return is;
}

```

## Octagon.h

```

#ifndef OOP_LAB4_OCTAGON_H
#define OOP_LAB4_OCTAGON_H

#include <iostream>
#include <cstdlib>
#include <stdint>
#include "Figure.h"

class Octagon : public Figure{
public:
    Octagon();
    Octagon(std::istream &is);
    Octagon(size_t i);
    size_t GetSide() override;
    double Square() override;
    void Print() override;

    bool operator==(const Octagon &obj) const;
    bool operator!=(const Octagon &obj) const;
    Octagon &operator=(const Octagon &obj);

```

```

    friend std::ostream &operator<<(std::ostream &os, const Octagon &obj);
    friend std::istream &operator>>(std::istream &is, Octagon &obj);

    virtual ~Octagon();

private:
    size_t side_a;

};

#endif //OOP_LAB4_OCTAGON_H

```

## Pentagon.cpp

```

#include "Pentagon.h"
#include <istream>
#include <cmath>

Pentagon::Pentagon() :Pentagon(0) {

}

Pentagon::Pentagon(size_t i) : side_a(i){
    std::cout << "Pentagon created: " << side_a << std::endl;
}

double Pentagon::Square() {
    return 1.5 * pow(side_a,2) * sqrt(3);
}

void Pentagon::Print() {
    std::cout << "Pentagon with side a = " << side_a;
}

size_t Pentagon::GetSide(){
    return side_a;
}

Pentagon::~~Pentagon() {}

Pentagon::Pentagon(std::istream &is) {
    while (true) {

```



```

        is >> *this;
        if (std::cin.peek() == '\n') {
            std::cin.get();
            break;
        }
        else {
            std::cerr << "Error: insert data" << "\n";
            std::cin.clear();
            while (std::cin.get() != '\n') {}
        }
    }
}

Pentagon &Pentagon::operator=(const Pentagon &obj) {
    if (this == &obj) return *this;
    side_a = obj.side_a;
    return *this;
}

bool Pentagon::operator==(const Pentagon &obj) const {
    return side_a == obj.side_a;
}

bool Pentagon::operator!=(const Pentagon &obj) const {
    return side_a != obj.side_a;
}

std::ostream &operator<<(std::ostream &os, const Pentagon &obj) {
    os << "Pentagon with side: " << obj.side_a << "\n";
    return os;
}

std::istream &operator>>(std::istream &is, Pentagon &obj) {
    std::cout << "Insert side of Pentagon: \n";
    is >> obj.side_a;
    return is;
}

```

## Pentagon.h

```

#ifndef OOP_LAB4_PENTAGON_H
#define OOP_LAB4_PENTAGON_H

```

```

#include <iostream>
#include <cstdlib>
#include <cstdlibint>
#include "Figure.h"

class Pentagon : public Figure{
public:
    Pentagon();
    Pentagon(std::istream &is);
    Pentagon(size_t i);
    size_t GetSide() override;
    double Square() override;
    void Print() override;

    bool operator==(const Pentagon &obj) const;
    bool operator!=(const Pentagon &obj) const;
    Pentagon &operator=(const Pentagon &obj);

    friend std::ostream &operator<<(std::ostream &os, const Pentagon &obj);
    friend std::istream &operator>>(std::istream &is, Pentagon &obj);

    virtual ~Pentagon();

private:
    size_t side_a;
};

#endif //OOP_LAB4_PENTAGON_H

```

## TNaryTree.cpp

```

#include <iostream>
#include "TNaryTree.h"
#include "Item.h"
#include "Figure.h"

template <class T>
TNaryTree<T>::TNaryTree() {
    this->root = nullptr;
}

```

```

template <class T>
TIterator<TTreeItem<T>, T> TNaryTree<T>::begin() {
    return TIterator<TTreeItem<T>, T>(root);
}

template <class T>
TIterator<TTreeItem<T>, T> TNaryTree<T>::end() {
    return TIterator<TTreeItem<T>, T>(nullptr);
}

template <class T>
void TreeDel(std::shared_ptr<TTreeItem<T>> item) { // Деструктор для дерева
    if(item) {
        TreeDel(item->GetBrother());
        TreeDel(item->GetSon());
        item.reset();
    }
}

template <class T>
TNaryTree<T>::~~TNaryTree() {
    TreeDel(root);
    std::cout << "Tree deleted" << std::endl;
}

template <class T>
int TNaryTree<T>::GetNodesOnLevel(std::shared_ptr<TTreeItem<T>> item, int level) { //
    if (item == nullptr) return 0;

    if (level <= 0) return 0;

    return GetNodesOnLevel(item->GetSon(), level - 1) + (level == 1) + GetNodesOnLevel(
}

template <class T>
bool TNaryTree<T>::empty() {
    return root == nullptr;
}

template <class T>
std::shared_ptr<TTreeItem<T>> TNaryTree<T>::Find(std::shared_ptr<TTreeItem<T>> node, siz

```

```

std::shared_ptr<TTreeItem<T>> tr = nullptr;
if (node->GetKey() == key)
    return node;

if (node->GetSon()) {
    tr = Find(node->GetSon(), key);
    if (tr != nullptr) {
        return tr;
    }
}

if (node->GetBrother()) {
    tr = Find(node->GetBrother(), key);

    if (tr != nullptr) {
        return tr;
    }
}

return nullptr;
}

template <class T>
std::shared_ptr<TTreeItem<T>> TNaryTree<T>::Insert(std::shared_ptr<T> figure, size_t par
std::shared_ptr<TTreeItem<T>> node = std::make_shared<TTreeItem<T>>(figure, key);
if (empty()) {
    root = node;
    root->SetParent(nullptr);
    std::cout << "Tree was empty. Item was set as root." << std::endl;
    return root;
}
std::shared_ptr<TTreeItem<T>> parent = Find(this->root, parentKey);
if (!parent) {
    std::cout << "Parent with this key not found." << std::endl;
    if (root->GetSon()) {
        std::shared_ptr<TTreeItem<T>> tmp = root->GetSon();

        while (tmp->GetBrother())
            tmp = tmp->GetBrother();

        tmp->SetBrother(node);
        node->SetParent(root);
    }
}

```

```

        return tmp->GetBrother();
    } else {
        root->SetSon(node);
        node->SetParent(root);
        return root->GetSon();
    }
}

if (parent->GetSon()) {
    std::shared_ptr<TTreeItem<T>> tmp = parent->GetSon();
    while (tmp->GetBrother())
        tmp = tmp->GetBrother();
    tmp->SetBrother(node);
    return tmp->GetBrother();
} else {
    parent->SetSon(node);
    node->SetParent(parent);
    return parent->GetSon();
}
}

template <class T>
std::shared_ptr<TTreeItem<T>> TNaryTree<T>::Push(std::shared_ptr<TTreeItem<T>> item)
{
    if (this->empty()) {
        this->root = item;
        return this->root;
    } else {
        if (root->GetSon()) {
            std::shared_ptr<TTreeItem<T>> tmp = root->GetSon();
            while (tmp->GetBrother())
                tmp = tmp->GetBrother();
            tmp->SetBrother(item);
            return tmp->GetBrother();
        }
        else {
            root->SetSon(item);
            return root->GetSon();
        }
    }
}
}

```

```

template <class T>
TTreeItem<T> *destroy_Tree(std::shared_ptr<TTreeItem<T>> pointer) {
    if(pointer == nullptr) {
        return nullptr;
    }

    if (pointer->GetSon() != nullptr) {
        destroy_Tree(pointer->GetSon());
    }

    if (pointer->GetBrother() != nullptr) {
        destroy_Tree(pointer->GetBrother());
    }

    if (pointer->GetSon() == nullptr && pointer->GetBrother() == nullptr) {
        pointer.reset();
        pointer = nullptr;
    }
    return nullptr;
}

```

```

template <class T>
void TNaryTree<T>::remove(size_t key)
{
    if (root->GetKey() == key) {
        root = nullptr;
    }
    else {
        remove(root, key);
    }
}

```

```

template <class T>
void TNaryTree<T>::remove(std::shared_ptr<TTreeItem<T>> node, size_t key) {
    if (node->GetSon()) {
        if (node->GetSon()->GetKey() == key) {
            std::shared_ptr<TTreeItem<T>> tr = node->GetSon();
            node->SetSon(node->GetSon()->GetBrother());
            tr->SetBrother(nullptr);
            return;
        } else {

```

```

        remove(node->GetSon(), key);
    }
}

if (node->GetBrother()) {
    if (node->GetBrother()->GetKey() == key) {
        std::shared_ptr<TTreeItem<T>> tr = node->GetBrother();
        node->SetBrother(node->GetBrother()->GetBrother());
        tr->SetBrother(nullptr);
        return;
    } else {
        remove(node->GetBrother(), key);
    }
}
}

template <class T>
void TNaryTree<T>::Pop(size_t value) // Вызов функции удаления
{
    if (root->GetFigure()->GetSide() == value) {
        root = nullptr;
    }
    else {
        Pop(root, value);
    }
}

template <class T>
void TNaryTree<T>::Pop(std::shared_ptr<TTreeItem<T>> item, size_t value)
// Удаление элемента из дерева
{
    if (item->GetSon()) {
        if (item->GetSon()->GetFigure()->GetSide() == value) {
            std::shared_ptr<TTreeItem<T>> ptr = item->GetSon();
            item->SetSon(item->GetSon()->GetBrother());
            ptr->SetBrother(nullptr);
            return;
        }
        else {
            Pop(item->GetSon(), value);
        }
    }
}

```

```

    }
    if (item->GetBrother()) {
        if (item->GetBrother()->GetFigure()->GetSide() == value) {
            std::shared_ptr<TTreeItem<T>> ptr = item->GetBrother();
            item->SetBrother(item->GetBrother()->GetBrother());
            ptr->SetBrother(nullptr);
            return;
        }
        else {
            Pop(item->GetBrother(), value);
        }
    }
}

void TSpace(size_t n){ // Функция расставляющая пробелы
    for (size_t i=0;i<=n;i++)
        std::cout << "    ";
}

template <class T>
void TreeRun(std::ostream &os, std::shared_ptr<TTreeItem<T>> item, size_t space){
// Функция обхода дерева для печати
    if (item) {
        TSpace(space);
        //os << *item << std::endl;
        std::cout << "[";
        item->GetFigure()->Print();
        std::cout << " , key = " << item->GetKey() << "]" << "\n";
        if (item->GetBrother() != nullptr) {
            TreeRun(os, item->GetBrother(),space);
        }
        if (item->GetSon() != nullptr) {
            TreeRun(os, item->GetSon(),space+1);
        }
    }
}

template <class A>
std::ostream& operator<<(std::ostream &os, const TNaryTree<A> &tree) {
//Оператор вывода дерева
    std::shared_ptr<TTreeItem<A>> obj = tree.root;

```



```

        os << "Printed tree:" << std::endl;
        TreeRun(os, obj,1);
        return os;
}

template class TNaryTree<Figure>;
template std::ostream &operator<<(std::ostream &os, const TNaryTree<Figure> &obj);

```

## TNaryTree.h

```

#ifdef OOP_LAB4_TREE_H
#define OOP_LAB4_TREE_H

```

```

#include "Figure.h"
#include "Iterator.h"
#include "Item.h"
#include <stdbool.h>
#include <memory>
#include <cstdint>

```

```

template <class T> class TNaryTree{
public:
    TNaryTree();

    int GetNodesOnLevel(int level) {
        return GetNodesOnLevel(root, level);
    }

    TIterator<TTreeItem<T>, T> begin();
    TIterator<TTreeItem<T>, T> end();

    std::shared_ptr<TTreeItem<T>> Insert(std::shared_ptr<T> figure, size_t parentKey, si
    std::shared_ptr<TTreeItem<T>> Find(std::shared_ptr<TTreeItem<T>> tree, size_t key);
    std::shared_ptr<TTreeItem<T>> Push(std::shared_ptr<TTreeItem<T>> item);

    bool empty();

    void remove(size_t key);
    void remove(std::shared_ptr<TTreeItem<T>> tree, size_t key);

    void Pop(size_t value);
    void Pop(std::shared_ptr<TTreeItem<T>> tree, size_t value);

```

```

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

virtual ~TNaryTree();

private:
    int GetNodesOnLevel(std::shared_ptr<TTreeItem<T>> item, int level);
    std::shared_ptr<TTreeItem<T>> root;
};

#endif //OOP_LAB4_TREE_H

```

## TAllocationBlock.cpp

```

#include "TAllocationBlock.h"

TAllocationBlock::TAllocationBlock(size_t size, size_t count)
{
    _used_blocks = (Byte *)malloc(size * count);

    for(size_t i = 0; i < count; ++i) {
        void *ptr = (void *)malloc(sizeof(void *));
        ptr = _used_blocks + i * size;
        _free_blocks.Push(ptr);
    }
}

void *TAllocationBlock::Allocate()
{
    if(!_free_blocks.IsEmpty()) {
        void *res = _free_blocks.Top();
        _free_blocks.Pop();
        return res;
    } else {
        throw std::bad_alloc();
    }
}

void TAllocationBlock::Deallocate(void *ptr)
{
    _free_blocks.Push(ptr);
}

```

```

bool TAllocationBlock::Empty()
{
    return _free_blocks.IsEmpty();
}

size_t TAllocationBlock::Size()
{
    return _free_blocks.GetSize();
}

TAllocationBlock::~TAllocationBlock()
{
    while(!_free_blocks.IsEmpty()) {
        _free_blocks.Pop();
    }
    free(_used_blocks);
    //delete _free_blocks;
    //delete _used_blocks;
}

```

## TAllocationBlock.h

```

#ifndef OOP_LAB4_TALLOCATIONBLOCK_H
#define OOP_LAB4_TALLOCATIONBLOCK_H

#include <iostream>
#include <cstdlib>
#include "TStack.h"

typedef unsigned char Byte;

class TAllocationBlock
{
public:
    TAllocationBlock(size_t size, size_t count);
    void *Allocate();
    void Deallocate(void *ptr);
    bool Empty();
    size_t Size();

    virtual ~TAllocationBlock();

```

```

private:
    Byte *_used_blocks;
    TStack<void *>_free_blocks;
};

#endif //OOP_LAB4_TALLOCATIONBLOCK_H

```

## TStack.cpp

```

#include <iostream>
#include <memory>
#include "TStack.h"

template <class T>
TStack<T>::TStack()
{
    head = nullptr;
    count = 0;
}

template <class T>
void TStack<T>::Push(const T &item)
{
    TStackItem<T> *tmp = new TStackItem<T>(item, head);
    head = tmp;
    ++count;
}

template <class T>
bool TStack<T>::IsEmpty() const
{
    return !count;
}

template <class T>
uint32_t TStack<T>::GetSize() const
{
    return count;
}

template <class T>
void TStack<T>::Pop()
{

```

```

        if(head) {
            TStackItem<T> *tmp = &head->GetNext();
            delete head;
            head = tmp;
            --count;
        }
    }

template <class T>
T &TStack<T>::Top()
{
    return head->Pop();
}

template <class T>
TStack<T>::~~TStack()
{
    for(TStackItem<T> *tmp = head, *tmp2; tmp; tmp = tmp2) {
        tmp2 = &tmp->GetNext();
        delete tmp;
    }
}

typedef unsigned char Byte;

template class
TStack<void *>;

```

## TStack.h

```

#ifndef OOP_LAB4_TSTACK_H
#define OOP_LAB4_TSTACK_H

#include <iostream>
#include <memory>
#include "TStackItem.h"

template <class T>
class TStack
{
public:
    TStack();
    virtual ~TStack();

```

```

    void Push(const T &item);
    void Pop();
    T &Top();
    bool IsEmpty() const;
    uint32_t GetSize() const;

    template <class A> friend std::ostream& operator<<(std::ostream &os, const TStack<A>

private:
    TStackItem<T> *head;
    uint32_t count;
};

#endif //OOP_LAB4_TSTACK_H

```

## TStackItem.cpp

```

#include <iostream>
#include <memory>
#include "TStackItem.h"

template <class T>
TStackItem<T>::TStackItem(const T &val, TStackItem<T> *item)
{
    value = new T(val);
    next = item;
}

template <class T>
void TStackItem<T>::Push(const T &val)
{
    *value = val;
}

template <class T>
T &TStackItem<T>::Pop() const
{
    return *value;
}

template <class T>
void TStackItem<T>::SetNext(TStackItem<T> *item)

```

```

{
    next = item;
}

template <class T>
TStackItem<T> &TStackItem<T>::GetNext() const
{
    return *next;
}

template <class T>
TStackItem<T>::~~TStackItem()
{
    delete value;
}

typedef unsigned char Byte;

template class
TStackItem<void *>;

```

## TStackItem.h

```

#ifndef OOP_LAB4_TSTACKITEM_H
#define OOP_LAB4_TSTACKITEM_H

#include <iostream>
#include <memory>

template <class T>
class TStackItem
{
public:
    TStackItem(const T &val, TStackItem<T> *item);
    virtual ~TStackItem();

    void Push(const T &val);
    T &Pop() const;
    void SetNext(TStackItem<T> *item);
    TStackItem<T> &GetNext() const;

private:

```

```
    T *value;  
    TStackItem<T> *next;  
};  
  
#endif //OOP_LAB4_TSTACKITEM_H
```