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

ЛАБОРАТОРНАЯ РАБОТА №7

по курсу “Объектно-ориентированное программирование”
I семестр, 2021/22 учебный год

Студент: *Степанов Данила Михайлович, группа М8О-207Б-20*
Преподаватель: *Дорохов Евгений Павлович, каф. 806*

Задание:

Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

> Итератор должен позволять использовать структуру данных в операторах типа for. Например:
> ```
> for (auto i : stack) {
> std::cout << *i << std::endl;
> }
> ```

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

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

Вариант №24:

- Фигура: 8-угольник (Octagon)
- Контейнер: N-дерево (TNaryTree)

Описание программы:

Исходный код разделён на 10 файлов:

- `figure.h` – описание класса фигуры
- `point.h` – описание класса точки
- `point.cpp` – реализация класса точки
- `octagon.h` – описание класса 8-угольника
- `octagon.cpp` – реализация класса 8-угольника
- `TNaryTree_item.h` – описание элемента N-дерева
- `TNaryTree.h` – описание N-дерева
- `TNaryTree.cpp` – реализация N-дерева

- titerator.h – описание итератора
- main.cpp – основная программа

Дневник отладки:

Проблем не возникло.

Тестирование программы:

The tree is empty !
Octagon: (1, 4) (1, 2) (5, 6) (2, 8) (3, 1) (2, 6) (9, 5) (5, 4)

Octagon: (2, 5) (1, 5) (16, 6) (3, 6) (1, 8) (4, 2) (7, 3) (1, 15)

Octagon: (6, 5) (9, 1) (7, 3) (1, 8) (5, 6) (4, 8) (9, 5) (6, 4)

Octagon: (10, 5) (9, 1) (7, 3) (1, 8) (5, 6) (4, 8) (9, 5) (6, 4)

Octagon: (11, 5) (9, 1) (7, 3) (1, 8) (5, 6) (4, 8) (9, 5) (6, 4)

Octagon: (7, 4) (1, 2) (5, 6) (2, 8) (3, 1) (2, 6) (9, 5) (5, 4)

Octagon: (3, 5) (9, 1) (7, 3) (1, 8) (5, 6) (4, 8) (9, 5) (6, 4)

Octagon: (8, 5) (1, 5) (16, 6) (3, 6) (1, 8) (4, 2) (7, 3) (1, 15)

Octagon: (9, 5) (9, 1) (7, 3) (1, 8) (5, 6) (4, 8) (9, 5) (6, 4)

Octagon: (4, 4) (1, 2) (5, 6) (2, 8) (3, 1) (2, 6) (9, 5) (5, 4)

Octagon: (5, 5) (1, 5) (16, 6) (3, 6) (1, 8) (4, 2) (7, 3) (1, 15)

0.5: [36: [12: [18, 19.5], 6.5], 7.5: [6, 16.5], 3.5: [21]]
44
0.5: [36: [12: [18, 19.5], 6.5], 7.5: [6, 16.5], 3.5: [21]]
Octagon: (1, 4) (1, 2) (5, 6) (2, 8) (3, 1) (2, 6) (9, 5) (5, 4)
The tree is not empty !

Вывод:

В данной лабораторной работе я познакомился с итераторами и реализовал свой для работы с N-деревом. Итераторы осуществляют доступ к элементам контейнеров и навигацию по ним, поэтому итераторы крайне необходимы для разработки алгоритмов работы с контейнерами.

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 getX();
  double getY();

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

private:
  double x_;
  double y_;
};

#endif
```

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::getX()
{
   return x_;
}

double Point::getY()
{
   return y_;
}

std::istream& operator>>(std::istream& is, Point& p) {
   is >> p.x_ >> p.y_;
   return is;
}

std::ostream& operator<<(std::ostream& os, Point& p) {
   os << "(" << p.x_ << ", " << p.y_ << ")";
   return os;
}
```

figure.h:

```
#ifndef FIGURE_H
#define FIGURE_H

#include "point.h"

class Figure
{
public:
   virtual size_t VertexesNumber() = 0;
   virtual double Area() = 0;
   virtual void Print(std::ostream& os) = 0;
};

#endif
```

octagon.h:

```
#ifndef OCTAGON_H
#define OCTAGON_H

#include "figure.h"

class Octagon : Figure
{
public:
   Octagon(std::istream& is);
   size_t VertexesNumber();
   double Area();
   void Print(std::ostream& os);

private:
   Point a_, b_, c_, d_;
   Point e_, f_, g_, h_;
};

#endif
```

octagon.cpp:

```
#include "octagon.h"

Octagon::Octagon(std::istream& is)
{
   std::cin >> a_ >> b_ >> c_ >> d_;
   std::cin >> e_ >> f_ >> g_ >> h_;
}

size_t Octagon::VertexesNumber()
{
   return (size_t)8;
}

double Octagon::Area()
{
   return 0.5 * abs((a_.getX() * b_.getY() + b_.getX() * c_.getY() + c_.getX() * d_.getY() + d_.getX() * e_.getY() + e_.getX() *
f_.getY() +
   f_.getX() * g_.getY() + g_.getX() * h_.getY() + h_.getX() * a_.getY() - (b_.getX() * a_.getY() + c_.getX() * b_.getY() +
   d_.getX() * c_.getY() + e_.getX() * d_.getY() + f_.getX() * e_.getY() + g_.getX() * f_.getY() + h_.getX() * g_.getY() +
   a_.getX() * h_.getY())));
}

void Octagon::Print(std::ostream& os)
{
   std::cout << "Octagon: " << a_ << " " << b_ << " ";
   std::cout << c_ << " " << d_ << " " << e_ << " ";
   std::cout << f_ << " " << g_ << " " << h_ << "\n";
}
```

TNaryTree_item.h:

```
#ifndef TNARYTREE_ITEM
#define TNARYTREE_ITEM

#include "octagon.h"
#include <memory>

template<class T>
class TreeItem
{
public:
   std::shared_ptr<T> figure;
   int cur_size;
   std::shared_ptr<TreeItem<T>> son;
   std::shared_ptr<TreeItem<T>> brother;
   std::shared_ptr<TreeItem<T>> parent;
};

#endif
```

TnaryTree.h:

```
#ifndef TNARY_TREE
#define TNARY_TREE

#include "octagon.h"
#include "TNaryTree_item.h"
#include <memory>
```

```
template<class T>
class TNaryTree
{
public:
   TNaryTree(int n);
   TNaryTree(const TNaryTree<T>& other);
   TNaryTree();

   void Update(const std::shared_ptr<T> &&polygon, const std::string &&tree_path)
   {
      Update(&root, polygon, tree_path);
   }

   void Update(const std::shared_ptr<T> &polygon, const std::string &tree_path)
   {
      Update(&root, polygon, tree_path);
   }

   const std::shared_ptr<T>& GetItem(const std::string& tree_path)
   {
      return GetItem(&root, tree_path);
   }

   void RemoveSubTree(const std::string &&tree_path);
   void RemoveSubTree(const std::string &tree_path);
   bool Empty();
   double Area(std::string&& tree_path);
   double Area(std::string& tree_path);
   template<class A> friend std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree);
   virtual ~TNaryTree();

private:
   int size;
   std::shared_ptr<TreeItem<T>> root;
   void Update(std::shared_ptr<TreeItem<T>>* root, std::shared_ptr<T> polygon, std::string tree_path);
   const std::shared_ptr<T>& GetItem(std::shared_ptr<TreeItem<T>>* root, const std::string tree_path);
};

#endif
```

TNaryTree.cpp:

```
#include "TNaryTree.h"
#include "TNaryTree_item.h"

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

template<class T>
std::shared_ptr<TreeItem<T>> tree_copy(std::shared_ptr<TreeItem<T>> root)
{
   if (root != nullptr) {
      std::shared_ptr<TreeItem<T>> new_root (new TreeItem<T>);
      new_root->figure = root->figure;
```

```
        new_root->son = nullptr;
        new_root->brother = nullptr;
        if (root->son != nullptr) {
            new_root->son = tree_copy(root->son);
        }
        if (root->brother != nullptr) {
            new_root->brother = tree_copy(root->brother);
        }
        return new_root;
    }
    return nullptr;
}

template<class T>
TNaryTree<T>::TNaryTree(const TNaryTree<T>& other)
{
    this->root = tree_copy(other.root);
    this->root->cur_size = 0;
    this->size = other.size;
}

template<class T>
void TNaryTree<T>::Update(std::shared_ptr<TreeItem<T>>* root, std::shared_ptr<T> polygon, std::string tree_path)
{
    if (tree_path == "") {
        if (*root == nullptr) {
        *root = std::shared_ptr<TreeItem<T>>(new TreeItem<T>);
        (*root)->figure = std::shared_ptr<T>(new T);
        (*root)->figure = polygon;
        (*root)->brother = nullptr;
        (*root)->son = nullptr;
        (*root)->parent = nullptr;
        } else {
            (*root)->figure = polygon;
        }
        return;
    }
    if (tree_path == "b") {
        std::cout << "Cant add brother to root\n";
        return;
    }
    std::shared_ptr<TreeItem<T>> cur = *root;
    if (cur == NULL) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
        return;
    }
    for (int i = 0; i < tree_path.size() - 1; i++) {
        if (tree_path[i] == 'c') {
            cur = cur->son;
        } else {
            cur = cur->brother;
        }
        if (cur == nullptr && i < tree_path.size() - 1) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
    }
    if (tree_path[tree_path.size() - 1] == 'c' && cur->son == nullptr) {
        if (cur->cur_size + 1 > this->size) {
            throw std::out_of_range("Tree is overflow\n");
```

```
            return;
        }
        if (cur->son == nullptr) {
            cur->son = std::shared_ptr<TreeItem<T>>(new TreeItem<T>);
            cur->son->figure = std::shared_ptr<T>(new T);
            cur->son->figure = polygon;
            cur->son->son = nullptr;
            cur->son->brother = nullptr;
            cur->son->parent = cur;
            cur->son->parent->cur_size++;
        } else {
            cur->son->figure = polygon;
        }
    } else if (tree_path[tree_path.size() - 1] == 'b' && cur->brother == nullptr) {
        if (cur->parent->cur_size + 1 > this->size) {
            throw std::out_of_range("Tree is overflow\n");
            return;
        }
        if (cur->brother == nullptr) {
            cur->brother = std::shared_ptr<TreeItem<T>>(new TreeItem<T>);
            cur->brother->figure = std::shared_ptr<T>(new T);
            cur->brother->figure = polygon;
            cur->brother->son = nullptr;
            cur->brother->brother = nullptr;
            cur->brother->parent = cur->parent;
            cur->brother->parent->cur_size++;
        } else {
            cur->brother->figure = polygon;
        }
    }
}

template<class T>
void delete_tree(std::shared_ptr<TreeItem<T>>* root)
{
    if ((*root)->son != nullptr) {
        delete_tree(&((*root)->son));
    }
    if ((*root)->brother != nullptr) {
        delete_tree(&((*root)->brother));
    }
    *root = nullptr;
}

template<class T>
void delete_undertree(std::shared_ptr<TreeItem<T>>* root, char c)
{
    if (*root == nullptr) {
        return;
    }
    if (c == 'b') {
        if ((*root)->brother != nullptr) {
            std::shared_ptr<TreeItem<T>> cur = (*root)->brother;
            if ((*root)->brother->brother != nullptr) {
                (*root)->brother = (*root)->brother->brother;
                cur->brother = nullptr;
                delete_tree(&cur);
            } else {
                delete_tree(&((*root)->brother));
            }
```

```
        }
    } else if (c == 'c') {
        std::shared_ptr<TreeItem<T>> cur = (*root)->son;
        if ((*root)->son->brother != nullptr) {
            (*root)->son = (*root)->son->brother;
            if (cur->son != nullptr) {
                delete_tree(&(cur->son));
            }
            cur = nullptr;
        } else {
            delete_tree(&((*root)->son));
        }
    }
}

template<class T>
void TNaryTree<T>::RemoveSubTree(const std::string &&tree_path)
{
    if (tree_path == "" && this->root != nullptr) {
        std::shared_ptr<TreeItem<T>>* iter = &(this->root);
        delete_tree(iter);
        return;
    } else if (tree_path == "" && this->root == nullptr) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
        return;
    }
    std::shared_ptr<TreeItem<T>> cur = this->root;
    for (int i = 0; i < tree_path.size() - 1; i++) {
        if (tree_path[i] == 'c') {
            if (cur->son == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->son;
        } else if (tree_path[i] == 'b') {
            if (cur->brother == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->brother;
        }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
        if (cur->son == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'c');
    } else if (tree_path[tree_path.size() - 1] == 'b') {
        if (cur->brother == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'b');
    }
    return;
}

template<class T>
void TNaryTree<T>::RemoveSubTree(const std::string &tree_path)
```

```
{
    if (tree_path == "" && this->root != nullptr) {
        std::shared_ptr<TreeItem<T>>* iter = &(this->root);
        delete_tree(iter);
        return;
    } else if (tree_path == "" && this->root == nullptr) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
        return;
    }
    std::shared_ptr<TreeItem<T>> cur = this->root;
    for (int i = 0; i < tree_path.size() - 1; i++) {
        if (tree_path[i] == 'c') {
            if (cur->son == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->son;
        } else if (tree_path[i] == 'b') {
            if (cur->brother == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
                return;
            }
            cur = cur->brother;
        }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
        if (cur->son == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'c');
    } else if (tree_path[tree_path.size() - 1] == 'b') {
        if (cur->brother == nullptr) {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
            return;
        }
        delete_undertree(&cur, 'b');
    }
    return;
}

template<class T>
bool TNaryTree<T>::Empty()
{
    if (this->root != nullptr) {
        return false;
    } else {
        return true;
    }
}

template<class T>
double TNaryTree<T>::Area(std::string &&tree_path)
{
    if (tree_path == "") {
        if (this->root != nullptr) {
            return this->root->figure->Area();
        } else {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
        }
```

```
    }
    std::shared_ptr<TreeItem<T>> cur = this->root;
    double square = 0;
    for (int i = 0; i < tree_path.size(); i++) {
        if (tree_path[i] == 'c') {
            if (cur->son != nullptr) {
                cur = cur->son;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        } else {
            if (cur->brother != nullptr) {
                cur = cur->brother;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        }
        square += cur->figure->Area();
    }
    return square + this->root->figure->Area();
}

template<class T>
double TNaryTree<T>::Area(std::string &tree_path)
{
    if (tree_path == "") {
        if (this->root != nullptr) {
            return this->root->figure->Area();
        } else {
            throw std::invalid_argument("Vertex doesn't exist in the path\n");
        }
    }
    std::shared_ptr<TreeItem<T>> cur = this->root;
    double square = 0;
    for (int i = 0; i < tree_path.size(); i++) {
        if (tree_path[i] == 'c') {
            if (cur->son != nullptr) {
                cur = cur->son;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        } else {
            if (cur->brother != nullptr) {
                cur = cur->brother;
            } else {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
        }
        square += cur->figure->Area();
    }
    return square + this->root->figure->Area();
}

template<class T>
void Print(std::ostream& os, std::shared_ptr<TreeItem<T>> vertex)
{
    if (vertex != nullptr) {
        os << vertex->figure->Area();
        if (vertex->son != nullptr) {
            os << ": " << "[";
```

```
            Print(os, vertex->son);
            if ((vertex->son->brother == nullptr && vertex->brother != nullptr) || (vertex->son->brother == nullptr &&
vertex->brother == nullptr)) {
                os << "]";
            }
        }
        if (vertex->brother != nullptr) {
            os << ", ";
            Print(os, vertex->brother);
            if (vertex->brother->brother == nullptr) {
                os << "]";
            }
        }
    } else {
        return;
    }
}

template<class A>
std::ostream& operator<<(std::ostream& os, const TNaryTree<A>& tree)
{
    if (tree.root != nullptr) {
        Print(os, tree.root); os << "\n";
        return os;
    } else {
        os << "Tree has no vertex\n";
        return os;
    }
}

template<class T>
const std::shared_ptr<T>& TNaryTree<T>::GetItem(std::shared_ptr<TreeItem<T>>* root, const std::string tree_path)
{
    if (tree_path == "" && *root == nullptr) {
        throw std::invalid_argument("Vertex doesn't exist in the path\n");
    }
    std::shared_ptr<TreeItem<T>> cur  = *root;
    for (int i = 0; i < tree_path.size(); i++) {
        if (tree_path[i] == 'c') {
            if (cur->son == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
            cur = cur->son;
        } else if (tree_path[i] == 'b') {
            if (cur->brother == nullptr) {
                throw std::invalid_argument("Vertex doesn't exist in the path\n");
            }
            cur = cur->brother;
        }
    }
    return cur->figure;
}

template<class T>
TNaryTree<T>::~TNaryTree()
{
    if (this->root != nullptr) {
        this->RemoveSubTree("");
    }
}
```

```
template class TNaryTree<octagon>;
template std::ostream& operator<< <octagon>(std::ostream&, TNaryTree<octagon> const&);
```

titerator.h:

```
#ifndef TITERATOR_H
#define TITERATOR_H

#include <iostream>
#include <memory>

template<class node, class T>
class TIterator
{
public:
    TIterator(std::shared_ptr<node> n) {node_ptr = n;}
    std::shared_ptr<T> operator*() {return node_ptr->figure;}
    std::shared_ptr<T> operator->() {return node_ptr->figure;}
    void operator++() {node_ptr = node_ptr->GetNext();}

      TIterator operator++(int) {

        TIterator iter(*this);
        ++(*this);
        return iter;
    }

  bool operator==(TIterator const& i) { return node_ptr == i.node_ptr; }

  bool operator!=(TIterator const& i) { return !(*this == i); }

private:
    std::shared_ptr<node> node_ptr;
};

#endif
```

main.cpp:

```
#include "figure.h"
#include "TNaryTree.h"
#include "TNaryTree_item.h"
#include "octagon.h"
#include <string>

int main()
{
   TNaryTree<octagon> a(4);
   if (a.Empty()) {
      std::cout << "The tree is empty !\n";
   } else {
      std::cout << "The tree is not empty !\n";
   }
   a.Update(std::shared_ptr<octagon>(new octagon(Point(1, 4), Point(1, 2), Point(5, 6), Point(2, 8),
   Point(3, 1), Point(2, 6), Point(9, 5), Point(5, 4))), ""); // 1
   a.Update(std::shared_ptr<octagon>(new octagon(Point(2, 5), Point(1, 5), Point(16, 6), Point(3, 6),
   Point(1, 8), Point(4, 2), Point(7, 3), Point(1, 15))), "c"); // 2
   a.Update(std::shared_ptr<octagon>(new octagon(Point(3, 5), Point(9, 1), Point(7, 3), Point(1, 8),
   Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4))), "cb"); // 3
   a.Update(std::shared_ptr<octagon>(new octagon(Point(4, 4), Point(1, 2), Point(5, 6), Point(2, 8),
```

```
    Point(3, 1), Point(2, 6), Point(9, 5), Point(5, 4))), "cbb"); // 4
    a.Update(std::shared_ptr<octagon>(new octagon(Point(5, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2), Point(7, 3), Point(1, 15))), "cbbc"); // 5
    a.Update(std::shared_ptr<octagon>(new octagon(Point(6, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4))), "cc"); // 6
    a.Update(std::shared_ptr<octagon>(new octagon(Point(7, 4), Point(1, 2), Point(5, 6), Point(2, 8),
    Point(3, 1), Point(2, 6), Point(9, 5), Point(5, 4))), "ccb"); // 7
    a.Update(std::shared_ptr<octagon>(new octagon(Point(8, 5), Point(1, 5), Point(16, 6), Point(3, 6),
    Point(1, 8), Point(4, 2), Point(7, 3), Point(1, 15))), "cbc"); // 8
    a.Update(std::shared_ptr<octagon>(new octagon(Point(9, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4))), "cbcb"); // 9
    a.Update(std::shared_ptr<octagon>(new octagon(Point(9, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4))), "ccc"); // 10
    a.Update(std::shared_ptr<octagon>(new octagon(Point(9, 5), Point(9, 1), Point(7, 3), Point(1, 8),
    Point(5, 6), Point(4, 8), Point(9, 5), Point(6, 4))), "cccb"); // 11
    std::cout << a;
    std::cout << a.Area("cb") << "\n";
    TNaryTree<octagon> b(a);
    std::cout << b;
    std::shared_ptr<octagon> c = a.GetItem("");
    std::cout << *c;
    a.RemoveSubTree("cbc");
    if (a.Empty()) {
        std::cout << "The tree is empty !\n";
    } else {
        std::cout << "The tree is not empty !\n";
    }
    return 0;
}
```