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

ЛАБОРАТОРНАЯ РАБОТА №7

по курсу объектно-ориентированное программирование I семестр, 2021/22 уч. год

Студент: *Соколов Даниил Витальевич группа М8О-207Б-20*
Преподаватель: *Дорохов Евгений Павлович*

Условие

Задание: Вариант 23: N-арное дерево (Шестиугольник). 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.

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 нескольких файлах:

1. figure.h: родительский класс-интерфейс для фигур
2. point.h: описание класса точки
3. point.cpp: реализация класса точки
4. triangle.h: описание класса треугольника, наследующегося от figure
5. triangle.cpp: реализация класса треугольника
6. main.cpp: тестирование кода
7. titerator.h: описание и реализация итераторов
8. tqueue.h: структура очереди
9. tqueue.cpp: реализация очереди
10. TNaryTree.cpp
11. TNaryTree.h
12. TNaryTree_item.cpp
13. TNaryTree_item.h

14. titerator.h

Дневник отладки

Ошибок не было.

Недочёты

Недочётов не заметил.

Вывод

В данной лабораторной работе были реализованы итераторы для очереди. Итераторы необходимы для удобного перемещения по последовательным контейнерам. Работа сложной не была, так как был опыт в реализации итераторов на первом курсе. Однако, лабораторная работа всё же было интересной и познавательной.

Исходный код

main.cpp

```
#include "TNaryTree.h"
#include "hexagon.h"
#include "titerator.h"
#include "TNaryTree_item.h"
#include <string>

int main()
{
    TNaryTree<hexagon> 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<hexagon>(new
   hexagon(Point(1, 4), Point(1, 2), Point(5, 6),
   Point(2, 8),
    Point(3, 1), Point(2, 6))), ""); // 1
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(2, 5), Point(1, 5), Point(16, 6),
   Point(3, 6),
    Point(1, 8), Point(4, 2))), "c"); // 2
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(3, 5), Point(9, 1), Point(7, 3),
   Point(1, 8),
    Point(5, 6), Point(4, 8))), "cb"); // 3
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(4, 4), Point(1, 2), Point(5, 6),
   Point(2, 8),
    Point(3, 1), Point(2, 6))), "cbb"); // 4
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(5, 5), Point(1, 5), Point(16, 6),
   Point(3, 6),
    Point(1, 8), Point(4, 2))), "cbbc"); // 5
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(6, 5), Point(9, 1), Point(7, 3),
   Point(1, 8),
    Point(5, 6), Point(4, 8))), "cc"); // 6
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(7, 4), Point(1, 2), Point(5, 6),
   Point(2, 8),
```

```
    Point(3, 1), Point(2, 6))), "ccb"); // 7
    a.Update(std::shared_ptr<hexagon>(new
   hexagon(Point(8, 5), Point(1, 5), Point(16, 6),
   Point(3, 6),
    Point(1, 8), Point(4, 2))), "cbc"); // 8
    // a.Update(std::shared_ptr<hexagon>(new
   hexagon(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<hexagon>(new
   hexagon(Point(10, 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<hexagon>(new
   hexagon(Point(11, 5), Point(9, 1), Point(7, 3),
   Point(1, 8),
    // Point(5, 6), Point(4, 8), Point(9, 5),
   Point(6, 4))), "cccb"); // 11
    for (auto i: a) {
        std::cout << *i << std::endl;
    }
    std::cout << a;
    std::cout << a.Area("cb") << "\n";
    TNaryTree<hexagon> b(a);
    std::cout << b;
    std::shared_ptr<hexagon> 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;
}
```

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;
            ~Figure() {};
};

#endif
```

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();

  bool operator==(Point& other);

  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_;
}

bool Point::operator==(Point& other)
{
    return this->x_ == other.x_ && this->y_ == other.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;
}
```

hexagon.h

```
#ifndef HEXAGON_H
#define HEXAGON_H

#include "point.h"
#include "figure.h"

class hexagon : figure
{
public:
    hexagon(std::istream& is);
    hexagon();
    ~hexagon();
    hexagon(Point a, Point b, Point c, Point d, Point e, Point f);

    size_t VertexesNumber();
    double Area();
    void Print(std::ostream& os);

    hexagon& operator=(const hexagon& other);
    bool operator==(hexagon& other);
    friend std::ostream& operator<<(std::ostream& os, hexagon& other);
    friend std::istream& operator>>(std::istream& is, hexagon& other);

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

#endif
```

hexagon.cpp

```
#include "hexagon.h"
#include "point.h"

hexagon::hexagon(std::istream& is)
{
```

```
    std::cin >> a_ >> b_ >> c_ >> d_;
    std::cin >> e_ >> f_;
}

hexagon::hexagon() : a_(0,0), b_(0,0), c_(0,0), d_(0, 0), e_(0,0), f_(0,0)
{}

hexagon::hexagon(Point a, Point b, Point c, Point d, Point e, Point f)
{
    this->a_ = a; this->b_ = b;
    this->c_ = c; this->d_ = d;
    this->e_ = e; this->f_ = f;
}

size_t hexagon::VertexesNumber()
{
    return (size_t)6;
}

double hexagon::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()
     - (b_.getX() * a_.getY() + c_.getX() * b_.getY() + d_.getX() * c_.getY() + e_.getX() * d_.getY() + f_.getX() *
e_.getY())));
}

hexagon& hexagon::operator=(const hexagon& other)
{
    this->a_ = other.a_; this->b_ = other.b_;
    this->c_ = other.c_; this->d_ = other.d_;
    this->e_ = other.e_; this->f_ = other.f_;
    return *this;
}

bool hexagon::operator==(hexagon& other)
{
    return this->a_ == other.a_ && this->b_ == other.b_ &&
    this->c_ == other.c_ && this->d_ == other.d_ &&
    this->e_ == other.e_ && this->f_ == other.f_;
}

std::ostream& operator<<(std::ostream& os, hexagon& oct)
{
    os << "Octagon: " << oct.a_ << " " << oct.b_ << " ";
    os << oct.c_ << " " << oct.d_ << " " << oct.e_ << " ";
    os << oct.f_ << "\n";
    return os;
}
```

```
std::istream& operator>>(std::istream& is, hexagon& other)
{
    is >> other.a_ >> other.b_ >> other.c_ >> other.d_;
    is >> other.e_ >> other.f_;
    return is;
}

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

hexagon::~hexagon(){}
```

TNaryTree.h

```
#ifndef TNARY_TREE
#define TNARY_TREE

#include "hexagon.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);
   }

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

   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"

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 T>
TIterator<TreeItem<T>, T> TNaryTree<T>::begin() {
```

```
 return TIterator<TreeItem<T>, T>(root);
}

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

#include "hexagon.h"
template class TNaryTree<hexagon>;
template std::ostream& operator<<(std::ostream& os, const TNaryTree<hexagon>& tree);
```

TNaryTree_item.cpp

```
#include "TNaryTree_item.h"

template<class T>
std::shared_ptr<TreeItem<T>> TreeItem<T>::GetNext()
{
    if (this->son != nullptr) {
        return this->son;
    }
    if (this->brother != nullptr) {
        return this->brother;
    }
    std::shared_ptr<TreeItem<T>> cur = this->parent;
    if (cur->brother != nullptr) {
        return cur->brother;
    }
    return nullptr;
}

#include "hexagon.h"
template class TreeItem<hexagon>;
```

TNaryTree_item.h

```
#ifndef TNARYTREE_ITEM
#define TNARYTREE_ITEM

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

template<class T>
class TreeItem
{
```

```
public:

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

    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
```

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
```