МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

ЛАБОРАТОРНАЯ РАБОТА №4

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

Студент *Ханнанов Руслан Маратович, группа М8О-208Б-20*

Преподаватель *Дорохов Евгений Павлович*

Условие

Задание: Вариант 22: N-дерево. Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру (колонка фигура 1), согласно вариантам задания. Классы должны удовлетворять следующим правилам:

1. Требования к классу фигуры аналогичны требованиям из лабораторной работы №1.
2. Классы фигур должны содержать набор следующих методов:
    * Перегруженный оператор ввода координат вершин фигуры из потока std::istream (»). Он должен заменить конструктор, принимающий координаты вершин из стандартного потока.
    * Перегруженный оператор вывода в поток std::ostream («), заменяющий метод Print из лабораторной работы 1.
    * Оператор копирования (=)
    * Оператор сравнения с такими же фигурами (==)
3. Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).
4. Класс-контейнер должен содержать набор следующих методов:
    * InsertFirst() – метод, добавляющий элемент в начало списка
    * InsertLast() – метод, добавляющий фигуру в конец списка
    * Insert() - метод, добавляющий фигуру в произвольное место списка
    * RemoveFirst() - метод, удаляющий первый элемент списка
    * RemoveLast() - метод, удаляющий последний элемент списка
    * Remove() - метод, удаляющий произвольный элемент списка
    * Empty() - метод, проверяющий пустоту списка
    * Length() - метод, возвращающий длину массива
    * operator« – выводит связанный список в соответствии с заданным форматом в поток вывода
    * Clear() - метод, удаляющий все элементы контейнера, но позволяющий пользоваться им.

Нельзя использовать:

1. Стандартные контейнеры std.

2. Шаблоны (template).
3. Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

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

Описание программы

Исходный код лежит в 10 файлах:

1. src/main.cpp: основная программа, взаимодействие с пользователем посредством комманд из меню
2. include/figure.h: описание абстрактного класса фигур
3. include/point.h: описание класса точки
4. include/node.h: описание класса ноды дерева
5. include/pentagon.h: описание класса пятиугольника, наследующегося от figures
6. include/tnary_tree.h: описание класса N-дерева
7. include/point.cpp: реализация класса точки
8. include/pentagon.cpp: реализация класса пятиугольника, наследующегося от figures
9. include/node.cpp: реализация класса ноды дерева
10. include/tnary_tree.cpp: реализация класса N-дерева

Протокол работы

Tree is not empty
Pentagon:(0, 0) (0, 2) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 3) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 8) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 4) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 6) (1, 3) (2, 3) (3, 0)

Pentagon:(0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
Pentagon:(0, 0) (0, 3) (1, 3) (2, 3) (3, 0)

Pentagon:(0, 0) (0, 8) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 4) (1, 3) (2, 3) (3, 0)

Pentagon:(0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
Pentagon:(0, 0) (0, 3) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 8) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 4) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 6) (1, 3) (2, 3) (3, 0)
34.5

Pentagon:(0, 0) (0, 0) (0, 0) (0, 0) (0, 0)
Pentagon:(0, 0) (0, 3) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 8) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 4) (1, 3) (2, 3) (3, 0)
Pentagon:(0, 0) (0, 6) (1, 3) (2, 3) (3, 0)

0: [7.5: [10], 8, 9]

Дневник отладки

Было непонятно, каким именно образом реализовывать добавление элемента в дерево, то есть по какому принципу, но после выбора ввода точного пути - реализовать получилось достаточно быстро.

Недочёты

Недочетов не заметил

Выводы

В процессе выполнения данной лабораторной работы я реализовал класс-контейнер N-дерево и методы работы с ним. Также мной было применено динамическое выделение памяти с помощью операторов new и delete.

Исходный код:

figure.h

```
#ifndef LAB2_FIGURE_H
#define LAB2_FIGURE_H

#include "point.h"

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

#endif
```

point.h

```
#ifndef LAB2_POINT_H
#define LAB2_POINT_H

#include <istream>

class Point {
 public:
  Point();
  Point(std::istream &is);
  Point(double x, double y);
  double dist(Point &other);
  friend double getx(Point &p);
  friend double gety(Point &p);
  friend std::istream &operator>>(std::istream &is, Point &p);
  friend std::ostream &operator<<(std::ostream &os, Point &p);
 private:
  double x_;
  double y_;
};
#endif //LAB2_POINT_H
```

point.cpp

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

```cpp
#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);
}

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;
}

double getx(Point& p) {
    return p.x_;
}

double gety(Point& p) {
    return p.y_;
}
```

pentagon.h

```cpp
#ifndef LAB2_PENTAGON_H
#define LAB2_PENTAGON_H
#include "figure.h"
#include "point.h"
#include <iostream>

class Pentagon : public Figure {
```

```
 public:
  Pentagon();
  Pentagon(Point v1, Point v2, Point v3, Point v4, Point v5);
  explicit Pentagon(std::istream &is);
  Pentagon(Pentagon &other);
  void Print(std::ostream &os) override;
  double Area() override;
  size_t VertexesNumber() override;
  ~Pentagon();
 private:
  Point v1, v2, v3, v4, v5;
};
#endif
```

pentagon.cpp

```
#include <cmath>
#include "pentagon.h"

Pentagon::Pentagon() : v1(0, 0), v2(0, 0), v3(0, 0), v4(0, 0), v5(0, 0) {
  //std::cout << "Default pentagon created" << std::endl;
}

Pentagon::Pentagon(Point v_1, Point v_2, Point v_3, Point v_4, Point v_5) :
    v1(v_1), v2(v_2), v3(v_3), v4(v_4), v5(v_5) {
  //std::cout << "Pentagon created" << std::endl;
}

Pentagon::Pentagon(Pentagon &other) :
    Pentagon(other.v1, other.v2, other.v3, other.v4, other.v5) {
  //std::cout << "Made copy of pentagon";
}

Pentagon::Pentagon(std::istream &is) {
  is >> v1 >> v2 >> v3 >> v4 >> v5;
  //std::cout << "Pentagon created via istream" << std::endl;
}

void Pentagon::Print(std::ostream &os) {
  os << "Pentagon:" << v1 << v2 << v3 << v4 << v5 << "\n";
}

Pentagon::~Pentagon() {
```

```
  //std::cout << "Object Pentagon ";
  //Print(std::cout);
  //std::cout << "deleted" << std::endl;
}

double Pentagon::Area() {
  Point ar[5];
  ar[0] = v1;
  ar[1] = v2;
  ar[2] = v3;
  ar[3] = v4;
  ar[4] = v5;
  double res = 0;
  for (unsigned i = 0; i < 5; i++) {
    Point p = i ? ar[i - 1] : ar[4];
    Point q = ar[i];
    res += (getx(p) - getx(q)) * (gety(p) + gety(q));
  }
  return fabs(res) / 2;
}

size_t Pentagon::VertexesNumber() {
  return 5;
}
```

node.h

```
#ifndef LAB2_NODE_H
#define LAB2_NODE_H
#include "pentagon.h"
class Node {
 public:
  Node();
  Node(const Pentagon &pentagon);

  Node *last_son();
  void Print(std::ostream &os);

  Node *son;
  Node *brother;
  Pentagon key;
};
#endif //LAB2_NODE_H
```

node.cpp

```
#include "node.h"

Node::Node(const Pentagon &pentagon) {
  key = pentagon;
  son = nullptr;
  brother = nullptr;
}

Node::Node() {
  son = nullptr;
  brother = nullptr;
}

Node *Node::last_son() {
  Node *ls = son;
  int number_of_sons = 0;
  while (ls->brother != nullptr) {
    ls = ls->brother;
    ++number_of_sons;
  }
  return ls;
}

void Node::Print(std::ostream &os) {
  key.Print(os);
}
```

tnary_tree.h

```
#ifndef LAB2_TNARY_TREE_H
#define LAB2_TNARY_TREE_H

#include "node.h"

class TNaryTree {
 public:
  TNaryTree();
  TNaryTree(int n);
  TNaryTree(const TNaryTree &other);
  void Update(const Pentagon &&pentagon, const std::string &&tree_path = "");
  void Print(std::ostream &os) const;
```

```
  void RemoveSubTree(const std::string &&tree_path);
  bool Empty();
  double Area(const std::string &&tree_path);
  friend std::ostream &operator<<(std::ostream &os, const TNaryTree &tree);
  Pentagon &GetItem(const std::string &&tree_path = "");
  virtual ~TNaryTree();

 private:
  int N;
  Node *root;

  Node *get_root();
  void set_root(const Pentagon &pentagon);
  void sub_copy(Node *&cur, Node *cp);
  void sub_print(Node *cur, std::ostream &os) const;
  void sub_remove(Node *cur);
  void sub_area(Node *cur, double &sum);
  void sub_print_operator(Node *cur, std::ostream &os) const;
};

#endif //LAB2_TNARY_TREE_H
```

tnary_tree.cpp

```
#include "tnary_tree.h"

//Constructor
TNaryTree::TNaryTree(int n) {
  N = n;
  root = nullptr;
}

TNaryTree::TNaryTree() {
  N = 3;
  root = nullptr;
}

void TNaryTree::Update(const Pentagon &&pentagon, const std::string &&tree_path) {
  if (tree_path == "") {
    if (root == nullptr) {
      set_root(pentagon);
    } else root->key = pentagon;
  } else {
```

```
    Node *cur = get_root();
    int degree = 1;
    for (int i = 0; i < tree_path.size() - 1; ++i) {
      if (tree_path[i] == 'c') {
        degree = 1;
        cur = cur->son;
        if (cur == nullptr) {
          throw std::invalid_argument("The node doesn't exist");
        }
      }
      if (tree_path[i] == 'b') {
        ++degree;
        cur = cur->brother;
        if (cur == nullptr) {
          throw std::invalid_argument("The node doesn't exist");
        }
      }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      if (cur->son == nullptr) {
        cur->son = new Node(pentagon);
      } else cur->son->key = pentagon;
    }
    if (tree_path[tree_path.size() - 1] == 'b') {
      ++degree;
      if (degree > N) {
        throw std::out_of_range("Node cannot be added due to overflow");
      }
      if (cur->brother == nullptr) {
        cur->brother = new Node(pentagon);
      } else cur->brother->key = pentagon;
    }
  }
}

Node *TNaryTree::get_root() {
  return root;
}

void TNaryTree::set_root(const Pentagon &pentagon) {
  root = new Node(pentagon);
}
```

```
void TNaryTree::Print(std::ostream &os) const {
  Node *cur = root;
  sub_print(cur, os);
}

void TNaryTree::sub_print(Node *cur, std::ostream &os) const {
  cur->Print(os);
  if (cur->son != nullptr) {
    sub_print(cur->son, os);
  }
  if (cur->brother != nullptr) {
    sub_print(cur->brother, os);
  }
  return;
}

bool TNaryTree::Empty() {
  if (root == nullptr)
    return true;
  else return false;
}

void TNaryTree::RemoveSubTree(const std::string &&tree_path) {
  Node *cur;
  if (tree_path == "") {
    sub_remove(root);
  } else {
    cur = get_root();
    for (int i = 0; i < tree_path.size() - 1; ++i) {
      if (tree_path[i] == 'c') {
        cur = cur->son;
      }
      if (tree_path[i] == 'b') {
        cur = cur->brother;
      }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      Node *cur_d = cur->son;
      cur->son = nullptr;
      sub_remove(cur_d);
    }
```

```
    if (tree_path[tree_path.size() - 1] == 'b') {
      Node *cur_d = cur->brother;
      cur->brother = nullptr;
      sub_remove(cur_d);
    }
  }
}

void TNaryTree::sub_remove(Node *cur) {
  if (cur->son != nullptr) {
    sub_remove(cur->son);
  }
  if (cur->brother != nullptr) {
    sub_remove(cur->brother);
  }
  delete (cur);
  return;
}

double TNaryTree::Area(const std::string &&tree_path) {
  double sum = 0;
  Node *cur;
  if (tree_path == "") {
    sub_area(root, sum);
  } else {
    cur = get_root();
    for (int i = 0; i < tree_path.size() - 1; ++i) {

      if (tree_path[i] == 'c') {

        cur = cur->son;

      }
      if (tree_path[i] == 'b') {

        cur = cur->brother;

      }

    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      sub_area(cur->son, sum);
```

```
    }
    if (tree_path[tree_path.size() - 1] == 'b') {
      sub_area(cur->brother, sum);
    }
  }
  return sum;
}

void TNaryTree::sub_area(Node *cur, double &sum) {
  if (cur->son != nullptr) {
    sub_area(cur->son, sum);
  }
  if (cur->brother != nullptr) {
    sub_area(cur->brother, sum);
  }
  sum += cur->key.Area();
}

TNaryTree::~TNaryTree() {
  sub_remove(root);
}

TNaryTree::TNaryTree(const TNaryTree &other) {
  N = other.N;
  Node *cp = other.root;
  root = new Node(cp->key);
  Node *cur = root;
  if (cp->son != nullptr) {
    sub_copy(cur->son, cp->son);
  }
  if (cp->brother != nullptr) {
    sub_copy(cur->brother, cp->brother);
  }
}

void TNaryTree::sub_copy(Node *&cur, Node *cp) {
  cur = new Node(cp->key);
  if (cp->son != nullptr) {
    sub_copy(cur->son, cp->son);
  }
  if (cp->brother != nullptr) {
    sub_copy(cur->brother, cp->brother);
```

```
  }
  return;
}

std::ostream &operator<<(std::ostream &os, const TNaryTree &tree) {
  Node *cur = tree.root;
  tree.sub_print_operator(cur, os);
  return os;
}

void TNaryTree::sub_print_operator(Node *cur, std::ostream &os) const {
  os << cur->key.Area();
  if (cur->son != nullptr) {
    os << ": [";
    sub_print_operator(cur->son, os);
    os << "]";
  }
  if (cur->brother != nullptr) {
    os << ", ";
    sub_print_operator(cur->brother, os);
    //os << "]";
  }
  return;
}

Pentagon &TNaryTree::GetItem(const std::string &&tree_path) {
  Node *cur;
  if (tree_path == "") {
    return root->key;
  } else {
    cur = get_root();
    for (int i = 0; i < tree_path.size() - 1; ++i) {
      if (tree_path[i] == 'c') {
        cur = cur->son;
      }
      if (tree_path[i] == 'b') {
        cur = cur->brother;
      }
    }
    if (tree_path[tree_path.size() - 1] == 'c') {
      return cur->son->key;
    }
```

```
    if (tree_path[tree_path.size() - 1] == 'b') {
      return cur->brother->key;
    }
  }
}
```

main.cpp

```
#include <iostream>
#include "pentagon.h"
#include "tnary_tree.h"

int main() {
  TNaryTree t(3);
  t.Update(Pentagon({0, 0}, {0, 2}, {1, 3}, {2, 3}, {3, 0}), "");
  if (t.Empty()) std::cout << "Tree is empty\n";
  else std::cout << "Tree is not empty\n";
  t.Update(Pentagon({0, 0}, {0, 3}, {1, 3}, {2, 3}, {3, 0}), "c");
  t.Update(Pentagon({0, 0}, {0, 4}, {1, 3}, {2, 3}, {3, 0}), "cb");
  t.Update(Pentagon({0, 0}, {0, 6}, {1, 3}, {2, 3}, {3, 0}), "cbb");
  t.Update(Pentagon({0, 0}, {0, 8}, {1, 3}, {2, 3}, {3, 0}), "cc");
  t.Print(std::cout);
  std::cout << "_____________________________________________\n";
  t.Update(Pentagon({0, 0}, {0, 0}, {0, 0}, {0, 0}, {0, 0}), "");

  t.RemoveSubTree("cbb");
  t.Print(std::cout);
  std::cout << "_____________________________________________\n";

  t.Update(Pentagon({0, 0}, {0, 6}, {1, 3}, {2, 3}, {3, 0}), "cbb");
  t.Print(std::cout);
  std::cout << t.Area("") << "\n";
  std::cout << "_____________________________________________\n";

  TNaryTree z(t);
  z.Print(std::cout);
  std::cout << "_____________________________________________\n";
  std::cout << z;

  return 0;
}
```