

Федеральное государственное бюджетное образовательное учреждение
высшего образования «Сибирский государственный университет
телекоммуникаций и информатики»

Кафедра ПМиК

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Тема: «Нарисовать двоичное дерево»

Выполнил: студент группы ИС-241

Кулик П.Е.

Проверил: доцент кафедры ПМиК

Ситняковская Е.И.

Новосибирск – 2023

Оглавление

ПОСТАНОВКА ЗАДАЧИ.....	3
ТЕХНОЛОГИИ ООП	4
СТРУКТУРА КЛАССОВ	5
ПРОГРАММНАЯ РЕАЛИЗАЦИЯ.....	7
РЕЗУЛЬТАТ РАБОТЫ	9
ЗАКЛЮЧЕНИЕ	15
ИСПОЛЬЗУЕМЫЕ ИСТОЧНИКИ	16
ПРИЛОЖЕНИЕ	17

ПОСТАНОВКА ЗАДАЧИ

Опираясь на формулировку темы курсовой работы была поставлена следующая задача: разработать графическое приложение, дающее пользователю возможность в реальном времени построить двоичное дерево путём добавления в него узлов с произвольными ключами. Помимо этого, в приложении должна быть возможность в реальном времени удалять узел с любым ключом. Также, дерево должно отображаться правильно, несмотря на количество добавленных в него узлов, то есть расстояние между теми узлами, которые находятся ближе к корню должно увеличиваться по мере роста высоты дерева для того, чтобы новые узлы могли отображаться корректно и не перекрывали друг друга. На тот случай, если дерево станет слишком большим и перестанет помещаться в окне, должен быть функционал, позволяющий изменять масштаб отображения, а также перемещать всё, что изображено в окне.

Исходя из требований к курсовой работе, проект должен быть реализован с применением различных технологий объектно-ориентированного программирования.

ТЕХНОЛОГИИ ООП

При разработке приложения применялись следующие технологии объектно-ориентированного программирования:

1. Инкапсуляция – все поля классов защищены типом доступа `protected` для того, чтобы они были недоступны во внешних функциях, но при этом были доступны для наследования. Обращение к этим полям и взаимодействие с ними осуществляется через специальные методы.
2. Наследование – при разработке некоторых классов было применено наследование как от тех классов, которые были написаны при реализации проекта, так и от классов графической библиотеки.
3. Полиморфизм - при реализации классов происходило переопределении некоторых функций.
4. Конструкторы. Перегрузка конструкторов – для классов были написаны как конструкторы по умолчанию (если это имело смысл), так и конструкторы с параметрами.
5. Списки инициализации – все конструкторы описаны вместе со списками инициализации для инициализации полей класса.
6. Виртуальные функции – использовались при создании абстрактного класса, который содержал в себе только определение функций.
7. Множественное наследование – данная технология применялась для построения чёткой и логической структуры проекта.
8. Массивы указателей на объекты – от библиотечного класса был создан производный класс специально для того, чтобы после этого можно было создать массив объектов такого класса.
9. Параметры по умолчанию – данная технология использовалась для конструкторов по умолчанию и для тех данных, ввод которых может быть не обязательным при вызове конструктора.
10. Объекты использовались в качестве аргументов и возвращаемых значений.

СТРУКТУРА КЛАССОВ

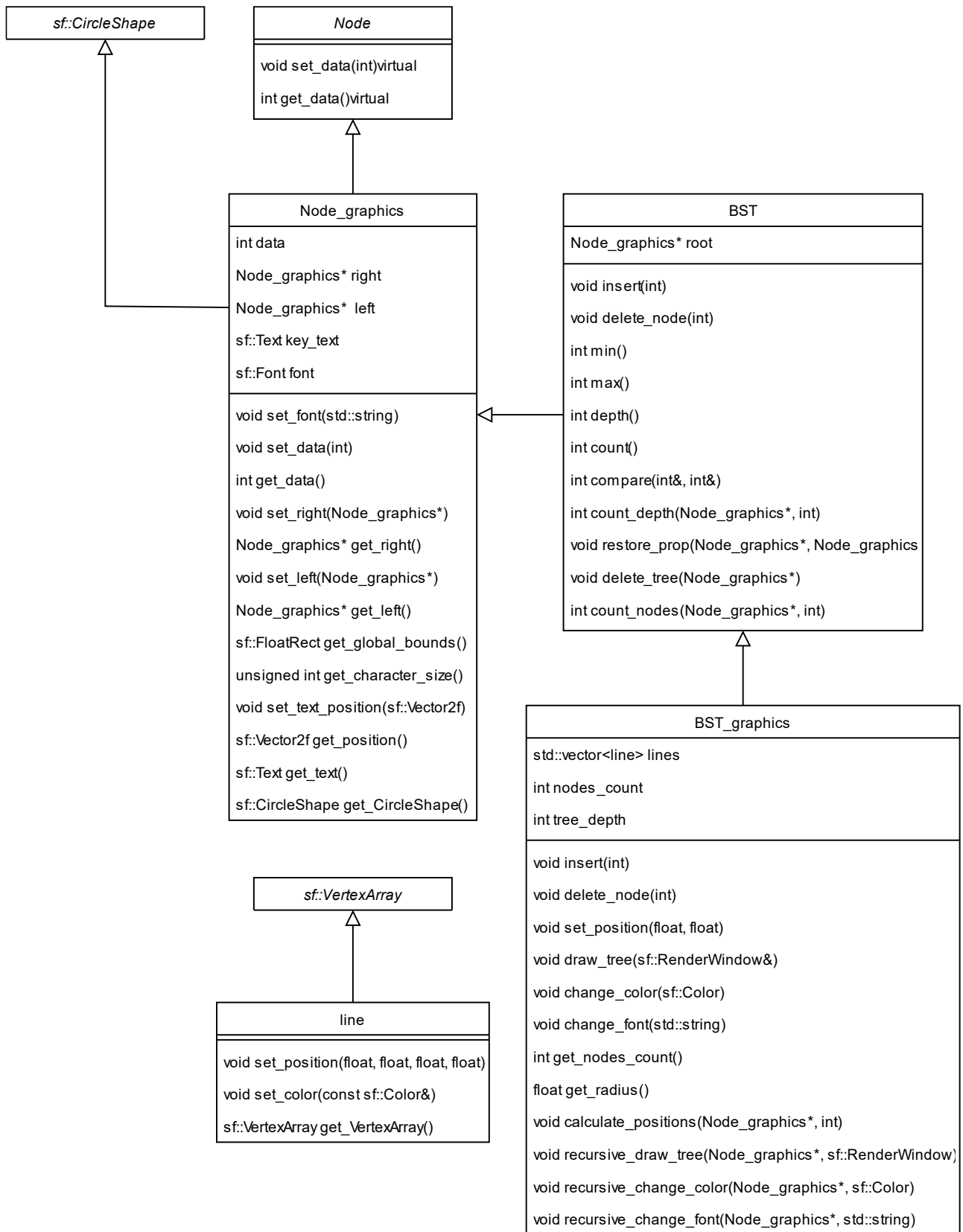


Рисунок 1. Структура классов.

На рисунке 1 схематично представлена структура наследования всех созданных для приложения классов. Всё начинается с абстрактного класса Node, который содержит в себе 2 экземпляра виртуальных методов. На основе этого класса может быть описан любой производный класс узла, хранящего в себе целочисленные значения.

Класс Node_graphics является наследованием от класса Node и CircleShape из библиотеки SFML. Данный класс представляет собой описание узла бинарного дерева, предназначенного для отрисовки в виде круга.

Класс BST является дочерним классом узла, что логично, так как даже один узел бинарного дерева, лежащий в его корне, сам по себе является бинарным деревом с одним узлом. В этом классе описаны основные операции такой структуры данных как бинарное дерево поиска.

От класса BST наследуется класс BST_graphics, который является расширением для класса BST и содержит в себе методы добавления и удаления вершин при работе с нарисованным деревом. Методы подсчёта координат всех вершин дерева, а также его рёбер, которые представлены в виде массива объектов класса line и правильной отрисовки дерева. Помимо этого, в этом классе описаны методы для изменения шрифта в дереве и изменения его цвета.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

При разработке программы использовались такие библиотеки как SFML и ImGUI. Первая библиотека позволяет создать окно и нарисовать в нём какие-либо двухмерные фигуры. Конкретно в той программе, которая была написана для данной курсовой работы, использовалась возможность рисования кругов и линий, а также текста для того, чтобы обозначать то, какие ключи хранятся в узлах дерева. Вторая библиотека позволяет добавить к приложению меню с пользовательским вводом. Данная особенность позволяет создать удобное интерактивное приложение, которое наглядно демонстрирует то, как происходит добавление узлов в дерево, а также то, как перестраивается дерево при удалении узлов из него.

Сама программа реализована внутри функции `main` с использованием всех разработанных классов. Внутри главной функции имеется цикл, который выполняется до тех пор, пока открыто окно приложения и в этом же цикле обрабатываются все пользовательские действия и отрисовка дерева.

Изначально имеется объект класса `BST_graphics`, созданный с конструктором по умолчанию и хранящий в себе пустое дерево. Всё, что доступно пользователю – это меню, в котором есть поле для ввода чисел, кнопка “Add Node” и кнопка “Delete Node”. У пользователя есть возможность управлять программой с помощью данных кнопок, либо же с помощью клавиатуры, так как добавление узла происходит также при нажатии на клавишу “Enter”, а удаление происходит при нажатии на клавишу “Delete”.

После того, как программа получает сигнал о том, что нужно добавить или удалить узел, она вызывает методы класса `BST_graphics` для того, чтобы сделать это. В том случае, если дерево пустое и в него добавляется первый узел, вызывается метод смены позиции для того, чтобы этот узел был сверху и по центру. Также смена местоположения происходит при каждом удалении узла для того, чтобы корень дерева всегда оставался на одном и том же месте.

Данная реализация программы также позволяет масштабировать то, что изображено в окне при помощи колёсика мыши. Для этого используется обработка события вращения колёсика мыши и стандартные методы библиотеки SFML. Помимо этого, имеется возможность перемещения отрисованной области с помощью перетягивания курсором. Для этого в каждом кадре, в котором зажата левая кнопка мыши сохраняются координаты курсора, а потом сравниваются с новыми координатами и на основании разницы между ними происходит смещение обзора.

РЕЗУЛЬТАТ РАБОТЫ

В конечном итоге получилось интерактивное приложение, которое могло бы использоваться в обучающих целях. Стартовое положение приложения продемонстрировано на рисунке 2.

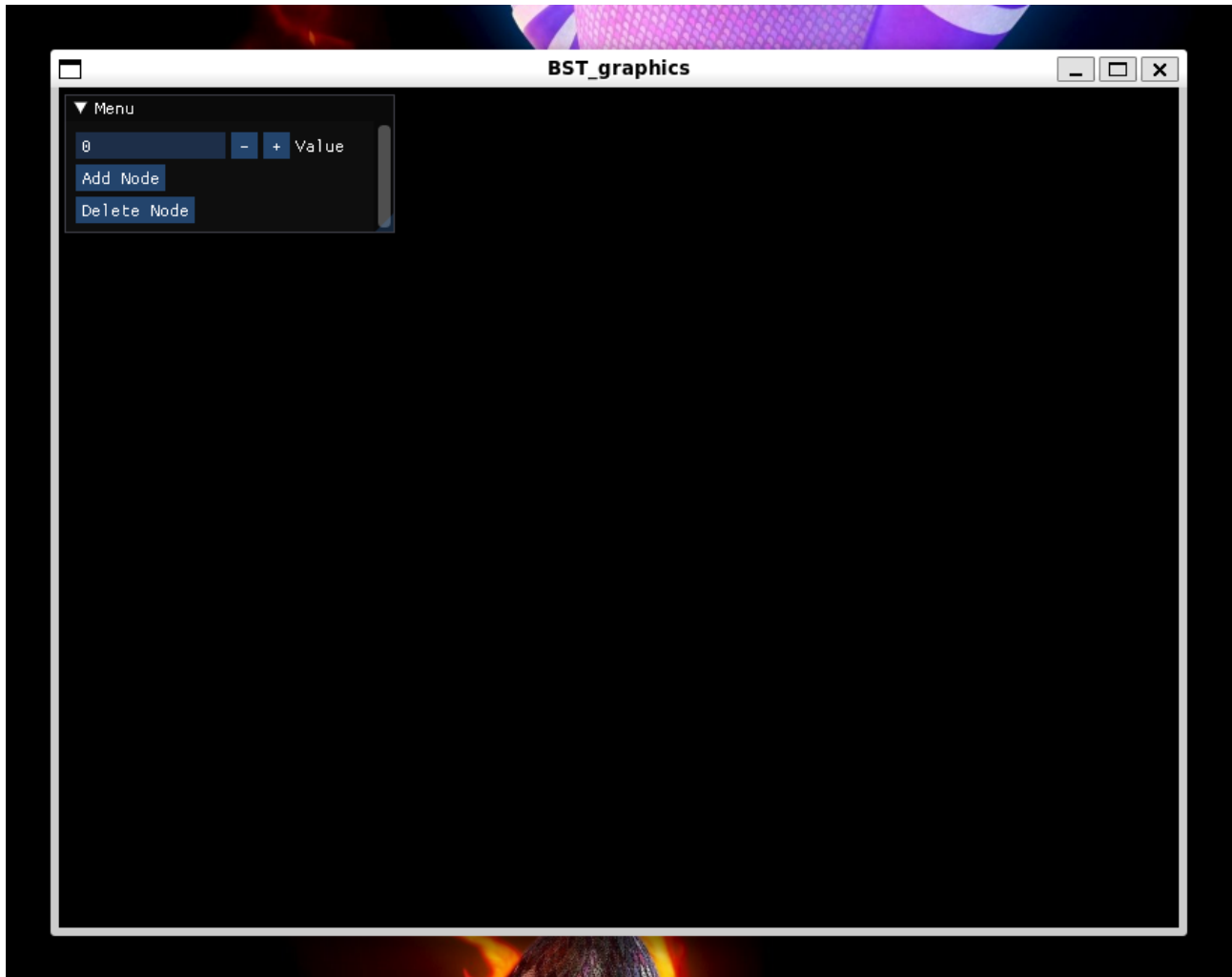


Рисунок 2. Начальное положение приложения.

На рисунке наглядно видно пустой “холст”, на котором пока ещё не нарисовано никакое дерево и меню для добавления узлов.

На рисунке 3 изображено состояние приложения, в котором в дереве содержатся 3 узла, а высота дерева 2.

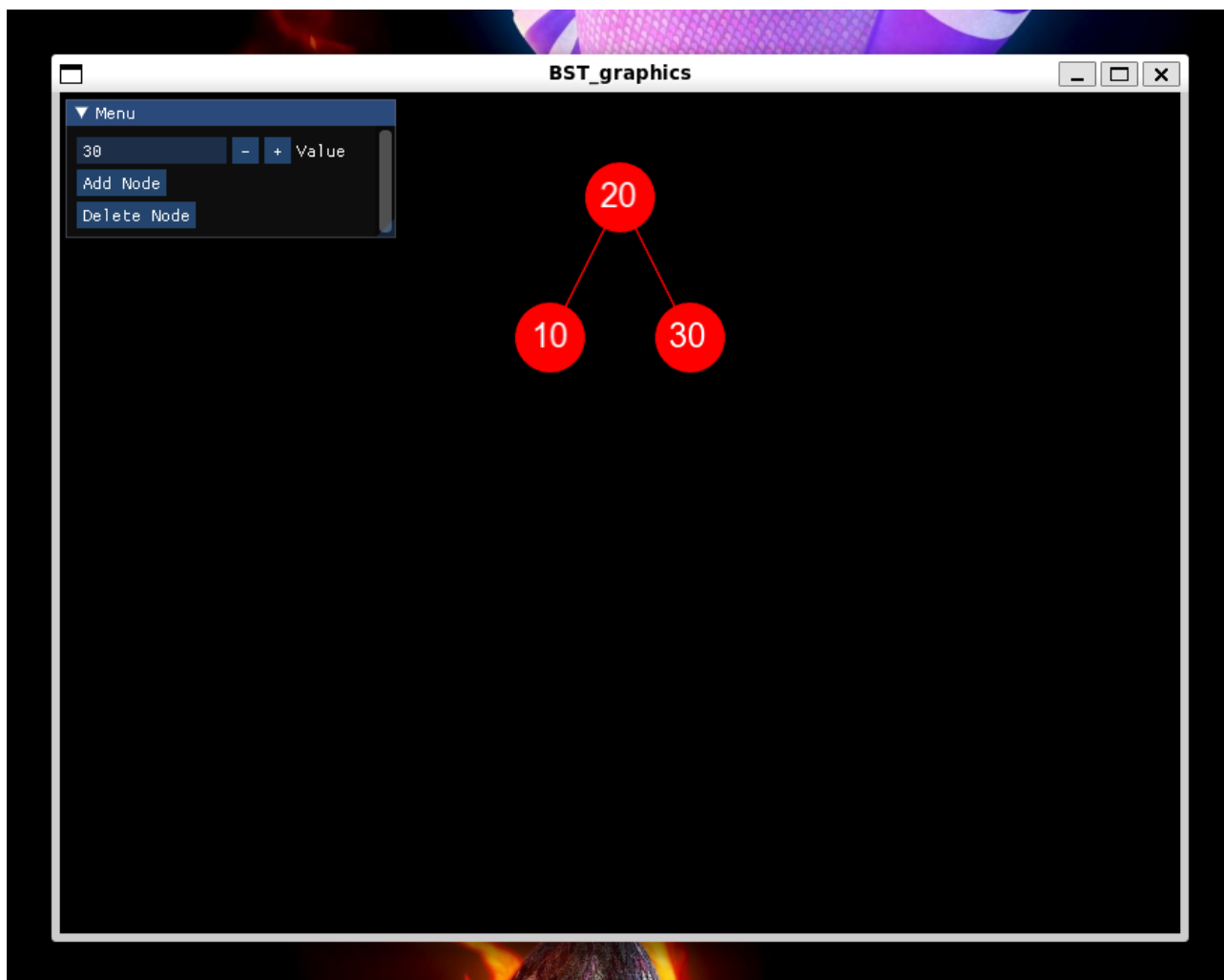


Рисунок 3. Дерево с тремя узлами.

Если добавить в дерево большее количество узлов, то у него увеличится высота, и координаты всех его и рёбер будут пересчитаны для того, чтобы правильно нарисовать все узлы. Это наглядно видно на рисунке 4.

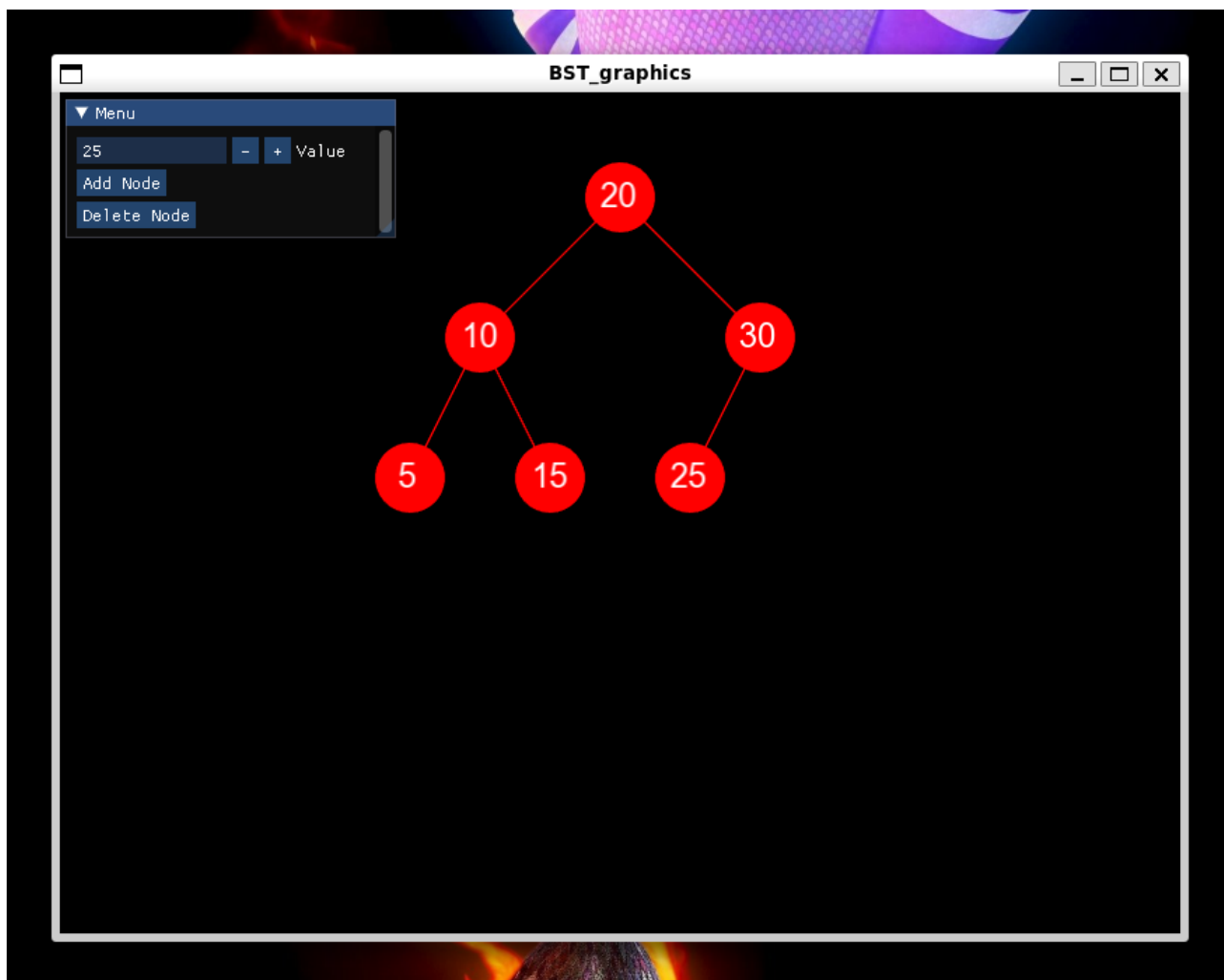


Рисунок 4. Дерево с семью узлами.

Если же добавить в дерево ещё больше узлов, то оно может стать настолько большим, что выйдет за пределы области видимости окна. Для таких случаев предусмотрено масштабирование и возможность передвигать область видимости окна. Данные возможности продемонстрированы на рисунках 5 и 6.

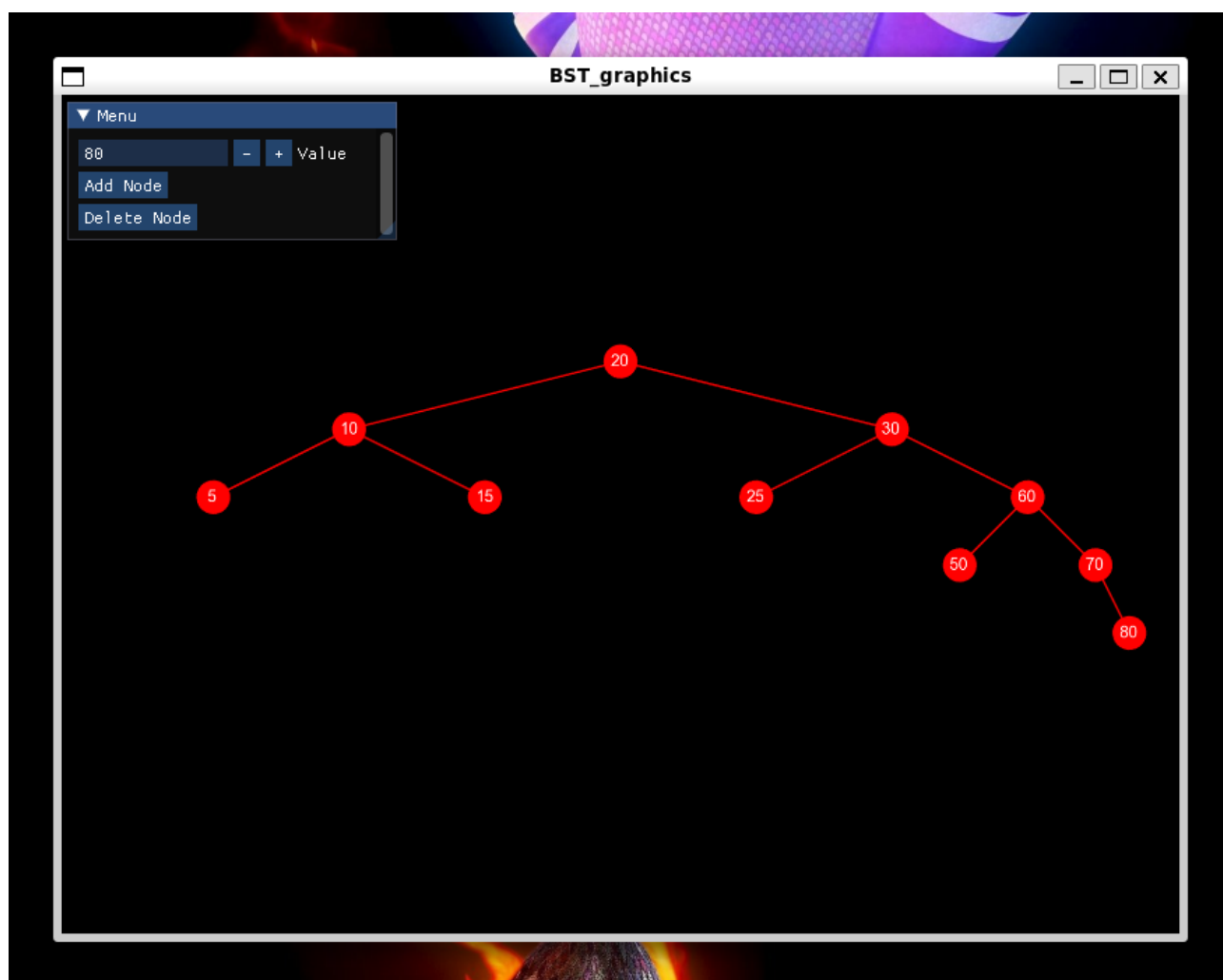


Рисунок 5. Масштабирование.

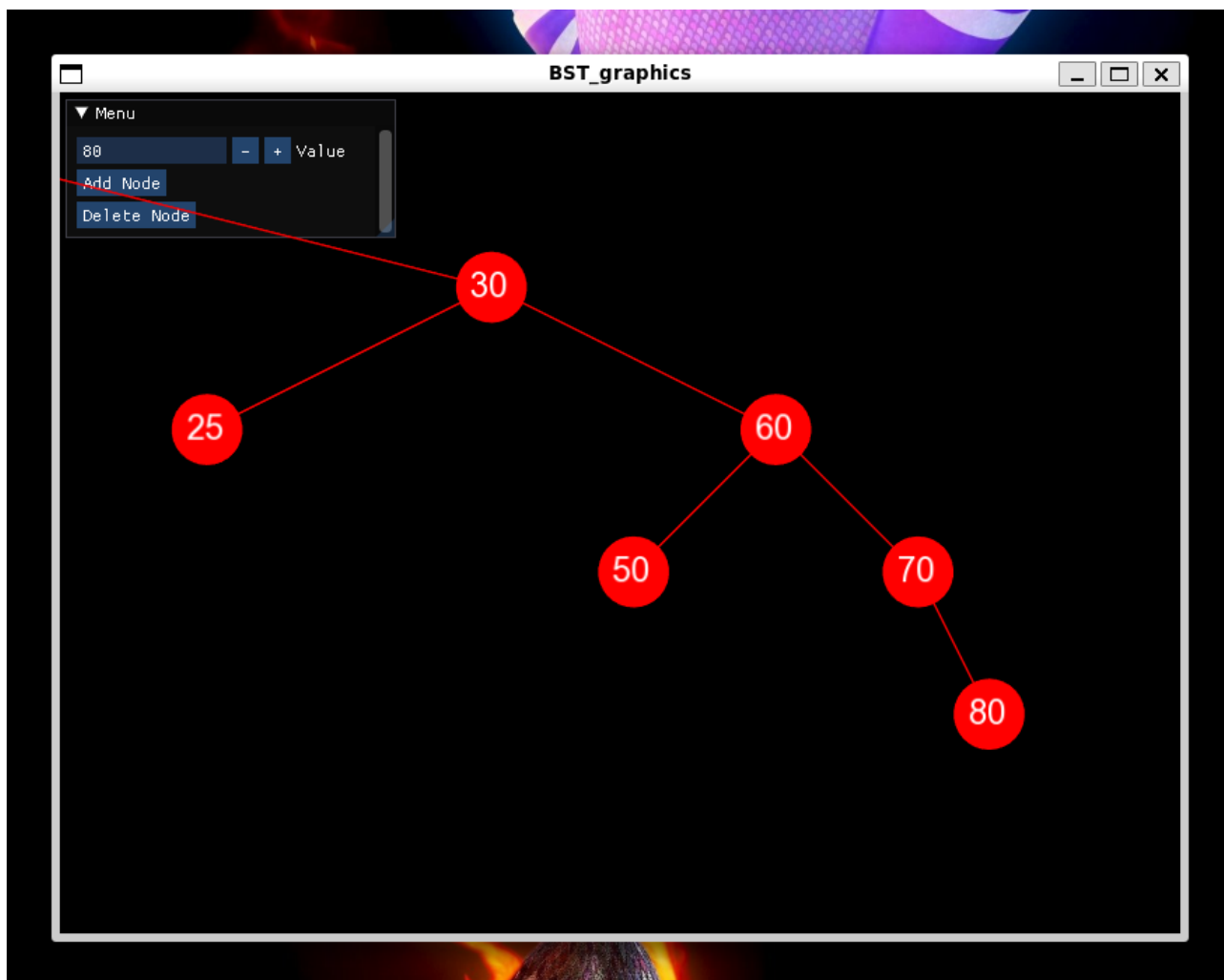


Рисунок 6. Перемещение области видимости.

Помимо этого, в программе предусмотрена возможность удаления узлов. При удалении происходит перестройка дерева с соблюдением свойств бинарного дерева, а также происходит пересчёт координат узлов для правильной отрисовки, а также для того, чтобы корень всегда оставался в одном и том же положении. На рисунке 7 продемонстрировано состояние дерева после удаления из него корня.

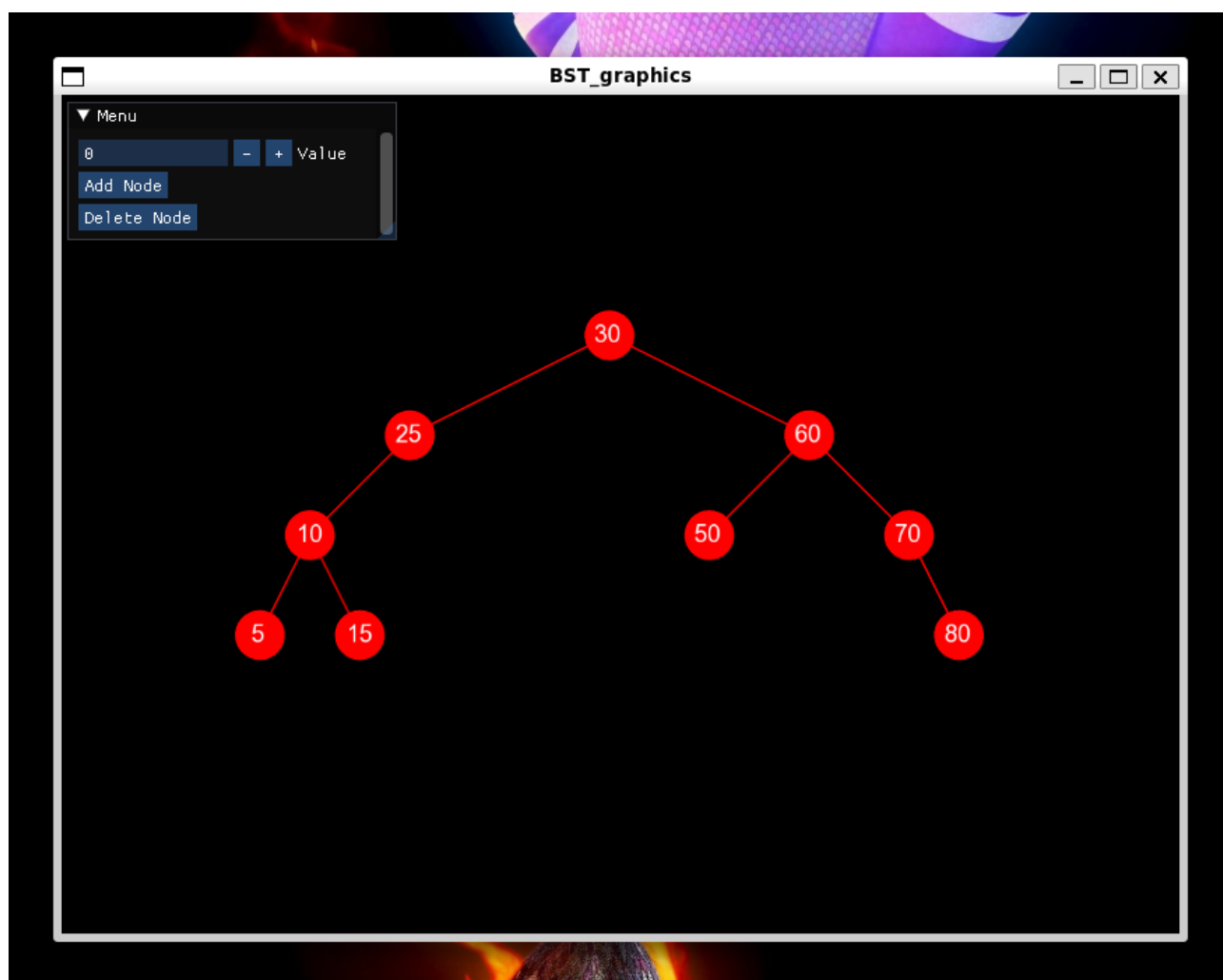


Рисунок 7. Удаление корня из дерева.

Таким образом, приложение наглядно демонстрирует основные свойства бинарного дерева, позволяя изучить дерево на интерактивном и понятном примере.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы были продемонстрированы различные технологии ООП, применённые для разработки полноценного оконного приложения, представляющего собой интерактивную модель двоичного дерева. Реализованное приложение полностью соответствует поставленной задаче, а также принципам объектно-ориентированного программирования.

Данная курсовая работа наглядно демонстрирует те возможности в сфере разработки приложений, которые предоставляет объектно-ориентированное программирование.

ИСПОЛЬЗУЕМЫЕ ИСТОЧНИКИ

1. Курносов, М. Г., Берлизов, Д. М. АЛГОРИТМЫ И СТРУКТУРЫ ОБРАБОТКИ ИНФОРМАЦИИ [Текст] / М. Г. Курносов, Д. М. Берлизов — 1-е изд. — Новосибирск: Параллель, 2019 — 227 с.
2. Laurent Gomila Tutorials for SFML 2.6 / Laurent Gomila [Электронный ресурс] // sfml-dev : [сайт]. — URL: <https://www.sfm1-dev.org/tutorials/2.6/> (дата обращения: 07.12.2023).

ПРИЛОЖЕНИЕ

Исходный код программы

main.cpp

```
1  #include <BST_graphics.h>
2  #include <SFML/Graphics.hpp>
3  #include <imgui-SFML.h>
4  #include <imgui.h>
5  #include <iostream>
6
7  enum { x_size = 800, y_size = 600 };
8
9  int main()
10 {
11     BST_graphics tree;
12
13     sf::ContextSettings settings;
14
15     settings.antialiasingLevel = 8;
16
17     sf::RenderWindow window(
18         sf::VideoMode(x_size, y_size),
19         "BST_graphics",
20         sf::Style::Default,
21         settings);
22     window.setPosition(sf::Vector2i(0, 0));
23
24     if (!ImGui::SFML::Init(window)) {
25         std::cout << "Can't open ImGUI!\n";
26         return 1;
27     }
28
29     sf::View view;
30     view.reset(sf::FloatRect(0, 0, x_size, y_size));
31     window.setView(view);
32
33     sf::Vector2f prev_mouse_pos;
34     bool is_dragging = false;
35
36     int input_value = 0;
37     bool add_node_button_pressed = false;
38     bool delete_node_button_pressed = false;
39
40     while (window.isOpen()) {
41         sf::Event event;
42         while (window.pollEvent(event)) {
43             ImGui::SFML::ProcessEvent(window, event);
44
45             if (event.type == sf::Event::Closed)
46                 window.close();
47
48             if (event.type == sf::Event::KeyPressed) {
49                 if (event.key.code == sf::Keyboard::Enter) {
50                     if (tree.get_nodes_count() == 0) {
51                         tree.insert(input_value);
52                         tree.set_position(
53                             x_size / 2 - tree.get_radius(),
54                             tree.get_radius() * 2);
```

```

55         } else {
56             tree.insert(input_value);
57         }
58         input_value = 0;
59     }
60     if (event.key.code == sf::Keyboard::Delete) {
61         if (tree.get_nodes_count() != 0) {
62             tree.delete_node(input_value);
63             if (tree.get_nodes_count() != 0)
64                 tree.set_position(
65                     x_size / 2 - tree.get_radius(),
66                     tree.get_radius() * 2);
67         }
68         input_value = 0;
69     }
70     if (event.key.code == sf::Keyboard::Escape)
71         window.close();
72 }
73 if (event.type == sf::Event::MouseButtonPressed)
74     if (event.mouseButton.button == sf::Mouse::Left) {
75         is_dragging = true;
76         prev_mouse_pos = window.mapPixelToCoords(
77             sf::Mouse::getPosition(window));
78     }
79
80     if (event.type == sf::Event::MouseButtonReleased)
81         if (event.mouseButton.button == sf::Mouse::Left) {
82             is_dragging = false;
83         }
84
85     if (event.type == sf::Event::MouseWheelScrolled) {
86         if (event.mouseWheelScroll.delta > 0) {
87             view.zoom(0.9f);
88         } else if (event.mouseWheelScroll.delta < 0) {
89             view.zoom(1.1f);
90         }
91     }
92     window.setView(view);
93 }
94 }
95
96 if (is_dragging) {
97     sf::Vector2f mousePos
98         = window.mapPixelToCoords(sf::Mouse::getPosition(window));
99     sf::Vector2f delta = prev_mouse_pos - mousePos;
100     view.move(delta);
101     window.setView(view);
102 }
103
104 ImGui::SFML::Update(window, sf::seconds(1.f / 60.f));
105
106 ImGui::Begin("Menu");
107
108 ImGui::InputInt("Value", &input_value);
109
110 if (ImGui::Button("Add Node"))
111     add_node_button_pressed = true;
112
113 if (ImGui::Button("Delete Node"))
114     delete_node_button_pressed = true;
115
116 ImGui::End();
117

```

```

118         if (add_node_button_pressed) {
119             if (tree.get_nodes_count() == 0) {
120                 tree.insert(input_value);
121                 tree.set_position(
122                     x_size / 2 - tree.get_radius(), tree.get_radius() * 2);
123             } else {
124                 tree.insert(input_value);
125             }
126             add_node_button_pressed = false;
127             input_value = 0;
128         }
129
130         if (delete_node_button_pressed) {
131             if (tree.get_nodes_count() != 0) {
132                 tree.delete_node(input_value);
133                 if (tree.get_nodes_count() != 0)
134                     tree.set_position(
135                         x_size / 2 - tree.get_radius(),
136                         tree.get_radius() * 2);
137             }
138
139             delete_node_button_pressed = false;
140             input_value = 0;
141         }
142
143         window.clear();
144
145         if (tree.get_nodes_count() != 0)
146             tree.draw_tree(window);
147         ImGui::SFML::Render(window);
148         window.display();
149     }
150
151     ImGui::SFML::Shutdown();
152     return 0;
153 }

```

BST.cpp

```
1  #include <BST.h>
2
3  void Node_graphics::set_data(int new_data)
4  {
5      data = new_data;
6  }
7
8  int Node_graphics::get_data()
9  {
10     return data;
11 }
12
13 void Node_graphics::set_right(Node_graphics* new_right)
14 {
15     right = new_right;
16 }
17
18 void Node_graphics::set_left(Node_graphics* new_left)
19 {
20     left = new_left;
21 }
22
23 Node_graphics* Node_graphics::get_right()
24 {
25     return right;
26 }
27
28 Node_graphics* Node_graphics::get_left()
29 {
30     return left;
31 }
32
33 void Node_graphics::set_font(std::string font_name)
34 {
35     if (!font.loadFromFile(font_name)) {
36         std::cerr << "Failed to load font!" << std::endl;
37     }
38     key_text.setFont(font);
39 }
40
41 sf::FloatRect Node_graphics::get_global_bounds()
42 {
43     return key_text.getGlobalBounds();
44 }
45
46 unsigned int Node_graphics::get_character_size()
47 {
48     return key_text.getCharacterSize();
49 }
50
51 void Node_graphics::set_text_position(sf::Vector2f &new_position)
52 {
53     key_text.setPosition(new_position);
54 }
55
56 sf::Vector2f Node_graphics::get_position()
57 {
58     return this->getPosition();
59 }
60
```

```

61 sf::Text Node_graphics::get_text()
62 {
63     return key_text;
64 }
65
66 sf::CircleShape Node_graphics::get_CircleShape()
67 {
68     return *this;
69 }
70
71 BST::~~BST()
72 {
73     delete_tree(root);
74 };
75
76 void BST::insert(int value)
77 {
78     Node_graphics* new_Node_graphics = new Node_graphics(value);
79     if (root == nullptr) {
80         root = new_Node_graphics;
81         return;
82     }
83
84     Node_graphics* current = root;
85     Node_graphics* prev = nullptr;
86
87     int compare_result;
88
89     while (current) {
90         compare_result = compare(value, current->get_data());
91         if (compare_result == 1) {
92             prev = current;
93             current = current->get_right();
94         }
95         if (compare_result == -1) {
96             prev = current;
97             current = current->get_left();
98         }
99         if (compare_result == 0) {
100             new_Node_graphics->~Node_graphics();
101             return;
102         }
103     }
104     switch (compare_result) {
105     case 1:
106         prev->set_right(new_Node_graphics);
107         break;
108     case -1:
109         prev->set_left(new_Node_graphics);
110         break;
111     }
112 }
113
114 void BST::delete_node(int value)
115 {
116     if (root == nullptr) {
117         std::cout << "delete_Node_graphics() called on an empty tree\n";
118         return;
119     }
120     Node_graphics* current = root;
121     Node_graphics* prev = nullptr;
122
123     int compare_result;

```

```

124     while (current) {
125         compare_result = compare(value, current->get_data());
126         if (compare_result == 1) {
127             prev = current;
128             current = current->get_right();
129         }
130         if (compare_result == -1) {
131             prev = current;
132             current = current->get_left();
133         }
134         if (compare_result == 0) {
135             if (prev == nullptr) {
136                 restore_prop(prev, current, compare_result);
137                 return;
138             }
139             compare_result = compare(value, prev->get_data());
140             restore_prop(prev, current, compare_result);
141             current->~Node_graphics();
142             return;
143         }
144     }
145 }
146
147 int BST::min()
148 {
149     if (root == nullptr) {
150         std::cout << "min() called on an empty tree\n";
151         return 0;
152     }
153
154     Node_graphics* current = root;
155
156     while (current->get_left())
157         current->set_left(current->get_left());
158     return current->get_data();
159 }
160
161 int BST::max()
162 {
163     if (root == nullptr) {
164         std::cout << "max() called on an empty tree\n";
165         return 0;
166     }
167
168     Node_graphics* current = root;
169
170     while (current->get_right())
171         current->set_right(current->get_right());
172     return current->get_data();
173 }
174
175 int BST::depth()
176 {
177     return count_depth(root, 1);
178 }
179
180 int BST::count()
181 {
182     return count_nodes(root, 1);
183 }
184
185 int BST::compare(const int& a, const int& b)
186 {

```

```

187     if (a < b)
188         return -1;
189     if (a > b)
190         return 1;
191
192     return 0;
193 }
194
195 int BST::count_depth(Node_graphics* root, int depth)
196 {
197     if (root == nullptr)
198         return 0;
199     int left = depth, right = depth;
200     if (root->get_left())
201         left = count_depth(root->get_left(), depth + 1);
202     if (root->get_right())
203         right = count_depth(root->get_right(), depth + 1);
204     return right > left ? right : left;
205 }
206
207 void BST::restore_prop(
208     Node_graphics* prev, Node_graphics* current, int compare_result)
209 {
210     Node_graphics* right = current->get_right();
211     Node_graphics* left = current->get_left();
212     if (right) {
213         current = current->get_right();
214         if (left) {
215             while (current->get_left())
216                 current = current->get_left();
217             current->set_left(left);
218         }
219     } else {
220         right = left;
221     }
222     switch (compare_result) {
223     case -1:
224         prev->set_left(right);
225         break;
226     case 1:
227         prev->set_right(right);
228         break;
229     case 0:
230         root = right;
231     }
232 }
233
234 void BST::delete_tree(Node_graphics* root)
235 {
236     if (root == nullptr)
237         return;
238     if (root->get_right())
239         delete_tree(root->get_right());
240     if (root->get_left())
241         delete_tree(root->get_left());
242     delete root;
243 }
244
245 int BST::count_nodes(Node_graphics* root, int count)
246 {
247     if (root == nullptr)
248         return 0;
249     if (root->get_right())

```

250	count = count_nodes(root->get_right(), count + 1);
251	if (root->get_left())
252	count = count_nodes(root->get_left(), count + 1);
253	return count;
254	}

BST.h

```

1  #pragma once
2
3  #include <cstring>
4  #include <iostream>
5
6  #include <SFML/Graphics.hpp>
7
8  class Node {
9  public:
10     virtual void set_data(int data) = 0;
11     virtual int get_data() = 0;
12 };
13
14 class Node_graphics : public Node, public sf::CircleShape {
15 protected:
16     int data;
17     Node_graphics* right;
18     Node_graphics* left;
19     sf::Text key_text;
20     sf::Font font;
21
22 public:
23     Node_graphics(){};
24     Node_graphics(int value) : data(value), right(nullptr), left(nullptr)
25     {
26         if (!font.loadFromFile("arial.ttf")) {
27             std::cerr << "Failed to load font!" << std::endl;
28         }
29         key_text.setFont(font);
30         key_text.setFillColor(sf::Color::White);
31         key_text.setCharacterSize(24);
32         key_text.setString(std::to_string(value));
33         key_text.setPosition(0, 0);
34         this->setRadius(25.0f);
35         this->setFillColor(sf::Color::Red);
36     };
37     ~Node_graphics(){};
38
39     void set_font(std::string font_name);
40     void set_data(int new_data) override;
41     int get_data() override;
42     void set_right(Node_graphics* new_right);
43     Node_graphics* get_right();
44     void set_left(Node_graphics* new_left);
45     Node_graphics* get_left();
46     sf::FloatRect get_global_bounds();
47     unsigned int get_character_size();
48     void set_text_position(sf::Vector2f &new_position);
49     sf::Vector2f get_position();
50     sf::Text get_text();
51
52     sf::CircleShape get_CircleShape();
53 };
54
55 class BST : public Node_graphics {

```



```

56 protected:
57     Node_graphics* root;
58
59 public:
60     BST() : root(nullptr){};
61     BST(int value) : root(new Node_graphics(value)){};
62     ~BST();
63
64     virtual void insert(int value);
65
66     virtual void delete_node(int value);
67
68     int min();
69
70     int max();
71
72     int depth();
73
74     int count();
75
76 private:
77     int compare(const int& a, const int& b);
78
79     int count_depth(Node_graphics* root, int depth);
80
81     void restore_prop(Node_graphics* prev, Node_graphics* current, int compare_result);
82
83     void delete_tree(Node_graphics* root);
84
85     int count_nodes(Node_graphics* root, int count);
86 };

```

line.cpp

```

1  #include <SFML/Graphics.hpp>
2  #include <iostream>
3
4  #include <line.h>
5
6  line::line()
7  {
8      this->setPrimitiveType(sf::Lines);
9      this->resize(2);
10     set_color(sf::Color::Red);
11 }
12 line::line(float x1, float y1, float x2, float y2)
13 {
14     this->setPrimitiveType(sf::Lines);
15     this->resize(2);
16     set_color(sf::Color::Red);
17     set_position(x1, y1, x2, y2);
18 }
19 void line::set_position(float x1, float y1, float x2, float y2)
20 {
21     (*this)[0].position = sf::Vector2f(x1, y1);
22     (*this)[1].position = sf::Vector2f(x2, y2);
23 }
24 void line::set_color(const sf::Color& color)
25 {
26     (*this)[0].color = color;
27     (*this)[1].color = color;
28 }
29 sf::VertexArray line::get_VertexArray()

```

```

30 {
31     return *this;
32 }

```

line.h

```

1 #pragma once
2
3 #include <SFML/Graphics.hpp>
4 #include <iostream>
5
6 class line : public sf::VertexArray {
7 public:
8     line();
9     line(float x1, float y1, float x2, float y2);
10    void set_position(float x1, float y1, float x2, float y2);
11    void set_color(const sf::Color& color);
12    sf::VertexArray get_VertexArray();
13 };

```

BST_graphics.cpp

```

1 #include <BST_graphics.h>
2
3 void BST_graphics::insert(int value)
4 {
5     BST::insert(value);
6     tree_depth = this->depth();
7     nodes_count = this->count();
8     lines.clear();
9     calculate_positions(root, tree_depth - 1);
10 }
11 void BST_graphics::delete_node(int value)
12 {
13     BST::delete_node(value);
14     tree_depth = BST::depth();
15     nodes_count = BST::count();
16     lines.clear();
17     if (root != nullptr)
18         calculate_positions(root, tree_depth - 1);
19 }
20 void BST_graphics::set_position(float x, float y)
21 {
22     root->setPosition(sf::Vector2f(x, y));
23
24     sf::FloatRect textBounds = root->get_global_bounds();
25     sf::Vector2f textPosition(
26         root->getPosition().x
27         + (root->getRadius() - textBounds.width / 2.0f)
28         - root->get_character_size() / 12,
29         root->getPosition().y
30         + (root->getRadius() - textBounds.height / 2.0f)
31         - root->get_character_size() / 3);
32     root->set_text_position(textPosition);
33
34     lines.clear();
35     calculate_positions(root, tree_depth - 1);
36 }
37 void BST_graphics::draw_tree(sf::RenderWindow& window)
38 {
39     for (auto& line : lines)
40         window.draw(line.get_VertexArray());
41 }

```

```

42     recursive_draw_tree(root, window);
43 }
44 void BST_graphics::change_color(sf::Color color)
45 {
46     for (auto& line : lines)
47         line.set_color(color);
48     recursive_change_color(root, color);
49 }
50 void BST_graphics::change_font(std::string font_name)
51 {
52     recursive_change_font(root, font_name);
53 }
54
55 void BST_graphics::calculate_positions(Node_graphics* node, int current_depth)
56 {
57     if (current_depth == 0)
58         return;
59     sf::Vector2f node_position = node->get_position();
60     float radius = node->getRadius();
61     if (node->get_right()) {
62         node->get_right()->setPosition(node_position);
63         node->get_right()->move(std::pow(2, current_depth) * radius, 4 * radius);
64
65         sf::FloatRect textBounds = node->get_right()->get_global_bounds();
66         sf::Vector2f textPosition(
67             node->get_right()->get_position().x
68                 + (radius - textBounds.width / 2.0f)
69                 - root->get_character_size() / 12,
70             node->get_right()->get_position().y
71                 + (radius - textBounds.height / 2.0f)
72                 - root->get_character_size() / 3);
73         node->get_right()->set_text_position(textPosition);
74
75         lines.push_back(
76             line(node_position.x + radius,
77                 node_position.y + radius,
78                 node->get_right()->getPosition().x + radius,
79                 node->get_right()->getPosition().y + radius));
80         calculate_positions(node->get_right(), current_depth - 1);
81     }
82     if (node->get_left()) {
83         node->get_left()->setPosition(node_position);
84         node->get_left()->move(-(std::pow(2, current_depth) * radius), 4 * radius);
85
86         sf::FloatRect textBounds = node->get_left()->get_global_bounds();
87         sf::Vector2f textPosition(
88             node->get_left()->getPosition().x + (radius - textBounds.width / 2.0f)
89                 - root->get_character_size() / 12,
90             node->get_left()->getPosition().y
91                 + (radius - textBounds.height / 2.0f)
92                 - root->get_character_size() / 3);
93
94         node->get_left()->set_text_position(textPosition);
95
96         lines.push_back(
97             line(node_position.x + radius,
98                 node_position.y + radius,
99                 node->get_left()->getPosition().x + radius,
100                 node->get_left()->getPosition().y + radius));
101
102         calculate_positions(node->get_left(), current_depth - 1);
103     }
104 }

```

```

105 void BST_graphics::recursive_draw_tree(
106     Node_graphics* node, sf::RenderWindow& window)
107 {
108     window.draw(node->get_CircleShape());
109     window.draw(node->get_text());
110     if (node->get_right())
111         recursive_draw_tree(node->get_right(), window);
112     if (node->get_left())
113         recursive_draw_tree(node->get_left(), window);
114 }
115 void BST_graphics::recursive_change_color(Node_graphics* node, sf::Color color)
116 {
117     node->setFillColor(color);
118     if (node->get_right())
119         recursive_change_color(node->get_right(), color);
120     if (node->get_left())
121         recursive_change_color(node->get_left(), color);
122 }
123 void BST_graphics::recursive_change_font(Node_graphics* node, std::string font_name)
124 {
125     node->set_font(font_name);
126     if (node->get_right())
127         recursive_change_font(node->get_right(), font_name);
128     if (node->get_left())
129         recursive_change_font(node->get_left(), font_name);
130 }
131 int BST_graphics::get_nodes_count()
132 {
133     return nodes_count;
134 }
135
136 float BST_graphics::get_radius()
137 {
138     return root->getRadius();
139 }

```

BST_graphics.h

```

1  #pragma once
2
3  #include <cmath>
4  #include <cstring>
5  #include <iostream>
6
7  #include <SFML/Graphics.hpp>
8
9  #include <BST.h>
10 #include <line.h>
11
12 class BST_graphics : public BST{
13 protected:
14     std::vector<line> lines;
15     int nodes_count;
16     int tree_depth;
17
18 public:
19     BST_graphics() : BST(), nodes_count(0), tree_depth(0){};
20     BST_graphics(int value) : BST(value), nodes_count(1), tree_depth(1){};
21
22     void insert(int value);
23     void delete_node(int value);
24     void set_position(float x, float y);
25     void draw_tree(sf::RenderWindow& window);

```

```
26     void change_color(sf::Color color);
27     void change_font(std::string font_name);
28     int get_nodes_count();
29     float get_radius();
30
31 private:
32     void calculate_positions(Node_graphics* node, int current_depth);
33     void recursive_draw_tree(Node_graphics* node, sf::RenderWindow& window);
34     void recursive_change_color(Node_graphics* node, sf::Color color);
35     void recursive_change_font(Node_graphics* node, std::string font_name);
36 };
```