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](#_Toc152886518)

[ТЕХНОЛОГИИ ООП 4](#_Toc152886519)

[СТРУКТУРА КЛАССОВ 5](#_Toc152886520)

[ПРОГРАММНАЯ РЕАЛИЗАЦИЯ 7](#_Toc152886521)

[РЕЗУЛЬТАТ РАБОТЫ 9](#_Toc152886522)

[ЗАКЛЮЧЕНИЕ 15](#_Toc152886523)

[ИСПОЛЬЗУЕМЫЕ ИСТОЧНИКИ 16](#_Toc152886524)

[ПРИЛОЖЕНИЕ 17](#_Toc152886525)

# 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

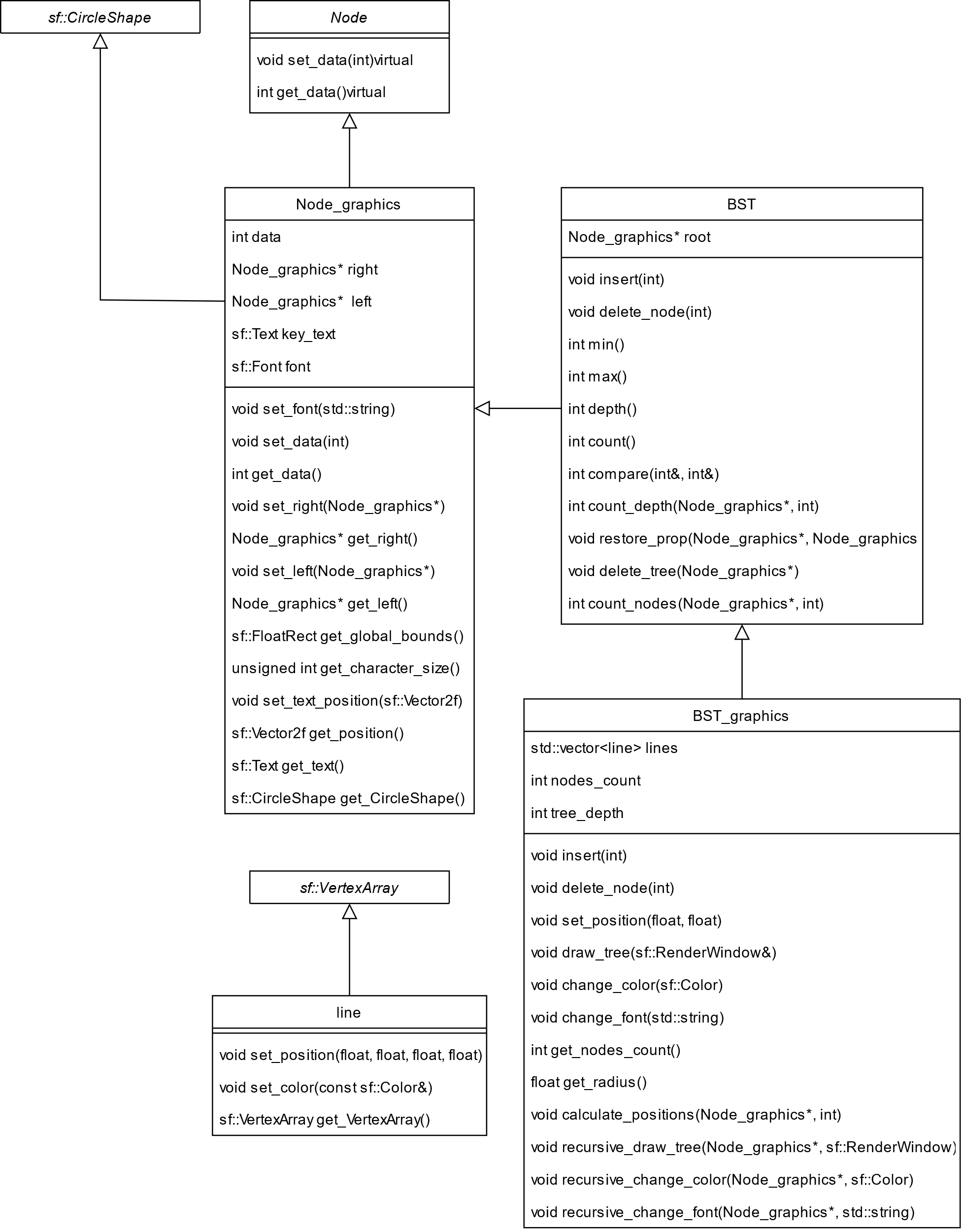


Рисунок 1. Структура классов.

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.

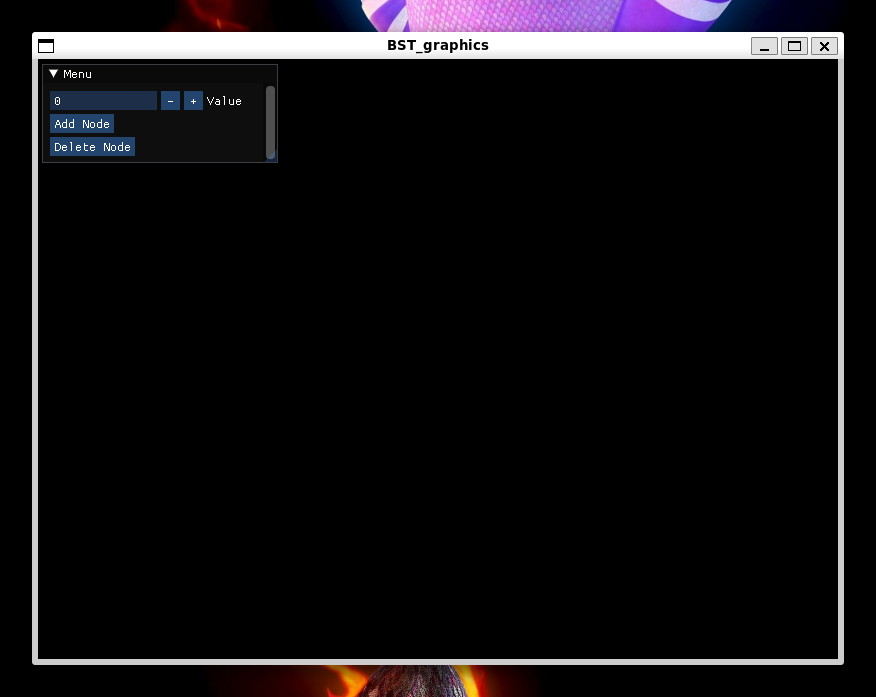


Рисунок 2. Начальное положение приложения.

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.

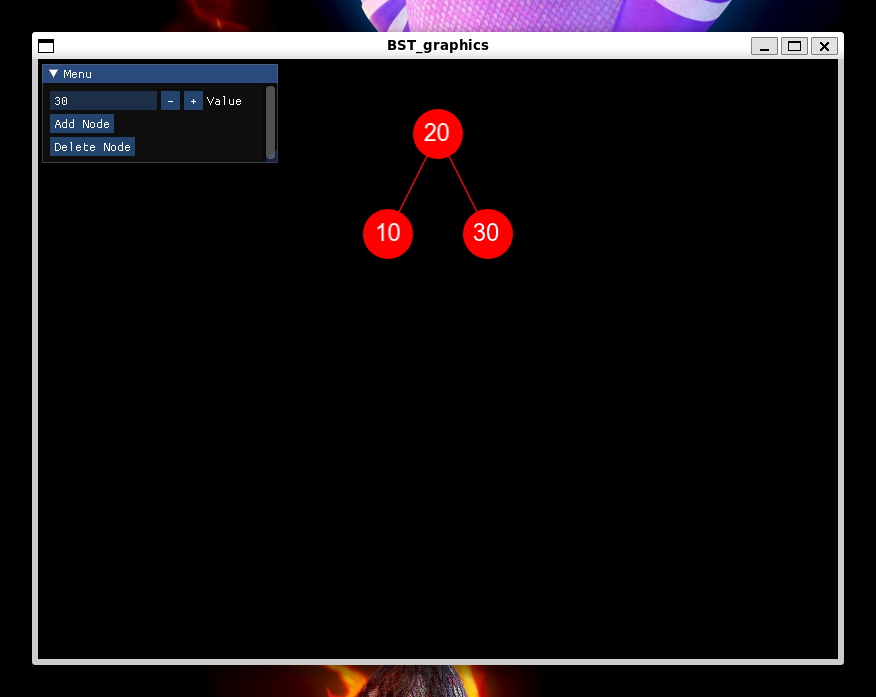


Рисунок 3. Дерево с тремя узлами.

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.

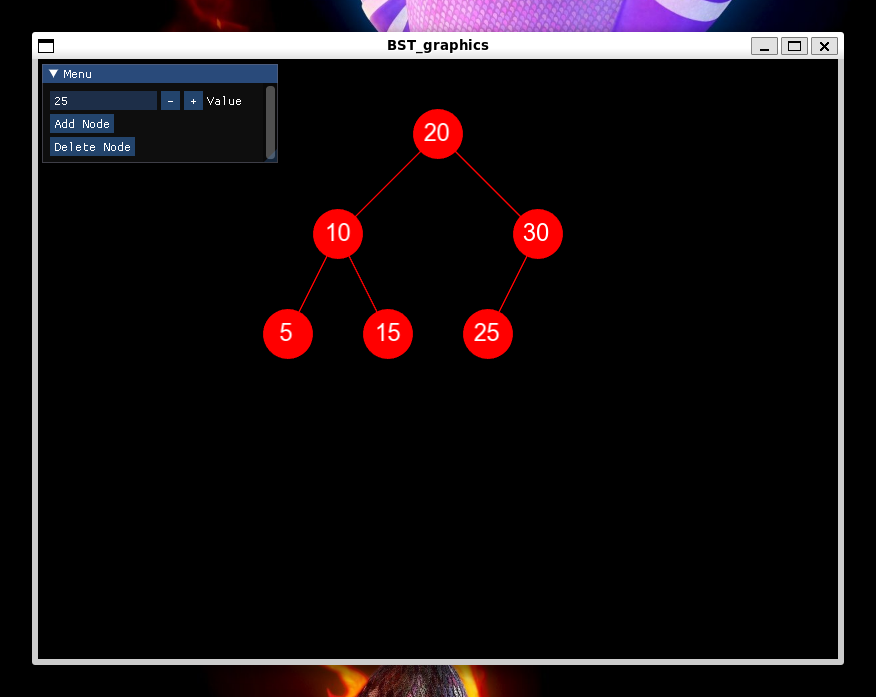


Рисунок 4. Дерево с семью узлами.

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.

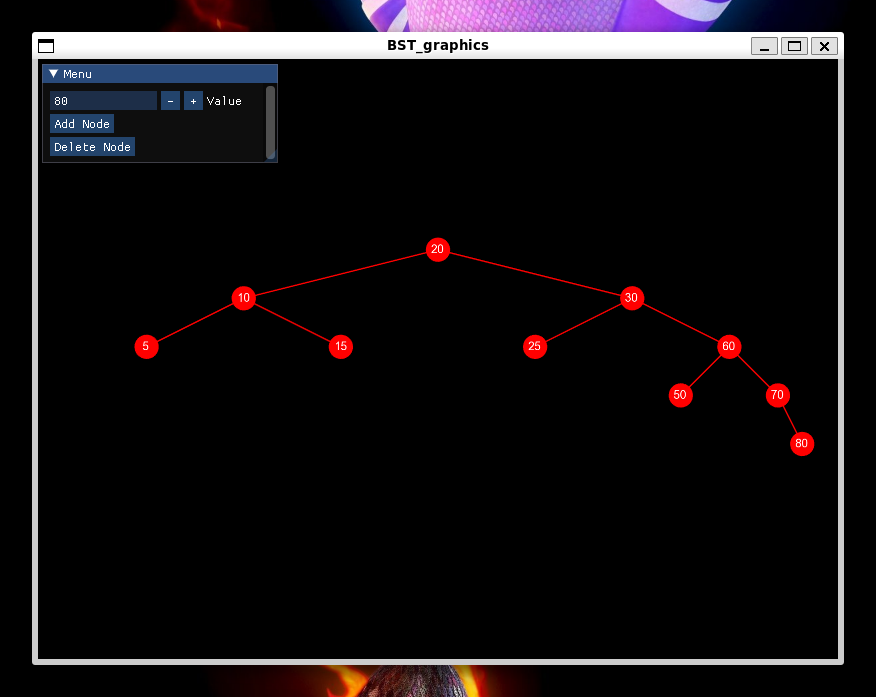


Рисунок 5. Масштабирование.

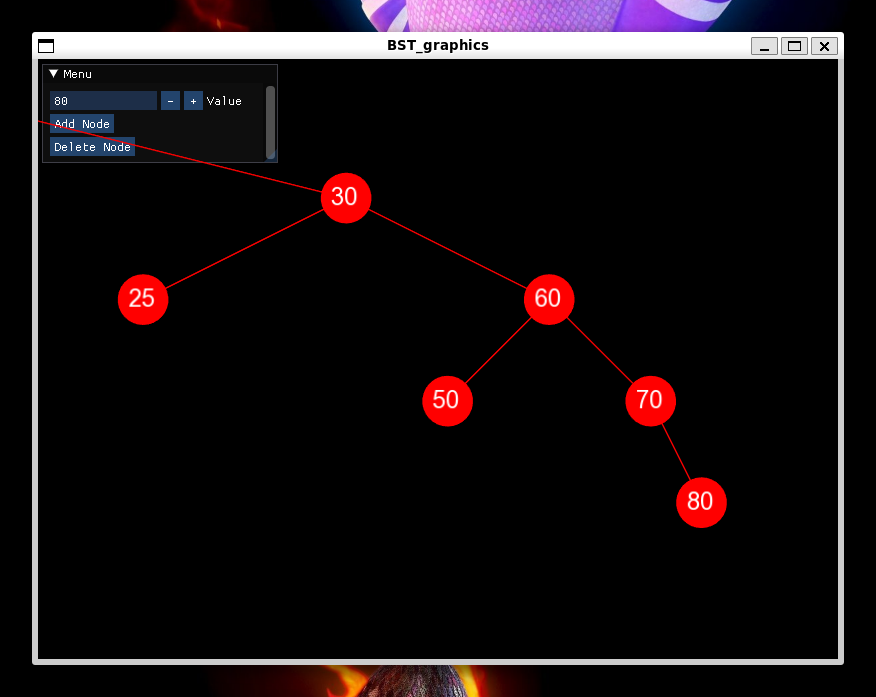


Рисунок 6. Перемещение области видимости.

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.

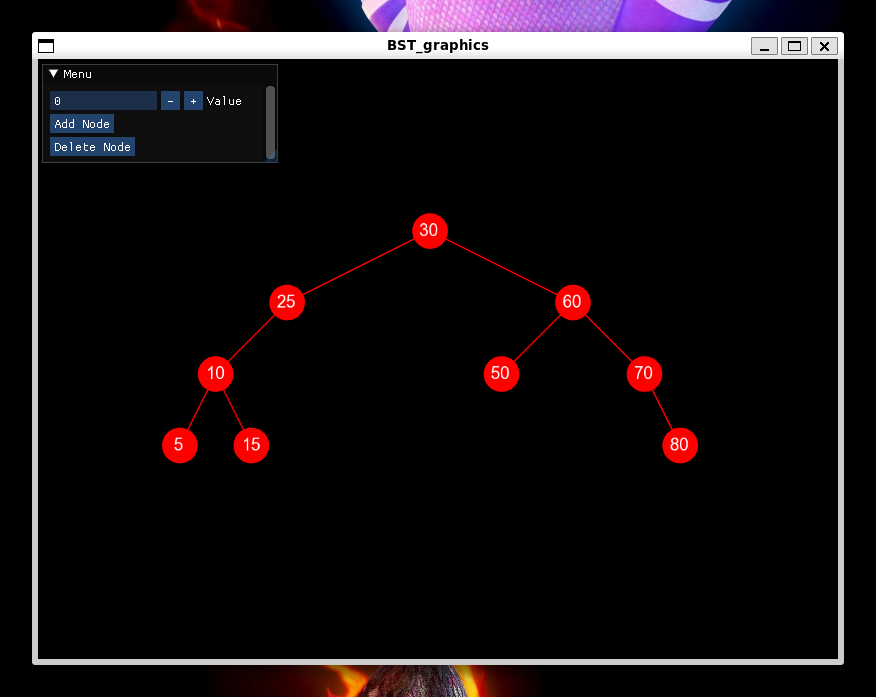


Рисунок 7. Удаление корня из дерева.

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 c.
2. Laurent Gomila Tutorials for SFML 2.6 / Laurent Gomila [Электронный ресурс] // sfml-dev : [сайт]. — URL: https://www.sfml-dev.org/tutorials/2.6/ (дата обращения: 07.12.2023).

# ПРИЛОЖЕНИЕ

Исходный код программы

main.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153 | #include <BST\_graphics.h>  #include <SFML/Graphics.hpp>  #include <imgui-SFML.h>  #include <imgui.h>  #include <iostream>  enum { x\_size = 800, y\_size = 600 };  int main()  {  BST\_graphics tree;  sf::ContextSettings settings;  settings.antialiasingLevel = 8;  sf::RenderWindow window(  sf::VideoMode(x\_size, y\_size),  "BST\_graphics",  sf::Style::Default,  settings);  window.setPosition(sf::Vector2i(0, 0));  if (!ImGui::SFML::Init(window)) {  std::cout << "Can't open ImGUI!\n";  return 1;  }  sf::View view;  view.reset(sf::FloatRect(0, 0, x\_size, y\_size));  window.setView(view);  sf::Vector2f prev\_mouse\_pos;  bool is\_dragging = false;  int input\_value = 0;  bool add\_node\_button\_pressed = false;  bool delete\_node\_button\_pressed = false;  while (window.isOpen()) {  sf::Event event;  while (window.pollEvent(event)) {  ImGui::SFML::ProcessEvent(window, event);  if (event.type == sf::Event::Closed)  window.close();  if (event.type == sf::Event::KeyPressed) {  if (event.key.code == sf::Keyboard::Enter) {  if (tree.get\_nodes\_count() == 0) {  tree.insert(input\_value);  tree.set\_position(  x\_size / 2 - tree.get\_radius(),  tree.get\_radius() \* 2);  } else {  tree.insert(input\_value);  }  input\_value = 0;  }  if (event.key.code == sf::Keyboard::Delete) {  if (tree.get\_nodes\_count() != 0) {  tree.delete\_node(input\_value);  if (tree.get\_nodes\_count() != 0)  tree.set\_position(  x\_size / 2 - tree.get\_radius(),  tree.get\_radius() \* 2);  }  input\_value = 0;  }  if (event.key.code == sf::Keyboard::Escape)  window.close();  }  if (event.type == sf::Event::MouseButtonPressed)  if (event.mouseButton.button == sf::Mouse::Left) {  is\_dragging = true;  prev\_mouse\_pos = window.mapPixelToCoords(  sf::Mouse::getPosition(window));  }  if (event.type == sf::Event::MouseButtonReleased)  if (event.mouseButton.button == sf::Mouse::Left) {  is\_dragging = false;  }  if (event.type == sf::Event::MouseWheelScrolled) {  if (event.mouseWheelScroll.delta > 0) {  view.zoom(0.9f);  } else if (event.mouseWheelScroll.delta < 0) {  view.zoom(1.1f);  }  window.setView(view);  }  }  if (is\_dragging) {  sf::Vector2f mousePos  = window.mapPixelToCoords(sf::Mouse::getPosition(window));  sf::Vector2f delta = prev\_mouse\_pos - mousePos;  view.move(delta);  window.setView(view);  }  ImGui::SFML::Update(window, sf::seconds(1.f / 60.f));  ImGui::Begin("Menu");  ImGui::InputInt("Value", &input\_value);  if (ImGui::Button("Add Node"))  add\_node\_button\_pressed = true;  if (ImGui::Button("Delete Node"))  delete\_node\_button\_pressed = true;  ImGui::End();  if (add\_node\_button\_pressed) {  if (tree.get\_nodes\_count() == 0) {  tree.insert(input\_value);  tree.set\_position(  x\_size / 2 - tree.get\_radius(), tree.get\_radius() \* 2);  } else {  tree.insert(input\_value);  }  add\_node\_button\_pressed = false;  input\_value = 0;  }  if (delete\_node\_button\_pressed) {  if (tree.get\_nodes\_count() != 0) {  tree.delete\_node(input\_value);  if (tree.get\_nodes\_count() != 0)  tree.set\_position(  x\_size / 2 - tree.get\_radius(),  tree.get\_radius() \* 2);  }  delete\_node\_button\_pressed = false;  input\_value = 0;  }  window.clear();  if (tree.get\_nodes\_count() != 0)  tree.draw\_tree(window);  ImGui::SFML::Render(window);  window.display();  }  ImGui::SFML::Shutdown();  return 0;  } |

BST.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254 | #include <BST.h>  void Node\_graphics::set\_data(int new\_data)  {  data = new\_data;  }  int Node\_graphics::get\_data()  {  return data;  }  void Node\_graphics::set\_right(Node\_graphics\* new\_right)  {  right = new\_right;  }  void Node\_graphics::set\_left(Node\_graphics\* new\_left)  {  left = new\_left;  }  Node\_graphics\* Node\_graphics::get\_right()  {  return right;  }  Node\_graphics\* Node\_graphics::get\_left()  {  return left;  }  void Node\_graphics::set\_font(std::string font\_name)  {  if (!font.loadFromFile(font\_name)) {  std::cerr << "Failed to load font!" << std::endl;  }  key\_text.setFont(font);  }  sf::FloatRect Node\_graphics::get\_global\_bounds()  {  return key\_text.getGlobalBounds();  }  unsigned int Node\_graphics::get\_character\_size()  {  return key\_text.getCharacterSize();  }  void Node\_graphics::set\_text\_position(sf::Vector2f &new\_position)  {  key\_text.setPosition(new\_position);  }  sf::Vector2f Node\_graphics::get\_position()  {  return this->getPosition();  }  sf::Text Node\_graphics::get\_text()  {  return key\_text;  }  sf::CircleShape Node\_graphics::get\_CircleShape()  {  return \*this;  }  BST::~BST()  {  delete\_tree(root);  };  void BST::insert(int value)  {  Node\_graphics\* new\_Node\_graphics = new Node\_graphics(value);  if (root == nullptr) {  root = new\_Node\_graphics;  return;  }  Node\_graphics\* current = root;  Node\_graphics\* prev = nullptr;  int compare\_result;  while (current) {  compare\_result = compare(value, current->get\_data());  if (compare\_result == 1) {  prev = current;  current = current->get\_right();  }  if (compare\_result == -1) {  prev = current;  current = current->get\_left();  }  if (compare\_result == 0) {  new\_Node\_graphics->~Node\_graphics();  return;  }  }  switch (compare\_result) {  case 1:  prev->set\_right(new\_Node\_graphics);  break;  case -1:  prev->set\_left(new\_Node\_graphics);  break;  }  }  void BST::delete\_node(int value)  {  if (root == nullptr) {  std::cout << "delete\_Node\_graphics() called on an empty tree\n";  return;  }  Node\_graphics\* current = root;  Node\_graphics\* prev = nullptr;  int compare\_result;  while (current) {  compare\_result = compare(value, current->get\_data());  if (compare\_result == 1) {  prev = current;  current = current->get\_right();  }  if (compare\_result == -1) {  prev = current;  current = current->get\_left();  }  if (compare\_result == 0) {  if (prev == nullptr) {  restore\_prop(prev, current, compare\_result);  return;  }  compare\_result = compare(value, prev->get\_data());  restore\_prop(prev, current, compare\_result);  current->~Node\_graphics();  return;  }  }  }  int BST::min()  {  if (root == nullptr) {  std::cout << "min() called on an empty tree\n";  return 0;  }  Node\_graphics\* current = root;  while (current->get\_left())  current->set\_left(current->get\_left());  return current->get\_data();  }  int BST::max()  {  if (root == nullptr) {  std::cout << "max() called on an empty tree\n";  return 0;  }  Node\_graphics\* current = root;  while (current->get\_right())  current->set\_right(current->get\_right());  return current->get\_data();  }  int BST::depth()  {  return count\_depth(root, 1);  }  int BST::count()  {  return count\_nodes(root, 1);  }  int BST::compare(const int& a, const int& b)  {  if (a < b)  return -1;  if (a > b)  return 1;  return 0;  }  int BST::count\_depth(Node\_graphics\* root, int depth)  {  if (root == nullptr)  return 0;  int left = depth, right = depth;  if (root->get\_left())  left = count\_depth(root->get\_left(), depth + 1);  if (root->get\_right())  right = count\_depth(root->get\_right(), depth + 1);  return right > left ? right : left;  }  void BST::restore\_prop(  Node\_graphics\* prev, Node\_graphics\* current, int compare\_result)  {  Node\_graphics\* right = current->get\_right();  Node\_graphics\* left = current->get\_left();  if (right) {  current = current->get\_right();  if (left) {  while (current->get\_left())  current = current->get\_left();  current->set\_left(left);  }  } else {  right = left;  }  switch (compare\_result) {  case -1:  prev->set\_left(right);  break;  case 1:  prev->set\_right(right);  break;  case 0:  root = right;  }  }  void BST::delete\_tree(Node\_graphics\* root)  {  if (root == nullptr)  return;  if (root->get\_right())  delete\_tree(root->get\_right());  if (root->get\_left())  delete\_tree(root->get\_left());  delete root;  }  int BST::count\_nodes(Node\_graphics\* root, int count)  {  if (root == nullptr)  return 0;  if (root->get\_right())  count = count\_nodes(root->get\_right(), count + 1);  if (root->get\_left())  count = count\_nodes(root->get\_left(), count + 1);  return count;  } |

BST.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | #pragma once  #include <cstring>  #include <iostream>  #include <SFML/Graphics.hpp>  class Node {  public:  virtual void set\_data(int data) = 0;  virtual int get\_data() = 0;  };  class Node\_graphics : public Node, public sf::CircleShape {  protected:  int data;  Node\_graphics\* right;  Node\_graphics\* left;  sf::Text key\_text;  sf::Font font;  public:  Node\_graphics(){};  Node\_graphics(int value) : data(value), right(nullptr), left(nullptr)  {  if (!font.loadFromFile("arial.ttf")) {  std::cerr << "Failed to load font!" << std::endl;  }  key\_text.setFont(font);  key\_text.setFillColor(sf::Color::White);  key\_text.setCharacterSize(24);  key\_text.setString(std::to\_string(value));  key\_text.setPosition(0, 0);  this->setRadius(25.0f);  this->setFillColor(sf::Color::Red);  };  ~Node\_graphics(){};  void set\_font(std::string font\_name);  void set\_data(int new\_data) override;  int get\_data() override;  void set\_right(Node\_graphics\* new\_right);  Node\_graphics\* get\_right();  void set\_left(Node\_graphics\* new\_left);  Node\_graphics\* get\_left();  sf::FloatRect get\_global\_bounds();  unsigned int get\_character\_size();  void set\_text\_position(sf::Vector2f &new\_position);  sf::Vector2f get\_position();  sf::Text get\_text();  sf::CircleShape get\_CircleShape();  };  class BST : public Node\_graphics {  protected:  Node\_graphics\* root;  public:  BST() : root(nullptr){};  BST(int value) : root(new Node\_graphics(value)){};  ~BST();  virtual void insert(int value);  virtual void delete\_node(int value);  int min();  int max();  int depth();  int count();  private:  int compare(const int& a, const int& b);  int count\_depth(Node\_graphics\* root, int depth);  void restore\_prop(Node\_graphics\* prev, Node\_graphics\* current, int compare\_result);  void delete\_tree(Node\_graphics\* root);  int count\_nodes(Node\_graphics\* root, int count);  }; |

line.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <SFML/Graphics.hpp>  #include <iostream>  #include <line.h>  line::line()  {  this->setPrimitiveType(sf::Lines);  this->resize(2);  set\_color(sf::Color::Red);  }  line::line(float x1, float y1, float x2, float y2)  {  this->setPrimitiveType(sf::Lines);  this->resize(2);  set\_color(sf::Color::Red);  set\_position(x1, y1, x2, y2);  }  void line::set\_position(float x1, float y1, float x2, float y2)  {  (\*this)[0].position = sf::Vector2f(x1, y1);  (\*this)[1].position = sf::Vector2f(x2, y2);  }  void line::set\_color(const sf::Color& color)  {  (\*this)[0].color = color;  (\*this)[1].color = color;  }  sf::VertexArray line::get\_VertexArray()  {  return \*this;  } |

line.h

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

BST\_graphics.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139 | #include <BST\_graphics.h>  void BST\_graphics::insert(int value)  {  BST::insert(value);  tree\_depth = this->depth();  nodes\_count = this->count();  lines.clear();  calculate\_positions(root, tree\_depth - 1);  }  void BST\_graphics::delete\_node(int value)  {  BST::delete\_node(value);  tree\_depth = BST::depth();  nodes\_count = BST::count();  lines.clear();  if (root != nullptr)  calculate\_positions(root, tree\_depth - 1);  }  void BST\_graphics::set\_position(float x, float y)  {  root->setPosition(sf::Vector2f(x, y));  sf::FloatRect textBounds = root->get\_global\_bounds();  sf::Vector2f textPosition(  root->getPosition().x  + (root->getRadius() - textBounds.width / 2.0f)  - root->get\_character\_size() / 12,  root->getPosition().y  + (root->getRadius() - textBounds.height / 2.0f)  - root->get\_character\_size() / 3);  root->set\_text\_position(textPosition);  lines.clear();  calculate\_positions(root, tree\_depth - 1);  }  void BST\_graphics::draw\_tree(sf::RenderWindow& window)  {  for (auto& line : lines)  window.draw(line.get\_VertexArray());  recursive\_draw\_tree(root, window);  }  void BST\_graphics::change\_color(sf::Color color)  {  for (auto& line : lines)  line.set\_color(color);  recursive\_change\_color(root, color);  }  void BST\_graphics::change\_font(std::string font\_name)  {  recursive\_change\_font(root, font\_name);  }  void BST\_graphics::calculate\_positions(Node\_graphics\* node, int current\_depth)  {  if (current\_depth == 0)  return;  sf::Vector2f node\_position = node->get\_position();  float radius = node->getRadius();  if (node->get\_right()) {  node->get\_right()->setPosition(node\_position);  node->get\_right()->move(std::pow(2, current\_depth) \* radius, 4 \* radius);  sf::FloatRect textBounds = node->get\_right()->get\_global\_bounds();  sf::Vector2f textPosition(  node->get\_right()->get\_position().x  + (radius - textBounds.width / 2.0f)  - root->get\_character\_size() / 12,  node->get\_right()->get\_position().y  + (radius - textBounds.height / 2.0f)  - root->get\_character\_size() / 3);  node->get\_right()->set\_text\_position(textPosition);  lines.push\_back(  line(node\_position.x + radius,  node\_position.y + radius,  node->get\_right()->getPosition().x + radius,  node->get\_right()->getPosition().y + radius));  calculate\_positions(node->get\_right(), current\_depth - 1);  }  if (node->get\_left()) {  node->get\_left()->setPosition(node\_position);  node->get\_left()->move(-(std::pow(2, current\_depth) \* radius), 4 \* radius);  sf::FloatRect textBounds = node->get\_left()->get\_global\_bounds();  sf::Vector2f textPosition(  node->get\_left()->getPosition().x + (radius - textBounds.width / 2.0f)  - root->get\_character\_size() / 12,  node->get\_left()->getPosition().y  + (radius - textBounds.height / 2.0f)  - root->get\_character\_size() / 3);  node->get\_left()->set\_text\_position(textPosition);  lines.push\_back(  line(node\_position.x + radius,  node\_position.y + radius,  node->get\_left()->getPosition().x + radius,  node->get\_left()->getPosition().y + radius));  calculate\_positions(node->get\_left(), current\_depth - 1);  }  }  void BST\_graphics::recursive\_draw\_tree(  Node\_graphics\* node, sf::RenderWindow& window)  {  window.draw(node->get\_CircleShape());  window.draw(node->get\_text());  if (node->get\_right())  recursive\_draw\_tree(node->get\_right(), window);  if (node->get\_left())  recursive\_draw\_tree(node->get\_left(), window);  }  void BST\_graphics::recursive\_change\_color(Node\_graphics\* node, sf::Color color)  {  node->setFillColor(color);  if (node->get\_right())  recursive\_change\_color(node->get\_right(), color);  if (node->get\_left())  recursive\_change\_color(node->get\_left(), color);  }  void BST\_graphics::recursive\_change\_font(Node\_graphics\* node, std::string font\_name)  {  node->set\_font(font\_name);  if (node->get\_right())  recursive\_change\_font(node->get\_right(), font\_name);  if (node->get\_left())  recursive\_change\_font(node->get\_left(), font\_name);  }  int BST\_graphics::get\_nodes\_count()  {  return nodes\_count;  }  float BST\_graphics::get\_radius()  {  return root->getRadius();  } |

BST\_graphics.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #pragma once  #include <cmath>  #include <cstring>  #include <iostream>  #include <SFML/Graphics.hpp>  #include <BST.h>  #include <line.h>  class BST\_graphics : public BST{  protected:  std::vector<line> lines;  int nodes\_count;  int tree\_depth;  public:  BST\_graphics() : BST(), nodes\_count(0), tree\_depth(0){};  BST\_graphics(int value) : BST(value), nodes\_count(1), tree\_depth(1){};  void insert(int value);  void delete\_node(int value);  void set\_position(float x, float y);  void draw\_tree(sf::RenderWindow& window);  void change\_color(sf::Color color);  void change\_font(std::string font\_name);  int get\_nodes\_count();  float get\_radius();  private:  void calculate\_positions(Node\_graphics\* node, int current\_depth);  void recursive\_draw\_tree(Node\_graphics\* node, sf::RenderWindow& window);  void recursive\_change\_color(Node\_graphics\* node, sf::Color color);  void recursive\_change\_font(Node\_graphics\* node, std::string font\_name);  }; |