AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Algorytm binarnego drzewa poszukań z zastosowaniem GitHub

Autor:
Mateusz Stanek
Dawid Szołdra

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest stworzenie programu tworzące drzewo BTS działające na stercie oraz kontrolowanie jego wersji za pomocą narzędzia git. Do zadań programu będzie należało: dodawanie elementu, usuwanie elementu, usunięcie całego drzewa, szukanie drogi do wskazanego elementu, wyświetlenie drzewa graficznego, zapis do pliku tekstowego wygenerowanego drzewa.
Program będzie podzielony na 3 pliki: plik main - w którym zostanie zaimplementowane menu sterujące drzewem BTS, plik z drzewem - zawierający klasę drzewa bTS oraz jej implementację oraz plik który będzie odczytywał drzewo oraz zapisywał je do pliku tekstowego pliku tekstwoego.
Wynikiem projektu powinno być działające drzeewo BTS, repozytorium git z kodem oraz dokumentacja w doxygenie.

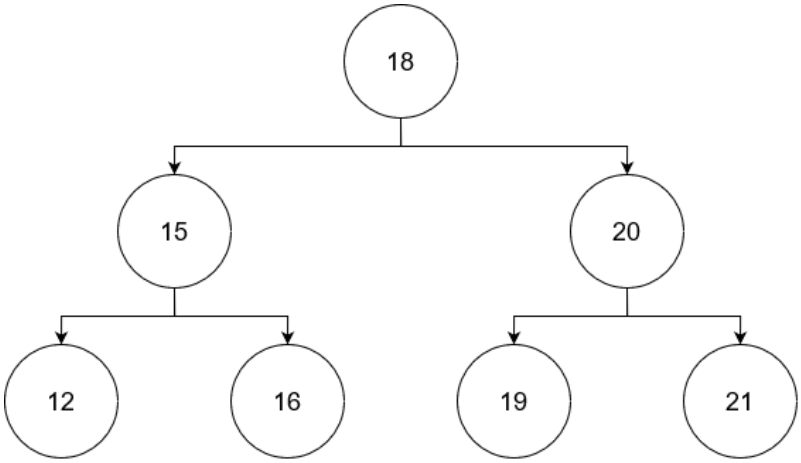
2. Analiza problemu

2.1. Drzewo BST

Drzewo binarne BST[1] jet strukturą przechowującą dane w sposób posortowany. Poszczególne instancje danych są reprezentowane za pomocą węzłów połączonych ze sobą gałęziami. Każdy węzeł posiada dwoje dzieci - lewe oraz prawe. Prawe dziecko jest zawsze większe lub równe od swojego rodzica. Natomiast lewe dziecko jest mniejsze od rodzica i prawego dziecka.

Taka struktura drzewa zapewnia efektywne sortowanie zawartych w nich danych.

Graficzna reprezentacja drzewa BST zaprezentowana na rysunku 2.1



Rys. 2.1. Reprezentacja drzewa BST

Na ryunku 2.1 węzły są reprezentowane przez okręgi a gałęzie przez strzałki.

2.2. Git

Kolejnym konceptem, którym zajmuje się projekt jest narzędzie git[2]. Pozwala ono zarządzać poszczególnymi wersjami projektów. Głównym korzeniem gita jest system commitów, czyli zapisania zmian w pliku w stosunku do commita starszego. To, w połączeniu z jego innymi możliwościami pozwala na tworzenie długich i skomplikowanych osi czasu danych projektów.

Użycie gita można zademonstrować na prostym przykładzie. Tworzymy katalog a w nim repozytorium, uzywając komendy `git init`, jak widać na rys. 2.2.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/mattys/skrypty-i-syfy/studia/rok2/progr
amowanie-zaawansowane/p1/git-test/.git/
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % ls
drwxr-xr-x - mattys  6 Nov 15:54 .git/
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test %
```

Rys. 2.2. Puste repozytorium git

Stwórzmy jakiś plik i dodajmy go do repozytorium. Plik można dodać do repozytorium komendą `git add`

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo "commit1" > plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git add plik.txt
```

Rys. 2.3. Stworznie pliku w repozytorium

Następnie należy scommitować zmiany.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git commit -am "utworzenie pl
ik.txt"
[master (root-commit) bb3b898] utworzenie plik.txt
 1 file changed, 1 insertion(+)
 create mode 100644 plik.txt
```

Rys. 2.4. Commit nr. 1

Na rysunku 2.4 użyta komenda `git commit` commituje wszystkie dodane pliki (`-a`) z jakimś komunikatem (`-m`).

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo "commit2" >> plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git commit -am "uzupelnienie
plik.txt"
[master 40694c2] uzupelnienie plik.txt
 1 file changed, 1 insertion(+)
```

Rys. 2.5. Commit nr. 2

Na rys. 2.5, został utworzony kolejny commit, dodajacy zmiany do `plik.txt`.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git log
commit 40694c2e73758368445cb817f4524ee5c5afbece (HEAD → master)
Author: mattys1 <mattys0082@gmail.com>
Date:   Wed Nov 6 16:26:07 2024 +0100

    uzupelnienie plik.txt

commit bb3b898df760395b5763575e204c3c43b513d2f6
Author: mattys1 <mattys0082@gmail.com>
Date:   Wed Nov 6 16:12:31 2024 +0100

    utworzenie plik.txt
```

Rys. 2.6. Log gita

Jak na rys. 2.6 jest pokazane, używając komendy `git log`, można wyświetlić log commitów w repozytorium.

```
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % git checkout bb3b898df760395b
5763575e204c3c43b513d2f6
Note: switching to 'bb3b898df760395b5763575e204c3c43b513d2f6'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at bb3b898 utworzenie plik.txt
~/skrypty-i-syfy/studia/rok2/programowanie-zaawansowane/p1/git-test % echo plik.txt
plik.txt
```

Rys. 2.7. Demonstracja checkout

Jak widać na rys. 2.7, komenda `git checkout`, pozwala na przejście repozytorium w inny stan, w tym przypadku przechodzi się do commita o danym ID, pokazanym na rys. 2.6. Jako, że jest to pierwszy commit, nie ma w nim zmian z drugiego.

2.3. Doxygen

Doxygen[3] jest narzędziem automatycznie generującym dokumentację programu z komentarzy w kodzie źródłowym. Potrafi on generować strony HTML, gdzie można dynamicznie nawigować się miedzy rożnymi częściami kodu oraz pliki LaTeX, które można konwertować na różne, statyczne formaty.

3. Projektowanie

3.1. Implementacja drzewa BTS

Do zaimplementowania drzewa BTS zostanie użyty Język `C++` z kompilatorem `g++` oraz `MVSC`. Wersja standardu `C++`to `C++23`. Wersja ta została użyta, ze względu na zawartą w niej funkcję `std::print()`. Jako, że projekt ma być rozdzielony na dwa pliki, zostanie zastosowany `CMake` w celu automatyzacji procesu budowania. `CMake` pozwala na generowanie plików budujących dany projekt, zgodnie z określoną konfiguracją. Oszczędza to programiście, szczególnie przy większych projektach, manualne pisanie Makefileów. Plik konfiguracyjny `CMakeLists.txt` może wyglądać jak na rysunku

```
1  cmake_minimum_required(VERSION 3.5)
2
3  set(PROJECT_NAME proj1)
4
5  project(${PROJECT_NAME} VERSION 0.1 LANGUAGES CXX)
6
7  set(CMAKE_CXX_STANDARD 23)
8  set(CMAKE_CXX_STANDARD_REQUIRED ON)
9  set(CMAKE_EXPORT_COMPILE_COMMANDS True)
10 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_CURRENT_LIST_DIR}/out)
11
12 add_subdirectory(src)
13
```

Listing 1. Plik konfiguracyjny CMake

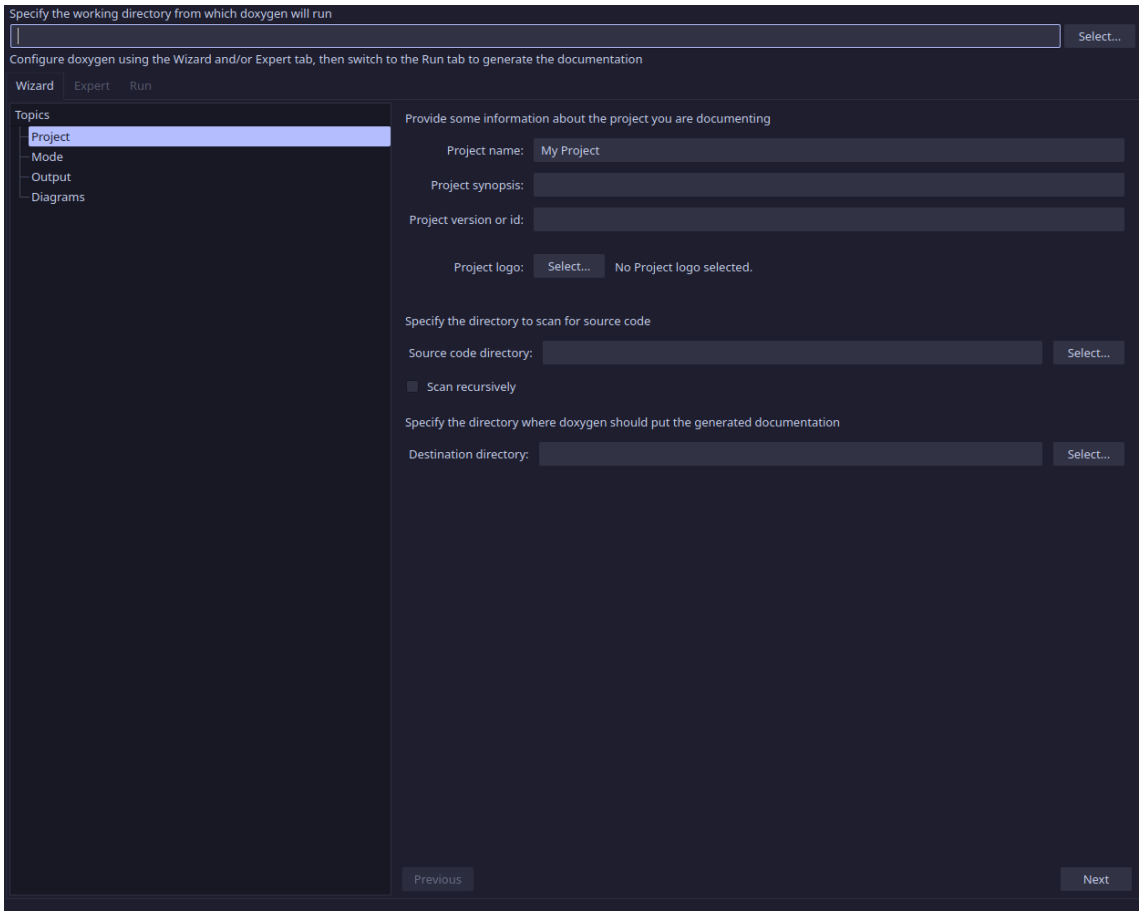
Edytorem będzie program Neovim oraz Visual Studio 2022. Jest to terminalowy edytor tekstu z możliwością poszerzenia funkcjonalności przy użyciu wszelkiego rodzaju pluginów. Wybrany został, dlatego że jest on już skonfigurowany na moim komputerze zgodnie z moimi preferencjami.

3.2. Git

Dla ułatwienia pracy, zastosowany został front-end dla gita o nazwie lazygit. Jest to terminalowy program, którego główną zaletą jest łatwa nawigacja przy użyciu klawiatury. Ponadto, jest on lekki i szybki.

3.3. Doxygen

Konfiguracja dla Doxygena jest wygenerowana przy użyciu programu doxywizard, pokazany na rys. 3.1, pozwalającego na graficzne zmienianie ustawień. Po wygenerowaniu konfiguracji, Doxygen wywoływany jest przy użyciu komendy.



Rys. 3.1. Interfejs programu doxywizard

4. Implementacja

4.1. Ogólne informacje o implementacji klas

4.1.1. Klasa BSTTree

Drzewo jest zaimplementowane jako jeden plik `.hpp`. Nie jest podzielone na plik implementacji oraz nagłówek, ponieważ jest ono szablonem. Deklaracja klasy oraz prywatne elementy wyglądają następująco:

```
1  template <typename T>
2  class BSTTree {
3    private:
4    struct Tree {
5      T contents;
6      Tree* parent;
7      Tree* left;
8      Tree* right;
9
10     Tree(T _contents, Tree* _parent = nullptr, Tree* _left =
   nullptr, Tree* _right = nullptr):
11     contents { _contents },
12     parent { _parent },
13     left { _left },
14     right { _right } {}
15
16     ~Tree() {
17       delete left;
18       delete right;
19     }
20   };
21
22   Tree* root;
23
24   void recursive_add(const T& element, Tree*& node, Tree*
   parentNode = nullptr) {
25     if(node == nullptr) {
26       node = new Tree(element);
27
28       if(parentNode == nullptr) {
29         return;
30       }
31
32       node->parent = parentNode;
33
```

```
34        if(node->contents >= parentNode->contents) {
35          parentNode->right = node;
36        } else {
37          parentNode->left = node;
38        }
39
40        return;
41      }
42
43      if(element >= node->contents) {
44        recursive_add(element, node->right, node);
45      } else {
46        recursive_add(element, node->left, node);
47      }
48    }
49
50    void preorder_traverse_recursive(Tree* node, std::vector<Tree
    *>& traversedTrees) {
51      if(node == nullptr) {
52        return;
53      }
54
55      traversedTrees.push_back(node);
56
57      preorder_traverse_recursive(node->left, traversedTrees);
58      preorder_traverse_recursive(node->right, traversedTrees);
59    }
60
61    void inorder_traverse_recursive(Tree* node, std::vector<Tree*>&
     traversedTrees) {
62      if(node == nullptr) {
63        return;
64      }
65
66      inorder_traverse_recursive(node->left, traversedTrees);
67
68      traversedTrees.push_back(node);
69
70      inorder_traverse_recursive(node->right, traversedTrees);
71    }
72
73    void postorder_traverse_recursive(Tree* node, std::vector<Tree
    *>& traversedTrees) {
74      if(node == nullptr) {
75        return;
```

```
76        }
77
78        postorder_traverse_recursive(node->left, traversedTrees);
79        postorder_traverse_recursive(node->right, traversedTrees);
80
81        traversedTrees.push_back(node);
82      }
83      public:
84        ...
```

Listing 2. Deklaracja drzewa BST

Jak widać w kodzie zamieszczonym na listingu nr 2., Klasa jest wrapperem dla structa `Tree`, zadeklarowanego na linijce nr. 4. Struct ten ma trzy wskaźniki - `left` dla elementu lewego, `right` dla elementu prawego, `parent` dla elementu będącego rodzicem oraz zawartość `contents`, opsującą element zawarty w danym węźle.

W linii 10 zadeklarowany jest konstruktor, który domyślnie ustawia wartości wskaźników rodzica, dziecka lewego oraz prawego na `nullptr`. Istnieje możliwość sprecyzowania wartości wskaźników.

Od linii 16 do 18 znajduje się destruktor drzewa. W destruktorze węzeł lewy i prawy jest usuwany.

Od linijki nr 24, do końca fragmentu, zawarty jest szereg pomocniczych metod prywatnych. Ich działanie zostanie omówione przy dyskusji korzystających z nich metod publicznych.

Jedną z tych metod publicznych jest `add()`, którą widać na listingu nr 3

```
1  void add(const T& element) {
2    recursive_add(element, root);
3  }
```

Listing 3. Metoda `add()`

Metoda ta przyjmuje parametr `element`, który będzie wartością nowo dodanego węzła. W swoim ciele, wywołuje ona prywatną metodę `recursive_add()` z parametrami `element` i korzeniem drzewa `root`. Jej definicja jest zawarta w linijce 24, listingu nr 2.

Metoda `recursive_add()` działa na zasadzie rekurencji. Na początku funkcji sprawdzane jest czy teraźniejszy `node` to `nullptr`. Jeżeli tak, oznacza to, że doszło się do końca drzewa i można ten wskaźnik ustawić jako nowy węzeł. Na linijce nr. 28, sprawdzane jest czy rodzic wskaźnika `node` to `nullptr`. Jeżeli tak, oznaczałoby to, że pracujemy ze wskaźnikiem `root`. Wracamy wtedy wcześnie z funkcji, dlatego że `root` nie ma rodzica, więc nie chcemy manipulować wskaźnikami tego rodzica.

Następnie, po zakończeniu bloku `if`, ustawiane są wskaźniki rodzica. Na linijce nr. 43, metoda wywoływana jest ponownie, w zależności od tego czy zawartość `node` jest większa od liczby która ma być dodana, czy nie.

Kolejną metodą na jaką można zwrócić uwagę to `traverse_preorder()`, zawarta na listingu nr 4.

```
1  std::vector<T> traverse_preorder(void) {
2    std::vector<Tree*> traversedTrees;
3
4    preorder_traverse_recursive(root, traversedTrees);
5
6    return traversedTrees
7    | std::ranges::views::transform(
8        [](const Tree* tree) { return tree->contents; }
9    )
10   | std::ranges::to<std::vector>();
11 }
```

Listing 4. Metoda `add()`

Metoda `traverse_preorder()`, ma za zadanie zwrócić wektor wartości węzłów ułożonych w kolejności preorder drzewa. Ku temu celu, na początku metody, w linijce nr 2 listingu nr 4, deklarowany jest wektor `traversedTrees`, który będzie wypełniony wskaźnikami `Tree*` w kolejności preorder. Do wypełnienia tego wektora, używana jest prywatna metoda `preorder_traverse_recursive()` z parametrami `root` i `traversedTrees`.

Metoda `preorder_traverse_recursive()`, podobnie jak inne prywatne metody jest zawarta na listingu nr 2, konkretnie od linijki nr. 50 do linijki nr. 59. Na początku metody, sprawdzane jest czy parametr `node` jest równy `nullptr` - oznaczałoby to, że dotarło się do końca danej ścieżki w drzewie i należy się wrócić. Następnie bieżący `node` jest pushowany do wektora `traversedTrees`, po czym w linijkach nr. 57 i 58, wywoływana jest dwukrotnie metoda `preorder_traverse_recursive()`. Za pierwszym razem na lewy, węzeł, a za drugim na prawym. Takie wywołanie sprawi, że najpierw odwiedzone zostaną lewe gałęzie, a dopiero później prawe, zgodnie z porządkiem preorder.

Wracając z metody `preorder_traverse_recursive()` do metody `traverse_preorder()` i listingu nr 4, w linijce nr. 6, widać że do `traversedTrees` aplikowana jest transformacja przy użyciu biblioteki `std::ranges`, powodująca, że zwrócony zostanie wektor zawartości poszczególnych węzłów.

4.1.2. Klasa FileTree

Głównym zadaniem klasy FileTree jest odczyt i zapis zawartość drzewa z, jak i do pliku. Odbywa się to na dwa sposoby: 1 - do zwykłego pliku tekstowego(.txt) oraz 2 - do pliku binarnego(.bin). Klasa zawiera 4 metody publiczne:

- save_to_text() - Ta metoda odpowiedzialna jest za zapisywanie zawartości drzewa binarnego do pliku tekstowego.
- load_from_file() - Ta metoda odpowiedzialna jest za wczytywanie zawartości pliku tekstowego do drzewa binarnego.
- analogicznie działają metody do drzewa binarnego.

Na listingu nr 5 przedstawiona jest klasa FileTree.

```
1
2  template <typename T>
3  class FileTree {
4    public:
5    void save_to_text(const std::string& filename, const std::
   vector<T>& elements) {
6      std::ofstream file(filename);
7      if (!file) {
8        std::cerr << "Error: Could not open file for writing.\n";
9        return;
10     }
11     for (const auto& elem : elements) {
12       file << elem << " ";
13     }
14   }
15   void load_from_text(const std::string& filename, BSTTree<T>&
   tree, std::vector<T>& elements, bool append = false) {
16     std::ifstream file(filename);
17     if (!file) {
18       std::cerr << "Error: Could not open file for reading.\n";
19       return;
20     }
21     if (!append) {
22       tree.delete_tree();
23       elements.clear();
24     }
25     T value;
26     while (file >> value) {
27       elements.push_back(value);
```

```
28          tree.add(value);
29        }
30      }
31      void save_to_binary(const std::string& filename, const std::
    vector<T>& elements) {
32        std::ofstream file(filename, std::ios::binary);
33        if (!file) {
34          std::cerr << "Error: Could not open file for binary writing
    .\n";
35          return;
36        }
37        size_t size = elements.size();
38        file.write(reinterpret_cast<const char*>(&size), sizeof(size)
    );
39        for (const auto& elem : elements) {
40          file.write(reinterpret_cast<const char*>(&elem), sizeof(T))
    ;
41        }
42      }
43      void load_from_binary(const std::string& filename, BSTTree<T>&
    tree, std::vector<T>& elements, bool append = false) {
44        std::ifstream file(filename, std::ios::binary);
45        if (!file) {
46          std::cerr << "Error: Could not open file for binary reading
    .\n";
47          return;
48        }
49        if (!append) {
50          tree.delete_tree();
51          elements.clear();
52        }
53        size_t size = 0;
54        file.read(reinterpret_cast<char*>(&size), sizeof(size));
55        for (size_t i = 0; i < size; ++i) {
56          T elem;
57          file.read(reinterpret_cast<char*>(&elem), sizeof(T));
58          elements.push_back(elem);
59          tree.add(elem);
60        }
61      }
62  };
```

Listing 5. Klasa `FileTree`

jak widać na listingu, klasa nie posiada żadnych metod prywatnych. Klasa składa

się z 4 metod publicznych, zadeklarowanych w wiersach: 5, 15, 31, 43. Poniżej dokładniejsze objaśnienie działania metod.

```
1  void save_to_text(const std::string& filename, const std::vector<
    T>& elements) {
2    std::ofstream file(filename);
3    if (!file) {
4      std::cerr << "Error: Could not open file for writing.\n";
5      return;
6    }
7    for (const auto& elem : elements) {
8      file << elem << " ";
9    }
10 }
```

Listing 6. Metoda `save_to_file`

Jak widać na listingu nr 6 metoda ma dwa parametry: filename - które definiuje nazwę pliku do którego zostanie zapisana zawartość tablicy oraz elements - który jest wektorem elementów które zostaną zapisane do pliku. Polecenie w wierszu 2 otwiera plik, jeżeli nie istnieje jest tworzony. Następnie w wierszu 3 mamy instrukcję if. Jeżeli plik nie może być otwarty wysyłany jest komunikat o błędzie. Jeżeli plik się otworzy, to w wierszu 7 zadeklarowana jest pętla która iteruje przez wszystkie elementy drzewa binarnego patrząc na kolejność ich dodania oraz dodaje je do pliku poleceniem w wierszu 8.

```
1  void load_from_text(const std::string& filename, BSTTree<T>& tree
    , std::vector<T>& elements, bool append = false) {
2    std::ifstream file(filename);
3    if (!file) {
4      std::cerr << "Error: Could not open file for reading.\n";
5      return;
6    }
7    if (!append) {
8      tree.delete_tree();
9      elements.clear();
10   }
11   T value;
12   while (file >> value) {
13     elements.push_back(value);
14     tree.add(value);
15   }
16 }
```

Listing 7. Metoda `load_to_file`

Na listingu nr 7 przedstawiona jest metoda wczytywania danych z pliku tekstowego do drzewa binarnego. Metoda zawiera 4 parametry: filename - nazwa pliku, tree - drzewo binarne do którego będą dodane będą wartości, elements - wketor przechowujący wartości do dodania oraz append - który polega na kontroli dodawania wartości do już istniejącego drzewa z wartościami. Append jest ustawiony na false dlatego drzewo będzie czyszczone oraz następne wartości zostaną dodane do drzewa.

W wierszu 2 przedstawiona jest instrukcja wyświetlenia komunikatu błędu w przypadku gdy pliku nie da się otworzyć.

W wierszach 7, 8 i 9 przedstawiona jest seria instrukcji odpowiedzialna za czyszczenie drzewa binarnego z poprzednich wartości.

W wierszach 11, 12, 13 oraz 14 przedstawiona jest pętla while która dodaje wartości do tablicy w takiej samej kolejności w jakiej są zapisane w pliku tekstowym.

```
1  void save_to_binary(const std::string& filename, const std::
    vector<T>& elements) {
2    std::ofstream file(filename, std::ios::binary);
3    if (!file) {
4      std::cerr << "Error: Could not open file for binary writing.\
    n";
5      return;
6    }
7    size_t size = elements.size();
8    file.write(reinterpret_cast<const char*>(&size), sizeof(size));
9    for (const auto& elem : elements) {
10     file.write(reinterpret_cast<const char*>(&elem), sizeof(T));
11   }
12 }
```

Listing 8. Metoda `save_to_binary`

Na listingu 8 przedstawiona jest metoda zapisu danych z drzewa do pliku binarnego. Metoda posiada 2 parametry: filename - nazwa pliku oraz elements - wektor elementów do zapisania.

W wierszu 2 znajduje się instrukcja tworząca plik binarny, aby stworzyć taki plik trzeba zastosować ”std::ios::binary”.

W wierszach 3 i 4 znajduje się instrukcja if która wyświetli komunikat o błędzie w przypadku gdy pliku nie uda się otworzyć.

W wierszach od 7 do 11 znajduje się szereg instrukcji odpowiedzialnych za zapis danych z drzewa do pliku binarnego.

Polecenie w wierszu 7 odpowiedzialne jest za ustalenie liczby elementów w wektorze. Polecenie w wierszu 8 zapisuje bity z pamięci do pliku za pomocą ”file.write()”.

Polecenie "reinterpret_cast¡const char*¿ jest odpowiedzialne za konwersję adresu size na wskaźnik const char*, jest to wymagane ponieważ write zapisuje bity bezpośrednio z pamięci. sizeof() definiuje ile bitów należy zapisać.

Następnie za pomocą pętli for w wierszu 9, 10 oraz 11 każdy element zapisywany jest do pliku binarnego.

```
1  void load_from_binary(const std::string& filename, BSTTree<T>&
    tree, std::vector<T>& elements, bool append = false) {
2    std::ifstream file(filename, std::ios::binary);
3    if (!file) {
4      std::cerr << "Error: Could not open file for binary reading.\
    n";
5      return;
6    }
7    if (!append) {
8      tree.delete_tree();
9      elements.clear();
10   }
11   size_t size = 0;
12   file.read(reinterpret_cast<char*>(&size), sizeof(size));
13   for (size_t i = 0; i < size; ++i) {
14     T elem;
15     file.read(reinterpret_cast<char*>(&elem), sizeof(T));
16     elements.push_back(elem);
17     tree.add(elem);
18   }
19 }
```

Listing 9. Metoda `load_to_binary`

Na listingu 9 przedstawiona jest metoda wczytywania elementów z pliku binarnego do drzewa binarnego. Metoda zawiera 4 parametry: filename - nazwa pliku, tree - wskazujące na drzewo do którego zostaną wczytane wartości, elements - wektor przechowujący elementy do wczytania oraz append - decydujący o dodaniu wartości z drzewa binarnego do istniejącej tablicy lub nie.

W wierszu 2 otwierany jest plik binarny poleceniem "std::ifstream()". Instrukcja if w wierszach od 3 do 6 odpowiedzialna jest za wyświetlenie komunikatu o błędzie w przypadku gdy pliku nie uda się otworzyć. Instrukcja if w wiersach od 7 do 10 odpowiedzialna jest za czyszczenie drzewa z poprzednich wartości przed dodaniem nowych. Dodawanie wartości do drzewa binarnego odbywa się w wierszach od 11 do 18. W wierszu 11 oraz 12 czytana jest część pliku bin przechowująca liczbę przechowywanych elementów. "reinterpret_cast¡char*¿(...) odpowiedzialne jest za konwersję bitów na postać bardziej odpowiednią do odczytu. sizeof(...) odpowiedzialne jest

za liczbę bitów do odczytania. Następnie w wierszach od 13 do 18 sczytywane są wszystkie wartości oraz dodawane do drzewa. Polecenie w wierszu 16 sprawdza jest odpowiedzialne za dodawanie elementów w takiej samej kolejności w jakiej są zapisane na pliku bin. Polecenie w wierszu 17 dodaje elementy.

4.1.3. Klasa main

```
1  int main(int argc, char* argv[]) {
2    BSTTree<int> tree;
3    BSTTree<int> tree2;
4    FileTree<int> fileTree;
5    std::vector<int> elements;
6
7    int option;
8    int option2;
9    int value;
10   std::println("List of options:");
11   std::println("1 - add element | 2 - remove element | 3 - print
     traversal preordern");
12   std::println("4 - print traversal inorder | 5 - print traversal
     postorder | 6 - export to file");
13   std::println("7 - import from file | 8 - delete tree | 9 - fid
     path to element");
14   std::println();
15   do {
16     std::print("What do you want to do? "), std::cin >> option;
17
18     switch (option) {
19       case 1:
20       std::print("Insert element to add: ");
21       std::cin >> value;
22       tree.add(value);
23       elements.push_back(value);
24       break;
25
26       case 2:
27       std::print("Insert element to remove: ");
28       std::cin >> value;
29       tree.delete_element(value);
30       break;
31
32       case 3:
33       std::print("Preorder traversal:\n");
34       for (const auto& item : tree.traverse_preorder()) {
```

```
35            std::print("{}, ", item);
36          }
37          std::print("\n");
38          break;
39
40          case 4:
41          std::print("Inorder traversal:\n");
42          for (const auto& item : tree.traverse_inorder()) {
43            std::print("{}, ", item);
44          }
45          std::print("\n");
46          break;
47
48          case 5:
49          std::print("Postorder traversal:\n");
50          for (const auto& item : tree.traverse_postorder()) {
51            std::print("{}, ", item);
52          }
53          std::print("\n");
54          break;
55
56          case 6:
57          std::print("1 - To text | 2 - To binary\n");
58          std::print("Do you want to export to text or binary file? "
    );
59          std::cin >> option2;
60          if (option2 == 1) {
61            fileTree.save_to_text("tree.txt", elements);
62            std::print("Tree saved to text file.\n");
63          }
64          else if (option2 == 2) {
65            fileTree.save_to_binary("tree.bin", elements);
66            std::print("Tree saved to binary file.\n");
67          }
68          break;
69
70          case 7:
71          std::print("1 - From text | 2 - From binary\n");
72          std::print("Do you want to import from text or binary file?
     ");
73          std::cin >> option2;
74          if (option2 == 1) {
75            fileTree.load_from_text("tree.txt", tree, elements);
76            std::print("Tree loaded from text file.\n");
77          }
```

```
78          else if (option2 == 2) {
79            fileTree.load_from_binary("tree.bin", tree, elements);
80            std::print("Tree loaded from binary file.\n");
81          }
82          break;
83          case 8:
84          tree.delete_tree();
85          std::print("Tree deleted.\n");
86          break;
87
88          case 9: {
89            std::print("Input the value to search: "), std::cin >>
    value;
90            auto path = tree.find_path(value);
91            if (path.empty()) {
92              std::print("Element not found.\n");
93            }
94            else {
95              std::print("Path: ");
96              for (const auto& item : path) {
97                std::print("{} ", item);
98              }
99              std::print("\n");
100           }
101         }
102         break;
103
104
105         default:
106         std::print("Invalid option. Try again.\n");
107         break;
108       }
109     } while (lauf());
110
111     return 0;
112   }
```

Listing 10. Klasa `main`

Na listingu nr 10 przedstawiona jest klasa main. Klasa main odpowiedzialna jest za sterowaniem operacjami na drzewie binarnym. Do przeprowadzania operacji stworzone zostało menu, które podgląd wyboru dostępnych opcji oraz prośbę o wybranie opcji działania. Następnie, w zależności od wyboru opcji instrukcja switch wywołuje odpowiednią metodę z klasy `BSTTree`. Po wykonaniu operacji wyświetlany jest komunikat o kontynuacji lub zakończeniu działania programu. Wykorzystywana jest

do tego funkcja "bool lauf"przedstawiona na listingu nr 11

```
1  bool lauf()
2  {
3    std::string input;
4    std::cout << "\nDo you want to repeat? (type yes ocontinue, no
     to stop): ", std::cin >> input;
5    do {
6      if (input == "Yes" || input == "yes" || input== "y") {
7        return true;
8      }
9      else if (input == "no" || input == "No" || input == "n") {
10       return false;
11     }
12     else {
13       std::cout << "insert ONLY yes or no", std::cin >> input;
14     }
15   } while (true);
```

Listing 11. Funkcja `lauf()`

Funkcja jest odpowiedzialna za sprawdzanie czy podano poprawną wartość w komunikacje o kontynuacji lub zakończeniu programu.

4.2. Ciekawe fragmenty kodu

```
1  int delete_element(T value) {
2  ...
3    if (elementOfValue->left == nullptr && elementOfValue->right ==
     nullptr) {
4      if (elementOfValue->parent != nullptr) {
5        if(elementOfValue->parent->left == elementOfValue) {
6          elementOfValue->parent->left = nullptr;
7        } else {
8          elementOfValue->parent->right = nullptr;
9        }
10     }
11
12     delete elementOfValue;
13   } else {
14 ...
15 }
```

Listing 12. Metoda `delete_element()`

Na listingu nr. 12, widać od linijki nr. 3 zanestowane zdania `if`. Służą one do

sprawdzania, czy dziecko `elementOfValue` jest połączone z rodzicem od lewej czy prawej strony, po czym wskaźnik rodzica jest zmieniany na `nullptr`. Dlatego, że C++ nie posiada łatwej możliwości tworzenia referencji do wskaźników, np. przy użyciu operatora `?`, czy lambdy, należy użyć takich niezbyt ładnych zabiegów.

```
1  std::vector<T> find_path(const T& value) {
2    std::vector<T> path;
3    Tree* curr = root;
4
5    while (root != nullptr) {
6      path.push_back(curr->contents);
7
8      if (curr->contents == value) {
9        return path;
10     }
11     else if (value < curr->contents) {
12       curr = curr->left;
13     }
14     else {
15       curr = curr->right;
16     }
17   }
18   return {};
19 }
```

Listing 13. Metoda `Filepath`

na listingu przedstawiona jest metoda szukająca drogi do podanej wartości. Metoda przyjmuje jeden parametr: value - poszukiwana wartość. W wektorze path w wierszu 2 przechowywane będą wartości. Wskaźnik curr na root w wierszu 3 wskazuje na root w drzewie, użyty jest aby nie modyfikować bezpośrednio root. droga jest wyszukiwana za pomocą pętli while. za pomocą instrukcji w wierszu 6 wartości są dodawane do wektora path. Instrukcja if służy do przejścia na dziecko lewe lub prawe.

5. Wnioski

- Konstrukcja drzewa binarnego przydaje się wtedy, kiedy mamy zamiar kilkakrotnie wyszukiwać dane z nieposortowanego zbioru
- Branchowanie w git bardzo pomaga w pracy kolaboracyjnej
- Rekurencja w pewnych algorytmach jest prostszym rozwiązaniem do zaimplementowania niż metody iteracyjne

Bibliografia

[1] *Artykuł Wikipedii o drzewie BST*. URL: https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwa%C5%84.

[2] *Strona Gita*. URL: https://git-scm.com/.

[3] *Strona Doxygena*. URL: https://www.doxygen.nl/.

Spis rysunków

Spis tabel

Spis listingów