a

AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Algorytm drzewa BST z zastosowaniem GitHub

Autor:
Dawid Michura
Sebastian Tatara
Dominik Jonik

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Celem niniejszego projektu jest opracowanie kompletnego systemu operującego na strukturze danych BST (Binary Search Tree)[1] zrealizowanej w języku C++. Projekt obejmuje implementację podstawowych operacji drzewa, moduł zapisu i odczytu danych z plików oraz interfejs użytkownika bazujący na menu konsolowym. Istotnym elementem projektu jest również udokumentowane wykorzystanie narzędzi Git i GitHub[2] jako platformy współpracy zespołowej, umożliwiającej kontrolę wersji, pracę z gałęziami i rozwiązywanie konfliktów.

Projekt ma umożliwić:

- zapoznanie się z drzewami BST i ich własnościami,
- praktyczne zastosowanie paradygmatu programowania obiektowego,
- zaprojektowanie systemu zgodnego z zasadami modularności,
- zdobycie doświadczenia w korzystaniu z Git i GitHub poprzez pracę równoległą, scalanie gałęzi, rozwiązywanie konfliktów oraz rejestrowanie historii zmian.

Projekt kończy się implementacją działającego programu oraz przygotowaniem szczegółowej dokumentacji technicznej.

1.1. Cel i zakres pracy

Celem pracy jest opracowanie działającego systemu BST, który umożliwia:

1. dodawanie i usuwanie węzłów,
2. wyszukiwanie elementów,
3. drukowanie drzewa w porządku preorder, inorder i postorder,
4. graficzne wyświetlanie struktury drzewa,
5. zapis i odczyt danych do/z plików binarnych.

Ponadto projekt zakłada użycie narzędzi Git i GitHub do wersjonowania oraz pracy zespołowej, z wymogiem tworzenia gałęzi funkcjonalnych, wykonania określonej liczby commitów i rozwiązywania konfliktów scalania.

[1] Binarne drzewo poszukiwań - Wikipedia [1].

[2] Więcej informacji o pracy zespołowej z użyciem GitHub można znaleźć na stronie [2].

1.2. Uzasadnienie wyboru problemu

Drzewo BST jest jedną z najważniejszych i najczęściej omawianych struktur danych. Jego prostota, a zarazem wysoka efektywność sprawia, że stanowi fundament dla licznych wariantów takich jak AVL, Red-Black Tree czy struktury B-Drzew wykorzystywane w systemach baz danych. Implementacja BST pozwala zrozumieć:

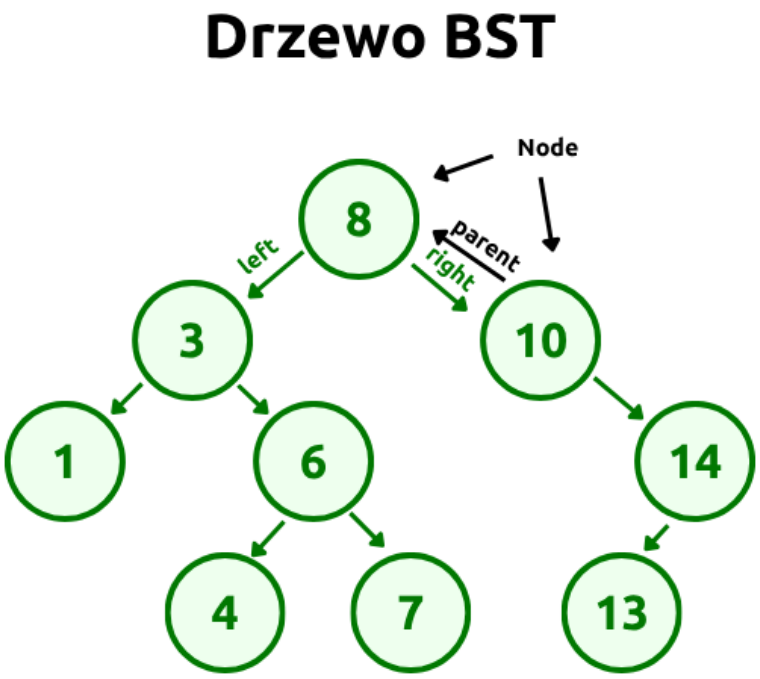
- działanie rekurencji,
- zarządzanie dynamiczną pamięcią,
- tworzenie abstrakcji klas i modularyzację,
- analizę złożoności algorytmicznej.

BST jest również idealnym przykładem do prezentacji w repozytorium GitHub, gdyż implementacja kolejnych funkcjonalności dzieli się naturalnie na mniejsze moduły, które można realizować w osobnych gałęziach.

1.3. Przykład działania struktury BST

Poniżej przedstawiono przykładowe drzewo BST zbudowane na podstawie sekwencji wejściowej: **8, 3, 10, 1, 6, 14, 4, 7, 13**.

Rysunek 1.1 przedstawia graficzną strukturę drzewa BST.



Rys. 1.1. Przykładowe drzewo BST

2. Analiza problemu

W niniejszym rozdziale dokonano analizy teoretycznych podstaw algorytmu drzewa binarnego poszukiwań (BST).

2.1. Zastosowanie algorytmu

Drzewa BST, dzięki zachowaniu porządku (lewy ¡ rodzic ¡ prawy), są powszechnie stosowane. Główne obszary zastosowań to:

- **Implementacja map i zbiorów:** Stanowią bazę dla ‘std::map‘ i ‘std::set‘ w C++, gwarantując logarytmiczny czas operacji ($O(\log n)$).
- **Systemy baz danych:** Warianty (B-drzewa) są podstawą mechanizmów indeksujących, umożliwiając szybkie wyszukiwanie rekordów.
- **Tablice symboli w kompilatorach:** Efektywne przechowywanie i wyszukiwanie informacji o zmiennych i funkcjach podczas kompilacji.

- **Algorytmy sortujące:** Realizacja sortowania (TreeSort)[3] przez wstawienie wszystkich elementów i odczytanie ich metodą *inorder*.

2.2. Opis działania algorytmu

Logika programu opiera się na implementacji kluczowych operacji:

- **Dodawanie elementu:** Rekurencyjne porównywanie wartości (mniejsza w lewo, większa w prawo) od korzenia, aż do znalezienia pustego miejsca (‘nullptr‘) i wstawienia tam nowego węzła.
- **Usuwanie elementu:** Wymaga rozpatrzenia trzech przypadków:
  1. **Węzeł jest liściem (brak dzieci):** Węzeł jest usuwany, rodzic otrzymuje wskaźnik ‘nullptr‘.
  2. **Węzeł ma jedno dziecko:** Rodzic usuwanego węzła zaczyna wskazywać bezpośrednio na jedyne dziecko (wnuka).
  3. **Węzeł ma dwoje dzieci:** Należy znaleźć jego następnika (najmniejszy element w prawym poddrzewie), skopiować jego wartość do usuwanego węzła, a następnie rekursywnie usunąć następnika (co sprowadza się do przypadku 1. lub 2.).
- **Przechodzenie (Traversal):** Implementacja trzech metod wyświetlania: *preorder* (Korzeń, Lewy, Prawy), *inorder* (Lewy, Korzeń, Prawy) oraz *postorder* (Lewy, Prawy, Korzeń).

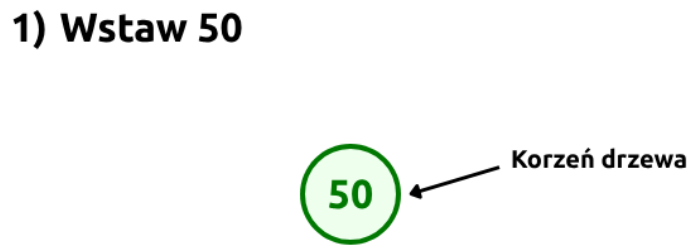
2.3. Ręczna analiza przykładu

Aby zobrazować działanie algorytmu, przeanalizujmy ręcznie operacje na zestawie danych: [**50, 30, 70, 20, 40, 60**].

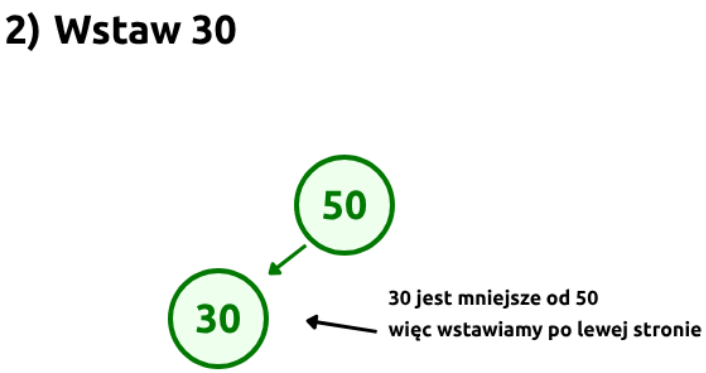
[3] Więcej o algorytmie Tree Sort [3].

2.3.1. Proces wstawiania

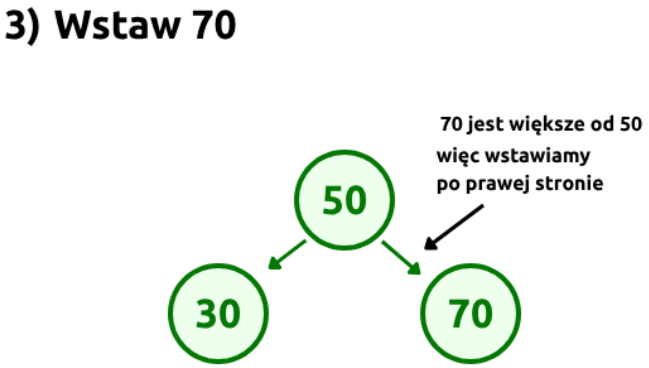
1. **Wstaw 50**: Drzewo puste. 50 staje się korzeniem. Przedstawia to Rysunek 2.1
2. **Wstaw 30**: $30 < 50$ (lewo). Wstaw 30 jako lewe dziecko 50. Przedstawia to Rysunek 2.2
3. **Wstaw 70**: $70 > 50$ (prawo). Wstaw 70 jako prawe dziecko 50. Przedstawia to Rysunek 2.3
4. **Wstaw 20**: $20 < 50$ (lewo) $\rightarrow 20 < 30$ (lewo). Wstaw 20 jako lewe dziecko 30. Przedstawia to Rysunek 2.4
5. **Wstaw 40**: $40 < 50$ (lewo) $\rightarrow 40 > 30$ (prawo). Wstaw 40 jako prawe dziecko 30. Przedstawia to Rysunek 2.5
6. **Wstaw 60**: $60 > 50$ (prawo) $\rightarrow 60 < 70$ (lewo). Wstaw 60 jako lewe dziecko 70. Przedstawia to Rysunek 2.6



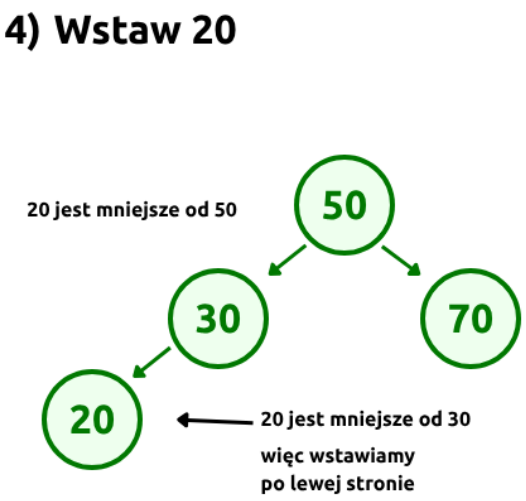
Rys. 2.1. Etap 1 tworzenia drzewka BST



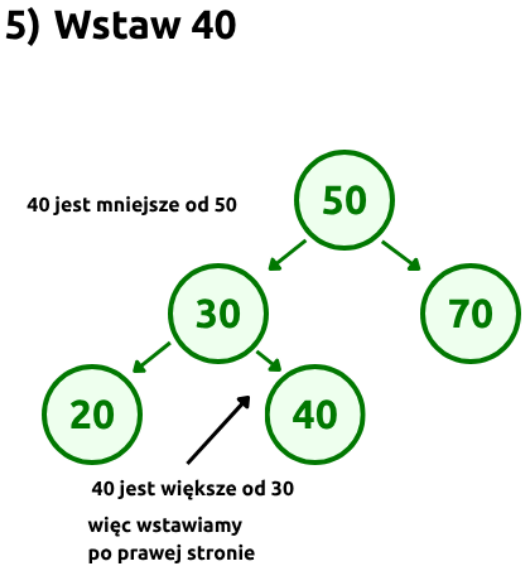
Rys. 2.2. Etap 2 tworzenia drzewka BST



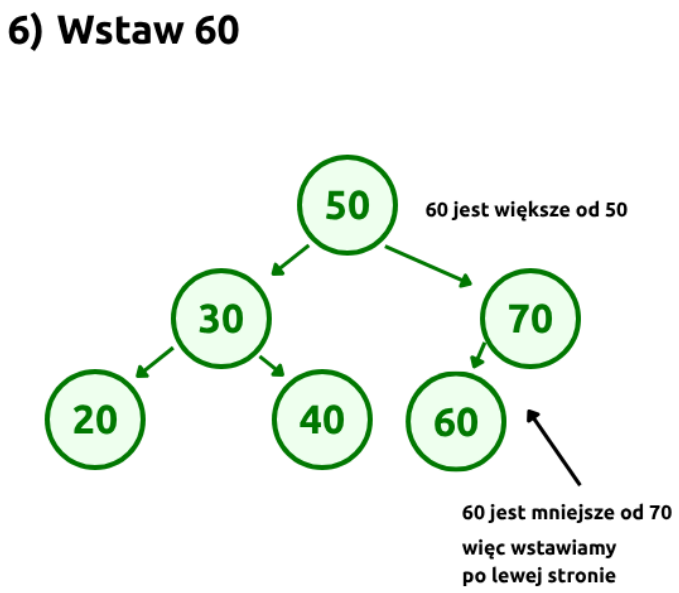
Rys. 2.3. Etap 3 tworzenia drzewka BST



Rys. 2.4. Etap 4 tworzenia drzewka BST



Rys. 2.5. Etap 5 tworzenia drzewka BST



Rys. 2.6. Etap 6 tworzenia drzewka BST

2.3.2. Proces usuwania (węzła 30)

Analiza usunięcia węzła 30 (przypadek z dwoma potomkami):

1. **Identyfikacja**: Znaleziono węzeł 30 (ma dwoje dzieci: 20 i 40).
2. **Znalezienie następnika**: Szukanie najmniejszego węzła w prawym poddrzewie (30). Następnikiem jest **40**.
3. **Kopiowanie wartości**: Wartość 40 jest kopiowana do węzła 30.
4. **Usunięcie następnika**: Rekursywne usunięcie węzła 40 z jego pierwotnej pozycji (jako prawe dziecko węzła o nowej wartości 40).
5. **Finalizacja**: Oryginalny węzeł 40 był liściem i zostaje usunięty.

3. Projektowanie

3.1. Wykorzystane narzędzia i technologie

Do realizacji projektu wykorzystano zestaw sprawdzonych narzędzi. Podstawowym językiem implementacji był **C++** w standardzie **C++17**, kompilowany przy użyciu **g++** (MinGW-w64).

Projekt opierał się w całości na **Bibliotece Standardowej (STL)**, bez zewnętrznych zależności. Kluczowe moduły STL to `<iostream>`, `<fstream>`, `<string>` oraz `<vector>`.

Kluczowym elementem zadania było wykorzystanie systemu kontroli wersji **Git** oraz platformy **GitHub** do zarządzania kodem i koordynacji pracy. Do przygotowania niniejszej dokumentacji wykorzystano system **LaTeX** (Overleaf), a do wygenerowania dokumentacji API kodu narzędzie **Doxygen**.

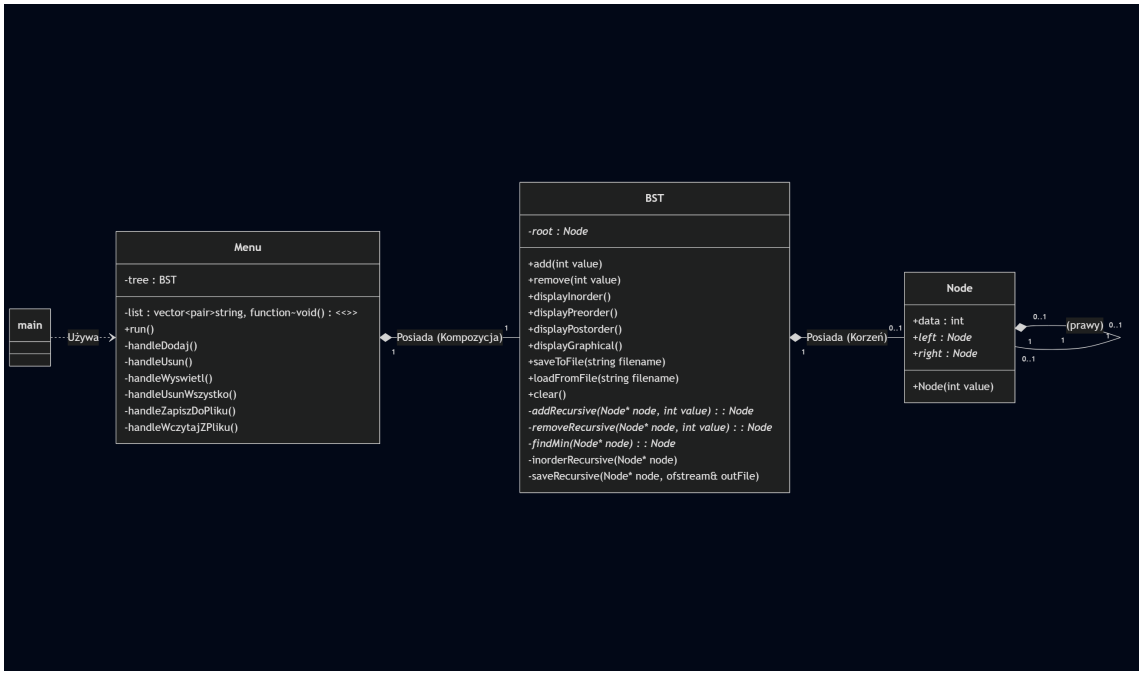
3.2. Architektura systemu i struktura klas

System zaprojektowano zgodnie z paradygmatem obiektowym, stosując separację odpowiedzialności. Logika drzewa (klasa 'BST') została oddzielona od logiki interfejsu (klasa 'Menu'), a plik 'main.cpp' pełni jedynie rolę startową. Architektura opiera się na następujących komponentach:

- **Struktura Node (wewnątrz BST.h):** Prosta struktura przechowująca wartość węzła ('data') oraz wskaźniki na potomków ('left', 'right').
- **Klasa BST (BST.h, BST.cpp):** Rdzeń aplikacji. Hermetyzuje całą logikę drzewa (posiada wskaźnik 'root') oraz, zgodnie z wymaganiami, implementuje metody zapisu/odczytu do plików.
- **Klasa Menu (Menu.h, Menu.cpp):** Warstwa odpowiedzialna za interfejs użytkownika. Posiada instancję klasy 'BST' i wywołuje jej publiczne metody (np. 'add', 'remove', 'saveToFile') w odpowiedzi na wybór użytkownika.
- **Plik main.cpp:** Punkt wejścia aplikacji, którego jedyną rolą jest utworzenie obiektu 'Menu' i uruchomienie jego głównej pętli.

Architektura systemu jest hierarchiczna i oparta na kompozycji, co precyzyjnie ilustruje Rysunek 3.1 (s. 12). Punkt wejścia aplikacji (`main`) pełni jedynie rolę startową – tworzy instancję klasy `Menu` i wywołuje jej główną pętlę (`run()`). Klasa

`Menu` (warstwa interfejsu) posiada na zasadzie kompozycji instancję klasy `BST` (widoczną jako pole `tree`). Z kolei rdzeń logiki, klasa `BST`, zarządza strukturą danych i posiada (poprzez wskaźnik `root`) obiekty typu `Node`.



Rys. 3.1. Diagram klas systemu BST.

3.3. Strategia pracy w Git

Zgodnie z wymaganiami projektu, zaprojektowano przepływ pracy (workflow) oparty na systemie Git i GitHub.

- **Model pracy:** Zastosowano model *feature branching*. Kilka osobnych gałęzi.
- **Gałąź główna:** Gałąź ‘main‘ jest traktowana jako stabilna. Zmiany są do niej dołączane wyłącznie przez scalanie (merge) ukończonych gałęzi funkcyjnych.
- **Praca równoległa i konflikty:** Projekt zakładał fazy pracy równoległej, celowo prowadzące do **konfliktów scalania**, aby przećwiczyć ich manualne rozwiązywanie.

3.4. Kluczowe decyzje algorytmiczne

Podczas projektowania logiki systemu podjęto następujące decyzje:

- **Rekurencja:** Większość logiki operacyjnej ‘BST‘ (dodawanie, usuwanie, przechodzenie) zostanie zaimplementowana rekurencyjnie.
- **Usuwanie węzła:** W przypadku węzła z dwoma potomkami, jego wartość zostanie nadpisana wartością jego **następnika** (najmniejszy element z prawego poddrzewa), który następnie zostanie usunięty.
- **Wyświetlanie graficzne:** Implementacja przejścia ”reverse in-order”(Prawy-Korzeń-Lewy) w celu wydruku drzewa obróconego o 90 stopni.
- **Serializacja binarna:** Zapis binarny nastąpi w kolejności **Pre-order (KLP)**. Gwarantuje to odtworzenie tej samej struktury drzewa podczas odczytu przez zwykłe wywoływanie metody ‘add()‘.

4. Implementacja

4.1. Narzędzia i struktura

Projekt zaimplementowano w języku **C++17**, kompilując go za pomocą **g++** z flagami ‘-Wall -Wextra‘ (Zobacz linię 8). Proces budowania został zautomatyzowany skryptami powłoki dla Windows (Listing 1) oraz systemów Unixowych (Listing 2). Wykorzystano wyłącznie bibliotekę STL, w tym ‘<fstream>‘ do obsługi plików oraz ‘<vector>‘ i ‘<functional>‘ do budowy dynamicznego menu.

```
1  @echo off
2  setlocal enabledelayedexpansion
3
4  REM Nazwa pliku wynikowego
5  set OUT=main.exe
6
7  REM Kompilacja wszystkich cpp w katalogu
8  g++ -Wall -Wextra -std=c++17 *.cpp -o %OUT%
9  if ERRORLEVEL 1 (
10     pause
11     exit /b 1
12 )
13
14 %OUT%
15 pause
```

Listing 1. Skrypt kompilacyjny ‘run.bat‘ dla Windows.

```
1  #!/bin/zsh
2
3  # Nazwa wyj Żciowego programu
4  OUT="main"
5
6  # Wyczy Ż  ekran
7  clear
8
9  # Kompilacja wszystkich plik w .cpp w bie  cym katalogu
10 g++ -Wall -Wextra -std=c++17 *.cpp -o $OUT
11
12 # Sprawd  kod wyj Żcia
13 if [ $? -eq 0 ]; then
14     ./$OUT
15 else
16     echo "${NC}"
```

```
fi
```

Listing 2. Skrypt kompilacyjny ‘run.sh‘ dla Unix.

Implementację podzielono na trzy główne komponenty: strukturę ‘Node‘ (reprezentującą węzeł), klasę ‘BST‘ (hermetyzującą logikę drzewa, Listing 3) oraz klasę ‘Menu‘ (warstwa interfejsu użytkownika, Listing 4). Plik ‘main.cpp‘ jedynie inicjalizuje i uruchamia pętlę menu.

```
class BST {
private:
    Node* root;
    Node* removeRecursive(Node* node, int value);
    Node* findMin(Node* node);
    Node* addRecursive(Node* node, int value);
    void saveRecursive(Node* node, ofstream& outFile);
    // ... (prywatne metody pomocnicze do czyszczenia i
   wyswietlania) ...
public:
    BST();
    ~BST();
    void clear();
    void add(int value);
    void remove(int value);
    // ... (publiczne metody wyswietlania i zapisu) ...
};
```

Listing 3. Skrócona definicja klasy ‘BST‘ w ‘BST.h‘.

```
class Menu {
private:
    std::vector<std::pair<std::string, std::function<void()>>> list
   ;
    BST tree;

    void handleDodaj();
    // ... (metody obslugi dla kazdej opcji menu) ...
public:
    Menu();
    void run();
};
```

Listing 4. Skrócona definicja klasy ‘Menu‘ w ‘Menu.h‘.

4.2. Implementacja kluczowych funkcjonalności

Większość logiki klasy ‘BST‘ opiera się na rekurencji. Metoda ‘add()‘ jest publicznym interfejsem dla ‘addRecursive()‘ (Listing 5). Funkcja ta schodzi w dół drzewa, aż znajdzie puste miejsce (‘nullptr‘), gdzie wstawia nowy węzeł (linia 4).

```
1  Node* BST::addRecursive(Node* node, int value) {
2      // Przypadek bazowy: znaleziono puste miejsce
3      if (node == nullptr) {
4          return new Node(value);
5      }
6
7      // Krok rekurencyjny: szukanie miejsca w poddrzewach
8      if (value < node->data) {
9          node->left = addRecursive(node->left, value);
10     } else if (value > node->data) {
11         node->right = addRecursive(node->right, value);
12     }
13     return node;
14 }
```

Listing 5. Implementacja dodawania węzła (‘BST.cpp‘).

Najbardziej złożoną operacją jest usuwanie (Listing 7). Po znalezieniu węzła, funkcja ‘removeRecursive‘ obsługuje trzy przypadki:

- **Przypadek 1 (Liść):** Węzeł nie ma dzieci (linia 10) i jest po prostu usuwany, zwracając ‘nullptr‘.
- **Przypadek 2 (Jedno dziecko):** Węzeł (linie 16 i 20) jest usuwany, a jego miejsce zajmuje jedyne dziecko.
- **Przypadek 3 (Dwoje dzieci):** Węzeł zastępowany jest swoim następnikiem (linia 28) – czyli najmniejszą wartością z prawego poddrzewa, znalezioną przez ‘findMin()‘ (Listing 6). Następnie następnik jest usuwany ze swojego pierwotnego miejsca (linia 29).

```
1  Node* BST::findMin(Node* node) {
2      while (node != nullptr && node->left != nullptr) {
3          node = node->left;
4      }
5      return node;
6  }
```

Listing 6. Implementacja metody ‘findMin‘ (‘BST.cpp‘).

```
Node* BST::removeRecursive(Node* node, int value) {
    // ... (pominiecie wyszukiwania wezla) ...
    if (value < node->data) {
        node->left = removeRecursive(node->left, value);
    } else if (value > node->data) {
        node->right = removeRecursive(node->right, value);
    }
    else { // Znaleziono wezel
        // Przypadek 1: Lisc
        if (node->left == nullptr && node->right == nullptr) {
            delete node;
            return nullptr;
        }

        // Przypadek 2: Jedno dziecko
        else if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }

        // Przypadek 3: Dwoje dzieci
        Node* successor = findMin(node->right);
        node->data = successor->data;
        node->right = removeRecursive(node->right, successor->data)
   ;
    }
    return node;
}
```

Listing 7. Implementacja usuwania węzła (‘BST.cpp‘).

Kluczową funkcją jest binarna serializacja. Zapis (Listing 8) odbywa się w kolejności **Pre-order (KLP)**. Wartość węzła jest zapisywana do pliku (linia 7) *przed* rekurencyjnym zejściem do lewego i prawego poddrzewa.

```
void BST::saveRecursive(Node* node, ofstream& outFile) {
    if (node == nullptr) {
        return;
    }

    // KLP (Pre-order)
```

```
7       outFile.write(reinterpret_cast<const char*>(&node->data),
        sizeof(int));
8       saveRecursive(node->left, outFile);
9       saveRecursive(node->right, outFile);
10  }
```

Listing 8. Binarny zapis drzewa w kolejności Pre-order (‘BST.cpp‘).

Dzięki zapisowi Pre-order, odczyt (Listing 9) staje się trywialny. Wystarczy wyczyścić drzewo, a następnie w pętli wczytywać kolejne wartości i dodawać je standardową metodą ‘add()‘ (linia 12), która automatycznie odtworzy identyczną strukturę drzewa.

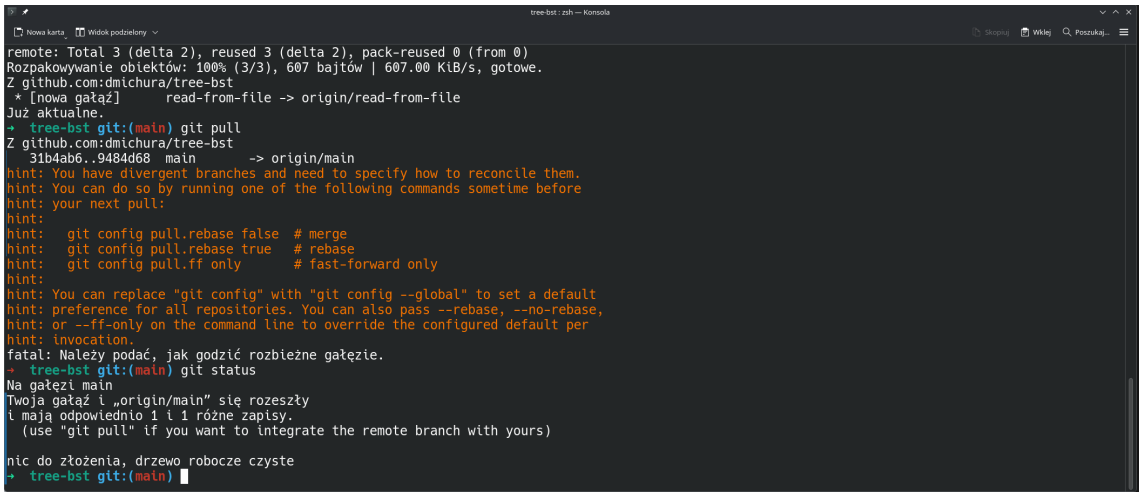
```
1   void BST::loadFromFile(const string& filename) {
2       ifstream inFile(filename, ios::binary);
3       if (!inFile.is_open()) {
4           cout << "Blad: Nie mozna otworzyc pliku " << filename << "
        do odczytu\n";
5           return;
6       }
7
8       clear();
9       int value;
10
11      while (inFile.read(reinterpret_cast<char*>(&value), sizeof(int)
        )) {
12          add(value);
13      }
14
15      inFile.close();
16  }
```

Listing 9. Implementacja binarnego odczytu (‘BST.cpp‘).

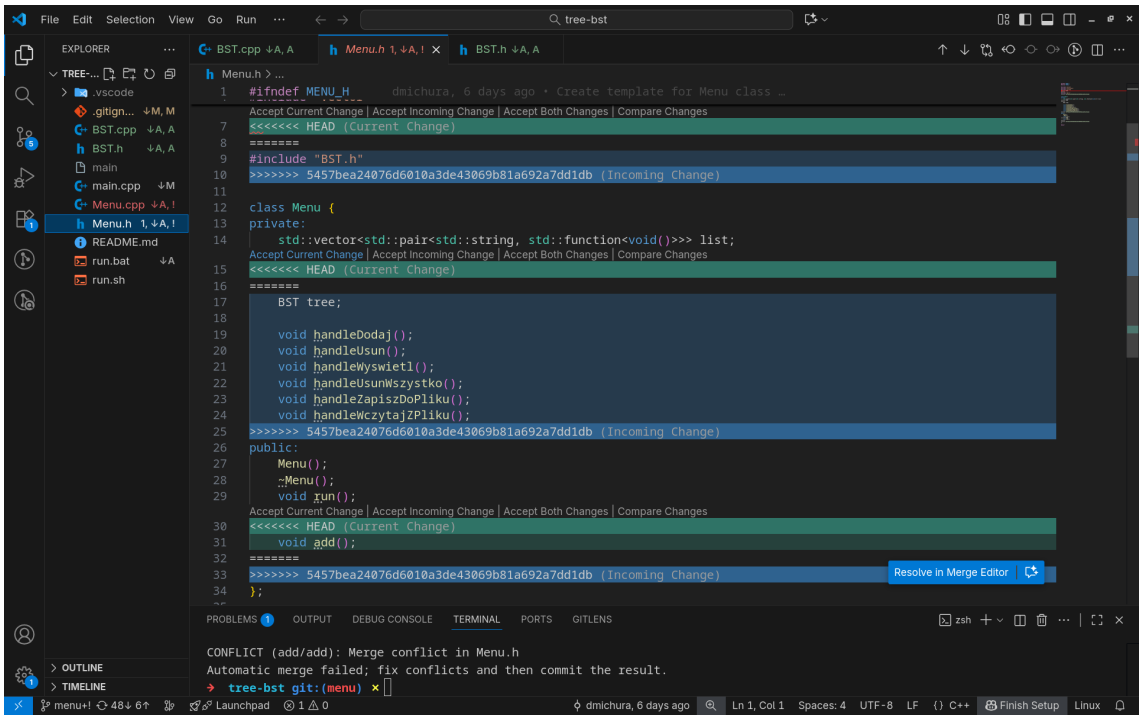
4.3. Implementacja procesu pracy z Git

Implementacja była zarządzana w Git zgodnie z metodyką *branching workflow*. Praca była dzielona na gałęzie. Kluczowym elementem była praca równoległa, co celowo doprowadziło do **konfliktów scalania** [4] w plikach ‘BST.cpp‘ i ‘BST.h‘. Zgodnie z założeniami, konflikty te były rozwiązywane manualnie i zatwierdzane w celu sfinalizowania scalenia. Wyniki tego procesu ilustruje (Rysunek 4.1) oraz rysunek (Rysunek 4.2).

[4] Więcej o konfliktach scalania [4].



Rys. 4.1. Konflikt scalania



Rys. 4.2. Rozwiązywanie konfliktu scalania

5. Wnioski

Realizacja niniejszego projektu była procesem wieloaspektowym. Osiągnięcie celu technicznego, jakim była implementacja w pełni funkcjonalnej struktury drzewa BST w C++, stanowiło jedynie część wyzwania. Równie istotnym, a z perspektywy zdobytego doświadczenia być może nawet ważniejszym, aspektem była nauka zarządzania kodem i organizacja pracy w zespole przy użyciu systemu Git.

Z perspektywy czysto programistycznej, projekt utrwalił naszą wiedzę na temat zarządzania pamięcią, rekurencji oraz złożoności algorytmicznej (szczególnie przy implementacji operacji usuwania węzła z dwoma potomkami).

Najcenniejsze wnioski płyną jednak bezpośrednio z wymogów dotyczących pracy zespołowej. Praca w trzyosobowej grupie na jednym repozytorium szybko zweryfikowała nasze nawyki. Konieczność tworzenia osobnych gałęzi (‘feature branching‘) dla każdej funkcjonalności przestała być teorią z wykładu, a stała się codzienną praktyką gwarantującą, że nie wchodzimy sobie nawzajem w drogę.

Wymóg celowego generowania i rozwiązywania konfliktów (łącznie 6) był najbardziej wartościową częścią zadania. Początkowa frustracja związana z ręcznym scalaniem rozbieżnych zmian w kodzie szybko przerodziła się w zrozumienie. Konflikty zmusiły nas do głębokiej analizy nie tylko własnego kodu, ale przede wszystkim kodu napisanego przez kolegów z zespołu. Nauczyło nas to więcej o komunikacji i planowaniu (”Kto dotyka ‘main.cpp‘? Jak modyfikujemy menu?”) niż jakakolwiek inna część projektu.

Podsumowując, projekt ”Drzewo BST”był bardziej lekcją inżynierii oprogramowania i efektywnej współpracy niż wyłącznie ćwiczeniem algorytmicznym. Zrozumieliśmy, że napisanie działającego fragmentu kodu to dopiero początek; prawdziwym wyzwaniem jest zintegrowanie go z pracą innych i utrzymanie spójnej, czytelnej historii projektu.

Bibliografia

[1] *Binarne drzewo poszukiwań - Wikipedia*. URL: https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwa%C5%84 (term. wiz. 14. 11. 2025).

[2] *Praca zespołowa przy użyciu systemu git*. URL: https://mareknowak.pl/notatki-z-gita-praca-zdalna-i-zespolowa/ (term. wiz. 14. 11. 2025).

[3] *Algorytm Tree Sort - Wikipedia*. URL: https://en.wikipedia.org/wiki/Tree_sort/ (term. wiz. 14. 11. 2025).

[4] *Więcej o konfliktach scalania*. URL: https://www.atlassian.com/pl/git/tutorials/using-branches/merge-conflicts/ (term. wiz. 14. 11. 2025).

Spis rysunków

Spis tabel

Spis listingów