AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Drzewa BST

Autor:
Kamil Gruca
Jakub Hajduk

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Wymagania w przypadku drzewa Binary search tree (BST) to funkcje jakie aplikacja będzie spełniać:

Wymagane funkcje:

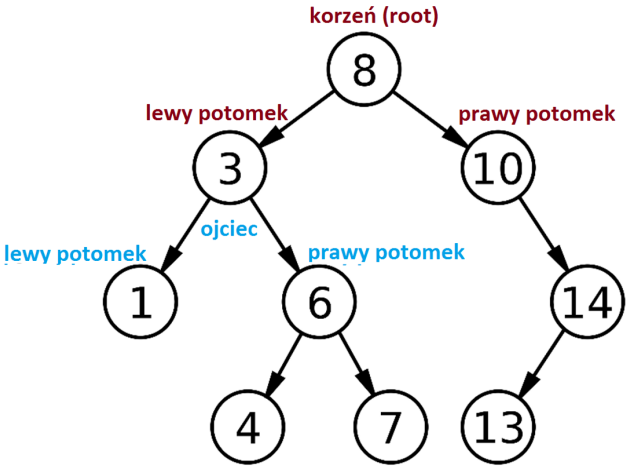
- **Dodawanie elementu** - Program umożliwia dodanie elementu do drzewa BST
- **Usuwanie elementu** - Program umożliwia dodanie elementu do drzewa BST
- **Usunięcie całego drzewa** - Program umożliwia usunąć całe drzewo BST
- **Szukanie drogi do podanego elementu** - Program umożliwia wyszukanie drogi do wybranego elementu drzewa BST
- **Wyświetlanie** - Program wyświetla graficznie na ekranie, użytkownik ma możliwość wyboru metody do wyświetlania drzewa. Użytkownik ma możliwość wyświetlania drzewa w kolejności: Postorder, Inorder, Preorder
- **Zapis do pliku tekstowego** - Program zapisuje wygenerowane drzewo do pliku tekstowego.
- **Zapisz i odczyt** - Program umożliwia zapisywanie i odczytywanie z pliku utworzonego drzewa BTS, plik musi być zapisany binarnie. Program powinien również mieć możliwość wczytania pliku tekstowego z cyframi, co daje możliwosc zbudowania drzewa BTS.
- **Wczytanie pliku** - Program powinien mieć możliwość wczytania pliku z liczbami do drzewa pustego lub istniejącego.
- **Menu z opcjami** - Funkcja main powinna wyświetlać menu z opcjami drzewa oraz odczytu i zapisu pliku.

2. Analiza problemu

2.1. Działanie algorytmu

Drzewo jest strukturą danych zbudowaną z elementów, które nazywamy węzłami. Dane przechowuje się w węzłach drzewa. Węzły są ze sobą powiązane w sposób hierarchiczny za pomocą krawędzi. Pierwszy węzeł drzewa nazywa się korzeniem. Od niego wychodzą pozostałe węzły, które nazywają się potomkami. Węzeł nadrzędny w stosunku do danego węzła potomnego nazywamy ojcem, rodzicem. Potomkowie są węzłami podrzędnymi w strukturze hierarchicznej.

Drzewo przeszukiwań binarnych - jest drzewem binarnym, w którym każdy węzeł spełnia następujące reguły: W momencie kiedy węzeł posiada lewe poddrzewo, to wszystkie węły w tym poddrzewie mają wartość niewiększą od wartości danego węzła, W momencie kiedy węzeł posiada prawe poddrzewo, to wszystkie węzły w tym poddrzewie są niemniejsze od wartości danego węzła. Przykładowe drzewo BST zostało przedstawione na rysunku 2.1 (s. 4). [1]



Rys. 2.1. Przykładowe drzewo BST[1]

2.1.1. Wyszukiwanie węzłów w drzewie BST

Drzewa BST pozwalają wyszukiwać zawarte w nich elementy z klasą złożoności obliczeniowej O(log n), gdzie n oznacza liczbę węzłów. Wyszukiwanie rozpoczyna się od korzenia drzewa. Porównuje się wartość węzła z wartością poszukiwania aż do momentu aż znajdzie się wyszukiwaną wartość.

[1] Zdjęcie ze strony: https://eduinf.waw.pl/inf/alg/001_search/0114.php

2.1.2. Dodawanie elementu do drzewa

Dodawanie elementu do drzewa BST polega na tym że bierzemy element który chcemy dodać następnie porównujemy go z wartością korzenia, jeśli jest mniejszy przechodzimy do lewego poddrzewia, jeśli większy przechodzimy do prawego poddrzewa. Następnie znowu porównujemy go z potomkiem korzenia, jeśli wartość jest większa przechodzimy na prawo, a jeśli wartość jest mniejsza to przechodzimy na lewo. Tą czynność należy powtarzać aż do momentu przejścia całego lewego bądź prawego poddrzewia.

2.1.3. Usuwanie elementu z drzewa

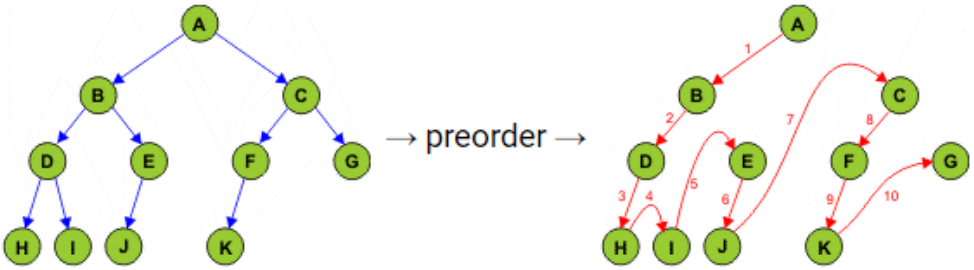
Proces usuwania węzła z drzewa BST jest zależne od struktury drzewa i konkretnego węzła którego należy sie pozbyć. Najpierw należy wyszukać węzeł który chcemy usunąć (wyszukiwanie zostało przedstawione w podrozdziale 2.1.1), następnie proces ten dzieli się na trzy przypadki:

- **Węzeł nie ma potomków** - Jeśli węzeł nie ma żadnych potomków to sie go usuwa.
- **Węzeł ma jednego potomka** - Jeśli węzeł, który chcemy usunąć, ma tylko jednego potomka (lewego lub prawego), wtedy zastępujemy ten węzeł jego jedynym potomkiem, oznacza to, że rodzic węzła, który usuwamy, zaczyna wskazywać na potomka.
- **Węzeł ma dwóch potomków** - W przypadku, gdy węzeł ma dwoje potomków, musimy znaleźć odpowiedni węzeł, który go zastąpi, najczęściej stosuje się następnika, czyli najmniejszy węzeł w prawym poddrzewie. Aby go znaleźć, przechodzimy w dół do lewego poddrzewa prawego potomka, ponieważ to tam znajduje się najmniejsza wartość większa od usuwanego węzła. Następnie zamieniamy wartość usuwanego węzła na wartość następnika. Ostatnim krokiem jest usunięcie następnika z jego oryginalnego miejsca, ponieważ jego wartość została już przeniesiona na miejsce usuwanego węzła.

2.1.4. Przejście drzewa BST

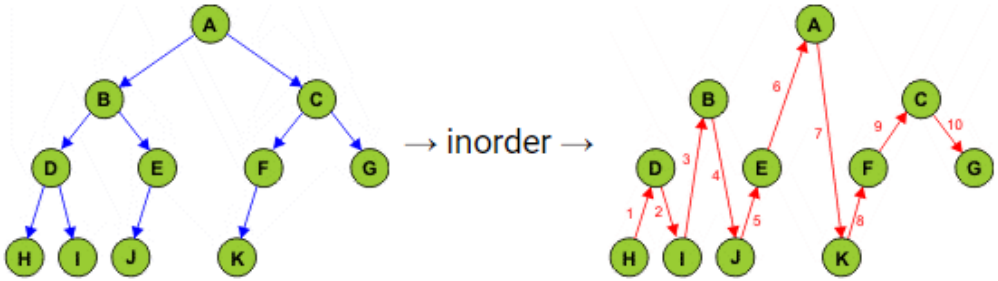
Przejście drzewa BST polega na odwiedzeniu wszystkich węzłów drzewa w określonym porządku. Istnieją trzy główne sposoby przejścia drzewa BST: in-order (porządkowe), pre-order (przedporządkowe) oraz post-order (poporządkowe). Każdy z tych sposobów różni się kolejnością odwiedzania węzłów drzewa.

- **Przejście preorder** - W przejściu pre-order najpierw odwiedzamy bieżący węzeł, a dopiero potem jego lewe i prawe poddrzewo. Przykładowe przejście drzewa metodą preorder zostało przedstawione na rysunku 2.2 (s. 6). [2]



Rys. 2.2. Przeszukanie preorder[2]

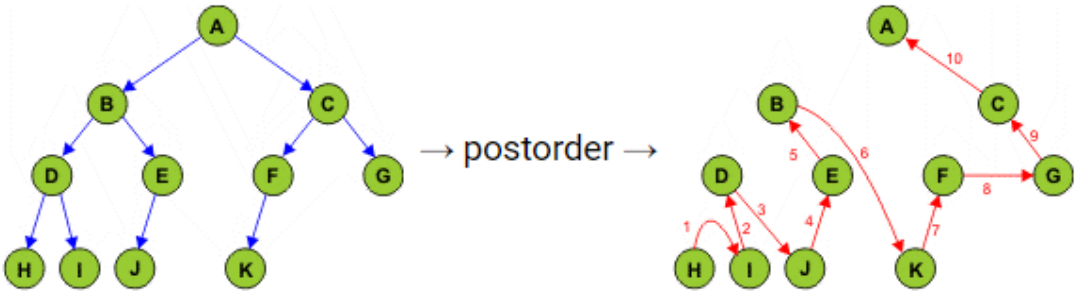
- **Przejście inorder** - W przypadku przejścia inorder odwiedzamy węzły w porządku rosnącym, co oznacza, że najpierw przechodzimy do lewego poddrzewa, potem odwiedzamy bieżący węzeł, a na końcu prawe poddrzewo. Przykładowe przejśćie drzewa metodą inorder zostało przedstawione na rysunku 2.3 (s. 6). [3]



Rys. 2.3. Przeszukanie inorder[2]

[2]Zdjęcie ze strony: https://eduinf.waw.pl/inf/alg/001_search/0114.php
[3]Zdjęcie ze strony: https://eduinf.waw.pl/inf/alg/001_search/0109.php

- **Przejście postorder** - Przejście postorder polega na tym, że najpierw przechodzimy przez lewe i prawe poddrzewo, a na końcu odwiedzamy bieżący węzeł. Przykładowe przejście drzewa metodą postorder zostało przedstawione na rysunku 2.4 (s. 7).[4]



Rys. 2.4. Przeszukanie postorder[2]

2.2. Zastosowanie drzew BST

- **Bazy danych** - Drzewa BST, są stosowane w implementacji indeksów w bazach danych. Umożliwiają one szybkie przeszukiwanie danych, wstawianie nowych rekordów i usuwanie istniejących przy zachowaniu odpowiedniego porządku.

- **Systemy plików** - Drzewa BST są wykorzystywane w systemach plików do organizacji hierarchicznej struktury katalogów i plików. Ułatwiają szybkie przeszukiwanie drzewa katalogów, znajdowanie plików czy zarządzanie systemem plików.

- **Sortowanie i przeszukiwanie** - Dzięki właściwościom drzewa BST, gdzie wartości węzłów są uporządkowane, drzewa te znajdują zastosowanie w algorytmach sortowania i przeszukiwania.

- **Kompilatory** - W kompilatorach i interpreterach drzewa BST są stosowane do organizacji i przeszukiwania tabel symboli, które przechowują informacje o zmiennych, funkcjach, klasach itp. Pozwala to na szybki dostęp do zdefiniowanych symboli w programie.

[4]Zdjęcie ze strony: https://eduinf.waw.pl/inf/alg/001_search/0109.php

3. Projektowanie

3.1. Języki programowania wykorzystane w projekcie

W tym projekcie tak jak i było w wymaganiach został wykorzystany tylko i wyłącznie język C++. Język ten jest dobrym językiem do implementacji drzewa BST (Binary Search Tree) z kilku istotnych powodów, które wynikają zarówno z cech języka, jak i jego zaawansowanych mechanizmów zarządzania pamięcią. Najważniejsze cechy:

- **Bezpośredni dostęp do wskaźników** - C++ daje możliwość korzystania z wskaźników, co pozwala na efektywną manipulację drzewami BST. Każdy węzeł w drzewie może mieć wskaźniki na swoich potomków, co umożliwia szybkie przechodzenie po strukturze drzewa. Wskaźniki zapewniają niski narzut na wydajność i precyzyjną kontrolę nad strukturą danych.

- **Dynamiczna alokacja pamięci** - C++ zapewnia pełną kontrole nad zarządzaniem pamięcią, co jest kluczowe przy implementacji struktur danych, takich jak drzewa BST.

3.2. Narzędzia które zostały wykorzystane w projekcie

W tym projekcie do pisania kodu został wykorzystany program Visual Studio 2022. Program ten to jedno z najpopularniejszych zintegrowanych środowisk programistycznych (IDE), które oferuje wiele funkcji i narzędzi ułatwiających programowanie w C++. Visual Studio ma wbudowaną obsługę systemu kontroli wersji Git, co ułatwia zarządzanie kodem źródłowym, śledzenie zmian i współpracę z innymi programistami. Visual Studio oferuje zaawansowane wsparcie dla języka C++ oraz standardowej biblioteki szablonów (STL), co ułatwia implementację i korzystanie z różnorodnych struktur danych i algorytmów.

Visual Studio używa kompilatora Microsoft Visual C++ (MSVC), który jest rozwijany i utrzymywany przez firmę Microsoft. MSVC jest jednym z najpopularniejszych kompilatorów C++ i jest szczególnie dobrze zintegrowany z ekosystemem Windows. MSVC dostarcza narzędzia do debugowania, które pozwalają na analizowanie błędów w kodzie źródłowym. Umożliwia śledzenie wartości zmiennych, ustawianie punktów przerwania oraz analizowanie stosu wywołań, co jest przydatne w przypadku złożonych aplikacji.

3.3. System kontroli wersji GIT

W trakcie realizacji projektu na bieżąco był używany system kontroli wersji GIT. Jest to system kontroli wersji, który pozwala na śledzenie zmian w plikach oraz współpracę z innymi programistami. Git jest rozproszonym systemem, co oznacza, że każdy programista posiada pełną kopię repozytorium na swoim lokalnym komputerze, a nie tylko jedną centralną wersję. Ważnym aspektem GITa jest to, że przechowuje on pełną historie zmian co pozwala na łatwe przywracanie wcześniejszych wersji kodu. Kolejnym ważnym aspektem Gita jest to, że pozwala on na tworzenie gałęzi (branchy), które pozwalają na rownoległą prace nad różnymi wersjami kodu, a nie wpływają na główną wersje kodu

W projekcie został wykorzystany hostingowy serwis internetowy przeznaczony do projektów programistycznych - GitHub - do zarządzania systemem kontroli wersji GIT. GitHub to serwis, w którym można przechowywać swój kod źródłowy, jak i zarządzać całym procesem wytwarzania oprogramowania: od ukrywania plików źródłowych przed publicznością dzięki repozytoriom prywatnym, przez możliwość ich pobierania dzięki komendzie git clone, po prośbę o wprowadzenie zmian dzięki opcji tworzenia pull request.

4. Implementacja

4.1. Struktura kodu

W naszym projekcie stworzyliśmy klasę, która reprezentuje drzewo BST (Binary Search Tree). Działa ona na zasadzie uporządkowania danych. Każdy węzeł drzewa ma wartość, która decyduje, czy dany element zostanie umieszczony po lewej stronie (jeśli jest mniejszy od bieżącego węzła) czy po prawej (jeśli jest większy). Dzięki tej strukturze możliwe jest efektywne dodawanie, usuwanie i szukanie elementów. Nasz program został podzielony na kilka różnych plików, zgodnie z założeniem. Ułatwia to zrozumienie takiego programu, a także edycję, gdyż łatwiej i szybciej jest znaleźć określone funkcje/metody w kodzie.

Projekt drzewa BST został podzielony na 5 plików, które odpowiadają za określone działanie programu:

1. Plik BST.cpp:
   W tym pliku znajduje się szczegółowa implementacja wszystkich metod klasy BST. Oznacza to, że dla każdej metody mamy odpowiednią funkcję, która pokazuje, jak działa ta metoda. Na przykład, w metodzie odpowiedzialnej za dodawanie nowego węzła do drzewa, wykorzystujemy porównania wartości, aby znaleźć właściwe miejsce, w którym nowy węzeł powinien zostać umieszczony. Dzięki tym porównaniom sprawdzamy, czy nowa wartość jest mniejsza czy większa od wartości w aktualnym węźle, pozwala nam to na kierowanie się w lewo lub w prawo w naszym drzewie.

   Natomiast dzięki oddzieleniu implementacji od definicji, możemy łatwo zmieniać lub dodawać nowe funkcje, nie wpływając na inne części programu. Taki podział sprawia, że kod staje się bardziej przejrzysty i zorganizowany oraz łatwiejszy do implementacji/edycji.

2. Plik BST.h:
   W tym pliku definiujemy strukturę Node oraz klasę BST, która zawiera wszystkie metody do obsługi drzewa. Klasa ta jest najważniejszym plikiem naszego programu i zarządza całą strukturą drzewa. Zawiera metody do:

   - dodawania elementu do drzewa,
   - usuwania elementu z drzewa,
   - usuwania całego drzewa,
   - szukania drogi do podanego elementu,

- wyświetlania drzewa graficznie w trzech metodach (preorder, inorder, postorder),
- zapisu do pliku tekstowego wygenerowanego drzewa.

3. Plik Plik.cpp:
   W tym pliku zawarta jest implementacja funkcji zadeklarowanych w Pliki.h. Odpowiada za konkretne działania związane z plikami:

   - Wczytywanie danych z pliku tekstowego: Otwiera plik tekstowy i każdą liczbę w nim zapisaną dodaje do drzewa BST. Umożliwia to szybkie przenoszenie danych z pliku do programu.
   - Zapis węzłów w pliku binarnym: Przetwarza każdy węzeł drzewa i zapisuje go do pliku w formie binarnej.
   - Zapis całego drzewa do pliku binarnego: Zapisuje stan drzewa BST do pliku binarnego, dzięki czemu można łatwo go później odtworzyć.
   - Odczyt drzewa z pliku binarnego: Wczytuje dane zapisane wcześniej w pliku binarnym i odbudowuje drzewo.

4. Plik Plik.h:
   W pliku Pliki.h znajduje się definicja klasy Pliki, która odpowiada za operacje związane z odczytem i zapisem drzewa BST do plików. Ten plik określa, jakie funkcje będą dostępne, ale nie opisuje ich działania.

5. Plik main.cpp:
   Ten plik działa jako interfejs, przez który użytkownik może korzystać z programu. Zawiera funkcję main, która jest głównym punktem uruchomienia programu. W pliku main.cpp użytkownik ma kilka opcji do wyboru:

   - Wybieranie operacji: Program pokazuje menu z różnymi opcjami do wyboru dla użytkownika, takimi jak dodawanie nowego węzła, usuwanie istniejącego węzła lub wyświetlanie struktury drzewa. Użytkownik może wpisać, co chce zrobić, a program odpowiada wykonując odpowiednie operacje wyświetlone na ekranie.
   - Wprowadzanie danych: Program prosi użytkownika o podanie konkretnych wartości, które mają być dodane do drzewa lub usunięte. Dzięki temu użytkownik ma pełną kontrolę nad danymi w drzewie, może je modyfikować w dowolny sposób.

- Zapis do pliku, a także odczyt: Użytkownik ma możliwość zapisania bieżącego drzewa do pliku, a także wczytania drzewa z pliku. To bardzo ułatwia zarządzanie danymi, ponieważ nie trzeba na nowo tworzyć drzewa za każdym razem, gdy program jest uruchamiany. Oprócz tego można dodać wartości do juz istniejącego pliku, co poprawia funkcjonalność takiego programu.

Do wyświetlania drzewa zaimplementowano trzy różne sposoby przeglądania jego węzłów:

- Preorder: najpierw wyświetlamy bieżący węzeł, potem lewe poddrzewo, na końcu prawe.
- Inorder: najpierw lewe poddrzewo, potem bieżący węzeł, na końcu prawe.
- Postorder: najpierw lewe poddrzewo, potem prawe, a na końcu bieżący węzeł.

Każdy z tych sposobów prezentuje drzewo w innym porządku, co pozwala użytkownikowi lepiej zrozumieć jego strukturę.

Bardzo przydatną implementacją w naszym kodzie jest dodanie funkcji do zapisu i odczytu z pliku. Umożliwiamy użytkownikom w ten sposób zapisanie drzewa do pliku tekstowego lub binarnego. Zapis binarny jest szybszy i zajmuje mniej miejsca, natomiast zapis tekstowy jest łatwiejszy do odczytania przez użytkownika. Odczyt z pliku pozwala na przywrócenie stanu drzewa, dzięki czemu można kontynuować pracę z tymi samymi danymi, które zostały wcześniej zapisane.

4.2. Rozłożenie metody BST_dodanie_elementu

Funkcja BST_dodanie_elementu wchodzi w skład klasy BST i służy do dodawania elementu do drzewa BST. Funkcja ta została przedstawiona na listingu 1 (s. 13).

```
1  void BST::BST_dodanie_elementu(int v) {
2      Node* newNode = new Node(v);
3
4      if (root == nullptr) {
5          root = newNode;
6          return;
7      }
8
9      Node* obecny_element = root;
10     Node* rodzic = nullptr;
11
12     while (obecny_element != nullptr) {
13
14         rodzic = obecny_element;
15         if (v < obecny_element->data) {
16             obecny_element = obecny_element->left;
17         }
18         else {
19             obecny_element = obecny_element->right;
20         }
21     }
22
23     if (v < rodzic->data) {
24         rodzic->left = newNode;
25     }
26     else {
27         rodzic->right = newNode;
28     }
29 }
```

Listing 1. Metoda BST_dodanie_elementu

Szczegółowe omówienie metody przedstawionej na listingu 1 (s. 13):

- Tworzymy nowy węzeł (Linia 2)
- Sprawdzamy za pomocą pętli if czy drzewo jest puste, jeśli tak to nowy element staje się korzeniem (Linia 4-7)
- Tworzemy dwa wskaźniki obecny_element oraz rodzic, wskaźnik obecny_element zaczyna od korzenia drzewa i będzie przechodził przez drzewo w poszukiwaniu odpowieniego miejsca dla nowego elementu (linia 9,10)
- Pętla while szuka właściwego miejsca do wstawienia nowego węzła, pętla trwa dopóki obecny_element nie stanie się nullptr, co oznacza że dotarliśmy do miejsca gdzie można wstawić nowy elemet. Rodzic przechowuje wskaźnik na aktualny węzeł (obecny_element). Pętla if porównuje wartość obecną do nowego elementu, jeśli jest mniejsza to idzie na lewo, jeśli większa to idzie na prawo. (Linia 12-21)
- W momencie kiedy wykonuje się ta pętla if obecny_element jest nullptr, a rodzic wskazuje na węzeł do którego należy dodać nowy węzeł, w zależności czy v jest mniejsze czy większe od wartości rodzica. Nowy element jest ustawiany jako lewe lub prawe dziecko rodzica. (Linia 23 - 29)

4.3. Rozłożenie metody BST_preorder

Metoda BST_preoder służy do wyświetlania drzewa w porządku preorder, kod tej metody został przedstawiony na listingu 2 (s. 15).

```
1  void BST::BST_preorder(Node* node) {
2
3      if (node == nullptr) {
4          return;
5      }
6
7      std::cout << node->data << " ";
8
9      BST_preorder(node->left);
10
11     BST_preorder(node->right);
12 }
13
14
15 void BST::BST_preorder_wyswietl() {
16
17     BST_preorder(root);
18
19     std::cout << std::endl;
20 }
```

Listing 2. Metoda BST_preorder

Szczegółowe omówienie metody przedstawionej na listingu 2 (s. 15):

- Pętla if sprawdza, Jeśli node jest nullptr, to znaczy, że dotarliśmy do końca gałęzi (liścia) i nie ma nic do przetwarzania. Wtedy funkcja kończy działanie dla tego węzła. (Linia 3-5)
- Wyświetlamy wartość aktualnego węzła (czyli korzenia) za pomocą std::cout (Linia 7)
- Rekurencyjnie wywołujemy funkcje BST_preorder dla lewego dziecka, i przechodzimy do lewego poddrzewia. (Linia 9)
- Na końcu wywołujemy rekurencyjnie funkcję BST_preorder dla prawego dziecka węzła, czyli przechodzimy do prawego poddrzewa.
- Funkcja BST_preorder_wyswietl() jest prostą funkcją pomocniczą, która wywołuje funkcję BST_preorder zaczynając od korzenia (Linia 15-20)

4.4. Rozłożenie metody BST_szukaj_drogi

Funkcja BST_szukaj_drogi służy do znalezienia konkretnej wartości „v"podanej przez użytkownika w drzewie BST i wyświetlenia ścieżki, którą podąża podczas szukania tego elementu, wypisując przy tym wszystkie wartości. Kod tej metody został przedstawiony na listingu 3 (s. 16):

```
1  Node* BST::BST_szukaj_drogi(Node* node, int v) {
2      if (node == nullptr) {
3          return nullptr;
4      }
5      std::cout << node->data;
6      if (v < node->data) {
7          std::cout << " -> ";
8          return BST_szukaj_drogi(node->left, v);
9      }
10     else if (v > node->data) {
11         std::cout << " -> ";
12         return BST_szukaj_drogi(node->right, v);
13     }
14     else {
15         return node;
16     }
17 }
```

Listing 3. Metoda BST_preorder

Szczegółowe omówienie metody przedstawionej na listingu 3 (s. 16):

- Na początku sprawdza, czy aktualny węzeł (node) jest pusty (nullptr). Jeśli jest pusty, zwraca nullptr, co oznacza, że elementu o wartości v nie ma w drzewie. (Linia 2-4)
- Następnie funkcja wypisuje wartość aktualnego węzła wypisując, przez które węzły przechodzi podczas szukania elementu. (Linia 5)
- Jeśli szukana wartość jest mniejsza niż wartość bieżącego węzła, funkcja wypisuje strzałkę i rekurencyjnie wywołuje się na lewym poddrzewie (Linia 6-9)
- Jeśli szukana wartość „v"jest większa, również wypisuje strzałkę i wywołuje się rekurencyjnie na prawym poddrzewie (Linia 10-13)
- Gdy funkcja natrafi na węzeł, którego wartość jest równa v, oznacza to, że znalazła poszukiwany element. Zwraca wtedy ten węzeł jako wynik, kończąc szukanie. (Linia 14-16)

4.5. Powstałe wyniki

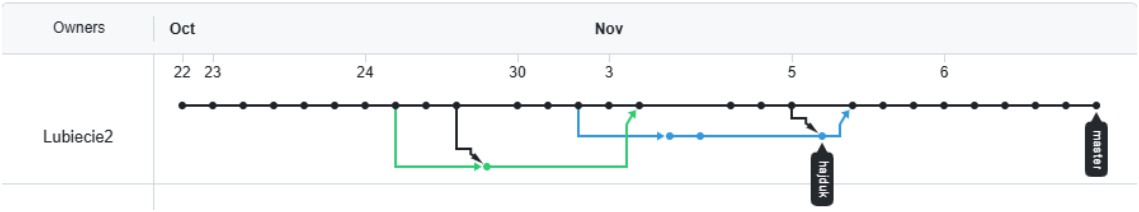
W efekcie naszej pracy stworzyliśmy w pełni funkcjonalny program, który realizuje wszystkie podstawowe operacje związane z drzewem BST. Dzięki niemu użytkownik ma możliwość dodawania nowych elementów, usuwania istniejących oraz wyszukiwania wartości w drzewie. Program oferuje także funkcję zapisu danych do pliku oraz ich późniejszego odczytu, co jest bardzo przydatne, gdy chcemy zachować bieżący stan drzewa na przyszłość. Użytkownik może również wyświetlać drzewo w różnych metodach, dzięki czemu może w łatwy sposób zrozumieć strukturę drzewa.

Zastosowanie narzędzi takich jak Git i GitHub znacząco ułatwiło nam pracę w zespole. Dzięki nim mogliśmy równolegle pracować nad różnymi częściami projektu, a później bez problemu łączyć nasze zmiany w jeden spójny kod. To z kolei przyczyniło się do lepszej efektywności oraz organizacji naszej pracy.

Dodatkowo, korzystanie z Doxygen umożliwiło nam tworzenie dokładnej dokumentacji naszego kodu. Używając programu Doxygen, byliśmy w stanie jasno opisać funkcje i klasy, które zostały użyte w naszym programie.

4.6. Drzewo commitów z Githuba

Na rysunku nr. 4.1 znajduję się drzewo commitów które powstało w naszym projekcie na zdalnym repozytorium GitHub, przedstawia ono commity na poszczególnych gałęziach (branch), oraz scalanie ich które było kluczowe w naszym projekcie



Rys. 4.1. Drzewo Network graph

5. Wnioski

Głównym celem naszego projektu było stworzenie działającego drzewa BST, które mogłoby wykonywać kluczowe operacje, takie jak dodawanie, usuwanie oraz wyszukiwanie elementów. Te podstawowe funkcje są niezwykle ważne, ponieważ umożliwiają skuteczne zarządzanie danymi w tej strukturze drzewa. Dodatkowo, zaimplementowaliśmy możliwość zapisywania aktualnego stanu drzewa do pliku oraz wczytywania danych z pliku. To pozwala użytkownikom na zachowanie swoich danych i łatwe przywracanie ich do programu w przyszłości. Taki sposób przechowywania danych pozwala na łatwe zarządzanie informacjami oraz szybkie ich przywracanie, dzięki czemu nasz projekt staje się bardziej funkcjonalny i użyteczny.

Podczas realizacji projektu korzystaliśmy z narzędzi takich jak Git oraz platformy GitHub, które pomogły nam nie tylko w śledzeniu zmian w kodzie, ale także w pracy zespołowej. Git umożliwił nam pracę równoległą, gdzie każdy mógł pracować nad innymi częściami projektu, a następnie scalaliśmy te zmiany. Ważnym elementem było zrozumienie, jak działa proces scalania w Gicie. Wtyczki zainstalowane w Visual Studio Code lub Visual Studio 2022 były niezwykle pomocne w śledzeniu zmian w repozytorium i ich łączeniu. Po scalaniu zawsze analizowaliśmy zmiany, by upewnić się, że wszystko działa poprawnie. Często sprawdzaliśmy pliki na GitHubie, aby zobaczyć, jak wyglądają po scaleniu. Pozwalało nam to wykryć ewentualne konflikty lub niezgodności w kodzie, zanim nasz kod trafiłby do ostatecznej wersji.

Natomiast Doxygen ułatwił nam dokumentowanie kodu. Dzięki niemu mogliśmy łatwo stworzyć dokumentację, która będzie przydatna w przyszłości – czy to do rozwijania projektu, czy lepszego zrozumienia działania programu.

W trakcie pracy nad projektem nauczyliśmy się również, jak ważne jest testowanie kodu na różnych etapach postępowych projektu oraz jak kluczowa jest komunikacja w zespole, szczególnie gdy pracujemy nad wspólnym repozytorium. Każdy etap projektu, od tworzenia metod/funkcji programu, przez testowanie, aż po scalanie, pomógł nam zrozumieć, jak odpowiednio zarządzać projektem i kontrolą wersji, co jest bardzo przydatne w tego typu projektach.

Bibliografia

[1] *Strona internetowa Zielony Buszmen*. URL: https://zielonybuszmen.github.io/2017/02/20/binarne-drzewo-poszukiwan-binary-search-tree-bst/ (term. wiz. 22. 10. 2024).

[2] *Serwis Edukacyjny Nauczycieli I LO w Tarnowie*. URL: https://eduinf.waw.pl/inf/alg/001_search/0109.php (term. wiz. 23. 10. 2024).

[3] *https://4programmers.net/*. URL: https://4programmers.net/C/Artyku%C5%82y/Odczyt_i_zapis_plik%C3%B3w_binarnych_w_Cpp (term. wiz. 02. 11. 2024).

[4] *pprogramowanie*. URL: https://www.p-programowanie.pl/matura-zadania/zapis-danych-do-pliku (term. wiz. 03. 11. 2024).

Spis rysunków

Spis listingów