AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Implementacja i analiza algorytmów pracy z drzewem wyszukiwania z zastosowaniem GitHub

Autor:
Blavitskyi Mykola
Stasiuk Oleh
Trudov Mykhailo

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

1.1. Cel Projektu

Celem projektu jest implementacja struktury danych drzewa binarnego wyszukiwania (BST) z następującymi funkcjonalnościami:

- Wstawianie elementów do drzewa.
- Usuwanie elementów z drzewa.
- Wyszukiwanie ścieżki do określonego elementu.
- Wyświetlanie drzewa w trzech porządkach przeszukiwania: preorder, inorder i postorder.
- Zapis struktury drzewa do pliku binarnego.
- Odczyt struktury drzewa z pliku binarnego.

1.2. Opis funkcjonalności

System umożliwia użytkownikowi interakcję z drzewem za pomocą menu:

1. Dodawanie nowego elementu do drzewa.
2. Usuwanie istniejącego elementu.
3. Wyświetlanie drzewa w wybranym porządku przeszukiwania.
4. Zapisywanie aktualnego stanu drzewa do pliku.
5. Wczytywanie drzewa z wcześniej zapisanego pliku.
6. Zakończenie działania programu.

1.3. Technologie

Projekt został zrealizowany w języku C++ z użyciem obiektowego podejścia programowania. Obsługa plików została zaimplementowana z wykorzystaniem strumieni wejścia i wyjścia (fstream). Dodatkowo zapewniono zgodność z nowoczesnymi standardami języka C++.

1.4. Przykładowe użycie

Użytkownik może wprowadzić wartości, wyświetlić drzewo w różnych porządkach, a następnie zapisać je do pliku. Po ponownym uruchomieniu programu może wczytać dane z pliku i kontynuować pracę.

2. Analiza problemu

2.1. Opis problemu

Zarządzanie strukturami danych, takimi jak drzewa binarne wyszukiwania (BST), jest kluczowym elementem wielu aplikacji informatycznych. BST pozwala na efektywne wykonywanie operacji, takich jak wstawianie, usuwanie i wyszukiwanie elementów, zachowując złożoność czasową $O(\log n)$ w przypadku zrównoważonego drzewa. Jednakże, istnieje wiele wyzwań związanych z zarządzaniem i przechowywaniem takiej struktury, w tym:

- Obsługa dynamicznych zmian w strukturze drzewa.
- Efektywne przechowywanie i odczyt drzewa z plików w celu zachowania danych pomiędzy sesjami.
- Zapewnienie intuicyjnej interakcji użytkownika z systemem.
- Rozwiązywanie potencjalnych problemów z niezrównoważonymi drzewami, które mogą prowadzić do degradacji wydajności.

2.2. Cele analizy

Celem analizy jest:

1. Określenie wymagań funkcjonalnych i niefunkcjonalnych systemu.
2. Zidentyfikowanie potencjalnych problemów, które mogą wystąpić podczas implementacji.
3. Zdefiniowanie struktury programu oraz powiązań między modułami (np. zarządzanie drzewem i obsługa plików).
4. Opracowanie strategii testowania, aby zapewnić poprawne działanie wszystkich funkcji.

2.3. Kluczowe wyzwania

Podczas projektowania i implementacji systemu mogą wystąpić następujące problemy:

- **Problem ze zrównoważeniem drzewa:** Drzewo BST może stać się niezrównoważone, co spowoduje wzrost złożoności operacji do $O(n)$.

- **Efektywność zapisu/odczytu z pliku:** Serializacja i deserializacja drzewa muszą być zgodne z wymaganiami dotyczącymi rozmiaru pliku oraz szybkości operacji.
- **Obsługa błędów:** System musi obsługiwać błędy, takie jak brak pliku do odczytu, próba wczytania uszkodzonego pliku lub wprowadzanie danych niezgodnych z wymaganiami.

2.4. Podsumowanie

Analiza problemu pokazuje, że kluczowym celem jest stworzenie systemu, który będzie efektywnie zarządzał strukturą drzewa BST, jednocześnie zapewniając prostotę użytkowania i bezpieczeństwo danych. Należy również uwzględnić potencjalne ograniczenia techniczne i opracować rozwiązania pozwalające na ich eliminację.

3. Projektowanie

3.1. Wprowadzenie

W niniejszym dokumencie opisano narzędzia i techniki, które zostaną wykorzystane w projekcie C++. Projekt będzie realizowany w języku C++ i z wykorzystaniem nowoczesnych narzędzi programistycznych.

3.2. Wykorzystane narzędzia

W projekcie zastosowane zostaną następujące narzędzia:

- **Język C++:** To język programowania wysokiego poziomu, który jest powszechnie używany w aplikacjach systemowych, gier oraz w programowaniu aplikacji wieloplatformowych.
- **Visual Studio 2022:** Zintegrowane środowisko programistyczne (IDE) od Microsoft, które ułatwia rozwijanie aplikacji w języku C++. Oferuje wiele narzędzi do debugowania, testowania oraz zarządzania projektami.
- **Git:** System kontroli wersji, który umożliwia śledzenie zmian w kodzie źródłowym. Umożliwia współpracę w zespołach oraz efektywne zarządzanie wersjami kodu.
- **GitHub:** Platforma oparta na systemie Git, która umożliwia hostowanie kodu, współpracę nad projektami oraz zarządzanie zadaniami.
- **Doxygen:** Narzędzie do generowania dokumentacji z kodu źródłowego. Umożliwia tworzenie dokumentacji w różnych formatach, takich jak HTML, PDF czy LaTeX, na podstawie specjalnych komentarzy w kodzie.
- **Overleaf:** Aplikacja webowa do edytowania dokumentów w LaTeX. Umożliwia łatwe współdzielenie dokumentów oraz ich wspólną edycję w czasie rzeczywistym.

3.3. Sposób użycia narzędzi

3.4. Git

Git to system kontroli wersji, który umożliwia zarządzanie historią zmian w kodzie źródłowym. Aby efektywnie korzystać z Git, wykonaj następujące kroki:

- **Inicjalizacja repozytorium:** Użyj komendy `git init` w katalogu projektu, aby utworzyć nowe repozytorium.
- **Dodawanie plików:** Użyj `git add .`, aby dodać wszystkie pliki do repozytorium.
- **Tworzenie commitów:** Użyj `git commit -m "Opis zmian"`, aby zapisać zmiany z odpowiednim komentarzem.
- **Tworzenie gałęzi:** Użyj `git branch nazwa_gałęzi`, aby utworzyć nową gałąź.
- **Przełączanie gałęzi:** Użyj `git checkout nazwa_gałęzi`, aby przełączyć się na inną gałąź.
- **Wysyłanie zmian na GitHub:** Użyj `git push origin nazwa_gałęzi`, aby wysłać zmiany na zdalne repozytorium.

3.5. Doxygen

Doxygen jest narzędziem do generowania dokumentacji z kodu źródłowego. Aby z niego skorzystać:

- **Instalacja:** Zainstaluj Doxygen na swoim systemie.
- **Konfiguracja:** Utwórz plik konfiguracyjny `Doxyfile` przy użyciu polecenia `doxygen -g`.
- **Dodawanie komentarzy:** Używaj specjalnych komentarzy w kodzie źródłowym, aby opisać funkcje, klasy i zmienne. Przykład:

```
/**
 * @brief Funkcja dodaje dwie liczby.
 * @param a Pierwsza liczba.
 * @param b Druga liczba.
 * @return Suma a i b.
 */
int add(int a, int b) {
    return a + b;
}
```

- **Generowanie dokumentacji:** Użyj polecenia `doxygen Doxyfile`, aby wygenerować dokumentację w formacie HTML lub PDF.

3.6. Overleaf

Overleaf to platforma do edytowania dokumentów w LaTeX. Aby z niej skorzystać:

- **Rejestracja:** Załóż konto na stronie Overleaf.
- **Tworzenie projektu:** Utwórz nowy projekt i wybierz szablon dokumentu.
- **Edycja:** Edytuj dokument w edytorze tekstu. Zmiany są automatycznie zapisywane.
- **Współpraca:** Zaproś innych użytkowników do współpracy, udostępniając link do projektu.
- **Eksport:** Eksportuj dokument do formatu PDF lub innego formatu, korzystając z opcji eksportu.

4. Implementacja

4.1. Wprowadzenie

W niniejszym dokumencie przedstawiono implementację algorytmu listy dwukierunkowej w języku C++. Opisano ciekawe fragmenty kodu oraz wyniki działania programu.

4.2. Implementacja algorytmu

Implementacja listy dwukierunkowej składa się z dwóch głównych części: struktury węzła (`Node`) oraz klasy listy dwukierunkowej (`DubleLinkedList`).

4.3. Drzewo binarne wyszukiwania (BST)

Ten kod implementuje klasę `BST`, która obsługuje podstawowe operacje na drzewie binarnym wyszukiwania, takie jak wstawianie, usuwanie, wyszukiwanie ścieżki oraz przechodzenie i zapis/ładowanie drzewa z/do pliku. Struktura `Node` reprezentuje węzeł z danymi i wskaźnikami na lewe i prawe poddrzewo.

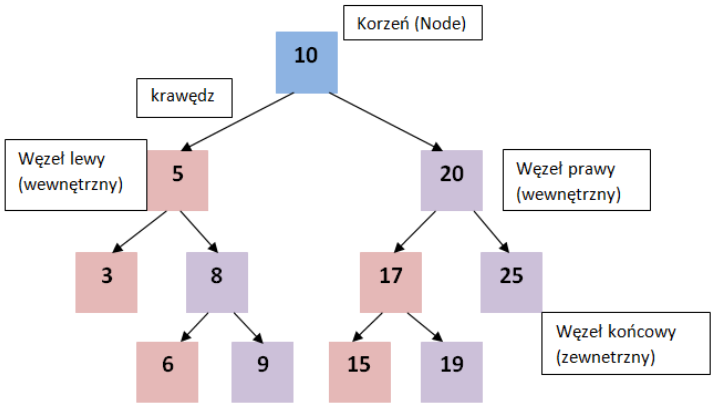
```
1  class BST {
2  private:
3      struct Node {
4          int data;
5          Node* left;
6          Node* right;
7
8          Node(int value) : data(value), left(nullptr), right(nullptr
       ) {}
9      };
10
11     Node* root;
12
13     void insert(Node*& node, int value);
14     bool remove(Node*& node, int value);
15     void clear(Node* node);
16     bool findPath(Node* node, int value, std::vector<int>& path);
17     void printPreOrder(Node* node);
18     void printInOrder(Node* node);
19     void printPostOrder(Node* node);
20     void saveToFile(Node* node, std::ofstream& outFile) const;
21     void loadFromFile(Node*& node, std::ifstream& inFile);
22
```

```
23 public:
24     BST();
25     ~BST();
```

Listing 1. Fragment kodu - Wyszukiwania (BST)

4.4. Metody klasu BST

4.4.1. Drzewo BST



Rys. 4.1. DrzewoBST

4.4.2. Dodawanie elementu do drzewa

Dodanie elementu do drzewa binarnego wyszukiwania polega na porównaniu wartości z korzeniem drzewa i wstawieniu nowego węzła w odpowiednie miejsce (lewe lub prawe poddrzewo).

Metoda `insert` realizuje to zadanie. Przeszukuje drzewo, aby znaleźć odpowiednie miejsce na nowy element.

```
1  void BST::insert(int value) {
2      insert(root, value); // Rekursywne wywolanie insert
3  }
4
5  void BST::insert(Node*& node, int value) {
6      if (node == nullptr) {
7          node = new Node(value);
8      }
9      else if (value < node->data) {
10         insert(node->left, value);  // Wstawienie do lewego
       poddrzewa
```

```
11      }
12      else {
13          insert(node->right, value);  // Wstawienie do prawego
    poddrzewa
14      }
15  }
```

Listing 2. Fragment kodu - Dodawanie elementu do drzewa

4.4.3. Usuwanie elementu z drzewa

Aby usunąć element z drzewa, należy znaleźć odpowiedni węzeł i usunąć go, zachowując właściwości drzewa. Usuwanie jest bardziej skomplikowane, gdy węzeł ma dwóch potomków, ponieważ trzeba znaleźć odpowiedni węzeł do zastąpienia.

```
1   bool BST::remove(int value) {
2       return remove(root, value);  // Wywolanie rekurencyjnej wersji
    metody remove
3   }
4
5   bool BST::remove(Node*& node, int value) {
6       if (node == nullptr) {
7           return false;
8       }
9
10      if (value < node->data) {
11          return remove(node->left, value);  // Usuwanie z lewego
    poddrzewa
12      }
13      else if (value > node->data) {
14          return remove(node->right, value); // Usuwanie z prawego
    poddrzewa
15      }
16      else {
17          if (node->left == nullptr) {
18              Node* temp = node;
19              node = node->right;
20              delete temp;
21          }
22          else if (node->right == nullptr) {
23              Node* temp = node;
24              node = node->left;
25              delete temp;
26          }
27          else {
```

```
28              Node* temp = node->right;
29              while (temp && temp->left) temp = temp->left;  //
    Znalezienie najmniejszego elementu w prawym poddrzewie
30              node->data = temp->data;  // Zastapienie danymi
    najmniejszego elementu
31              return remove(node->right, temp->data);  // Usuwanie
    elementu w prawym poddrzewie
32          }
33      }
34      return true;
35  }
```

Listing 3. Fragment kodu - Usuwanie elementu z drzewa

4.5. Metoda do czyszczenia drzewa

Metoda `clear` usuwa wszystkie węzły drzewa, rekurencyjnie usuwając lewą i prawą gałąź dla każdego węzła.

```
1  void BST::clear(Node* node) {
2      if (node) {
3          clear(node->left);  // Usuwanie lewego poddrzewa
4          clear(node->right); // Usuwanie prawego poddrzewa
5          delete node;        // Usuwanie wezla
6      }
7  }
8
9  void BST::clear() {
10     clear(root);  // Usuwanie wszystkich wezluw
11     root = nullptr;  // Ustawienie korzenia na null
12 }
```

Listing 4. Fragment kodu - Metoda do czyszczenia drzewa

4.6. Wypisywanie drzewa

Drzewo można wypisać w trzech różnych porządkach: preorder (przedwęzłowy), inorder (środkowy) i postorder (potemwęzłowy). Dla każdego porządku wywoływana jest odpowiednia metoda rekurencyjna.

```
1  void BST::printPreOrder(Node* node) {
2      if (node) {
3          std::cout << node->data << " ";
4          printPreOrder(node->left);  // Wypisanie lewego poddrzewa
```

```
        printPreOrder(node->right); // Wypisanie prawego poddrzewa
    }
}

void BST::printInOrder(Node* node) {
    if (node) {
        printInOrder(node->left);  // Wypisanie lewego poddrzewa
        std::cout << node->data << " ";
        printInOrder(node->right); // Wypisanie prawego poddrzewa
    }
}

void BST::printPostOrder(Node* node) {
    if (node) {
        printPostOrder(node->left);  // Wypisanie lewego poddrzewa
        printPostOrder(node->right); // Wypisanie prawego poddrzewa
        std::cout << node->data << " ";
    }
}
```

Listing 5. Fragment kodu - Wypisywanie drzewa

4.7. Zapis i wczytywanie drzewa z pliku

Drzewo może być zapisane do pliku binarnego, a także wczytane z pliku. Dane są zapisywane rekurencyjnie, a brakujące węzły są oznaczone specjalnym markerem (-1).

```
void BST::saveToFile(Node* node, std::ofstream& outFile) const {
    if (node) {
        outFile.write(reinterpret_cast<const char*>(&node->data),
    sizeof(node->data));
        saveToFile(node->left, outFile);  // Zapisanie lewego
    poddrzewa
        saveToFile(node->right, outFile); // Zapisanie prawego
    poddrzewa
    }
    else {
        int nullMarker = -1;
        outFile.write(reinterpret_cast<const char*>(&nullMarker),
    sizeof(nullMarker));
    }
}

void BST::saveToFile(const std::string& filename) const {
```

```
14      std::ofstream outFile(filename, std::ios::binary);
15      if (!outFile) {
16          throw std::ios_base::failure("Failed to open file for
    writing");
17      }
18      saveToFile(root, outFile);  // Zapisanie drzewa do pliku
19  }
```

Listing 6. Fragment kodu - Zapis i wczytywanie drzewa z pliku

4.8. Obsługa plików: FileManager.h

Klasa `FileManager` jest odpowiedzialna za obsługę zapisu i odczytu drzewa do/z plików binarnych.

```
1  #ifndef FILE_MANAGER_H
2  #define FILE_MANAGER_H
3
4  #include "BST.h"
5  #include <string>
6
7  class FileManager {
8  public:
9      static void saveTreeToBinaryFile(const BST& tree, const std::
    string& filename);
10     static void loadTreeFromBinaryFile(BST& tree, const std::string
    & filename);
11 };
12
13 #endif
```

Listing 7. Fragment kodu - obsługa plików: FileManager.h

4.9. Implementacja metod: FileManager.cpp

Poniżej przedstawiono implementację metod klasy `FileManager`, które zarządzają zapisem i odczytem plików.

```
1  #include "FileManager.h"
2  #include <fstream>
3
4  void FileManager::saveTreeToBinaryFile(const BST& tree, const std::
    string& filename) {
5      tree.saveToFile(filename);
6  }
```

```
7
8  void FileManager::loadTreeFromBinaryFile(BST& tree, const std::
       string& filename) {
9      tree.loadFromFile(filename);
10 }
```

Listing 8. Fragment kodu - Implementacja metod: FileManager.cpp

Funkcje `saveTreeToBinaryFile()` oraz `loadTreeFromBinaryFile()` bezpośrednio korzystają z metod zapisujących i odczytujących strukturę drzewa w klasie `BST`.

4.10. Funkcja główna programu: `main.cpp`

Główna funkcja programu obsługuje interakcję użytkownika za pomocą menu konsolowego. Poniżej znajduje się pełna implementacja pliku `main.cpp`:

```
1  #include <iostream>
2  #include "BST.h"
3  #include "FileManager.h"
4
5  void menu() {
6      std::cout << "1. Insert Element\n";
7      std::cout << "2. Remove Element\n";
8      std::cout << "3. Print Tree\n";
9      std::cout << "4. Save Tree to File\n";
10     std::cout << "5. Load Tree from File\n";
11     std::cout << "6. Exit\n";
12 }
13
14 int main() {
15     BST tree;
16     int choice;
17
18     while (true) {
19         menu();
20         std::cin >> choice;
21
22         if (choice == 1) {
23             int value;
24             std::cout << "Enter value to insert: ";
25             std::cin >> value;
26             tree.insert(value);
27         }
28         else if (choice == 2) {
29             int value;
```

```
30              std::cout << "Enter value to remove: ";
31              std::cin >> value;
32              tree.remove(value);
33          }
34          else if (choice == 3) {
35              std::cout << "Choose traversal order (1- Preorder, 2-
    Inorder, 3- Postorder): ";
36              int order;
37              std::cin >> order;
38              tree.printTree(order);
39          }
40          else if (choice == 4) {
41              std::string filename;
42              std::cout << "Enter filename to save: ";
43              std::cin >> filename;
44              tree.saveToFile(filename);
45          }
46          else if (choice == 5) {
47              std::string filename;
48              std::cout << "Enter filename to load: ";
49              std::cin >> filename;
50              tree.loadFromFile(filename);
51          }
52          else if (choice == 6) {
53              break;
54          }
55      }
56
57      return 0;
58  }
```

Listing 9. Fragment kodu - Funkcja główna

Funkcja `main` pozwala użytkownikowi na wykonanie następujących operacji:

- Dodawanie elementów do drzewa.
- Usuwanie elementów z drzewa.
- Wyświetlanie drzewa w trzech różnych porządkach (preorder, inorder, postorder).
- Zapis drzewa do pliku binarnego.
- Odczyt drzewa z pliku binarnego.

5. Podsumowanie

Przedstawiona implementacja programu ilustruje kompleksowe rozwiązanie do zarządzania drzewem binarnym (BST).

Program składa się z następujących modułów:

- `BST.cpp` i `BST.h` — implementacja drzewa binarnego wraz z operacjami dodawania, usuwania, wyświetlania w różnych porządkach (preorder, inorder, postorder) oraz obsługą zapisu i odczytu drzewa do pliku binarnego.
- `FileManager.cpp` i `FileManager.h` — moduł odpowiedzialny za obsługę operacji zapisu i odczytu drzewa z plików binarnych, współpracujący z modułem `BST`.
- `main.cpp` — główna funkcja programu, która integruje wszystkie moduły i umożliwia interakcję użytkownika poprzez menu tekstowe.

Funkcja główna (`main.cpp`) demonstruje, jak poszczególne moduły współpracują ze sobą w praktycznym zastosowaniu. Użytkownik końcowy ma możliwość:

- Dodawania i usuwania elementów w drzewie.
- Wyświetlania drzewa w różnych porządkach.
- Zapisywania i odczytu drzewa w formacie binarnym.

Struktura programu została zaprojektowana w sposób modułowy, co umożliwia łatwe rozszerzanie jego funkcjonalności. Możliwe przyszłe kierunki rozwoju to dodanie bardziej zaawansowanych algorytmów przetwarzania drzewa (np. balansowanie lub wizualizacja).

6. Wnioski

6.1. Wnioski Końcowe

W implementacji drzewa binarnego wyszukiwania (BST) zaprezentowaliśmy kluczowe operacje związane z manipulacją danymi w drzewie, takie jak dodawanie, usuwanie i wyszukiwanie elementów. Implementacja została wykonana z wykorzystaniem rekurencji, co jest typowe dla struktur danych opartych na drzewach, zapewniając czytelność i łatwość w zarządzaniu węzłami.

Wnioski końcowe dotyczące projektu implementacji drzewa BST:

- **Złożoność czasowa:** Wstawianie i usuwanie elementów w średnim przypadku mają złożoność $O(\log n)$, gdzie n to liczba węzłów w drzewie. Jednak w najgorszym przypadku (np. w przypadku drzewa zdegenerowanego do listy) złożoność może wynieść $O(n)$.Wyszukiwanie elementów także ma złożoność $O(\log n)$ w średnim przypadku.

- **Rekurencyjność:** Wszelkie operacje na drzewie, takie jak wstawianie, usuwanie, czyszczenie oraz wypisywanie, zostały zaimplementowane w sposób rekurencyjny. Takie podejście jest eleganckie i zapewnia prostą implementację, jednak dla bardzo dużych drzew może prowadzić do problemów związanych z limitem głębokości stosu.

- **Zapis i odczyt z pliku:** Drzewo zostało również zaimplementowane z możliwością zapisu do pliku i odczytu z niego. To pozwala na trwałe przechowywanie danych i łatwą rekonstruację drzewa. W implementacji użyto formatu binarnego, co umożliwia szybki zapis i odczyt, ale wymaga odpowiedniego zarządzania błędami.

- **Zastosowanie:** Drzewo binarne wyszukiwania jest jedną z najważniejszych struktur danych w informatyce, szczególnie w kontekście algorytmów wyszukiwania i sortowania. Implementacja BST może znaleźć szerokie zastosowanie w różnych dziedzinach, takich jak zarządzanie danymi, baza danych, czy struktury indeksujące.

- **Możliwości rozwoju:** Można rozważyć rozbudowę implementacji o funkcje takie jak balansowanie drzewa, aby poprawić jego wydajność w przypadku nieoptymalnych danych wejściowych. Można także dodać funkcje umożliwiające interakcję z użytkownikiem w czasie rzeczywistym, np. interfejs graficzny do wizualizacji drzewa.

Podsumowując, implementacja drzewa binarnego wyszukiwania jest efektywnym narzędziem do przechowywania i zarządzania danymi w sposób posortowany, a także jest podstawą wielu zaawansowanych algorytmów i struktur danych. Zastosowanie rekurencji sprawia, że kod jest czytelny i elegancki, ale warto rozważyć rozszerzenia mające na celu zwiększenie jego wydajności w przypadku dużych zbiorów danych.

Spis rysunków

Spis tabel

Spis listingów