AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

# DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

## ...Algorytm listy dwukierunkowej z zastosowaniem GitHub...

Autor:
Jakub Marek Tokarczyk

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

# Spis treści

# 1. Ogólne określenie wymagań

## 1.1. Celem projektu jest:

- Kluczowym celem w projekcie jest gruntowne zrozumienie koncepcji drzewa BST oraz zasad jego działania. Drzewa BST są jedną z fundamentalnych struktur danych w informatyce, wykorzystywanych w wielu aplikacjach do przechowywania i efektywnego przetwarzania danych. W trakcie pracy nad projektem zapoznaliśmy się z podstawowymi operacjami wykonywanymi na drzewach BST, takimi jak:

    1. Dodawanie elementów, czyli wstawianie nowych wartości zgodnie z regułami BST, które gwarantują porządek w strukturze drzewa,
    2. Usuwanie elementów, uwzględniające różne przypadki, takie jak usuwanie liścia, węzła z jednym dzieckiem lub węzła z dwójką dzieci,
    3. Wyszukiwanie elementów, umożliwiające szybkie odnalezienie danych w drzewie dzięki jego uporządkowanej strukturze,
    4. Wyświetlanie elementów, obejmujące różne metody, takie jak przejścia w porządku preorder, inorder czy postorder, co pozwala na uzyskanie różnych reprezentacji danych,

- Projekt stanowi doskonałą okazję do nauki pracy zespołowej oraz wykorzystywania narzędzi takich jak GitHub. Platforma ta odgrywa kluczową rolę w nowoczesnym zarządzaniu kodem źródłowym, szczególnie w środowiskach zespołowych. W trakcie realizacji projektu nauczyliśmy się korzystać z funkcji GitHuba, takich jak:

    1. Tworzenie gałęzi, umożliwiające równoległe rozwijanie różnych funkcji programu przez członków zespołu.
    2. Wykonywanie commitów, pozwalających na zapisywanie zmian w sposób czytelny i uporządkowany.
    3. Rozwiązywanie konfliktów w kodzie, stanowiących nieunikniony element pracy zespołowej. Nauka efektywnego radzenia sobie z konfliktami w trakcie scalania kodu to cenne doświadczenie, które będzie miało zastosowanie w przyszłych projektach.

- Jednym z istotnych elementów programu, który tworzymy, jest możliwość zarządzania danymi drzewa BST za pomocą operacji na plikach czyli:

1. Zapisywanie drzewa do pliku binarnego, co pozwala na długoterminowe przechowywanie danych w formacie zoptymalizowanym pod kątem efektywności i bezpieczeństwa,
2. Odczytywanie drzewa z pliku binarnego, co umożliwia szybkie przywrócenie zapisanych wcześniej danych do programu,
3. Wczytywanie danych z pliku tekstowego, co upraszcza proces inicjalizacji drzewa na podstawie zewnętrznych danych. Taka funkcjonalność jest szczególnie przydatna w przypadku konieczności pracy z dużymi zestawami danych lub importu danych z innych źródeł,

# 2. Analiza problemu

Algorytm drzewa BST (Binary Search Tree) jest używany w różnych dziedzinach informatyki i programowania, gdzie konieczne jest efektywne przechowywanie i zarządzanie danymi. Oto niektóre zastosowania drzewa BST:

- Wyszukiwanie: Drzewa BST są jedną z najbardziej popularnych struktur danych stosowanych w operacjach wyszukiwania. Dzięki uporządkowanej strukturze, w której elementy mniejsze od węzła znajdują się po jego lewej stronie, a większe po prawej, możliwe jest znaczne przyspieszenie wyszukiwania. Przykład zastosowania: Wyszukiwanie użytkownika w systemie rejestracji lub odnajdywanie określonego rekordu w bazie danych,

- Algorytmy sortowania: Drzewa BST mogą być używane do sortowania danych. Jednym z najbardziej efektywnych sposobów sortowania przy użyciu drzewa BST jest przechodzenie drzewa w porządku in-order. W tej technice odwiedza się węzły w sposób rosnący, co automatycznie generuje posortowaną listę elementów. Przykład zastosowania: generowanie posortowanych list rekordów w aplikacjach księgowych lub raportach sprzedażowych,

- Operacje na zbiorach Drzewa BST znajdują zastosowanie w operacjach na zbiorach danych, Przykład zastosowania: Systemy rekomendacji, gdzie analizowane są wspólne preferencje użytkowników lub różnice w zestawach danych,

- Drzewa BST są szeroko wykorzystywane w systemach zarządzania bazami danych. Umożliwiają one efektywne przechowywanie oraz indeksowanie danych, co przyspiesza operacje, takie jak wyszukiwanie rekordów, ich aktualizacja czy usuwanie. Choć klasyczne drzewa BST zostały w dużej mierze zastąpione bardziej zaawansowanymi strukturami, takimi jak drzewa B lub B+, ich zasady nadal stanowią podstawę działania. Przykład zastosowania: Indeksowanie danych w bazach relacyjnych, takich jak MySQL lub PostgreSQL,

- Drzewa BST mogą być stosowane w algorytmach wyszukiwania wzorców w tekście. Dzięki ich strukturze możliwe jest szybkie odnajdywanie słów kluczowych, fraz lub nawet liter w długich ciągach tekstowych. Przykład zastosowania: Systemy sprawdzania pisowni, wyszukiwarki internetowe lub narzędzia analizy tekstu.

- Drzewa BST są używane w problemach optymalizacyjnych, takich jak optymalizacja drzewa wyrażeń, gdzie dąży się do minimalizacji kosztów obliczeń

dla określonych operacji. Przykład zastosowania: Optymalizacja wyrażeń arytmetycznych w kompilatorach, gdzie analiza drzewa pomaga w minimalizacji liczby wykonywanych operacji,

- Drzewa BST są uniwersalną strukturą danych, którą często wykorzystuje się w programowaniu do przechowywania informacji i rozwiązywania różnorodnych problemów związanych z wyszukiwaniem, sortowaniem oraz zarządzaniem danymi. Przykład zastosowania: Systemy zarządzania plikami, systemy rezerwacji biletów czy zarządzanie sesjami użytkowników w aplikacjach internetowych.

## 2.1. Sposób działania Drzewa BST.

Drzewo BST (Binary Search Tree) to struktura danych oparta na zasadzie porównywania elementów i organizowania ich w hierarchiczną strukturę drzewa. Każdy węzeł w drzewie BST zawiera maksymalnie dwóch potomków: lewego i prawego. Kluczowa zasada działania BST polega na tym, że:
wartości mniejsze od wartości węzła nadrzędnego są umieszczane w lewym poddrzewie a wartości większe od wartości węzła nadrzędnego są umieszczane w prawym poddrzewie.

1. Wstawianie elementów:
    - Aby wstawić element do drzewa BST, rozpoczynamy od korzenia drzewa (pierwszego węzła),
    - Porównujemy element, który chcemy wstawić, z elementem w bieżącym węźle,
    - Jeśli element jest mniejszy, przechodzimy do lewego poddrzewa; jeśli większy, przechodzimy do prawego poddrzewa,
    - Kontynuujemy to porównywanie i przesuwanie się w dół drzewa, aż znajdziemy odpowiednie miejsce do wstawienia elementu jako nowy węzeł liściasty.
2. Wyszukiwanie elementów:
    - Aby znaleźć element w drzewie BST, rozpoczynamy od korzenia drzewa i porównujemy poszukiwany element z elementem w bieżącym węźle,
    - Jeśli są one równe, znaleźliśmy element,
    - Jeśli poszukiwany element jest mniejszy, przechodzimy do lewego poddrzewa; jeśli większy, przechodzimy do prawego poddrzewa,

- Kontynuujemy to porównywanie i przesuwanie się w dół drzewa, aż znaleźliśmy element lub doszliśmy do liścia, co oznacza, że elementu w drzewie nie ma.

3. Usuwanie elementów:

   - Jeśli usuwany węzeł ma:
     (a) Zero potomków: można go po prostu usunąć,
     (b) Jednego potomka: można podmienić go na swojego potomka,
     (c) Dwoje potomków: można znaleźć następnika lub poprzednika (czyli węzeł, który jest albo najmniejszym elementem w prawym poddrzewie albo największym elementem w lewym poddrzewie) i podmienić usuwany węzeł na niego. Następnie usuwany węzeł zostaje usunięty.

4. Przechodzenie drzewa:

   - Istnieją różne sposoby przechodzenia drzewa BST, takie jak In-order (lewy, korzeń, prawy), Pre-order (korzeń, lewy, prawy) i Post-order (lewy, prawy, korzeń). Każdy z tych sposobów pozwala na przeglądanie elementów w określonej kolejności

5. Operacje na drzewie BST

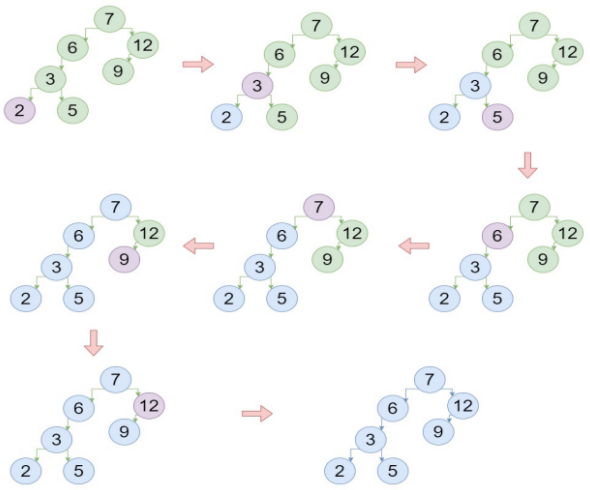
   - Możemy wykonywać różne operacje na drzewie BST, takie jak znajdowanie minimum i maksimum, obliczanie wysokości drzewa, sprawdzanie, czy drzewo jest zrównoważone, a także operacje na zbiorach, takie jak unia, przecięcie i różnica zbiorów

## 2.2. Przykłady przeszukiwania drzewa:

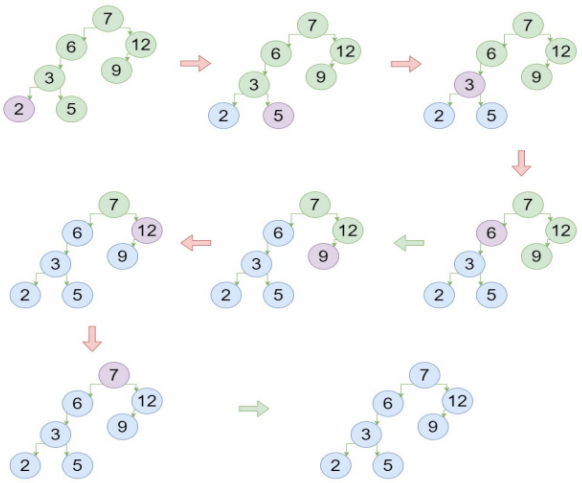
In-order (lewy, korzeń, prawy): Odwiedza najpierw lewego potomka, następnie węzeł główny (korzeń), a na końcu prawego potomka.

Pre-order (korzeń, lewy, prawy): Zaczyna od odwiedzenia węzła głównego (korzenia), następnie przechodzi do lewego poddrzewa, a na końcu do prawego poddrzewa

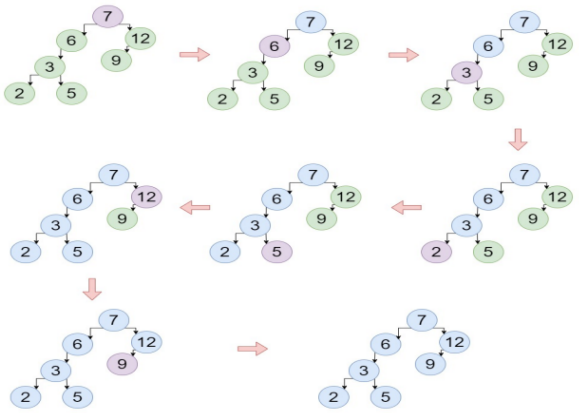
Post-order (lewy, prawy, korzeń): Odwiedza węzły w kolejności: najpierw lewy potomek, potem prawy, a na końcu węzeł główny (korzeń).



**Rys. 2.1.** in-order



**Rys. 2.2.** post-order



**Rys. 2.3.** pre-order

# 3. Projektowanie

W projekcie będę korzystać z następujących narzędzi:

- Visual Studio 2022 - zaawansowane i rozbudowane środowisko programistyczne (IDE) opracowane przez firmę Microsoft, które jest idealne do tworzenia aplikacji w różnych językach programowania, w tym C++. Oferuje ono intuicyjny interfejs, który wspiera programistów na każdym etapie tworzenia oprogramowania. W kontekście pracy nad projektem drzewa BST, Visual Studio 2022 udostępnia szereg narzędzi, takich jak inteligentne podpowiedzi kodu, które pomagają unikać błędów składniowych, debuger umożliwiający analizę działania programu w czasie rzeczywistym oraz funkcję zarządzania rozwiązaniami, która pozwala na łatwą organizację plików projektu. Ponadto Visual Studio oferuje integrację z systemami kontroli wersji, takimi jak Git, co umożliwia płynną współpracę z platformą GitHub.

- C++ - wszechstronny język programowania, który wyróżnia się wysoką wydajnością i możliwością operowania na niskim poziomie systemu. W przypadku projektu drzewa BST, C++ oferuje potężne możliwości zarządzania dynamiczną pamięcią i strukturami danych, co jest kluczowe w implementacji takich struktur jak drzewa binarne. Język ten umożliwia także łatwą organizację kodu w postaci klas i metod, co wspiera podejście obiektowe w programowaniu. Dzięki swojej elastyczności, C++ pozwala na bezpośrednie zarządzanie wskaźnikami, co jest istotne podczas dynamicznego przydzielania i zwalniania pamięci w projekcie.

- GitHub - platforma hostingowa dla systemu kontroli wersji Git, która umożliwia programistom zarządzanie i współdzielenie kodu źródłowego. W pracy nad projektem GitHub odgrywa kluczową rolę, umożliwiając tworzenie repozytoriów, które przechowują historię zmian kodu, a także pozwalając na współpracę z innymi programistami. Wykorzystanie GitHuba w projekcie drzewa BST zapewnia bezpieczeństwo kodu dzięki tworzeniu kopii zapasowych oraz ułatwia śledzenie postępów i zmian w kodzie. Dzięki funkcjom takim jak pull requests czy code reviews, zespoły mogą omawiać wprowadzone zmiany i dążyć do jak najlepszej jakości oprogramowania.

# 4. Implementacja

```
void BST::insert(Node*& node, int value) {
    if (!node) {
        node = new Node(value);
        return;
    }
    if (value < node->data)
        insert(node->left, value);
    else
        insert(node->right, value);
}

void BST::insert(int value) {
    insert(root, value);
}
```

**Listing 1.** Dodawanie elementu drzewa

Funkcja insert umożliwia dodawanie elementów do drzewa. Implementowana jest rekurencyjnie, gdzie porównywana jest wartość węzła z wartością do wstawienia. Jeśli nowa wartość jest mniejsza, dodawana jest do lewego poddrzewa, w przeciwnym razie do prawego. Funkcja ta działa na zasadzie porównań, dodając elementy do odpowiednich miejsc w drzewie, co pozwala na utrzymanie struktury drzewa binarnego.

```
BST::Node* BST::remove(Node* node, int value) {
    if (!node) return nullptr;

    if (value < node->data) {
        node->left = remove(node->left, value);
    }
    else if (value > node->data) {
        node->right = remove(node->right, value);
    }
    else {
        if (!node->left) {
            Node* rightChild = node->right;
            delete node;
            return rightChild;
        }
        else if (!node->right) {
            Node* leftChild = node->left;
            delete node;
            return leftChild;
```

```
        }
        else {
            Node* successor = node->right;
            while (successor->left)
                successor = successor->left;
            node->data = successor->data;
            node->right = remove(node->right, successor->data);
        }
    }
    return node;
}

void BST::remove(int value) {
    root = remove(root, value);
}
```

**Listing 2.** Usuwanie elemetu z drzewa

Funkcja remove odpowiedzialna jest za usuwanie węzłów. Działa również rekurencyjnie, porównując wartość, którą chcemy usunąć, z wartościami w drzewie. Jeżeli węzeł do usunięcia ma jedno dziecko, to jego miejsce zajmuje dziecko. Jeżeli węzeł ma dwoje dzieci, należy znaleźć jego następnika (najmniejszy element w prawym poddrzewie), skopiować jego wartość do usuwanego węzła, a następnie usunąć następnika.

```
void BST::inorder(Node* node, vector<int>& result) const {
    if (!node) return;
    inorder(node->left, result);
    result.push_back(node->data);
    inorder(node->right, result);
}

void BST::preorder(Node* node, vector<int>& result) const {
    if (!node) return;
    result.push_back(node->data);
    preorder(node->left, result);
    preorder(node->right, result);
}

void BST::postorder(Node* node, vector<int>& result) const {
    if (!node) return;
    postorder(node->left, result);
    postorder(node->right, result);
    result.push_back(node->data);
```

```
}
```

**Listing 3.** Wyświetlanie posortowanych drzew

Wyświetlanie drzewa odbywa się w trzech różnych porządkach: in-order, pre-order i post-order. Funkcje inorder, preorder i postorder są odpowiedzialne za przejście przez drzewo w określonej kolejności i zapisanie wyników do wektora. W przypadku porządku in-order, drzewo jest przeszukiwanie w sposób lewy-poddrzewo, węzeł, prawy-poddrzewo, co pozwala na uzyskanie posortowanych danych. Porządek pre-order to odwiedzanie węzła przed jego dziećmi, a post-order odwiedza dzieci przed węzłem.

```
void BST::removeTree(Node*& node) {
    if (!node) return;
    removeTree(node->left);
    removeTree(node->right);
    delete node;
    node = nullptr;
}

void BST::removeTree() {
    removeTree(root);
}
```

**Listing 4.** Usuwanie całego drzewa

Usuwanie całego drzewa realizowane jest przez funkcję removeTree. Działa ona rekurencyjnie, usuwając wszystkie węzły, zaczynając od liści i kończąc na korzeniu. Po usunięciu wszystkich węzłów, wskaźnik do korzenia jest ustawiany na nullptr, co oznacza, że drzewo zostało całkowicie usunięte.

```
void BST::saveToFile(const string& filename, int orderType) const {
    vector<int> result;
    if (orderType == 1) {
        inorder(root, result);
    }
    else if (orderType == 2) {
        preorder(root, result);
    }
    else if (orderType == 3) {
        postorder(root, result);
    }
    else {
        cerr << "Invalid order type" << endl;
        return;
    }
```

```
16
17      ofstream file(filename);
18      if (!file.is_open()) {
19          cerr << "Could not open file for writing" << endl;
20          return;
21      }
22
23      for (int value : result)
24          file << value << " ";
25      file.close();
26  }
```

**Listing 5.** Zapis drzewa do pliku

Zapis drzewa do pliku realizowany jest przez funkcję saveToFile. Funkcja ta zapisuje dane z drzewa do pliku tekstowego w jednym z trzech porządków, które użytkownik może wybrać (in-order, pre-order lub post-order). Dane zapisane w pliku są rozdzielone spacjami, co pozwala na ich późniejsze odczytanie i wykorzystanie w innym czasie.

# 5. Wnioski

Projekt drzewa BST umożliwił praktyczne zrozumienie i zastosowanie kluczowych koncepcji programowania obiektowego, zarządzania pamięcią oraz struktury danych. Implementacja drzewa BST wymagała zaprojektowania algorytmów umożliwiających efektywne wykonywanie operacji takich jak wstawianie, usuwanie, przeszukiwanie oraz wyświetlanie danych w różnych porządkach (in-order, pre-order, post-order). Dzięki temu projektowi można wyciągnąć następujące wnioski:

- Drzewo BST jest wydajną strukturą danych, która pozwala na przechowywanie danych w sposób hierarchiczny, umożliwiając szybki dostęp i operacje na danych. Poprawne zastosowanie zasad BST (np. przechowywanie mniejszych elementów po lewej stronie, większych po prawej) było kluczowe dla funkcjonowania drzewa i realizacji wymaganych operacji,
- Podczas pracy z dynamiczną alokacją pamięci w C++ istotne było zrozumienie i zastosowanie mechanizmów zarządzania pamięcią, takich jak new i delete. Błędy w tej sferze mogły prowadzić do wycieków pamięci, dlatego szczególną uwagę poświęcono metodom usuwania węzłów i całego drzewa. Stworzenie funkcji rekurencyjnych do usuwania wszystkich węzłów pozwoliło uniknąć potencjalnych problemów z zarządzaniem pamięcią,
- Dzięki debugerowi w Visual Studio mogliśmy skutecznie identyfikować błędy, takie jak niepoprawne operacje na wskaźnikach czy brak obsługi przypadków skrajnych, np. próby operacji na pustym drzewie. Debuger ułatwił zrozumienie, jak poszczególne operacje zmieniają strukturę drzewa w trakcie wykonywania programu,
- Wykorzystanie klasy do implementacji drzewa BST pozwoliło na lepszą organizację kodu i logiczne oddzielenie funkcji drzewa od innych elementów programu. Dzięki temu kod jest bardziej czytelny, łatwiejszy w utrzymaniu i gotowy do dalszej rozbudowy,
- Narzędzia takie jak Visual Studio 2022 i GitHub znacząco usprawniły proces programowania. Visual Studio umożliwiło szybkie pisanie, testowanie i debugowanie kodu, a GitHub pozwolił na tworzenie wersji projektu, co zabezpieczało postępy i ułatwiało współpracę.

# Spis rysunków

# Spis tabel

# Spis listingów