AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

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

Autor:
Kacper Kosal

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań **4**

2. Analiza problemu **6**

2.1. Lista dwukierunkowa — opis problemu 6
2.2. Funkcjonalność klasy listy dwukierunkowej 7
2.3. Problemy techniczne 7
2.4. Narzędzia wspomagające rozwój 8
2.5. Podsumowanie analizy problemu 8

3. Projektowanie **9**

3.1. Wykorzystane narzędzia 9
3.2. Ustawienia kompilatora 9
3.3. Diagramy UML 10
3.3.1. Diagram klas 10
3.4. Użycie systemu Git 10
3.5. Schemat działania algorytmu 11

4. Implementacja **22**

4.1. Klasa Node 22
4.2. Klasa DoublyLinkedList 22
4.3. Metody dodawania elementów 23
4.3.1. Dodawanie elementu na początek listy 23
4.3.2. Dodawanie elementu na koniec listy 23
4.3.3. Dodawanie elementu pod wskazanym indeksem 23
4.4. Metody usuwania elementów 24
4.4.1. Usuwanie elementu z początku listy 24
4.4.2. Usuwanie elementu z końca listy 25
4.4.3. Usuwanie elementu pod wskazanym indeksem 25
4.5. Metody wyświetlania elementów 26
4.5.1. Wyświetlanie elementów w kolejności 26
4.5.2. Wyświetlanie elementów w odwrotnej kolejności 26

5. Wnioski **27**

Literatura **29**

Spis rysunków **29**

Spis tabel **30**

Spis listingów **31**

1. Ogólne określenie wymagań

Napisz program listy dwukierunkowej działającej na stercie w języku C++. Działanie listy ma być zaimplementowane w klasie. Funkcjonalność klasy (metody):

- Dodaj element na początek listy,
- Dodaj element na koniec listy,
- Dodaj element pod wskazany indeks,
- Usuń element z początku listy,
- Usuń element z końca listy,
- Usuń element z pod wskazanego indeksu,
- Wyświetl całą listę,
- Wyświetl listę w odwrotnej kolejności,
- Wyświetl następny element,
- Wyświetl poprzedni element,
- Czyść całą listę.

Podczas wykonywania projektu należy:

- Wygenerować dokumentację automatycznie za pomocą programu „doxygen" do pliku „latex" z rozszerzeniem pdf,
- Do projektu dołączyć dokumentację projektową w latex (szablon u prowadzącego),
- Stworzyć konto na GitHub (osoby posiadające konto mogą działać na swoim koncie),
- Można zainstalować wtyczkę do Visual Code lub Visual Studio (zależy do środowiska z którego będziesz korzystać),
- Używać GitHuba przy pisaniu programu (możesz wypróbować wtyczki przy commitowaniu).

Przy pisaniu listy dwukierunkowej należy przetestować różne scenariusze z zastosowaniem gita:

- Co najmniej 5 commit'ów,
- Co najmniej jedno cofnięcie się o dwa commity,
- Usunięcie jednego commita,
- Pobranie projektu z GitHuba do innej lokalizacji na komputerze, dopisanie kolejnej metody w pobranym projekcie, zapisanie zmiany na serwerze. Pobranie zmian z GitHuba do pierwszej lokalizacji, dopisanie nowej metody i zapisanie zmiany na serwer. Obie lokalizacje mogą znajdować się na jednym komputerze tylko w różnych katalogach lub na dwóch komputerach,
- Usunięcie jakiegoś pliku w katalogu projektu (program nie może się nie kompilować). Ściągnąć ostatnią poprawną wersję z GitHuba aby znów projekt się kompilował.

Celem pierwszego projektu jest zapoznanie studenta z programem do kontroli wersji wraz ze stworzeniem dokumentacji projektowej, gdzie należy opisać program listy dwukierunkowej oraz działanie i obsługę programów doxygen i GitHuba. Tutaj może coś być wpisane.

2. Analiza problemu

Celem projektu jest stworzenie listy dwukierunkowej (ang. doubly linked list) działającej na stercie, z zaimplementowanymi metodami do zarządzania jej elementami. W projekcie zostaną również wykorzystane narzędzia takie jak system kontroli wersji Git oraz generator dokumentacji Doxygen. Analiza problemu obejmuje identyfikację wyzwań technicznych oraz funkcjonalnych związanych z implementacją listy, a także integrację narzędzi wspomagających rozwój oprogramowania.

2.1. Lista dwukierunkowa — opis problemu

Lista dwukierunkowa jest dynamiczną strukturą danych, w której każdy element (węzeł) posiada referencje do swojego poprzednika oraz następcy. Pozwala to na łatwą nawigację w obu kierunkach, co czyni ją bardziej elastyczną w porównaniu do list jednokierunkowych. Implementacja listy dwukierunkowej w języku C++ z alokacją na stercie wiąże się z kilkoma wyzwaniami:

- **Zarządzanie pamięcią:** Ponieważ lista ma być dynamicznie zarządzana na stercie, niezbędne będzie ręczne zarządzanie pamięcią. Każdy dodany element musi być odpowiednio zaalokowany, a usuwanie elementów powinno zwalniać pamięć, aby uniknąć wycieków pamięci.

- **Manipulowanie wskaźnikami:** Każdy element listy przechowuje dwa wskaźniki — na poprzedni i następny element. Każda operacja dodawania, usuwania i przemieszczania się po liście wymaga odpowiedniej manipulacji tymi wskaźnikami, co zwiększa ryzyko błędów, takich jak dereferencja wskaźników o wartości null czy modyfikacja niewłaściwych wskaźników.

- **Efektywność operacji:** Lista powinna umożliwiać szybkie operacje na początku i końcu (O(1)) oraz wydajną nawigację pomiędzy elementami. Implementacja musi być zaprojektowana tak, aby niepotrzebnie nie kopiować ani nie alokować dodatkowej pamięci.

- **Złożoność dodawania i usuwania elementów:** Wstawianie i usuwanie elementów z dowolnego miejsca listy wymaga odpowiedniego przesuwania wskaźników w elementach sąsiadujących, co wymaga szczególnej ostrożności, aby nie naruszyć struktury danych.

2.2. Funkcjonalność klasy listy dwukierunkowej

Klasa implementująca listę dwukierunkową powinna umożliwiać użytkownikowi następujące operacje:

- **Dodawanie elementów:** Powinny istnieć metody umożliwiające dodanie elementu na początek listy, na koniec listy, a także wstawienie elementu w dowolne miejsce listy na podstawie indeksu.
- **Usuwanie elementów:** Lista musi umożliwiać usunięcie elementów z początku, z końca oraz z dowolnej pozycji na podstawie indeksu. Podczas usuwania elementów, pamięć powinna być odpowiednio zwalniana, a wskaźniki sąsiadujących elementów poprawnie modyfikowane.
- **Nawigacja po liście:** Użytkownik musi mieć możliwość przechodzenia pomiędzy elementami, zarówno do przodu, jak i do tyłu, co jest jedną z głównych zalet listy dwukierunkowej.
- **Wyświetlanie zawartości:** Aplikacja powinna umożliwiać wyświetlenie całej zawartości listy w obu kierunkach — od początku do końca oraz od końca do początku.
- **Czyszczenie listy:** Lista musi oferować funkcję, która usunie wszystkie elementy i zwolni zajętą przez nie pamięć.

2.3. Problemy techniczne

Podczas implementacji listy dwukierunkowej można napotkać następujące problemy techniczne:

- **Wyciek pamięci:** Każdy nowy element jest dynamicznie alokowany na stercie, co wymaga ręcznego zwalniania pamięci podczas usuwania elementów z listy. Zaniedbanie tego procesu może prowadzić do wycieku pamięci.
- **Dereferencja pustych wskaźników:** Ponieważ wskaźniki na poprzedni i następny element mogą wskazywać na `nullptr`, istnieje ryzyko dereferencji pustych wskaźników, co może prowadzić do awarii programu.
- **Niezgodność wskaźników:** Dodawanie lub usuwanie elementów z listy wiąże się z koniecznością modyfikacji wskaźników poprzedniego i następnego elementu. Niewłaściwe przypisanie wskaźników może prowadzić do zerwania ciągłości listy, co może skutkować niepoprawnym działaniem programu.

- **Efektywność pamięciowa:** Przy dużej liczbie elementów lista może zajmować dużo pamięci, co wymaga odpowiedniej optymalizacji oraz unikania zbędnych operacji na stercie.

2.4. Narzędzia wspomagające rozwój

W trakcie realizacji projektu zostaną wykorzystane dwa kluczowe narzędzia wspomagające proces tworzenia oprogramowania:

- **Doxygen:** Narzędzie do generowania dokumentacji kodu źródłowego, które pozwala automatycznie wygenerować dokumentację w formacie PDF oraz HTML. Doxygen korzysta z komentarzy w kodzie, aby wygenerować szczegółowy opis funkcji, klas oraz ich zależności. Użycie Doxygen zapewnia lepszą czytelność kodu oraz ułatwia jego późniejszą rozbudowę.
- **Git:** System kontroli wersji Git umożliwia śledzenie zmian w kodzie, a także zarządzanie wersjami projektu. Dzięki GitHubowi projekt może być łatwo udostępniany oraz rozwijany w różnych lokalizacjach. Studenci, poprzez integrację Gita, nauczą się podstaw pracy z systemem kontroli wersji, co stanowi kluczową umiejętność w profesjonalnym programowaniu.

2.5. Podsumowanie analizy problemu

Analiza problemu wskazuje, że implementacja listy dwukierunkowej wiąże się z kilkoma wyzwaniami technicznymi, szczególnie w zakresie zarządzania pamięcią i wskaźnikami. Kluczowe będzie również właściwe przetestowanie wszystkich metod klasy oraz integracja z narzędziami wspomagającymi rozwój oprogramowania, takimi jak Doxygen i Git. Rozwiązanie problemów wymaga dokładnej implementacji klasy oraz testów w różnych scenariuszach użytkowania..

3. Projektowanie

3.1. Wykorzystane narzędzia

W projekcie do implementacji listy dwukierunkowej zastosowano następujące narzędzia i technologie:

- **Język programowania**: C++ – wybór tego języka umożliwia efektywne zarządzanie pamięcią oraz korzystanie z programowania obiektowego, co jest kluczowe w implementacji struktur danych.
- **Kompilator i środowisko programistyczne**: Visual Studio 2022 - zintegrowane środowisko programistyczne (IDE), które ułatwia pracę nad projektami C++ poprzez wbudowany kompilator, narzędzia do debugowania oraz wsparcie dla systemu kontroli wersji Git
- **System kontroli wersji**: Git – system umożliwiający zarządzanie wersjami kodu, co jest szczególnie ważne w pracy nad projektami programistycznymi. Użyto również GitHub jako zdalnej platformy do przechowywania repozytoriów.
- **Narzędzie do dokumentacji**: Doxygen – program do automatycznego generowania dokumentacji technicznej z komentarzy w kodzie źródłowym. Pozwala na tworzenie dokumentacji w formatach HTML i LaTeX.

3.2. Ustawienia kompilatora

Podczas kompilacji projektu w Visual Studio 2022 wykorzystano domyślne ustawienia kompilatora MSVC (Microsoft Visual C++), z możliwością wsparcia dla standardu C++11 oraz nowszych wersji. Ustawienia te zapewniają:

- /std:c++11: wsparcie dla standardu C++11, który ułatwia użycie nowoczesnych funkcji języka.
- /W4: poziom ostrzeżeń ustawiony na wysoki, co pomaga w wykrywaniu potencjalnych problemów w kodzie.
- /Zi: generowanie informacji debugowania, co jest przydatne podczas testowania i znajdowania błędów.

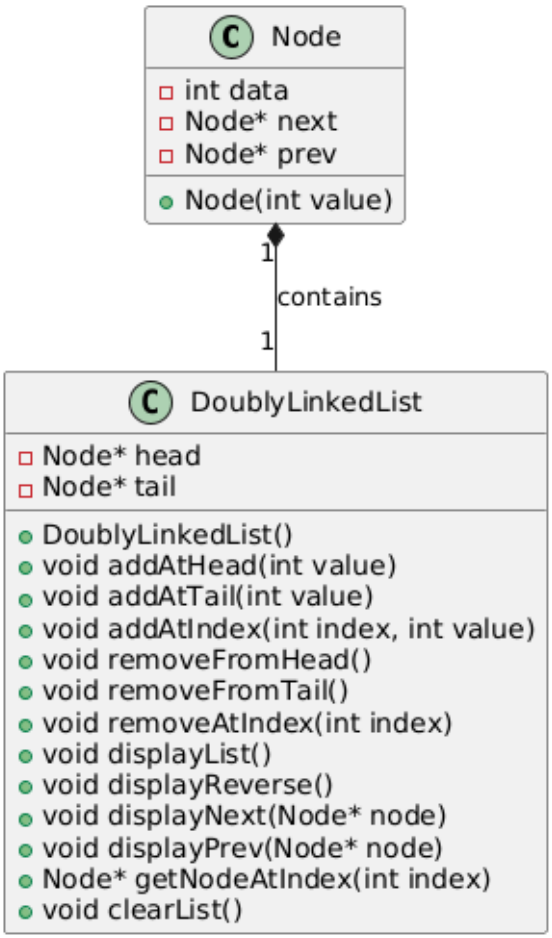
Projekt nie wymagał użycia zewnętrznych bibliotek poza standardową biblioteką języka C++ (iostream do obsługi wejścia i wyjścia).

3.3. Diagramy UML

Aby lepiej zobrazować strukturę klas oraz algorytmy użyte w projekcie, przygotowano odpowiednie diagramy UML.

3.3.1. Diagram klas

Diagram klas ilustruje relacje między klasą 'Node' a klasą 'DoublyLinkedList'. Klasa 'Node' zawiera atrybuty do przechowywania danych oraz wskaźników do węzłów sąsiednich, a klasa 'DoublyLinkedList' oferuje metody do zarządzania listą.



Rys. 3.1. Diagram klas dla listy dwukierunkowej

3.4. Użycie systemu Git

W projekcie zastosowano system Git do zarządzania wersjami kodu. Oto podstawowe polecenia i ich opis:

- **Inicjalizacja repozytorium**:

```
git init
```

Tworzy nowe repozytorium w bieżącym katalogu.

- **Dodawanie zmian do repozytorium**:

```
git add .
```

Dodaje wszystkie zmiany do obszaru staging.

- **Commitowanie zmian**:

```
git commit -m "Opis zmian"
```

Tworzy nowy commit z opisem.

- **Tworzenie gałęzi**:

```
git branch nazwa_gałęzi
```

Tworzy nową gałąź w repozytorium.

- **Przełączanie się między gałęziami**:

```
git checkout nazwa_gałęzi
```

Przełącza na wskazaną gałąź.

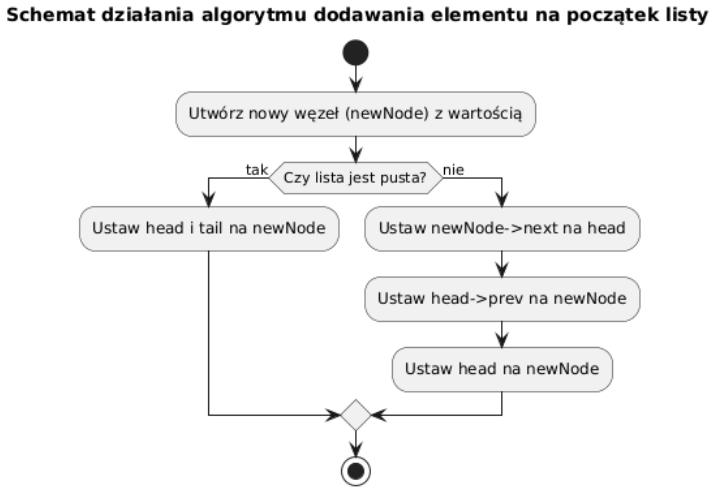
- **Pobieranie zmian z GitHub**:

```
git pull origin main
```

Pobiera zmiany z zdalnego repozytorium.

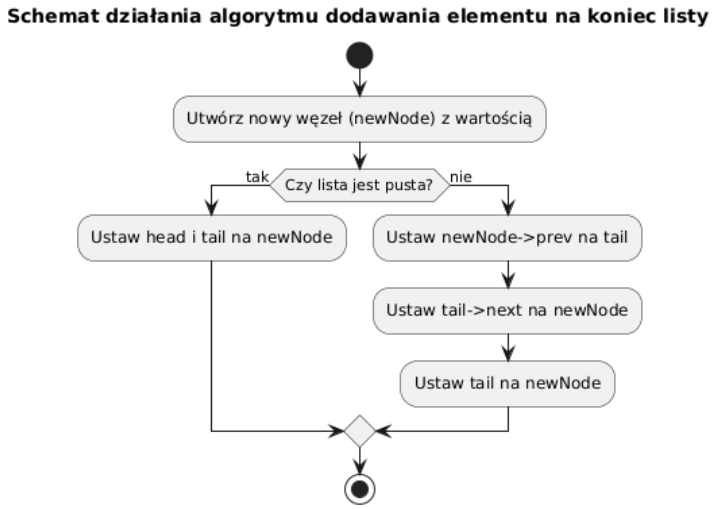
3.5. Schemat działania algorytmu

1. **Dodawanie elementu na początek listy (Prepend)**
   Algorytm dodawania elementu na początek listy polega na utworzeniu nowego węzła, który staje się nową głową listy. Jeśli lista jest pusta, nowy węzeł staje się również końcem listy. W przeciwnym razie, wskaźnik ‘prev‘ starej głowy jest ustawiany na nowy węzeł.



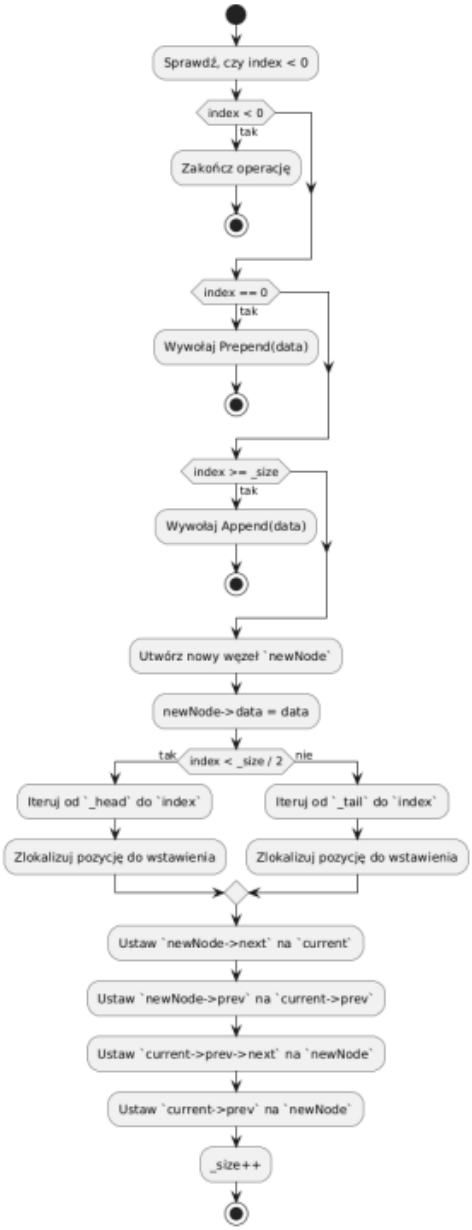
Rys. 3.2. Schemat działania algorytmu dodawania elementu na początek listy

2. **Dodawanie elementu na koniec listy (Append)**
   Algorytm dodawania elementu na koniec listy tworzy nowy węzeł, który staje się nowym końcem listy. Jeśli lista jest pusta, nowy węzeł staje się również głową listy. W przeciwnym razie, wskaźnik ‘next‘ starego końca jest ustawiany na nowy węzeł.



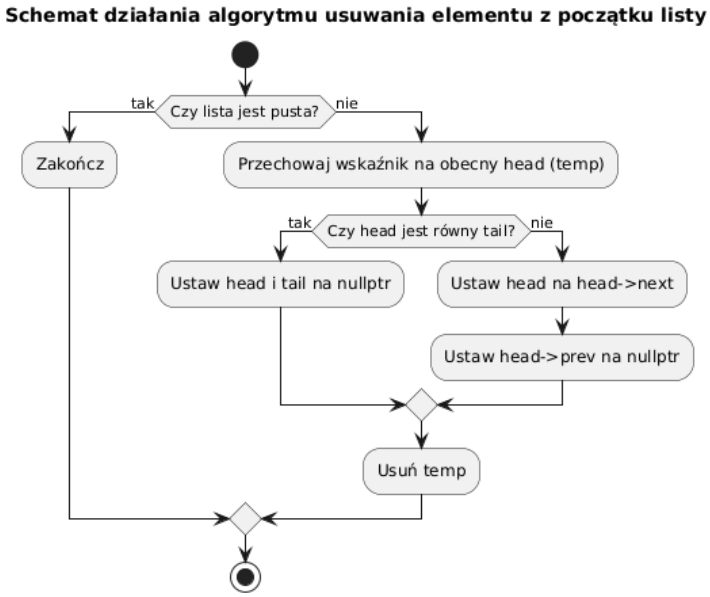
Rys. 3.3. Schemat działania algorytmu dodawania elementu na koniec listy

3. **Wstawianie elementu na określonej pozycji (Insert)**
   Algorytm wstawiania elementu na określonej pozycji polega na przejściu do odpowiedniego indeksu w liście i dodaniu nowego węzła w tej pozycji. W zależności od indeksu, może on być dodany na początek, koniec lub w środek listy.



Rys. 3.4. Schemat działania algorytmu wstawiania elementu na określonej pozycji

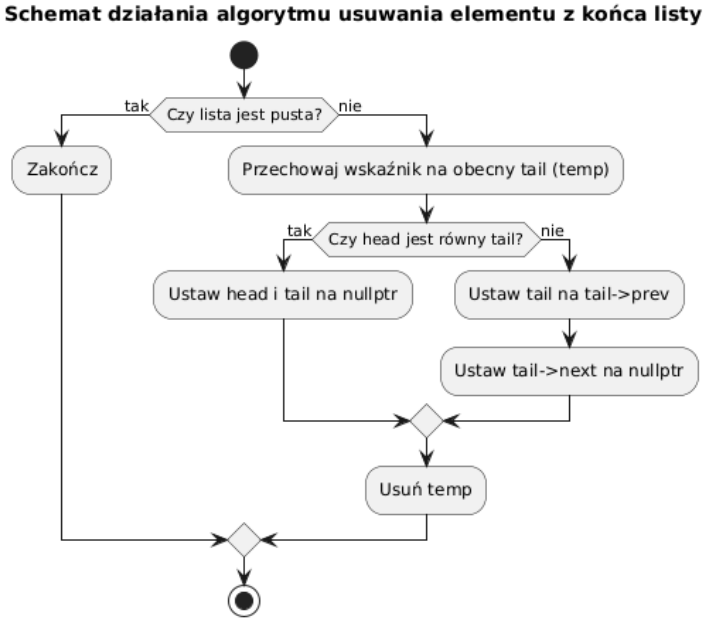
4. **Usuwanie pierwszego elementu (RemoveFirst)**
   Algorytm usuwania pierwszego elementu usuwa głowę listy. Jeśli lista po usunięciu staje się pusta, wskaźnik ‘tail‘ jest również ustawiany na ‘nullptr‘. W przeciwnym razie, nowa głowa ma wskaźnik ‘prev‘ ustawiony na ‘nullptr‘.



Rys. 3.5. Schemat działania algorytmu usuwania pierwszego elementu listy

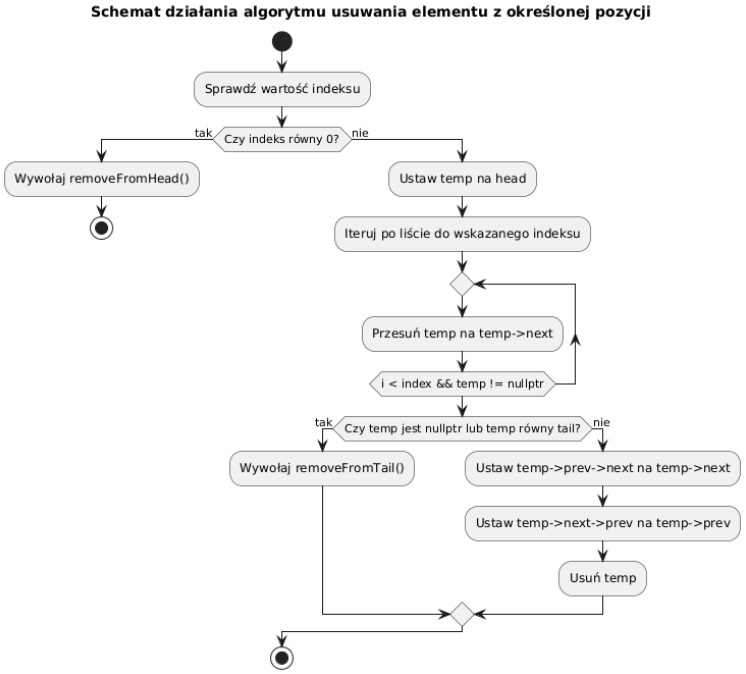
5. **Usuwanie ostatniego elementu (RemoveLast)**

   Algorytm usuwania ostatniego elementu usuwa ogon listy. Jeśli lista po usunięciu staje się pusta, wskaźnik ‘head‘ jest również ustawiany na ‘nullptr‘. W przeciwnym razie, nowy ogon ma wskaźnik ‘next‘ ustawiony na ‘nullptr‘.



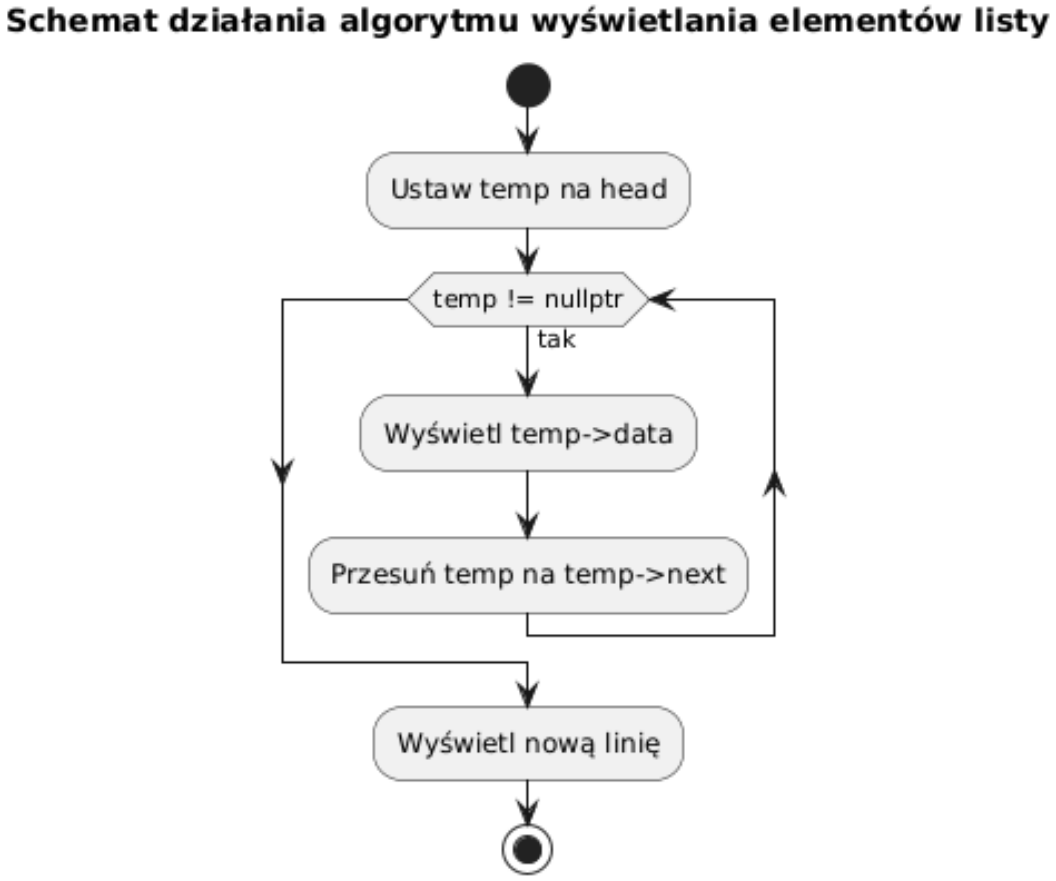
Rys. 3.6. Schemat działania algorytmu usuwania ostatniego elementu listy

6. **Usuwanie elementu z określonej pozycji (Remove)**
   Algorytm usuwania elementu z określonej pozycji przechodzi do wskazanego indeksu w liście, a następnie usuwa węzeł z tej pozycji. Wskaźniki ‘next‘ i ‘prev‘ są odpowiednio aktualizowane, aby zachować spójność listy.



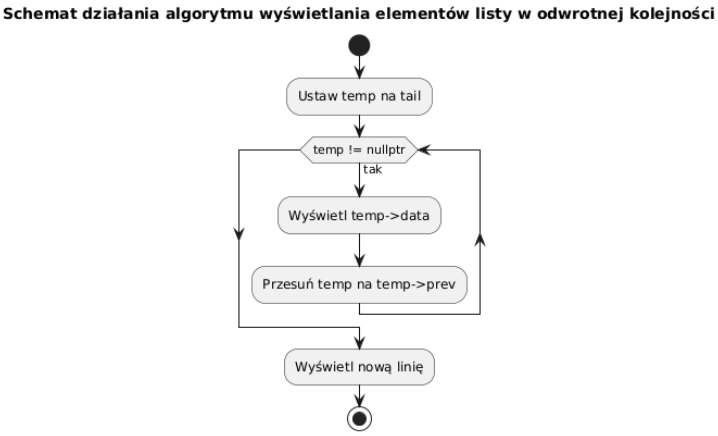
Rys. 3.7. Schemat działania algorytmu usuwania elementu z określonej pozycji

7. **Wyświetlanie elementów listy (Print)**
   Algorytm wyświetlania elementów listy iteruje przez wszystkie elementy, zaczynając od głowy, i wyświetla wartości każdego węzła aż do końca listy.



Rys. 3.8. Schemat działania algorytmu wyświetlania elementów listy

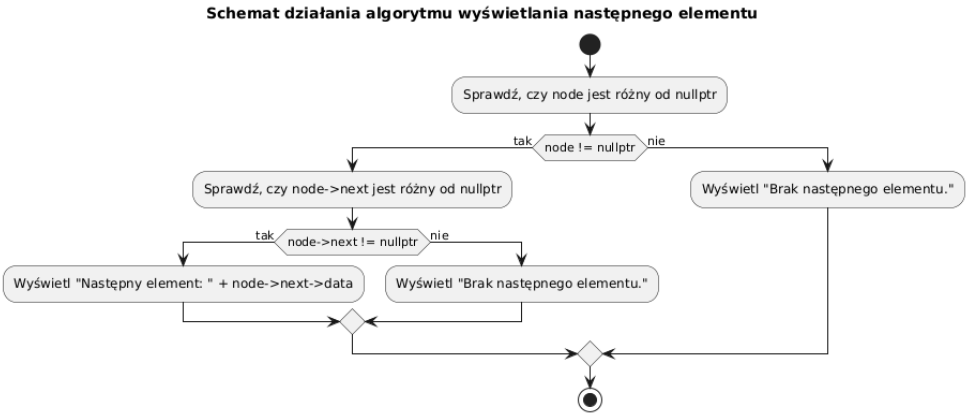
8. **Wyświetlanie elementów listy w odwrotnej kolejności (PrintReverse)**
   Algorytm wyświetlania elementów w odwrotnej kolejności iteruje przez wszystkie elementy, zaczynając od ogona, i wyświetla wartości każdego węzła aż do początku listy.



Rys. 3.9. Schemat działania algorytmu wyświetlania elementów listy w odwrotnej kolejności

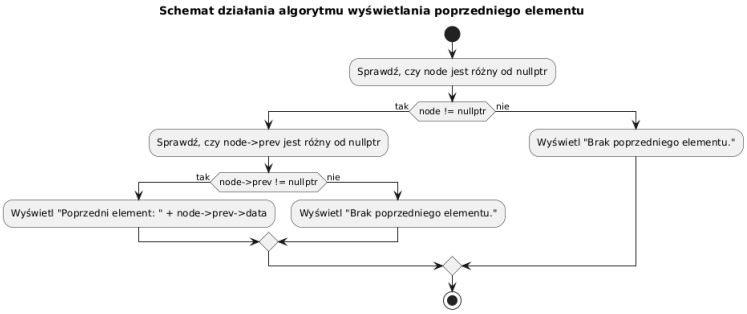
9. **Wyświetlanie następnego elementu (PrintNext)**

   Algorytm wyświetlania następnego elementu przesuwa wskaźnik na kolejny element w liście i wyświetla jego wartość. Jeśli osiągnie koniec listy, wraca do początku.



Rys. 3.10. Schemat działania algorytmu wyświetlania następnego elementu

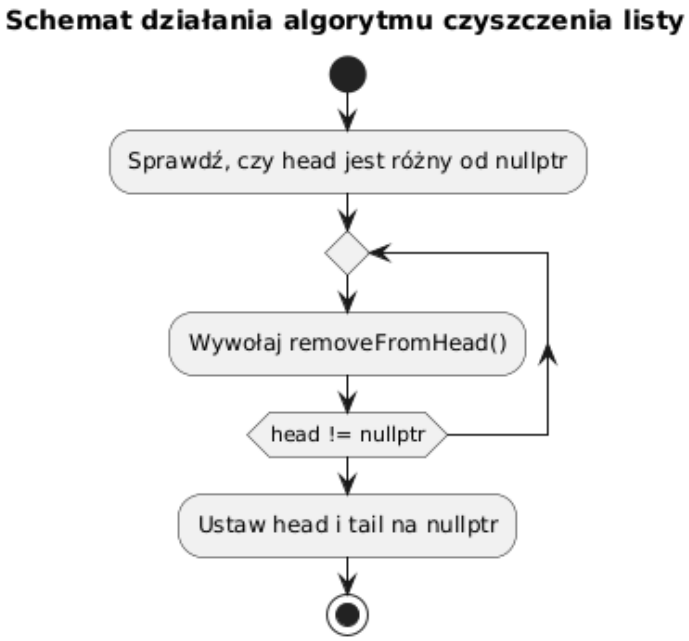
10. **Wyświetlanie poprzedniego elementu (PrintPrev)**
    Algorytm wyświetlania poprzedniego elementu przesuwa wskaźnik na poprzedni element w liście i wyświetla jego wartość. Jeśli osiągnie początek listy, wraca na koniec.



Rys. 3.11. Schemat działania algorytmu wyświetlania poprzedniego elementu

11. **Czyszczenie listy (Clear)**

    Algorytm czyszczenia listy iteruje przez całą listę, usuwając kolejne elementy, aż lista stanie się pusta. Po zakończeniu, wskaźniki ‘head‘ i ‘tail‘ są ustawiane na ‘nullptr‘, a rozmiar listy wynosi 0.



Rys. 3.12. Schemat działania algorytmu czyszczenia listy

4. Implementacja

W tym rozdziale zaprezentowane zostały niektóre operacje git oraz szczegółowa implementacja wszystkich metod z klasy LinkedList w postaci listingów. Znajdują się tu także wyniki z testów działania listy.

4.1. Klasa Node

Klasa `Node` reprezentuje pojedynczy węzeł w liście dwukierunkowej. Każdy węzeł zawiera wartość, wskaźnik do następnego węzła oraz wskaźnik do poprzedniego węzła.

```
class Node {
public:
    int data;      ///< Przechowywana warto Ż  w w Ź le.
    Node* next;    ///< Wska nik na nast Źpny element listy.
    Node* prev;    ///< Wska nik na poprzedni element listy.

    Node(int value) {
        data = value;  // Ustawia warto Ż  w Źz Ća.
        next = nullptr; // Inicjalizuje wska nik do nast Źpnego
    w Źz Ća jako nullptr.
        prev = nullptr; // Inicjalizuje wska nik do poprzedniego
    w Źz Ća jako nullptr.
    }
};
```

Listing 1. Klasa Node

4.2. Klasa DoublyLinkedList

Klasa `DoublyLinkedList` zarządza całą listą dwukierunkową i implementuje funkcje do dodawania, usuwania oraz wyświetlania elementów.

```
class DoublyLinkedList {
private:
    Node* head; ///< Wska nik na pierwszy element listy.
    Node* tail; ///< Wska nik na ostatni element listy.

public:
    DoublyLinkedList() {
        head = nullptr; // Inicjalizuje head jako nullptr.
        tail = nullptr; // Inicjalizuje tail jako nullptr.
    }
```

```
};
```

Listing 2. Klasa DoublyLinkedList

4.3. Metody dodawania elementów

4.3.1. Dodawanie elementu na początek listy

```
void addAtHead(int value) {
    Node* newNode = new Node(value); // Tworzy nowy w Źze Ć.
    if (head == nullptr) {
        head = tail = newNode; // Lista by Ća pusta.
    } else {
        newNode->next = head;  // Ustawia wska nik next nowego
    w Źz Ća.
        head->prev = newNode;   // Ustawia wska nik prev starego
    head.
        head = newNode;         // Nowy w Źze Ć staje si Ź nowym
    head.
    }
}
```

Listing 3. Dodawanie elementu na początek listy

4.3.2. Dodawanie elementu na koniec listy

```
void addAtTail(int value) {
    Node* newNode = new Node(value); // Tworzy nowy w Źze Ć.
    if (tail == nullptr) {
        head = tail = newNode; // Lista by Ća pusta.
    } else {
        newNode->prev = tail;   // Ustawia wska nik prev nowego
    w Źz Ća.
        tail->next = newNode;    // Ustawia wska nik next starego
    tail.
        tail = newNode;          // Nowy w Źze Ć staje si Ź nowym
    tail.
    }
}
```

Listing 4. Dodawanie elementu na koniec listy

4.3.3. Dodawanie elementu pod wskazanym indeksem

```
void addAtIndex(int index, int value) {
    if (index == 0) {
        addAtHead(value); // Dodaj na pocz tku , je Żli indeks to
    0.
        return;
    }

    Node* newNode = new Node(value);
    Node* temp = head;
    for (int i = 0; i < index - 1 && temp != nullptr; i++) {
        temp = temp->next; // Szuka odpowiedniego miejsca.
    }

    if (temp == nullptr || temp->next == nullptr) {
        addAtTail(value); // Dodaje na ko cu , je Żli temp jest
    nullptr.
    } else {
        newNode->next = temp->next; // Ustawia wska nik next
    nowego w Źz Ća.
        newNode->prev = temp;         // Ustawia wska nik prev
    nowego w Źz Ća.
        temp->next->prev = newNode;  // Ustawia wska nik prev
    nast Źpnego w Źz Ća.
        temp->next = newNode;         // Ustawia wska nik next
    w Źz Ća temp.
    }
}
```

Listing 5. Dodawanie elementu pod wskazanym indeksem

4.4. Metody usuwania elementów

4.4.1. Usuwanie elementu z początku listy

```
void removeFromHead() {
    if (head == nullptr) return; // Nic do usuni Ścia.

    Node* temp = head;
    if (head == tail) {
        head = tail = nullptr; // Usuni Ście jedynego w Źz Ća.
    } else {
        head = head->next;    // Przesuwa head na nast Źpny w Źze Ć
    .
        head->prev = nullptr; // Ustawia wska nik prev na nullptr.
    }
```

```
    delete temp; // Zwalnia pami Ź .
}
```

Listing 6. Usuwanie elementu z początku listy

4.4.2. Usuwanie elementu z końca listy

```
void removeFromTail() {
    if (tail == nullptr) return; // Nic do usuni Ścia.

    Node* temp = tail;
    if (head == tail) {
        head = tail = nullptr; // Usuni Ście jedynego w Źz Ća.
    } else {
        tail = tail->prev;    // Przesuwa tail na poprzedni w Źze Ć
    .
        tail->next = nullptr; // Ustawia wska nik next na nullptr.
    }
    delete temp; // Zwalnia pami Ź .
}
```

Listing 7. Usuwanie elementu z końca listy

4.4.3. Usuwanie elementu pod wskazanym indeksem

```
void removeAtIndex(int index) {
    if (index == 0) {
        removeFromHead(); // Usuwa z pocz tku , je Żli indeks to 0.
        return;
    }

    Node* temp = head;
    for (int i = 0; i < index && temp != nullptr; i++) {
        temp = temp->next; // Szuka w Źz Ća pod wskazanym indeksem.
    }

    if (temp == nullptr || temp == tail) {
        removeFromTail(); // Usuwa z ko ca , je Żli temp jest
    nullptr lub tail.
    } else {
        temp->prev->next = temp->next; // Ustawia wska nik next
    w Źz Ća poprzedniego.
        temp->next->prev = temp->prev; // Ustawia wska nik prev
    nast Źpnego w Źz Ća.
```

```
17          delete temp; // Zwalnia pami Ź .
18      }
19  }
```

Listing 8. Usuwanie elementu pod wskazanym indeksem

4.5. Metody wyświetlania elementów

4.5.1. Wyświetlanie elementów w kolejności

```
1  void displayList() {
2      Node* temp = head;
3      while (temp != nullptr) {
4          cout << temp->data << " "; // Wy Żwietla warto Ż  w Źz Ća.
5          temp = temp->next; // Przesuwa do nast Źpnego w Źz Ća.
6      }
7      cout << endl;
8  }
```

Listing 9. Wyświetlanie elementów w kolejności

4.5.2. Wyświetlanie elementów w odwrotnej kolejności

```
1  void displayReverse() {
2      Node* temp = tail;
3      while (temp != nullptr) {
4          cout << temp->data << " "; // Wy Żwietla warto Ż  w Źz Ća.
5          temp = temp->prev; // Przesuwa do poprzedniego w Źz Ća.
6      }
7      cout << endl;
8  }
```

Listing 10. Wyświetlanie elementów w odwrotnej kolejności

5. Wnioski

W ramach projektu stworzono dwukierunkową listę wiązaną, która umożliwia efektywne zarządzanie zbiorami danych poprzez operacje dodawania, usuwania oraz przeglądania elementów. Realizacja tego projektu pozwoliła na głębsze zrozumienie struktury list wiązanych oraz roli wskaźników w języku C++.

Podczas implementacji listy dwukierunkowej zwrócono szczególną uwagę na następujące aspekty:

- **Efektywność dodawania i usuwania elementów:** Operacje takie jak dodawanie na początek lub koniec listy oraz usuwanie elementów w tych pozycjach są niezwykle szybkie, ponieważ wymagają jedynie aktualizacji kilku wskaźników.

- **Przechodzenie przez listę:** Dzięki wskaźnikom `next` i `prev`, możliwe jest przeglądanie listy w obu kierunkach, co ułatwia wykonywanie złożonych operacji na danych, takich jak odwracanie listy czy iterowanie od końca do początku.

- **Wydajność wstawiania i usuwania elementów w środku listy:** Operacje te wymagają przeszukania listy do odpowiedniego indeksu, co może być czasochłonne w przypadku dużych zbiorów danych. Jednak w porównaniu do struktur takich jak tablice dynamiczne, lista dwukierunkowa oferuje większą elastyczność.

- **Zarządzanie pamięcią:** Implementacja listy dwukierunkowej wymagała dokładnego zarządzania pamięcią, w tym zwalniania pamięci przy usuwaniu elementów, aby uniknąć wycieków. Projekt ten przyczynił się do lepszego zrozumienia roli wskaźników i destruktorów w C++.

- **Złożoność algorytmów:** Podczas implementacji algorytmów i ich analizy w formie schematów blokowych można było dostrzec, jak złożoność czasowa i przestrzenna wpływają na działanie struktury danych, co jest istotne w kontekście optymalizacji.

Podsumowując, projekt umożliwił praktyczne zastosowanie teoretycznej wiedzy na temat struktur danych, takich jak listy wiązane, oraz rozwój umiejętności programowania w języku C++. Implementacja różnych metod operujących na liście oraz ich wizualizacja przy pomocy schematów blokowych ułatwiły zrozumienie działania

tej struktury. Uzyskane wyniki potwierdzają, że lista dwukierunkowa jest odpowiednią strukturą danych dla aplikacji, które wymagają częstego dodawania i usuwania elementów na początku lub końcu zbioru, a także tam, gdzie niezbędne jest przeglądanie danych w obu kierunkach.

Spis rysunków

3.1. Diagram klas dla listy dwukierunkowej 10
3.2. Schemat działania algorytmu dodawania elementu na początek listy . 12
3.3. Schemat działania algorytmu dodawania elementu na koniec listy . . 12
3.4. Schemat działania algorytmu wstawiania elementu na określonej pozycji 13
3.5. Schemat działania algorytmu usuwania pierwszego elementu listy . . . 14
3.6. Schemat działania algorytmu usuwania ostatniego elementu listy . . . 15
3.7. Schemat działania algorytmu usuwania elementu z określonej pozycji 16
3.8. Schemat działania algorytmu wyświetlania elementów listy 17
3.9. Schemat działania algorytmu wyświetlania elementów listy w odwrotnej kolejności 18
3.10. Schemat działania algorytmu wyświetlania następnego elementu . . . 19
3.11. Schemat działania algorytmu wyświetlania poprzedniego elementu . . 20
3.12. Schemat działania algorytmu czyszczenia listy 21

Spis tabel

Spis listingów

1. Klasa Node 22
2. Klasa DoublyLinkedList 22
3. Dodawanie elementu na początek listy 23
4. Dodawanie elementu na koniec listy 23
5. Dodawanie elementu pod wskazanym indeksem 24
6. Usuwanie elementu z początku listy 24
7. Usuwanie elementu z końca listy 25
8. Usuwanie elementu pod wskazanym indeksem 25
9. Wyświetlanie elementów w kolejności 26
10. Wyświetlanie elementów w odwrotnej kolejności 26