

AKADEMIA NAUK STOSOWANYCH W NOWYM SĄCZU

Wydział Nauk Inżynierskich
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA ZAAWANSOWANE PROGRAMOWANIE

Algorytm listy dwukierunkowej z zastosowaniem GitHub

Autor:
Tomasz Iwański

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań	4
1.1. Funkcjonalności programu:	4
2. Analiza problemu	5
2.1. Zastosowania dwukierunkowej listy wiązanej	5
2.2. Działanie algorytmu	6
2.3. Wykorzystanie narzędzia Git	6
3. Projektowanie	8
3.1. Narzędzia używane w projekcie	8
3.2. Wykorzystanie Git w projekcie	9
4. Implementacja	10
4.1. Definicja Struktury Wezla	10
4.2. Klasa Listy Dwukierunkowej	10
4.3. Konstruktor Klasy	10
4.4. Operacje na Liscie	10
4.5. Dodawanie Elementow	11
4.5.1. Dodawanie na początku listy	11
4.5.2. Dodawanie na koncu listy	11
4.5.3. Dodawanie elementu pod wskazany indeks	12
4.6. Usuwanie Elementow	12
4.6.1. Usuwanie elementu z początku listy	12
4.6.2. Usuwanie elementu z konca listy	13
4.6.3. Usuwanie elementu pod wskazanym indeksem	13
4.7. Wyszukiwanie Elementow	14
4.7.1. Wyszukiwanie listy od początku	14
4.7.2. Wyszukiwanie listy od konca	14
4.7.3. Wyszukiwanie następnego elementu dla danego wezla	14
4.7.4. Wyszukiwanie poprzedniego elementu dla danego wezla	15
4.8. Usuwanie całej listy	15

4.9. Destruktor Klasy	15
5. Wnioski	17
5.1. Efektywność operacji na liście	17
5.1.1. Dodawanie i usuwanie elementów:	17
5.1.2. Przechodzenie przez listę	17
5.2. Wydajność operacji w środku listy	18
5.3. Zarządzanie pamięcią	18
5.4. Analiza złożoności algorytmów	18
5.5. Zastosowania praktyczne	19
5.6. Refleksje i przyszłe kierunki rozwoju	19
Literatura	20
Spis rysunków	20
Spis tabel	21
Spis listingów	22

1. Ogólne określenie wymagań

Celem projektu jest opracowanie programu w języku C++, który implementuje dwukierunkową (podwójnie wiążaną) listę działającą na sterzie. Lista wiązana to struktura danych, w której każdy element (węzeł) zawiera odniesienia zarówno do poprzedniego, jak i do następnego elementu, co umożliwia łatwe poruszanie się po liście w obu kierunkach. Tego typu struktura danych znajduje szerokie zastosowanie w programowaniu, zwłaszcza gdy potrzebna jest efektywna manipulacja dynamicznie zarządzanymi danymi.

1.1. Funkcjonalności programu:

- Dodawanie elementów – zarówno na początku listy, jak i na końcu, a także w dowolnym miejscu, zgodnie z indeksem podanym przez użytkownika.
- Usuwanie elementów – na początku listy, na końcu oraz w określonym przez użytkownika miejscu.
- Wyświetlanie zawartości listy – w porządku, w jakim elementy zostały dodane, jak i w odwrotnej kolejności.
- Oczyszczanie listy – możliwość usunięcia wszystkich elementów listy jednocześnie, co zwalnia pamięć dynamicznie przydzieloną na sterzie.

Program ten zakłada stworzenie klasy reprezentującej listę dwukierunkową, która będzie posiadać metody odpowiedzialne za wszystkie powyższe operacje. Każdy element listy, czyli węzeł, będzie zawierał wskaźniki na poprzedni i następny element oraz dane użytkownika.

2. Analiza problemu

Implementacja listy dwukierunkowej w C++ stanowi złożone zagadnienie, które wymaga uwzględnienia kilku kluczowych aspektów. Przede wszystkim, konieczne jest zaprojektowanie struktury węzła zawierającego wskaźniki do poprzedniego i następnego elementu oraz przechowywaną wartość. Zarządzanie dynamiczną alokacją pamięci jest tutaj krytyczne; należy zapewnić prawidłowe tworzenie i usuwanie węzłów, aby uniknąć wycieków pamięci i innych problemów związanych z jej niepoprawnym zarządzaniem. Dodatkowo, implementacja powinna obsługiwać podstawowe operacje, takie jak wstawianie, usuwanie i iterowanie przez elementy listy, z uwzględnieniem wszystkich przypadków brzegowych (np. operacje na pustej liście lub usuwanie ostatniego węzła). Ważnym aspektem jest także zapewnienie bezpieczeństwa w kontekście wyjątków, co w praktyce oznacza obsługę potencjalnych błędów podczas alokacji pamięci czy operacji na wskaźnikach. Ponadto, warto rozważyć implementację iteratorów kompatybilnych ze standardową biblioteką STL, co umożliwi wykorzystanie listy w kontekście algorytmów bibliotecznych i zwiększy jej uniwersalność.

2.1. Zastosowania dwukierunkowej listy wiązanej

Dwukierunkowa lista wiązana to struktura danych, która jest często stosowana w programowaniu, zwłaszcza tam, gdzie istotne jest sprawne dodawanie i usuwanie elementów z dowolnej pozycji na liście. Dzięki swojej elastyczności, pozwala na efektywne manipulowanie danymi, co czyni ją przydatną w wielu obszarach. Ten algorytm znajduje zastosowanie między innymi w:

- W systemach operacyjnych, gdzie dwukierunkowe listy wiązane mogą być stosowane do zarządzania procesami lub zasobami.
- Implementacji struktur danych takich jak kolejki priorytetowe, deki (double-ended queues), oraz stosy.
- W aplikacjach wymagających łatwego przechodzenia w przód i w tył po elementach, takich jak edytory tekstu (np. do zarządzania cofaniem i powtarzaniem operacji).
- W programowaniu gier do zarządzania listami obiektów lub historii zdarzeń.

2.2. Działanie algorytmu

Dwukierunkowa lista wiązana składa się z węzłów, z których każdy przechowuje dane oraz dwa wskaźniki: jeden wskazujący na kolejny węzeł i drugi na poprzedni. Taka konstrukcja umożliwia poruszanie się po liście w obu kierunkach, zarówno do przodu, jak i do tyłu. Program realizuje następujące funkcje:

- Dodawanie elementów na początku i końcu listy.
- Wyświetlanie zawartości listy oraz jej odwrotności.
- Usuwanie elementów z początku, końca oraz z określonego indeksu
- Czyszczenie całej listy.

Główną zaletą takiej struktury jest możliwość łatwego dodawania i usuwania elementów bez konieczności przesuwania pozostałych elementów, co jest często wymagane w przypadku tablic.

2.3. Wykorzystanie narzędzia Git

W projekcie zastosowano system kontroli wersji Git, który pozwala na skuteczne zarządzanie wersjami kodu oraz ułatwia współpracę nad projektem. Git jest szeroko używanym narzędziem w programowaniu, umożliwiającym m.in.:

- Pracę nad projektem w zespole, poprzez możliwość tworzenia gałęzi (branches) i łączenia ich za pomocą operacji merge.
- Zarządzanie historią projektu, w tym dodawanie opisów do commitów, co ułatwia dokumentowanie postępów w projekcie.
- Śledzenie zmian w kodzie, co umożliwia przywracanie poprzednich wersji w razie potrzeby.
- Tworzenia co najmniej 5 commitów, dokumentujących kluczowe etapy pracy nad projektem, takie jak implementacja nowych funkcjonalności, testowanie oraz poprawki błędów.
- Synchronizacji projektu z repozytorium na GitHub, co umożliwia przechowywanie kopii zapasowej oraz pracę nad kodem z różnych lokalizacji.
- Cofania zmian, aby sprawdzić, jak można powrócić do poprzednich wersji projektu.

Zastosowanie Gita zapewniło pełną kontrolę nad zmianami w projekcie, co jest niezbędne do utrzymania porządku w kodzie oraz zapewnienia bezpieczeństwa danych. Dzięki Gitowi możliwe było śledzenie wszystkich modyfikacji, cofanie błędnych zmian oraz zarządzanie różnymi wersjami projektu w sposób uporządkowany i bezpieczny.

3. Projektowanie

3.1. Narzędzia używane w projekcie

W projekcie wykorzystano narzędzia i technologie:

- System kontroli wersji: Git - używany do śledzenia zmian w kodzie, zarządzania repozytoriami oraz synchronizacji zdalnego repozytorium na GitHub. 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
- Język programowania: C++ - język ten został wybrany ze względu na możliwość bezpośredniego zarządzania pamięcią oraz dostępność wskaźników, co jest istotne przy implementacji dwukierunkowej listy wiązanej.
- Generowanie dokumentacji: Doxygen - program używany do generowania dokumentacji z kodu źródłowego, co pozwala na automatyczne tworzenie plików LATEX z opisami klas, metod i funkcji

3.2. Wykorzystanie Git w projekcie

W trakcie realizacji projektu zastosowano system kontroli wersji Git do zarządzania kodem źródłowym oraz synchronizacji z repozytorium na GitHub. Poniżej przedstawiono użyte polecenia wraz z ich opisami:

- `git add .` - Dodaje wszystkie zmodyfikowane pliki do obszaru staging, przygotowując je do zapisania w następnym commicie.
- `git init` - Inicjuje nowe lokalne repozytorium Git w bieżącym katalogu. Tworzy ukryty folder `.git`, który zawiera wszystkie dane potrzebne do zarządzania wersjami
- `git commit -m "komentarz"` - Tworzy nowy commit z plików, które zostały wcześniej dodane do obszaru staging, z opisem zmian podanym w "komentarz".
- `git log` - Wyświetla historię commitów, umożliwiając śledzenie zmian wprowadzonych w projekcie
- `git status` - Wyświetla stan bieżącego repozytorium, informując o plikach, które zostały zmodyfikowane, dodane lub usunięte, oraz o ich stanie względem obszaru staging.
- `git checkout nazwa gałęzi` - Przełącza bieżącą gałąź na nazwa gałęzi, umożliwiając pracę nad kodem w innej gałęzi.
- `git push` - Przesyła lokalne commity do zdalnego repozytorium na GitHub, synchronizując zmiany z innymi użytkownikami.
- `git pull` - Pobiera najnowsze zmiany z zdalnego repozytorium i synchronizuje je z lokalnym repozytorium, co zapewnia, że mamy aktualną wersję projektu.
- `git revert commit hash` - Tworzy nowy commit, który odwraca zmiany wprowadzone w określonym commicie oznaczonym commit hash, co jest bardziej bezpiecznym sposobem cofania zmian.
- `git rm nazwa pliku` - Usuwa plik z obszaru śledzonego przez Git i przygotowuje zmianę do zapisania w następnym commicie.

4. Implementacja

W tym rozdziale przedstawiono wybrane operacje Gita oraz szczegółową implementację wszystkich metod z klasy `LinkedList` w formie listingów. Znajdą się tu również wyniki testów dotyczących działania listy.

4.1. Definicja Struktury Wezła

Pierwszym krokiem w implementacji listy dwukierunkowej jest zdefiniowanie struktury reprezentującej pojedynczy węzeł. Węzeł przechowuje dane oraz wskaźniki do poprzedniego i następnego węzła.

```
1 struct Node {  
2     int data;           // Przechowywane dane  
3     Node* prev;        // Wskaźnik na poprzedni węzeł  
4     Node* next;        // Wskaźnik na następny węzeł  
5  
6     // Konstruktor węzła  
7     Node(int value) : data(value), prev(nullptr), next(nullptr) {}  
8 };
```

Listing 1. Struktura węzła

4.2. Klasa Listy Dwukierunkowej

Następnie tworzymy klasę `DoublyLinkedList`, która będzie zarządzać węzłami. Klasa ta zawiera wskaźniki do pierwszego (`head`) i ostatniego (`tail`) węzła listy oraz metody do wykonywania różnych operacji na liście.

4.3. Konstruktor Klasy

Konstruktor klasy inicjalizuje wskaźniki `head` i `tail` na `nullptr`, co oznacza, że lista jest pusta:

```
1 DoublyLinkedList() : head(nullptr), tail(nullptr) {}
```

Listing 2. Konstruktor klasy

4.4. Operacje na Liście

W klasie `DoublyLinkedList` zdefiniowane są różne metody, które umożliwiają manipulację elementami listy.

4.5. Dodawanie Elementow

4.5.1. Dodawanie na poczatku listy

Metoda `insertAtBeginning` tworzy nowy wezel i dodaje go na poczatku listy. Jesli lista jest pusta, nowy wezel staje sie zarówno `head`, jak i `tail`.

```
1 void insertAtBeginning(int value) {
2     Node* newNode = new Node(value); // Tworzenie nowego wezla
3     if (head == nullptr) {           // Jesli lista jest pusta
4         head = tail = newNode;       // Head i tail wskazuja na
        nowy wezel
5     } else {
6         newNode->next = head;        // Nowy wezel wskazuje na
        obecny head
7         head->prev = newNode;        // Obecny head wskazuje na
        nowy wezel jako prev
8         head = newNode;              // Aktualizacja head do nowego
        wezla
9     }
10 }
```

Listing 3. Dodawanie elementu na poczatku listy

4.5.2. Dodawanie na koncu listy

Metoda `insertAtEnd` dziala podobnie jak `insertAtBeginning`, ale dodaje nowy wezel na koncu listy.

```
1 void insertAtEnd(int value) {
2     Node* newNode = new Node(value); // Tworzenie nowego wezla
3     if (tail == nullptr) {           // Jesli lista jest pusta
4         head = tail = newNode;       // Head i tail wskazuja na
        nowy wezel
5     } else {
6         tail->next = newNode;        // Ostatni wezel wskazuje na
        nowy wezel
7         newNode->prev = tail;        // Nowy wezel wskazuje na
        stary tail
8         tail = newNode;              // Aktualizacja tail do nowego
        wezla
9     }
10 }
```

Listing 4. Dodawanie elementu na koncu listy

4.5.3. Dodawanie elementu pod wskazany indeks

```
1 // Dodawanie elementu pod wskazany indeks
2 void insertAtIndex(int value, int index) {
3     if (index == 0) {
4         insertAtBeginning(value);
5         return;
6     }
7
8     Node* newNode = new Node(value);
9     Node* temp = head;
10    for (int i = 0; i < index - 1 && temp != nullptr; ++i) {
11        temp = temp->next;
12    }
13
14    if (temp == nullptr || temp == tail) {
15        insertAtEnd(value);
16    } else {
17        newNode->next = temp->next;
18        newNode->prev = temp;
19        if (temp->next != nullptr) {
20            temp->next->prev = newNode;
21        }
22        temp->next = newNode;
23    }
24 }
```

Listing 5. Dodawanie elementu pod wskazany indeks

4.6. Usuwanie Elementow

4.6.1. Usuwanie elementu z poczatku listy

```
1 // Usuwanie elementu z poczatku listy
2 void deleteFromBeginning() {
3     if (head == nullptr) return; // Lista pusta
4     Node* temp = head;
5     head = head->next;
6     if (head != nullptr) {
7         head->prev = nullptr;
8     } else {
9         tail = nullptr;
10    }
11    delete temp;
```

12 }

Listing 6. Usuwanie elementu z początku listy

4.6.2. Usuwanie elementu z końca listy

```

1 // Usuwanie elementu z końca listy
2 void deleteFromEnd() {
3     if (tail == nullptr) return; // Lista pusta
4     Node* temp = tail;
5     tail = tail->prev;
6     if (tail != nullptr) {
7         tail->next = nullptr;
8     } else {
9         head = nullptr;
10    }
11    delete temp;
12 }

```

Listing 7. Usuwanie elementu z końca listy

4.6.3. Usuwanie elementu pod wskazanym indeksem

```

1 // Usuwanie elementu pod wskazanym indeksem
2 void deleteAtIndex(int index) {
3     if (index == 0) {
4         deleteFromBeginning();
5         return;
6     }
7
8     Node* temp = head;
9     for (int i = 0; i < index && temp != nullptr; ++i) {
10        temp = temp->next;
11    }
12
13    if (temp == nullptr) return; // Indeks poza zakresem
14
15    if (temp == tail) {
16        deleteFromEnd();
17    } else {
18        temp->prev->next = temp->next;
19        if (temp->next != nullptr) {
20            temp->next->prev = temp->prev;
21        }

```

```
22     delete temp;
23 }
24 }
```

Listing 8. Usuwanie elementu pod wskazanym indeksem

4.7. Wyświetlanie Elementow

4.7.1. Wyświetlanie listy od początku

```
1 // Wyświetlanie listy od początku
2 void displayForward() {
3     Node* temp = head;
4     while (temp != nullptr) {
5         std::cout << temp->data << " -> ";
6         temp = temp->next;
7     }
8     std::cout << "nullptr" << std::endl;
9 }
```

Listing 9. Wyświetlanie listy od początku

4.7.2. Wyświetlanie listy od konca

```
1 // Wyświetlanie listy od konca
2 void displayBackward() {
3     Node* temp = tail;
4     while (temp != nullptr) {
5         std::cout << temp->data << " -> ";
6         temp = temp->prev;
7     }
8     std::cout << "nullptr" << std::endl;
9 }
```

Listing 10. Wyświetlanie listy od konca

4.7.3. Wyświetlanie następnego elementu dla danego węzła

```
1 // Wyświetlanie następnego elementu dla danego węzła
2 void displayNext(Node* node) {
3     if (node->next != nullptr) {
4         std::cout << "Następny element: " << node->next->data <<
5         std::endl;
6     } else {
7         std::cout << "Brak następnego elementu" << std::endl;
8     }
9 }
```

```

6         std::cout << "Brak następnego elementu." << std::endl;
7     }
8 }

```

Listing 11. Wyświetlanie następnego elementu

4.7.4. Wyświetlanie poprzedniego elementu dla danego węzła

```

1 // Wyświetlanie poprzedniego elementu dla danego węzła
2 void displayPrevious(Node* node) {
3     if (node->prev != nullptr) {
4         std::cout << "Poprzedni element: " << node->prev->data <<
std::endl;
5     } else {
6         std::cout << "Brak poprzedniego elementu." << std::endl;
7     }
8 }

```

Listing 12. Wyświetlanie poprzedniego elementu

4.8. Usuwanie całej listy

```

1 // Usuwanie całej listy
2 void deleteList() {
3     Node* current = head;
4     while (current != nullptr) {
5         Node* next = current->next; // Zapisanie wskaźnika na
następny węzeł
6         std::cout << "Usuwanie węzła o wartości: " << current->data
<< std::endl;
7         delete current;           // Zwalnianie obecnego węzła
8         current = next;           // Przechodzenie do następnego
węzła
9     }
10    head = tail = nullptr;        // Resetowanie wskaźników po
usuniecie listy
11 }

```

Listing 13. Usuwanie całej listy

4.9. Destruktor Klasy

```

1 // Destruktor listy dwukierunkowej
2 ~DoublyLinkedList() {

```

```
3     deleteList(); // Usuwanie całej listy przy niszczeniu obiektu
4 }
```

Listing 14. Destruktor klasy

5. Wnioski

W ramach realizacji projektu stworzono dwukierunkową listę wiązaną, co stanowi efektywną metodę zarządzania zbiorami danych. Implementacja ta umożliwiła przeprowadzenie operacji dodawania, usuwania oraz przeglądania elementów w sposób efektywny i elastyczny. Projekt dostarczył głębszego wglądu w strukturę list związanych oraz zastosowanie wskaźników w języku C++. Poniżej przedstawiono kluczowe wnioski oraz refleksje dotyczące implementacji i jej znaczenia.

5.1. Efektywność operacji na liście

5.1.1. Dodawanie i usuwanie elementów:

- Jednym z głównych atutów dwukierunkowej listy związanej jest szybkość operacji dodawania i usuwania elementów. W przypadku dodawania na początku lub końcu listy, operacje te wymagają jedynie aktualizacji kilku wskaźników. Takie podejście znacząco zmniejsza czas potrzebny na realizację tych operacji w porównaniu do innych struktur danych, takich jak tablice, gdzie proces ten często wiąże się z przesuwaniem pozostałych elementów.

Dzięki prostocie operacji wstawiania i usuwania, aplikacje mogą dynamicznie zarządzać danymi w odpowiedzi na zmieniające się potrzeby użytkowników lub systemu. Na przykład, w aplikacjach do zarządzania zadaniami, gdzie użytkownik często dodaje lub usuwa elementy, wydajność operacji na liście związanej może znacząco poprawić responsywność interfejsu użytkownika.

5.1.2. Przechodzenie przez listę

- Dzięki wskaźnikom `next` i `prev`, dwukierunkowa lista związana pozwala na swobodne przeglądanie danych w obu kierunkach. Ta cecha ułatwia realizację złożonych operacji, takich jak odwracanie listy czy iteracja od końca do początku. Możliwość przeglądania listy w obie strony jest szczególnie przydatna w przypadku aplikacji, które wymagają wykonywania operacji na danych w różnorodny sposób, takich jak systemy rekomendacji, które mogą potrzebować przeszukiwać dane zarówno w kolejności chronologicznej, jak i odwrotnej. W kontekście gier komputerowych, gdzie obiekty mogą być dynamicznie dodawane lub usuwane w różnych kierunkach, ta funkcjonalność znacząco zwiększa efektywność algorytmów przeszukiwania.

5.2. Wydajność operacji w środku listy

- Wstawianie i usuwanie elementów w środku listy związanej wymaga przeszukiwania listy do odpowiedniego indeksu, co może być czasochłonne, zwłaszcza w przypadku dużych zbiorów danych. Mimo tego, lista dwukierunkowa oferuje większą elastyczność w porównaniu do dynamicznych tablic, które wymagają większej ilości operacji w celu utrzymania struktury po dodaniu lub usunięciu elementów.

W kontekście wydajności, istotne jest zrozumienie, że podczas gdy operacje na końcu lub początku listy są bardzo szybkie, działania w środku wymagają podejścia do struktury danych. W takich przypadkach zastosowanie dodatkowych strategii, takich jak zrównoważone drzewo wyszukiwania lub tablice mieszające, może pomóc zminimalizować czas potrzebny na wyszukiwanie odpowiednich elementów. Zrozumienie tych zależności jest kluczowe dla projektowania efektywnych aplikacji.

5.3. Zarządzanie pamięcią

- Implementacja dwukierunkowej listy związanej wymagała starannego zarządzania pamięcią. Kluczowym aspektem była konieczność zwalniania pamięci przy usuwaniu elementów, aby uniknąć wycieków pamięci. Projekt pomógł w lepszym zrozumieniu roli wskaźników oraz destruktorów w C++.

Zarządzanie pamięcią jest szczególnie ważne w kontekście aplikacji o długotrwałym działaniu, takich jak serwery internetowe czy systemy operacyjne, gdzie efektywne wykorzystanie zasobów ma kluczowe znaczenie dla stabilności i wydajności. Implementacja mechanizmów automatycznego zwalniania pamięci, takich jak smart pointers w C++, pozwala na znaczące uproszczenie tego procesu i minimalizację ryzyka błędów.

5.4. Analiza złożoności algorytmów

- Podczas implementacji algorytmów oraz ich analizy w formie schematów blokowych zyskano cenne informacje dotyczące złożoności czasowej oraz przestrzennej. Zrozumienie tych aspektów jest kluczowe dla optymalizacji działania struktur danych.

Analiza złożoności algorytmów pozwala programistom lepiej przewidywać, jak różne struktury danych będą zachowywać się w różnych scenariuszach. W przypadku listy dwukierunkowej, operacje dodawania i usuwania mają złożoność

$O(1)$ w przypadku końców, natomiast złożoność wyszukiwania elementu wynosi $O(n)$. Dzięki tym informacjom, programiści mogą podejmować lepsze decyzje dotyczące wyboru odpowiednich struktur danych, co może mieć kluczowe znaczenie w kontekście wydajności aplikacji.

5.5. Zastosowania praktyczne

- Podsumowując, projekt stanowił praktyczne zastosowanie teoretycznej wiedzy z zakresu struktur danych, takich jak listy związane, oraz przyczynił się do rozwinięcia umiejętności programowania w języku C++. Implementacja różnorodnych metod operujących na liście oraz ich wizualizacja za pomocą schematów blokowych ułatwiły zrozumienie działania struktury.

Dwukierunkowa lista związana znajduje zastosowanie w wielu obszarach, od prostych aplikacji po bardziej złożone systemy. Przykłady obejmują aplikacje do zarządzania historią przeglądania, edytory tekstów, które muszą zarządzać cofaniem i powtarzaniem operacji, oraz systemy baz danych, w których efektywne zarządzanie rekordami ma kluczowe znaczenie. Uzyskane wyniki potwierdzają, że dwukierunkowa lista związana jest odpowiednią strukturą danych w aplikacjach wymagających częstego dodawania i usuwania elementów na początku lub końcu zbioru, jak również w sytuacjach, gdzie konieczne jest przeglądanie danych w obu kierunkach.

5.6. Refleksje i przyszłe kierunki rozwoju

- Dzięki realizacji tego projektu uczestnicy zyskali nie tylko praktyczne umiejętności programistyczne, ale również głębsze zrozumienie teoretycznych podstaw, co przyczyni się do ich przyszłego rozwoju w obszarze programowania i inżynierii oprogramowania. Implementacja dwukierunkowej listy związanej jest doskonałym przykładem zastosowania złożonych koncepcji w praktycznych projektach informatycznych.

Przyszłe prace mogą skupić się na rozszerzeniu funkcjonalności listy związanej, na przykład poprzez implementację dodatkowych metod wyszukiwania, sortowania czy filtrowania danych. Można również rozważyć integrację z innymi strukturami danych, aby zwiększyć elastyczność i wydajność aplikacji. W szczególności, badanie złożoności algorytmów w kontekście różnych struktur danych oraz ich wzajemnych interakcji może dostarczyć cennych informacji, które przyczynią się do dalszego rozwoju umiejętności uczestników.

Spis rysunków

Spis tabel

Spis listingów

1.	Struktura wezła	10
2.	Konstruktor klasy	10
3.	Dodawanie elementu na początku listy	11
4.	Dodawanie elementu na końcu listy	11
5.	Dodawanie elementu pod wskazany indeks	12
6.	Usuwanie elementu z początku listy	12
7.	Usuwanie elementu z końca listy	13
8.	Usuwanie elementu pod wskazanym indeksem	13
9.	Wyswietlanie listy od początku	14
10.	Wyswietlanie listy od końca	14
11.	Wyswietlanie następnego elementu	14
12.	Wyswietlanie poprzedniego elementu	15
13.	Usuwanie całej listy	15
14.	Destruktor klasy	15