AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Implementacja listy dwukierunkowej z zastosowaniem GitHub

Autor:
Szymon Wolski

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

1.1. Wstęp

Celem projektu jest stworzenie implementacji listy dwukierunkowej w języku C++, z wykorzystaniem platformy GitHub do zarządzania wersjami kodu. Lista dwukierunkowa to struktura danych pozwalająca na przemieszczanie się po jej elementach w obu kierunkach, czyli zarówno do przodu, jak i do tyłu. Dzięki temu struktura ta umożliwia bardziej elastyczne i wydajne wykonywanie operacji takich jak dodawanie, usuwanie i przeszukiwanie elementów, w porównaniu do list jednokierunkowych. Projekt ma na celu przygotowanie kompletnej implementacji listy dwukierunkowej, z obsługą błędów oraz dokumentacją, która ułatwi korzystanie z tej struktury danych.

1.2. Wymagania funkcjonalne

Wymagania funkcjonalne opisują zachowania, które system powinien realizować, aby spełniać swoje cele. W projekcie implementacji listy dwukierunkowej wymagania funkcjonalne obejmują:

1. **Tworzenie nowej listy dwukierunkowej:**
    - Umożliwienie tworzenia pustej listy dwukierunkowej.
2. **Dodawanie elementów do listy:**
    - Dodanie elementu na początek listy:
      Funkcja dodajNaPoczatek(int wartosc) powinna umożliwiać dodanie nowego elementu na początek listy. W przypadku, gdy lista jest pusta, nowy element staje się zarówno pierwszym, jak i ostatnim elementem listy.
    - Dodanie elementu na koniec listy:
      Funkcja dodajNaKoniec(int wartosc) powinna dodawać nowy element na końcu listy. Jeżeli lista jest pusta, nowy element staje się jednocześnie pierwszym i ostatnim elementem.
    - Dodanie elementu pod określonym indeksem:
      Funkcja dodajPodIndeksem(int indeks, int wartosc) pozwala na dodanie nowego elementu pod wybranym indeksem. Jeśli indeks wynosi 0, element jest dodawany na początku. Jeśli indeks przekracza długość listy, element jest dodawany na końcu.
3. **Usuwanie elementów z listy:**

- Usunięcie elementu z początku listy:
  Funkcja usunZPoczatku() ma umożliwiać usunięcie pierwszego elementu listy. Jeżeli lista jest pusta, funkcja nie powinna wykonywać żadnych operacji.
- Usunięcie elementu z końca listy:
  Funkcja usunZKonca() powinna usuwać ostatni element listy. Gdy lista jest pusta, funkcja nie wykonuje żadnej operacji.
- Usunięcie elementu pod określonym indeksem:
  Funkcja usunPodIndeksem(int indeks) powinna umożliwiać usunięcie elementu na danym indeksie. Jeżeli indeks wynosi 0, usuwany jest pierwszy element. Jeżeli indeks przekracza zakres listy, nie są wykonywane żadne operacje.
- Wyczyszczenie wszystkich elementów listy:
  Funkcja wyczysc() usuwa wszystkie elementy z listy, tak aby po jej wykonaniu lista była pusta.

4. **Wyświetlanie listy:**

- Wyświetlenie elementów od początku do końca:
  Funkcja wyswietl() wyświetla wszystkie elementy listy, zaczynając od pierwszego aż do ostatniego, w porządku rosnącym.
- Wyświetlenie elementów od końca do początku:
  Funkcja wyswietlOdTylu() wyświetla elementy listy w odwrotnej kolejności, zaczynając od ostatniego elementu.
- Wyświetlenie następnego elementu dla podanego węzła:
  Funkcja wyswietlNastepny(Wezel* wezel) pozwala na wyświetlenie wartości kolejnego elementu dla danego węzła. Jeżeli kolejny element nie istnieje, wyświetlana jest informacja o braku elementu.
- Wyświetlenie poprzedniego elementu dla podanego węzła:
  Funkcja wyswietlPoprzedni(Wezel* wezel) wyświetla wartość poprzedniego elementu danego węzła. W przypadku braku poprzedniego elementu, zwracana jest odpowiednia informacja.

Funkcje listy powinny być odpowiednio zabezpieczone przed błędami wynikającymi z niewłaściwego korzystania ze struktury, np. gdy próbuje się usunąć element z pustej listy. Dla każdej operacji użytkownik powinien otrzymać jasne komunikaty w sytuacji niepoprawnych danych wejściowych (jak np. podanie nieprawidłowego

indeksu czy próba dodania elementu do listy, jeśli jest pełna). Wszystkie funkcje powinny działać w sposób czytelny i intuicyjny, zapewniając proste zarządzanie elementami listy poprzez ich dodawanie, usuwanie oraz wyświetlanie.

1.3. Wymagania niefunkcjonalne

1.3.1. Wykorzystanie platformy GitHub

Platforma GitHub będzie wykorzystywana do zarządzania kodem źródłowym, wersjonowania zmian oraz wspólnej pracy nad projektem. Wymagania dotyczące GitHub obejmują:

- **Wersjonowanie kodu:** Wszystkie zmiany w projekcie będą systematycznie dokumentowane i przetrzymywane w repozytorium GitHub, co pozwala na kontrolowanie wersjami projektu i przywracanie wcześniejszych w razie awarri lub innych problemów.

1.3.2. Wydajność

Chociaż lista dwukierunkowa jest stosunkowo prosta w implementacji, ważne jest, aby operacje na niej były wykonane efektywnie. W projekcie należy uwzględnić:

- **Zarządzanie pamięcią:** Lista powinna być zarządzana w sposób efektywny pod względem pamięci i czasu wykonywania operacji, poprzez stosowanie wskaźników do przemieszczania się w obu kierunkach

2. Analiza problemu

Lista dwukierunkowa to jedna z fundamentalnych struktur danych w programowaniu, szczególnie używana w algorytmach, które wymagają sprawnego dodawania i usuwania elementów w dowolnej pozycji listy. Składa się z węzłów, z których każdy zawiera dane oraz dwa wskaźniki: jeden wskazuje na poprzedni węzeł, a drugi na kolejny. Dzięki tej budowie lista pozwala na przemieszczanie się w obu kierunkach – od początku do końca i w przeciwnym kierunku. Lista dwukierunkowa jest szeroko stosowana w wielu algorytmach oraz systemach, takich jak:

- **Implementacja stosów i kolejek:** Lista dwukierunkowa doskonale nadaje się do tworzenia dynamicznych kolejek i stosów, ponieważ umożliwia dodawanie i usuwanie elementów zarówno z początku, jak i z końca struktury.
- **Edytory tekstów:** W edytorach tekstu lista dwukierunkowa jest wykorzystywana do przechowywania i modyfikowania tekstu, co pozwala na sprawne przemieszczanie się po treści oraz szybkie wprowadzanie zmian.
- **Zarządzanie pamięcią:** Algorytmy zarządzania pamięcią, na przykład te stosowane do zarządzania blokami pamięci w systemach operacyjnych, często wykorzystują listę dwukierunkową, aby sprawnie przydzielać i zwalniać zasoby pamięci.
- **Przechodzenie przez elementy w obu kierunkach:** Lista dwukierunkowa sprawdza się w aplikacjach wymagających nawigacji w obu kierunkach, takich jak listy, które muszą umożliwiać łatwe przemieszczanie się po danych zarówno w przód, jak i w tył.

2.1. Sposób działania algorytmu

Algorytm działania listy dwukierunkowej jest prosty, ale wymaga staranności w zarządzaniu wskaźnikami. Podstawowe operacje to:

- **Dodawanie elementu:** Dodawanie elementu do listy dwukierunkowej może odbywać się na początku, na końcu lub w dowolnej pozycji listy. W zależności od lokalizacji, odpowiednie wskaźniki w węzłach są aktualizowane: wskaźnik poprzedniego węzła wskazuje na nowo dodany element, a wskaźnik następnego węzła prowadzi do kolejnego elementu.
- **Usuwanie elementu:** Usuwanie elementu polega na modyfikacji wskaźników sąsiednich węzłów. Gdy usuniemy element z początku listy, wskaźnik wskazujący na pierwszy element jest aktualizowany, aby wskazywał na kolejny element. Natomiast przy usuwaniu elementu ze środka lub końca listy, wskaźniki poprzedniego i następnego węzła są zmieniane, aby utrzymać ciągłość powiązań między pozostałymi elementami listy.
- **Przechodzenie po liście:** Operacja przechodzenia przez listę pozwala na poruszanie się w obu kierunkach. Dzięki wskaźnikom prev (poprzedni) i next (następny), możliwe jest rozpoczęcie przeglądania listy od pierwszego elementu do ostatniego, lub odwrotnie – od ostatniego do pierwszego.
- **Zarządzanie pamięcią:** W C++ lista dwukierunkowa wymaga dokładnego zarządzania pamięcią dynamiczną. Elementy listy są tworzone i usuwane w czasie działania programu, a wskaźniki muszą być poprawnie modyfikowane, aby zapobiec wyciekom pamięci.

Algorytm listy dwukierunkowej znajduje szerokie zastosowanie w praktyce, zwłaszcza w aplikacjach, które wymagają elastyczności w manipulacji danymi. Dzięki możliwościom dodawania, usuwania oraz przechodzenia po elementach listy w obu kierunkach, jest to struktura o dużej użyteczności, szczególnie w przypadkach takich jak:

- **Implementacja edytorów tekstu:** Listy dwukierunkowe znajdują zastosowanie w edytorach tekstu, umożliwiając użytkownikowi nawigację zarówno do przodu, jak i do tyłu wprowadzonymi danymi.
- **Zarządzanie pamięcią w systemach operacyjnych:** Algorytmy zarządzania pamięcią często wykorzystują listy dwukierunkowe do efektywnego dodawania i usuwania bloków pamięci, co pozwala na szybkie i elastyczne zarządzanie przydziałem pamięci.
- **Implementacja struktur danych:** Lista dwukierunkowa stanowi fundament dla bardziej złożonych struktur danych, takich jak kolejki i stosy, które wymagają operacji na danych zarówno z jednej, jak i z drugiej strony.

Algorytmy wykorzystujące listy dwukierunkowe oferują dużą elastyczność, szczególnie w środowiskach, gdzie operacje na danych muszą być szybkie i efektywne.

3. Projektowanie

3.1. Działanie listy dwukierunkowej – teoria

W strukturze danych typu lista dwukierunkowa, każdy element (zwany *węzłem*) zawiera trzy główne składniki:

- **Wartość elementu** – przechowywana węźle, która może być dowolnego typu (np. liczba całkowita, tekst, obiekt).
- **Wskaźnik na poprzedni element** – wskazuje na element poprzedzający bieżący węzeł w liście.
- **Wskaźnik na następny element** – wskazuje na element następujący po bieżącym węźle w liście.

Lista dwukierunkowa umożliwia poruszanie się zarówno w przód, jak i w tył. Oto jak działają podstawowe operacje na liście dwukierunkowej:

- **Dodawanie elementów**: Można dodawać elementy na początku, na końcu lub w określonym miejscu w liście. W zależności od miejsca, aktualizowane są wskaźniki poprzedniego i następnego elementu.
- **Usuwanie elementów**: Elementy mogą być usuwane zarówno z początku, jak i z końca listy, a także z dowolnej pozycji. Podczas usuwania elementu, wskaźniki są odpowiednio aktualizowane, aby nie utracić połączeń między węzłami.
- **Przemieszczanie się po liście**: Listy dwukierunkowe umożliwiają przeglądanie elementów w obu kierunkach. Można zacząć od początku i przechodzić do końca, lub zacząć od końca i wrócić na początek.
- **Zarządzanie pamięcią**: Operacje dodawania i usuwania elementów wymagają odpowiedniego zarządzania pamięcią, by uniknąć wycieków pamięci.

W przypadku implementacji w języku C++, wszystkie te operacje są wykonywane za pomocą wskaźników (ang. *pointers*), które pozwalają na dynamiczne zarządzanie pamięcią. Ponieważ lista dwukierunkowa przechowuje wskaźniki na elementy w obu kierunkach, dodanie i usunięcie elementu odbywa się w czasie stałym, co sprawia, że jest to wydajna struktura danych.

3.2. Wykorzystane narzędzia

Do realizacji projektu zastosowano następujące narzędzia i technologie:

1. **Język C++**: Jest to język programowania ogólnego przeznaczenia, który umożliwia zarządzanie pamięcią oraz łatwą implementację struktur danych takich jak lista dwukierunkowa. C++ oferuje duży zakres możliwości, w tym wskaźniki, klasy, oraz mechanizmy do zarządzania dynamiczną pamięcią.

2. **Kompilator g++ i MinGW**: Kompilator g++ jest jednym z najczęściej używanych kompilatorów dla języka C++, oferując wsparcie dla wielu platform i systemów operacyjnych. MinGW (Minimalist GNU for Windows) to środowisko umożliwiające kompilację kodu C++ na systemie Windows, zapewniając dostęp do bibliotek i narzędzi GNU.

3. **Git**: Git jest systemem kontroli wersji, który umożliwia zarządzanie historią zmian w projekcie. Używanie Gita pozwala na łatwe śledzenie postępu prac, tworzenie gałęzi do testowania nowych funkcji oraz współpracę w zespole.

4. **GitHub**: GitHub jest platformą internetową, która umożliwia przechowywanie i udostępnianie kodu źródłowego w repozytoriach Git. Jest to idealne miejsce do pracy zespołowej oraz publikowania gotowych projektów.

3.3. Ustawienia kompilatora

Do kompilacji projektu użyto środowiska MinGW w połączeniu z kompilatorem g++. Kompilator został skonfigurowany do pracy z systemem Windows, aby zapewnić kompatybilność z kodem C++ oraz odpowiednią obsługę wskaźników i zarządzania pamięcią.

3.4. Git i GitHub

Do zarządzania wersjami projektu użyto systemu kontroli wersji Git oraz platformy GitHub. Jako że projekt był realizowany jednoosobowo, głównie korzystano z Git Desktop w celu zarządzania historią zmian oraz organizowania pracy w ramach różnych gałęzi.

Praca jednoosobowa z gałęziami: W trakcie rozwoju projektu wykorzystywane były dodatkowa gałąź, co pozwoliło na eksperymentowanie z nowymi funkcjonalnościami, testowanie poprawek i wdrażanie ich do głównej gałęzi. Proces ten obejmował następujące kroki:

1. **Tworzenie gałęzi**: Na początku każdej nowej funkcjonalności comittowałem zmianny na gałęzi a potem pushowałęm je do repozytorium przy użyciu narzędzia Github Desktop
2. **Praca na gałęzi**: W trakcie pracy na gałęzi wprowadzałem odpowiednie zmiany w kodzie, testowałem je oraz sprawdzałem poprawność implementacji. Wszystkie zmiany były zapisywane poprzez pushowanie zmian na gałąź
3. **Przesyłanie zmian na GitHub**: Po zakończeniu wszystkich prac nad nowymi funkcjami, zmiany były przesyłane na GitHub, aby zachować kopię projektu w chmurze oraz móc do niego wrócić w przyszłości.
4. Usunięcie commita i cofanie
    - Niestety podczas pracy nad projektem nie udało mi się usunąć oraz cofnąć commitów w moim repozytorium

GitHub pełnił rolę repozytorium, w którym przechowywane były wszystkie zmiany i wersje projektu. Regularne synchronizowanie projektu z GitHubem pozwalało na tworzenie kopii zapasowych oraz umożliwiało wygodną organizację pracy z kodem, nawet w przypadku pracy jednoosobowej.

4. Implementacja

4.1. Konstruktory

4.1.1. Konstruktor klasy ListaDwukierunkowa

```
ListaDwukierunkowa() : pierwszy(nullptr), ostatni(nullptr) {}
```

Listing 1. Konstruktor klasy ListaDwukierunkowa.

Konstruktor klasy, który inicjalizuje pustą listę dwukierunkową, ustawiając wskaźniki pierwszy i ostatni na nullptr.

4.1.2. Destruktor klasy ListaDwukierunkowa

```
~ListaDwukierunkowa() {
    wyczysc();
}
```

Listing 2. Destruktor klasy ListaDwukierunkowa.

Destruktor, który wywołuje metodę wyczysc(), aby usunąć wszystkie węzły z listy i zwolnić pamięć.

4.1.3. Konstruktor klasy Wezel

```
Wezel(int wartosc) : dane(wartosc), poprzedni(nullptr), nastepny(
    nullptr) {}
```

Listing 3. Konstruktor klasy Wezel.

Tworzy nowy węzeł z wartością wartosc oraz ustawia wskaźniki poprzedni i nastepny na nullptr. Jest to konstruktor inicjalizujący nowy węzeł.

4.2. Metody

4.2.1. Metoda `dodajNaPoczatek`

```
void dodajNaPoczatek(int wartosc) {
    Wezel* nowyWezel = new Wezel(wartosc);
    if (!pierwszy) {
        pierwszy = ostatni = nowyWezel;
    } else {
        nowyWezel->nastepny = pierwszy;
        pierwszy->poprzedni = nowyWezel;
        pierwszy = nowyWezel;
    }
```

```
}
```

Listing 4. Metoda dodajNaPoczatek.

Tworzy nowy węzeł i ustawia go jako pierwszy element listy. W przypadku, gdy lista jest pusta, nowy węzeł staje się zarówno pierwszym, jak i ostatnim elementem listy.

4.2.2. Metoda `dodajNaKoniec`

```
void dodajNaKoniec(int wartosc) {
    Wezel* nowyWezel = new Wezel(wartosc);
    if (!ostatni) {
        pierwszy = ostatni = nowyWezel;
    } else {
        nowyWezel->poprzedni = ostatni;
        ostatni->nastepny = nowyWezel;
        ostatni = nowyWezel;
    }
}
```

Listing 5. Metoda dodajNaKoniect.

Tworzy nowy węzeł i ustawia go jako ostatni element listy. W przypadku pustej listy nowy węzeł staje się zarówno pierwszym, jak i ostatnim elementem listy.

4.2.3. Metoda `dodajPodIndeksem`

```
void dodajPodIndeksem(int indeks, int wartosc) {
    if (indeks == 0) {
        dodajNaPoczatek(wartosc);
        return;
    }

    Wezel* aktualny = pierwszy;
    int aktualnyIndeks = 0;
    while (aktualny && aktualnyIndeks < indeks) {
        aktualny = aktualny->nastepny;
        aktualnyIndeks++;
    }
    if (!aktualny) {
        dodajNaKoniec(wartosc);
    } else {
        Wezel* nowyWezel = new Wezel(wartosc);
        nowyWezel->poprzedni = aktualny->poprzedni;
        nowyWezel->nastepny = aktualny;
        if (aktualny->poprzedni) {
```

```
20              aktualny->poprzedni->nastepny = nowyWezel;
21          }
22          aktualny->poprzedni = nowyWezel;
23          if (aktualny == pierwszy) pierwszy = nowyWezel;
24      }
25  }
```

Listing 6. Metoda dodajPodIndeksem.

Jeśli indeks wynosi 0, element jest dodawany na początek. Jeśli element ma zostać dodany w środku, lista jest przechodzona, aby znaleźć odpowiednią pozycję i tam wstawia nowy element. Jeśli indeks przekroczy liczbę elementów w liście, element zostanie dodany na koniec.

4.2.4. Metoda `usunZPoczatku`

```
1  void usunZPoczatku() {
2      if (!pierwszy) return;
3
4      Wezel* temp = pierwszy;
5      if (pierwszy == ostatni) {
6          pierwszy = ostatni = nullptr;
7      } else {
8          pierwszy = pierwszy->nastepny;
9          pierwszy->poprzedni = nullptr;
10     }
11     delete temp;
12 }
```

Listing 7. Metoda usunZPoczatku.

Jeśli lista nie jest pusta, usuwa pierwszy węzeł. Jeżeli po usunięciu lista będzie pusta, oba wskaźniki pierwszy i ostatni są ustawiane na nullptr.

4.2.5. Metoda `usunZKonca`

```
1  void usunZKonca() {
2      if (!ostatni) return;
3
4      Wezel* temp = ostatni;
5      if (pierwszy == ostatni) {
6          pierwszy = ostatni = nullptr;
7      } else {
8          ostatni = ostatni->poprzedni;
9          ostatni->nastepny = nullptr;
10     }
11     delete temp;
```

```
}
```

Listing 8. Metoda usunZKonca.

Jeśli lista nie jest pusta, usuwa ostatni węzeł. Jeśli lista po usunięciu będzie pusta, oba wskaźniki pierwszy i ostatni są ustawiane na nullptr.

4.2.6. Metoda `usunPodIndeksem`

```
void usunPodIndeksem(int indeks) {
    if (indeks == 0) {
        usunZPoczatku();
        return;
    }

    Wezel* aktualny = pierwszy;
    int aktualnyIndeks = 0;

    while (aktualny && aktualnyIndeks < indeks) {
        aktualny = aktualny->nastepny;
        aktualnyIndeks++;
    }

    if (!aktualny) return;  // Index out of bounds

    if (aktualny->poprzedni) aktualny->poprzedni->nastepny =
   aktualny->nastepny;
    if (aktualny->nastepny) aktualny->nastepny->poprzedni =
   aktualny->poprzedni;

    if (aktualny == pierwszy) pierwszy = aktualny->nastepny;
    if (aktualny == ostatni) ostatni = aktualny->poprzedni;

    delete aktualny;
}
```

Listing 9. Metoda usunPodIndeksem.

Jeśli indeks to 0, element jest usuwany z początku. W przeciwnym przypadku, lista jest przechodzona, a odpowiedni węzeł jest usuwany, aktualizując wskaźniki w sąsiednich węzłach.

4.2.7. Metoda `wyswietl`

```
void wyswietl() {
    if (!pierwszy) {
        cout << "Lista jest pusta." << endl;
        return;
    }
    Wezel* aktualny = pierwszy;
    while (aktualny) {
        cout << aktualny->dane << " ";
        aktualny = aktualny->nastepny;
    }
    cout << endl;
}
```

Listing 10. Metoda wyswietl.

Przechodzi przez listę, wypisując dane każdego węzła.

4.2.8. Metoda `wyswietlodTylu`

```
void wyswietlOdTylu() {
    if (!ostatni) {
        cout << "Lista jest pusta." << endl;
        return;
    }
    Wezel* aktualny = ostatni;
    while (aktualny) {
        cout << aktualny->dane << " ";
        aktualny = aktualny->poprzedni;
    }
    cout << endl;
}
```

Listing 11. Metoda wyswietlodTylu.

Przechodzi przez listę od końca, wypisując dane każdego węzła.

4.2.9. Metody `wyswietlNastepnyiPoprzedni`

```
void wyswietlNastepny(Wezel* wezel) {
    if (wezel && wezel->nastepny) {
        cout << "Nastepny element: " << wezel->nastepny->dane <<
   endl;
    } else {
        cout << "Brak nastepnego elementu." << endl;
    }
}
```

```
8
9  void wyswietlPoprzedni(Wezel* wezel) {
10     if (wezel && wezel->poprzedni) {
11         cout << "Poprzedni element: " << wezel->poprzedni->dane <<
       endl;
12     } else {
13         cout << "Brak poprzedniego elementu." << endl;
14     }
15 }
```

Listing 12. Metody wyswietlNastepnyiPoprzedni.

Wyświetlają dane następnego lub poprzedniego węzła (jeśli istnieją), a jeśli sąsiedzi nie istnieją, wypisują odpowiedni komunikat.

4.2.10. Metoda `wyczysc`

```
1  void wyczysc() {
2      while (pierwszy) {
3          usunZPoczatku();
4      }
5  }
```

Listing 13. Metoda wyczysc.

Usuwa wszystkie elementy listy, ustawiając wskaźniki pierwszy i ostatni na nullptr.

5. Wnioski

Po zakończeniu implementacji listy dwukierunkowej w języku C++ można wyciągnąć następujące wnioski:

1. **Zrozumienie struktury danych:** Realizacja projektu pozwoliła na głębsze zrozumienie zasad działania listy dwukierunkowej. Dzięki przechowywaniu wskaźników na poprzedni i następny element w każdym węźle, możliwe jest efektywne przechodzenie po liście w obu kierunkach, co stanowi istotną przewagę nad listami jednokierunkowymi.

2. **Zarządzanie pamięcią:** Projekt wymagał precyzyjnego zarządzania pamięcią dynamiczną, szczególnie w kontekście tworzenia i usuwania elementów listy. Wykorzystanie wskaźników w C++ pozwoliło na pełną kontrolę nad pamięcią, ale wymagało także staranności w każdej operacji alokacji i zwalniania pamięci, aby uniknąć wycieków pamięci.

3. **Skalowalność:** Implementacja listy dwukierunkowej pokazała, że jest to elastyczna struktura, która łatwo może być rozbudowywana o dodatkowe operacje. Możliwość łatwego dodawania nowych funkcji, takich jak przeszukiwanie elementów czy zaawansowane operacje (np. sortowanie), pozwala na rozbudowę listy w miarę potrzeb projektu.

4. **Wykorzystanie GitHub:**Platforma GitHub okazała się pomocna w zarządzaniu wersjami kodu i śledzeniu postępów w projekcie. Umożliwiła także wygodne przechowywanie dokumentacji i współpracę nad kodem, co usprawniło proces tworzenia oprogramowania.

5. **Bezpieczeństwo operacji:** W trakcie implementacji zwrócono szczególną uwagę na zabezpieczenie funkcji przed błędami wynikającymi z nieprawidłowego użytkowania. Na przykład, operacje takie jak usuwanie elementu z pustej listy czy próba dostępu do nieistniejącego indeksu były odpowiednio obsługiwane przez generowanie komunikatów o błędach, co poprawiło niezawodność aplikacji.

6. **Użyteczność i dokumentacja:** Gotowy program, mimo swojej prostoty, okazał się funkcjonalny i łatwy w użyciu. Dzięki przejrzystej dokumentacji, użytkownicy mogli łatwo dodawać, usuwać i przeglądać elementy listy. Dodatkowo, kod został dobrze udokumentowany, co zwiększyło jego zrozumiałość i przyswajalność.

7. **Możliwości rozwoju:** Choć projekt realizuje podstawowe operacje na liście dwukierunkowej, istnieje wiele obszarów, w których można by go rozszerzyć. Przykładem może być dodanie funkcji do sortowania listy, wykrywania cykli w strukturze czy implementacja bardziej zaawansowanych funkcji, jak wskaźnik na środek listy, co umożliwiłoby szybszy dostęp do elementów.

8. **Zastosowanie w praktyce:** Lista dwukierunkowa znajduje szerokie zastosowanie w rzeczywistych aplikacjach informatycznych. Jest wykorzystywana m.in. w edytorach tekstów, systemach pamięci podręcznej, bazach danych czy innych systemach, w których istnieje potrzeba efektywnego zarządzania dynamicznymi danymi i umożliwia łatwy dostęp do zarówno poprzednich, jak i następnych elementów.

Podsumowując, implementacja listy dwukierunkowej stanowiła cenne ćwiczenie z zakresu algorytmiki, zarządzania pamięcią oraz współpracy z narzędziami do wersjonowania kodu. Projekt pozwolił na lepsze zrozumienie działania struktur danych, które są fundamentem wielu algorytmów i aplikacji w informatyce.

Spis rysunków

Spis listingów