AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

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

Autor:
Wiktoria Falowska

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem pracy jest stworzenie dwukierunkowej listy w języku C++. Lista powinna mieć możliwość wykonywania operacji dodawania, usuwania elementów. Dodatkowo, lista ma wbudowaną funkcję pokazywania zawartości listy w obu kierunkach oraz jej całkowite wyczyszczenie.

Przewidywane wyniki pracy to:

- Stworzenie klasy odpowiedzialnej za działanie listy dwukierunkowej,
- Możliwość dynamicznego zarządzania elementami listy,
- Implementacja funkcji obsługujących dodawanie i usuwanie elementów,
- Wyświetlanie zawartość listy od pierwszego i od ostatniego elementu,
- Zarządzanie pamięcią dzięki czyszczeniu listy.

2. Analiza problemu

Dwukierunkowa lista jest jednym z podstawowych algorytmów używanych w różnych dziedzinach informatyki. Dzięki swojej strukturze umożliwia szybki dostęp do elementów zarówno od początku, jak i od końca listy, co jest szczególnie przydatne w wielu zastosowaniach, takich jak:

2.1. Opis działania programu

Program z dwukierunkową listą składa się z klasy Node - węzeł oraz metod zarządzających listą. Każdy węzeł przechowuje wartość oraz wskaźniki na poprzedni i następny element listy. Klasa DoublyLinkedLIst zawiera metody umożliwiające:

- Dodawanie elementów
- Usuwanie elementów
- Wyświetlanie listy
- Czyszczenie całej listy.

2.2. Przykład ręcznego mnożenia macierzy

Mamy macierz A o wymiarach 1×3 oraz macierz B o wymiarach 3×1. macierzą C będzie macierz o wymiarach 1×1.

$$A = \begin{pmatrix} 2 & 4 & 6 \end{pmatrix}, \quad B = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix}$$

Macierz C równa się iloczynowi wiersza macierzy A z kolumną macierzy B.

Obliczenia:

$$C_{11} = (2 \times 1) + (4 \times 3) + (6 \times 5) = 2 + 12 + 30 = 44$$

Macierz C:

$$C = \begin{pmatrix} 44 \end{pmatrix}$$

Możemy wykonać to za pomocą DoublyLinkedList przechodząc po dwóch listach, gdzie mnożymy aktualne elementy, po czym idziemy do następnych i mnożymy je. Gdy listy A=[2,4,6] B=[[1],[3],[5]], wykonujemy operacje:

$$C_{11} = (2 \times 1) + (4 \times 3) + (6 \times 5) = 2 + 12 + 30 = 44$$

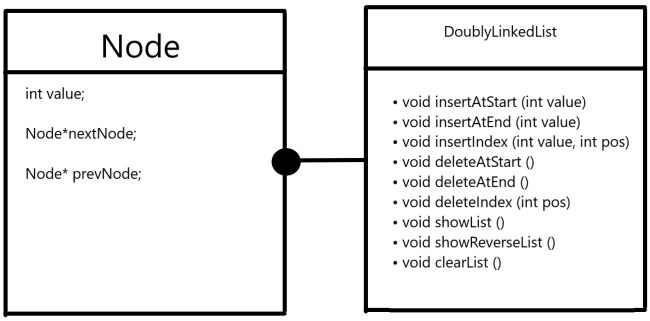
3. Projektowanie

3.1. Narzędzia i technologie

- **Kompilator:** g++ - Kompilator języka C++
- **Język programowania:** C++
- **VSC:** Git - Do zarządzania wersjami kodu
- **GitHub** - Do przechowywania kolejnych wersji kodu
- **Doxygen:** Do automatycznego generowania dokumentacji w formacie LateX

3.2. Diagram klas

Klasa `Node` reprezentuje pojedynczy element listy, a klasa `DoublyLinkedList` zarządza całą listą.



Rys. 3.1. Diagram klas dla listy dwukierunkowej

4. Implementacja

4.1. Opis implementacji

Implementacja struktury danych *doubly linked list* (lista dwukierunkowa), która umożliwia zarządzanie danymi poprzez dodawanie, usuwanie i przeglądanie elementów.

4.2. Ciekawe fragmenty kodu

4.2.1. Struktura węzła listy

Węzeł listy oznaczany jest przez klasę `Node`, która zawiera zmienną `value` przechowujące wartość oraz dwa wskaźniki: `nextNode` i `prevNode` na następny i poprzedni element.

```
class Node {
public:
    int value;
    Node* nextNode;
    Node* prevNode;

    Node(int value) {
        this->value = value;
        this->nextNode = nullptr;
        this->prevNode = nullptr;
    }
};
```

4.2.2. Dodawanie elementu na początku listy

Aby dodać element na pierwszym miejscu, musimy stworzyć nowy węzeł, a następnie zaaktaulizowanie węzłów w taki sposób, aby struktura dalej była poprawna

```
// Dodawanie elementu na początku listy
void insertAtStart(Node*& head, int value) {
    Node* newNode = new Node(value);

    if (head == nullptr) {
```

```
        head = newNode;
        return;
    }

    newNode->nextNode = head;
    head->prevNode = newNode;
    head = newNode;
}
```

4.2.3. Dodawanie elementu na końcu listy

Aby dodać element na końcu listy trzeba przejść przez całą listę, a następnie wstawić nowy węzeł na końcu

```
// Dodawanie elementu na końcu listy
void insertAtEnd(Node*& head, int value) {
    Node* newNode = new Node(value);

    if (head == nullptr) {
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->nextNode != nullptr) {
        temp = temp->nextNode;
    }

    temp->nextNode = newNode;
    newNode->prevNode = temp;
}
```

4.2.4. Usuwanie elementu na wskazanym indeksie

Aby usunąć element na danym indeksie trzeba do niego przejść od początku listy po czym usunąć go i zaaktualizować wskaźniki tak aby dalej lista działała

```
// Usuwanie elementu z podanego indeksu
void deleteIndex(Node*& head, int pos) {
```

```
    if (head == nullptr) {
        cout << "Lista jest pusta" << endl;
        return;
    }

    if (pos == 0) {
        deleteAtStart(head);
        return;
    }

    Node* temp = head;
    for (int i = 0; temp != nullptr && i < pos; i++) {
        temp = temp->nextNode;
    }

    if (temp == nullptr) {
        cout << "Zły indeks" << endl;
        return;
    }

    if (temp->nextNode != nullptr) {
        temp->nextNode->prevNode = temp->prevNode;
    }
    if (temp->prevNode != nullptr) {
        temp->prevNode->nextNode = temp->nextNode;
    }
    delete temp;
}
```

4.2.5. Wyświetlanie listy

Metoda wyświetlająca całą listę przechodzi przez wszystkie Nody od początku do końca

```
// Wyświetlanie listy
void showList(Node* head) {
    Node* temp = head;
    if (temp == nullptr) {
```

```
        cout << "Lista jest pusta" << endl;
        return;
    }
    cout << "Lista: ";
    while (temp != nullptr) {
        cout << temp->value << " ";
        temp = temp->nextNode;
    }
    cout << endl;
}
```

4.3. Wyniki działania programu

```
Dodajmy element na początku:
Lista: 1
Dodajmy element na końcu:
Lista: 1 3
Dodajmy element na indeks 1:
Lista: 1 2 3
Wyswietlmy liste na odwrot:
Lista odwrotnie: 3 2 1
Dodajmy jeszcze kilka wartości:
Lista: 1 2 3 6 13 10 0
Usunmy element na indeksie 4:
Lista: 1 2 3 6 10 0
Usunmy pierwszy element:
Lista: 2 3 6 10 0
Usunmy ostatni element:
Lista: 2 3 6 10
Czyscimy całą listę:
Lista jest pusta
```

Jak widać, algorytm działa poprawnie

5. Wnioski

Implementacja listy dwukierunkowej (*doubly linked list*) w języku C++ pokazała, jak efektywnie można zarządzać strukturami danych, umożliwiając dodawanie, usuwanie i przeglądanie elementów.

5.1. Największe zalety:

- Dynamiczny rozmiar
- Łatwość wstawiania/usuwania

5.2. Wady:

- Zajmuje więcej pamięci z uwagi na przechowywany wskaźnik do kolejnego ogniwa w szeregu.

Spis rysunków

Spis tabel

Spis listingów