AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

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

Autor:
Adrian Kądziołka

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2024

Spis treści

1. Ogólne określenie wymagań

Celem projektu jest opracowanie programu w języku C++, który implementuje dwukierunkową listę wiązaną, zarządzaną dynamicznie na stercie. Program ma umożliwiać przeprowadzanie różnych operacji na liście, takich jak dodawanie i usuwanie elementów, wyświetlanie zawartości listy w obu kierunkach oraz realizację innych funkcji zgodnie z założeniami specyfikacji.

Oczekiwane rezultaty obejmują prawidłowe działanie wszystkich metod klasy, w tym:

- Dodawanie i usuwanie elementów na początku oraz końcu listy.
- Dodawanie i usuwanie elementów pod wskazanym indeksem.
- Wyświetlanie całej listy oraz jej odwrotności.
- Czyszczenie listy.

Funkcjonalności programu obejmują:

- Testowanie wszystkich metod w funkcji main.
- Utworzenie dokumentacji za pomocą narzędzia Doxygen.
- Wykorzystanie GitHub do zarządzania projektem i kontroli wersji, w tym wykonanie co najmniej 5 commitów, cofanie zmian oraz synchronizację projektu między różnymi lokalizacjami.

Zakładane efekty realizacji projektu to:

- Praktyczne doskonalenie umiejętności programowania w C++, ze szczególnym uwzględnieniem zarządzania pamięcią.
- Zdobycie doświadczenia w korzystaniu z narzędzi Git i GitHub do zarządzania wersjami.
- Stworzenie kompletnej dokumentacji technicznej projektu.

2. Analiza problemu

2.1. Zastosowanie dwukierunkowej listy wiązanej

Dwukierunkowa lista wiązana to struktura danych, powszechnie stosowana w programowaniu, zwłaszcza tam, gdzie wymagane jest efektywne dodawanie i usuwanie elementów z dowolnej pozycji w liście. Algorytm ten jest wykorzystywany w:

- Implementacja struktur danych, takich jak kolejki priorytetowe, deki (double-ended queues) i stosy.
- Systemy operacyjne, gdzie są wykorzystywane do zarządzania procesami lub zasobami.
- Programowanie gier, służąc do zarządzania listami obiektów lub historią zdarzeń.

2.2. Opis działania

Dwukierunkowa lista wiązana składa się z węzłów, z których każdy przechowuje dane oraz dwa wskaźniki: jeden wskazujący na kolejny węzeł i drugi na poprzedni. Umożliwia to poruszanie się po liście w obu kierunkach. Program oferuje następujące funkcjonalności:

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

2.3. Wykorzystanie narzędzia Git

W projekcie zastosowano system kontroli wersji Git, który umożliwia efektywne zarządzanie wersjami kodu oraz ułatwia współpracę zespołową. Git jest powszechnie używanym narzędziem w programowaniu, oferującym:

- 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

3. Projektowanie

3.1. Narzędzia użyte w projekcie

W tym projekcie korzystano z następujących narzędzi:

- Język programowania: C++
- Środowisko programowania: Visual Studio 2022
- System kontroli wersji: Git
- Generowanie dokumentacji: Doxygen

3.2. Wykorzystanie Git'a w projekcie

W trakcie realizacji projektu wykorzystano system kontroli wersji Git do zarządzania kodem źródłowym oraz synchronizacji z repozytorium na platformie GitHub.

- 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

4.1. Implementacja metody AddToStart

```
/**
 * @brief Dodanie elementu na pocz tek listy.
 *
 * @param value Warto Ż  elementu do dodania na pocz tek listy.
 */
void addToStart(int value) {
    Node* newNode = new Node(value);
    if (!head) {
        head = tail = newNode;
    }
    else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
}
```

Listing 1. Implementacja metody AddToStart

4.2. Implementacja metody AddToEnd

```
/**
 * @brief Dodanie elementu na koniec listy.
 *
 * @param value Warto Ż  elementu do dodania na koniec listy.
 */
void addToEnd(int value) {
    Node* newNode = new Node(value);
    if (!tail) {
        head = tail = newNode;
    }
    else {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
}
```

Listing 2. Implementacja metody AddToEnd

4.3. Implementacja metody AddAtIndex

```
/**
 * @brief Dodanie elementu pod wskazany indeks.
 *
 * @param index Indeks, pod kt ry ma zosta  dodany element.
 * @param value Warto Ż  elementu do dodania.
 */
void addAtIndex(int index, int value) {
    if (index <= 0) {
        addToStart(value);
        return;
    }

    Node* newNode = new Node(value);
    Node* temp = head;
    int currentIndex = 0;

    while (temp && currentIndex < index) {
        temp = temp->next;
        currentIndex++;
    }

    if (!temp) {
        addToEnd(value);
    }
    else {
        newNode->next = temp;
        newNode->prev = temp->prev;
        if (temp->prev) {
            temp->prev->next = newNode;
        }
        temp->prev = newNode;
    }
}
```

Listing 3. Implementacja metody AddAtIndet

4.4. Implementacja metody RemoveFromStart

```
/**
 * @brief Usu  element z pocz tku listy.
 */
void removeFromStart() {
    if (!head) return;
```

```
6
7       Node* temp = head;
8       head = head->next;
9       if (head) {
10          head->prev = nullptr;
11      }
12      else {
13          tail = nullptr;
14      }
15      delete temp;
16  }
```

Listing 4. Implementacja metody RemoveFromStart

4.5. Implementacja metody RemoveFromEnd

```
1   /**
2    * @brief Usu  element z ko ca listy.
3    */
4   void removeFromEnd() {
5       if (!tail) return;
6
7       Node* temp = tail;
8       tail = tail->prev;
9       if (tail) {
10          tail->next = nullptr;
11      }
12      else {
13          head = nullptr;
14      }
15      delete temp;
16  }
```

Listing 5. Implementacja metody RemoveFromEnd

4.6. Implementacja metody DeleteAtIndex

```
1   /**
2    * @brief Usu  element pod wskazanym indeksem.
3    *
4    * @param index Indeks elementu do usuni Źcia.
5    */
6   void deleteAtIndex(int index) {
7       if (index < 0 || !head) return;
```

```
8
9       if (index == 0) {
10          removeFromStart();
11          return;
12      }
13
14      Node* temp = head;
15      int currentIndex = 0;
16
17      while (temp && currentIndex < index) {
18          temp = temp->next;
19          currentIndex++;
20      }
21
22      if (!temp) return;
23
24      if (temp->prev) temp->prev->next = temp->next;
25      if (temp->next) temp->next->prev = temp->prev;
26
27      if (temp == tail) tail = temp->prev;
28
29      delete temp;
30  }
```

Listing 6. Implementacja metody DeleteAtIndex

4.7. Implementacja metody Read

```
1  /**
2   * @brief Wyświetla całą listę od początku do końca.
3   */
4  void read() const {
5      Node* temp = head;
6      while (temp) {
7          cout << temp->data << " ";
8          temp = temp->next;
9      }
10     cout << endl;
11 }
```

Listing 7. Implementacja metody Read

4.8. Implementacja metody ReadReverse

```
/**
 * @brief Wyświetl listę w odwrotnej kolejności.
 */
void readReverse() const {
    Node* temp = tail;
    while (temp) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}
```

Listing 8. Implementacja metody ReadReverse

4.9. Implementacja metody printNextElement

```
/**
 * @brief Wyświetl następny element.
 */
void printNextElement() {
    if (!current) current = head;
    else if (current->next) current = current->next;
    else current = head; // Powrót do początku, jeśli koniec
    zostaĆ osiągnięty.

    if (current) cout << "Nastepny element: " << current->data <<
    endl;
}
```

Listing 9. Implementacja metody printNextElement

4.10. Implementacja metody printPrevElement

```
/**
 * @brief Wyświetl poprzedni element.
 */
void printPrevElement() {
    if (!current) current = tail;
    else if (current->prev) current = current->prev;
    else current = tail; // Powrót do końca, jeśli początek został ćosi gnięty.

    if (current) cout << "Poprzedni element: " << current->data << endl;
}
```

Listing 10. Implementacja metody printPrevElement

4.11. Implementacja metody clearList

```
/**
 * @brief Czyści całą listę.
 */
void clearList() {
    while (head) {
        removeFromStart();
    }
    current = nullptr;
}
```

Listing 11. Implementacja metody clearList

4.12. Destruktor klasy

```
/**
 * @brief Destruktor, który zwalnia zasoby klasy z pamięci
 */
~DoublyLinkedList() {
    head = nullptr;
    tail = nullptr;
    current = nullptr;
}
```

Listing 12. Destruktor klasy

4.13. Int main

```
int main() {
        DoublyLinkedList list;

        list.addToStart(10);
        list.addToEnd(20);
        list.addAtIndex(1, 15);
        list.read();

        list.removeFromStart();
        list.read();

        list.removeFromEnd();
        list.read();

        list.addToEnd(30);
        list.addToEnd(40);
        list.deleteAtIndex(1);
        list.read();

        list.readReverse();

        list.printNextElement();
        list.printNextElement();

        list.printPrevElement();
        list.printPrevElement();

        list.clearList();
        list.read();

        return 0;
}
```

Listing 13. int main

5. Wnioski

W projekcie stworzono dwukierunkową listę wiązaną, która pozwala na sprawne zarządzanie danymi poprzez operacje takie jak dodawanie, usuwanie oraz przeglądanie elementów. Realizacja tego projektu umożliwiła lepsze zrozumienie działania list wiązanych i roli wskaźników w języku C++.

Podczas implementacji listy dwukierunkowej zwrócono uwagę na następujące aspekty:

- Efektywność dodawania i usuwania elementów: Operacje takie jak dodawanie na początek lub koniec listy oraz usuwanie elementów z tych pozycji są bardzo szybkie, ponieważ wymagają jedynie aktualizacji kilku wskaźników.
- Przechodzenie przez listę: Dzięki wskaźnikom next i prev możliwe jest przeglądanie listy w obu kierunkach, co ułatwia realizację bardziej złożonych operacji, takich jak odwracanie listy lub iterowanie od końca do początku.
- Wydajność wstawiania i usuwania elementów w środku listy: Wstawianie i usuwanie elementów w środku listy wymaga przeszukania jej do odpowiedniego indeksu, co może być czasochłonne przy dużych zbiorach danych. Jednak w porównaniu do struktur, takich jak tablice dynamiczne, lista dwukierunkowa zapewnia większą elastyczność.
- Zarządzanie pamięcią: Implementacja listy dwukierunkowej wymagała starannego zarządzania pamięcią, w tym zwalniania jej przy usuwaniu elementów, aby uniknąć wycieków pamięci. Projekt przyczynił się do lepszego zrozumienia roli wskaźników i destruktorów w C++.
- Złożoność algorytmów: Implementacja algorytmów oraz ich analiza poprzez schematy blokowe pozwoliły zrozumieć, jak złożoność czasowa i przestrzenna wpływają na wydajność struktury danych, co jest kluczowe dla optymalizacji.

Podsumowując, projekt umożliwił praktyczne zastosowanie wiedzy teoretycznej dotyczącej struktur danych, takich jak listy wiązane, oraz rozwinięcie umiejętności programowania w języku C++. Implementacja różnych metod operujących na liście oraz ich wizualizacja za pomocą schematów blokowych ułatwiły zrozumienie działania tej struktury. Wyniki projektu potwierdzają, że lista dwukierunkowa jest odpowiednią strukturą danych dla aplikacji wymagających częstego dodawania i usuwania elementów na początku lub końcu zbioru, a także tam, gdzie istotne jest przeglądanie danych w obu kierunkach.

Spis rysunków

Spis tabel

Spis listingów