AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Informatyka Stosowana

DOKUMENTACJA PROJEKTOWA

ZAAWANSOWANE PROGRAMOWANIE

Algorytm listy dwukierunkowej z zastosowaniem GitHub

Autor:
Henryk Szambelan

Prowadzący:
mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

1. Ogólne określenie wymagań

Dzielimy wymagania na cztery kluczowe obszary: funkcjonalne (co ma robić kod), projektowe (jak ma być zbudowany), procesowe (jakie narzędzia zastosować) i dokumentacyjne (co dostarczyć).

1.1. Funkcjonalne

Program musi udostępniać strukturę danych **listy dwukierunkowej**[1], działającej w całości na **pamięci sterty** (heap). Oznacza to, że każdy nowy węzeł (`Node`) musi być alokowany dynamicznie zgodnie z zasadami **zarządzania pamięcią w C++** (`new`/`delete`). Lista przechowuje dane typu `int`.

1.2. Projektowe (architektura)

Implementacja musi być zgodna z zasadami programowania obiektowego. Wymagane jest zastosowanie następującej struktury:

- Każda główna klasa musi znajdować się w osobnym pliku nagłówkowym (`.h`).
- Zaimplementowany jest **Wzorzec Factory** (`ListFactory`) do centralizacji tworzenia instancji `DoublyLinkedList`.
- Zaimplementowany jest **Wzorzec Iterator** (`Iterator`) umożliwiający przechodzenie przez listę oraz jej użycie w pętlach `for-each`.

1.3. Procesowe (narzędzia)

Projekt musi być zarządzany przez Git i hostowany na GitHubie. Wymagane jest przeprowadzenie konkretnych, zaawansowanych operacji na repozytorium.

1.4. Dokumentacyjne

Dokumentacja API: Wygenerowana automatycznie przez Doxygen do formatu PDF (przez LaTeX) oraz dokumentacja projektowa.

[1] Struktura z wskaźnikami do poprzedniego i następnego elementu.

2. Analiza problemu

2.1. Gdzie używa się listy dwukierunkowej?

Lista dwukierunkowa to struktura danych wykorzystywana w wielu aplikacjach:

- **Aplikacje desktopowe i webowe:** edytory tekstów (undo/redo), przeglądarki (wstecz/do przodu), odtwarzacze muzyki (playlisty).
- **Systemy operacyjne:** zarządzanie procesami, cache LRU.
- **Gry komputerowe:** historia ruchów, zarządzanie obiektami na scenie.
- **Bazy danych:** implementacja kolejek FIFO/DEQUE, niektóre struktury indeksów.
- **Algorytmy:** sortowania (merge sort na listach), algorytmy grafowe (lista sąsiedztwa), implementacja stosu i kolejki jednocześnie.

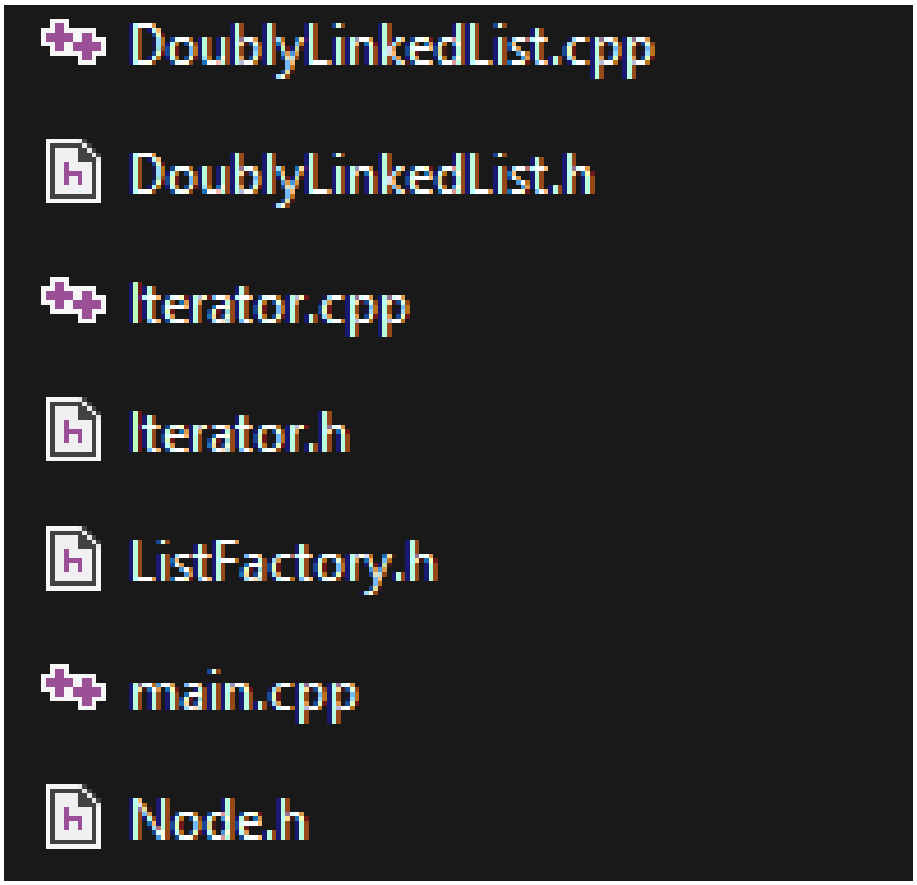
2.2. Sposób działania programu

Program implementuje listę dwukierunkową jako dynamiczną strukturę danych, w której każdy element (`Node`) jest połączony z poprzednim i następnym elementem za pomocą wskaźników.

2.2.1. Struktura węzła (`Node`)

Węzeł jest zdefiniowany jako struktura (domyślnie publiczna) przechowująca dane typu `int`.

- `data` – przechowywane dane typu `int`,
- `next` – wskaźnik do następnego węzła (`Node*`),
- `prev` – wskaźnik do poprzedniego węzła (`Node*`).



Rys. 2.1. Struktura projektu.

2.2.2. Klasa `DoublyLinkedList`

Klasa zarządza całą listą, przechowując wskaźniki na `head` (głowę) i `tail` (ogon) oraz licznik elementów `count`. Kluczowe operacje obejmują `addFront`, `addBack`, `removeFront`, `removeBack`, `addAt`, `removeAt`, `display` oraz `clear`.

2.2.3. Wzorzec Iterator (`Iterator`)

Klasa `Iterator` przechowuje wskaźnik `Node* currentNode`. Umożliwia poruszanie się po liście:

- `begin()` zwraca iterator wskazujący na `head`,
- `end()` zwraca iterator wskazujący na `nullptr` (za ostatnim elementem),
- operator `++` przesuwa do następnego węzła (`next`),
- operator `--` przesuwa do poprzedniego węzła (`prev`),
- dereferencja operator `*` daje dostęp do wartości `data`.

2.2.4. Wzorzec Factory (`ListFactory`)

`ListFactory` udostępnia statyczną metodę `createList()` do tworzenia instancji `DoublyLinkedList` na stercie, ukrywając operację `new` przed klientem (`main.cpp`).

3. Projektowanie

3.1. Narzędzia i środowisko programistyczne

- Język: C++ (zgodny ze standardem C++11/17),
- Dokumentacja: Doxygen,
- System kontroli wersji: Git,
- Biblioteki: standardowa biblioteka C++ (`<iostream>`, `<stdexcept>`).

3.2. Zarządzanie projektem w Git i GitHub

Do zarządzania kodem źródłowym projektu wykorzystano system kontroli wersji **Git**. Jako centralne, zdalne repozytorium (remote) posłużyła platforma **GitHub**, co umożliwiło śledzenie historii zmian, pracę na gałęziach oraz integrację z narzędziami zewnętrznymi.

Inicjalizacja projektu oraz połączenie z repozytorium na GitHubie przebiegło następująco:

```
# Inicjalizacja nowego repozytorium
git init

# Dodanie zdalnego repozytorium (np. GitHub)
git remote add origin https://github.com/EnriqeD/
   Zadanie1Zaawansowane.git
```

Listing 1. Utworzenie repozytorium Git

Podstawowy, codzienny cykl pracy z systemem Git polegał na sprawdzaniu statusu plików, dodawaniu zmodyfikowanych plików do "poczekalni"(staging area), zatwierdzaniu zmian (commit) oraz wysyłaniu ich na zdalny serwer.

```
# 1. Sprawdzenie statusu plikow
git status

# 2. Dodanie plikow do staging area
git add .

# 3. Zatwierdzenie zmian (commit)
git commit -m "feat: Implementacja klasy DoublyLinkedList"

# 4. Wyslanie zmian na serwer
```

```
git push -u origin main
```

Listing 2. Podstawowy cykl pracy z Git

3.2.1. Strategia gałęzi (Branching Strategy)

Prace nad projektem opierały się na strategii **feature branch**. Gałąź `main` pełniła rolę stabilnej, głównej linii rozwojowej, zawierającej kod gotowy do wydania.

Każda nowa funkcjonalność (np. implementacja iteratora, dodanie metod `addAt`/`removeAt`) lub poprawka błędu była rozwijana na osobnej, dedykowanej gałęzi, tworzonej z aktualnej wersji `main`. Po ukończeniu prac i (opcjonalnym) przejściu testów, gałąź była łączona z powrotem do `main` poprzez mechanizm `git merge` (najczęściej realizowany przez **Pull Request** na platformie GitHub).

```
# 1. Upewnij sie, ze jestes na glownej galezi i jest aktualna
git checkout main
git pull origin main

# 2. ...praca nad kodem (pliki Iterator.h, Iterator.cpp)...

# 3. Zapisz zmiany na nowej galezi
git add Iterator.h Iterator.cpp DoublyLinkedList.h
git commit -m "Utworzenie plikow Iterator.h Iterator.cpp
    DoublyLinkedList.h."

# 4. Wyslij galaz na serwer GitHub
git push -u origin

# 5. (Po zlaczeniu przez Pull Request na GitHub) Wroc do main
git checkout main
```

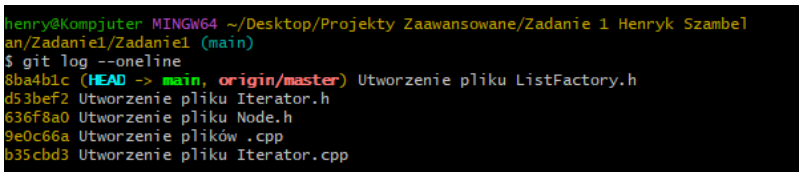
Listing 3. Praca z gałęziami (feature branch)

3.2.2. Konwencja nazewnictwa commitów

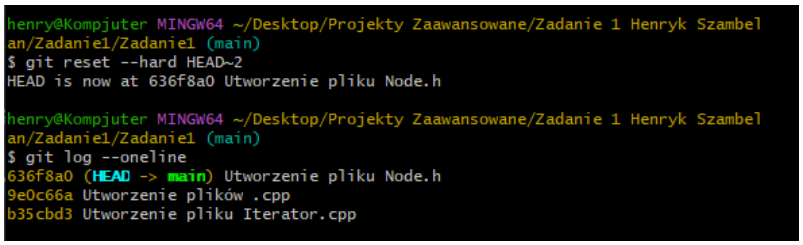
Aby utrzymać porządek w historii zmian i ułatwić jej przeglądanie, w projekcie stosowano zasady **Conventional Commits**. Każdy commit posiadał prefiks określający typ wprowadzonej zmiany:

- `feat:` (nowa funkcjonalność, np. dodanie `addAt`)
- `fix:` (naprawa błędu, np. wycieku pamięci w `clear`)

- `docs:` (zmiany w dokumentacji, np. komentarze Doxygen, aktualizacja README)
- `refactor:` (zmiany w kodzie nie wpływające na działanie, np. optymalizacja `getNode`)
- `test:` (dodanie lub poprawa testów w `main.cpp`)



Rys. 3.1. Przykładowa historia commitów z komendy `git log --oneline`.



Rys. 3.2. Przykład zaawansowanej operacji - cofnięcie o 2 commity (`git reset --hard`).

4. Implementacja

4.1. Struktura projektu

Projekt składa się z następujących plików: `Node.h`, `Iterator.h`, `Iterator.cpp`, `DoublyLinkedList.h`, `DoublyLinkedList.cpp`, `ListFactory.h`, `main.cpp`.

4.2. Implementacja struktury Node

Plik: `Node.h`

```
/**
 * @file Node.h
 * @brief Definiuje strukture wezla (Node) dla listy dwukierunkowej.
 *
 * Ten plik zawiera definicje podstawowego bloku budujacego liste.
 */

  // #pragma once to dyrektywa, ktora mowi kompilatorowi,
  // aby wczytal ten plik only jeden raz.
#pragma once

/**
 * @struct Node
 * @brief Reprezentuje pojedynczy element (wezel) w liscie dwukierunkowej.
 *
 * Przechowuje wartosc (data) oraz wskazniki na nastepny (next)
 * i poprzedni (prev) element listy.
 * W 'struct' wszystko jest domyslnie publiczne.
 */
struct Node {
    int data;     ///< Dane (typu int) przechowywane w wezle.
    Node* next;   ///< Wskaznik na nastepny wezel (lub `nullptr`, jesli to ogon).
    Node* prev;   ///< Wskaznik na poprzedni wezel (lub `nullptr`, jesli to glowa).

    /**
     * @brief Konstruktor wezla.
     *
     * Inicjalizuje wezel podana wartoscia (val).
```

```
29      * Wskazniki ‘next‘ i ‘prev‘ sa domyslnie ustawiane na ‘nullptr
    ‘.
30      * Uzywamy "listy inicjalizacyjnej" (po dwukropku).
31      *
32      * @param val Wartosc (typu int) do przechowania w wezle.
33      */
34     Node(int val) : data(val), next(nullptr), prev(nullptr) {
35
36         // "nullptr" to specjalna wartosc oznaczajaca "wskaznik
    donikad".
37         // Nowy wezel domyslnie nie jest polaczony z innymi.
38     }
39 };
```

Listing 4. Node.h

4.3. Implementacja klasy DoublyLinkedList

Plik: `DoublyLinkedList.h`

```
1  /**
2   * @file DoublyLinkedList.h
3   * @brief Deklaracja klasy DoublyLinkedList.
4   */
5
6  #pragma once
7  #include "Node.h"
8  #include "Iterator.h"
9
10  /**
11   * @class DoublyLinkedList
12   * @brief Implementuje liste dwukierunkowa.
13   */
14 class DoublyLinkedList {
15 private:
16     Node* head;  ///< Wskaznik na poczatek (glowe).
17     Node* tail;  ///< Wskaznik na koniec (ogon).
18     int count; ///< Liczba elementow.
19
20     /**
21      * @brief Pomocnik: Pobiera wezel o zadanym indeksie.
22      * @param index Indeks (0-based) wezla.
23      * @return Wskaznik (Node*) do wezla lub nullptr.
24      */
25     Node* getNode(int index);
```

```
26
27 public:
28     /**
29      * @brief Konstruktor. Tworzy pusta liste.
30      */
31     DoublyLinkedList();
32
33     /**
34      * @brief Destruktor. Zwalnia pamiec (wywoluje clear()).
35      */
36     ~DoublyLinkedList();
37
38     /**
39      * @brief Zwraca iterator na poczatek listy.
40      * @return Iterator na 'head'.
41      */
42     Iterator begin();
43
44     /**
45      * @brief Zwraca iterator "za koncem" (nullptr).
46      * @return Iterator na 'nullptr' (koniec petli).
47      */
48     Iterator end();
49
50     /**
51      * @brief Dodaje element na poczatek.
52      * @param data Wartosc do dodania.
53      */
54     void addFront(int data);
55
56     /**
57      * @brief Dodaje element na koniec.
58      * @param data Wartosc do dodania.
59      */
60     void addBack(int data);
61
62     /**
63      * @brief Dodaje element na indeks.
64      * @param data Wartosc do dodania.
65      * @param index Indeks (0-based) wstawienia.
66      */
67     void addAt(int data, int index);
68
69     /**
70      * @brief Usuwa element z poczatku.
```

```
71      */
72     void removeFront();
73
74     /**
75      * @brief Usuwa element z konca.
76      */
77     void removeBack();
78
79     /**
80      * @brief Usuwa element z indeksu.
81      * @param index Indeks (0-based) do usuniecia.
82      */
83     void removeAt(int index);
84
85     /**
86      * @brief Wyswietla liste (Head -> Tail).
87      */
88     void display();
89
90     /**
91      * @brief Wyswietla liste odwrotnie (Tail -> Head).
92      */
93     void displayReverse();
94
95     /**
96      * @brief Czysci liste (usuwa wszystkie elementy).
97      */
98     void clear();
99
100    /**
101     * @brief Zwraca liczbe elementow.
102     * @return Liczba elementow (int).
103     */
104    int getSize();
105
106    /**
107     * @brief Sprawdza, czy lista jest pusta.
108     * @return ‘true‘ jesli pusta, ‘false‘ w innym razie.
109     */
110    bool isEmpty();
111 };
```

Listing 5. DoublyLinkedList.h

Plik: `DoublyLinkedList.cpp` (wybrane metody)

```
/**
 * @file DoublyLinkedList.cpp
 * @brief Implementacja metod dla klasy DoublyLinkedList.
 *
 * Ten plik zawiera definicje metod zadeklarowanych w ‘
   DoublyLinkedList.h‘.
 */

#include "DoublyLinkedList.h"
#include <iostream>     // Potrzebne do wyswietlania (std::cout)
#include <stdexcept>    // Potrzebne do rzucania wyjatkow (bledow)
using namespace std;

 /**
  * @brief Konstruktor domyslny.
  * Tworzy nowa, pusta liste. Ustawia head/tail na nullptr i count
    na 0.
  */
DoublyLinkedList::DoublyLinkedList() {
    head = nullptr; // Na starcie pusta lista nie ma glowy
    tail = nullptr; // Ani ogona
    count = 0;      // Licznik jest 0
}

/**
 * @brief Destruktor.
 * Zwalnia cala pamiec (wywoluje clear()).
 */
DoublyLinkedList::~DoublyLinkedList() {
    clear(); // Sprzata pamiec
}

/**
 * @brief Zwraca iterator na poczatek listy.
 * @return Iterator wskazujacy na 'head'.
 */
Iterator DoublyLinkedList::begin() {
    return Iterator(head); // Iterator zaczyna od glowy
}

/**
 * @brief Zwraca iterator "za koncem" listy (nullptr).
 * Uzywany do oznaczania konca w petlach.
 * @return Iterator wskazujacy na ‘nullptr‘.
 */
```

```
Iterator DoublyLinkedList::end() {
    return Iterator(nullptr); // Iterator konczy na nullptr
}

/**
 * @brief Sprawdza, czy lista jest pusta.
 * @return ‘true‘ jesli ’count == 0’, ‘false‘ w przeciwnym razie.
 */
bool DoublyLinkedList::isEmpty() {
    return count == 0; // Lista jest pusta, jesli licznik jest 0
}

/**
 * @brief Dodaje element na poczatek listy (przed ’head’).
 * @param data Wartosc do dodania.
 */
void DoublyLinkedList::addFront(int data) {
    Node* newNode = new Node(data); // Stworz nowy wezel
    if (isEmpty()) {
        head = tail = newNode; // Jest to jedyny element
    }
    else {
        newNode->next = head; // Nowy wezel wskazuje na stara glowe
        head->prev = newNode; // Stara glowa wskazuje wstecz na
   nowy wezel
        head = newNode;       // Nowy wezel staje sie glowa
    }
    count++; // Zwieksz licznik
}

/**
 * @brief Dodaje element na koniec listy (za ’tail’).
 * @param data Wartosc do dodania.
 */
void DoublyLinkedList::addBack(int data) {
    Node* newNode = new Node(data); // Stworz nowy wezel
    if (isEmpty()) {
        head = tail = newNode; // Jest to jedyny element
    }
    else {
        tail->next = newNode;    // Stary ogon wskazuje na nowy
   wezel
        newNode->prev = tail;    // Nowy wezel wskazuje wstecz na
   stary ogon
        tail = newNode;          // Nowy wezel staje sie ogonem
```

```
86      }
87      count++; // Zwieksz licznik
88  }
89
90  /**
91   * @brief Usuwa element z poczatku listy ('head').
92   * @throws std::runtime_error Jesli lista jest pusta.
93   */
94  void DoublyLinkedList::removeFront() {
95      if (isEmpty()) {
96          throw std::runtime_error("Blad: Proba usuniecia z pustej
        listy!");
97      }
98      Node* nodeToDelete = head; // Zapamietaj stara glowe
99      if (count == 1) {
100         head = tail = nullptr; // Lista staje sie pusta
101     }
102     else {
103         head = head->next;     // Drugi element staje sie glowa
104         head->prev = nullptr;  // Nowa glowa nie ma nic przed soba
105     }
106     delete nodeToDelete; // Usun stara glowe
107     count--; // Zmniejsz licznik
108 }
109
110 /**
111  * @brief Usuwa element z konca listy ('tail').
112  * @throws std::runtime_error Jesli lista jest pusta.
113  */
114 void DoublyLinkedList::removeBack() {
115     if (isEmpty()) {
116         throw std::runtime_error("Blad: Proba usuniecia z pustej
        listy!");
117     }
118     Node* nodeToDelete = tail; // Zapamietaj stary ogon
119     if (count == 1) {
120         head = tail = nullptr; // Lista staje sie pusta
121     }
122     else {
123         tail = tail->prev;     // Przedostatni element staje sie
        ogonem
124         tail->next = nullptr;  // Nowy ogon nie ma nic po sobie
125     }
126     delete nodeToDelete; // Usun stary ogon
127     count--; // Zmniejsz licznik
```

```
128 }
129
130 /**
131  * @brief Usuwa wszystkie elementy z listy.
132  * Zwalnia pamiec i resetuje liste do stanu pustego.
133  */
134 void DoublyLinkedList::clear() {
135     // Usuwaj pierwszy element, dopoki lista nie bedzie pusta
136     while (!isEmpty()) {
137         removeFront();
138     }
139 }
140
141 /**
142  * @brief Zwraca liczbe elementow na liscie.
143  * @return Aktualna liczba elementow (int).
144  */
145 int DoublyLinkedList::getSize() {
146     return count;
147 }
148
149 /**
150  * @brief Wyswietla liste od poczatku do konca (Head -> Tail).
151  * Wypisuje elementy na konsole.
152  */
153 void DoublyLinkedList::display() {
154     cout << "Lista (Head -> Tail): ";
155     Node* current = head; // Zacznij od glowy
156     while (current != nullptr) { // Idz do przodu, az dojdziesz do
    konca
157         cout << current->data << " <-> ";
158         current = current->next;
159     }
160     cout << "NULL" << std::endl;
161 }
162
163 /**
164  * @brief Wyswietla liste od konca do poczatku (Tail -> Head).
165  * Wypisuje elementy na konsole.
166  */
167 void DoublyLinkedList::displayReverse() {
168     cout << "Lista (Tail -> Head): ";
169     Node* current = tail; // Zacznij od ogona
170     while (current != nullptr) { // Idz do tylu, az dojdziesz do
    poczatku
```

```
171         cout << current->data << " <-> ";
172         current = current->prev;
173     }
174     cout << "NULL" << std::endl;
175 }
176 
177 /**
178  * @brief Prywatna metoda pomocnicza do pobierania wezla o zadanym
        indeksie.
179  * Optymalizacja: szuka od 'head' lub 'tail' (co blizej).
180  * @param index Indeks (0-based) wezla.
181  * @return Wskaznik (Node*) do wezla lub `nullptr` (zly indeks).
182  */
183 Node* DoublyLinkedList::getNode(int index) {
184     if (index < 0 || index >= count) {
185         return nullptr; // Zly indeks
186     }
187     // Optymalizacja: Szukaj od poczatku lub od konca
188     if (index < count / 2) {
189         // Blizej od glowy
190         Node* current = head;
191         for (int i = 0; i < index; ++i) {
192             current = current->next;
193         }
194         return current;
195     }
196     else {
197         // Blizej od ogona
198         Node* current = tail;
199         for (int i = count - 1; i > index; --i) {
200             current = current->prev;
201         }
202         return current;
203     }
204 }
205 
206 /**
207  * @brief Dodaje element pod wskazany indeks.
208  * @param data Wartosc (int) do wstawienia.
209  * @param index Indeks (0-based), [0, count].
210  * @throws std::out_of_range Jesli indeks jest poza zakresem [0,
        count].
211  */
212 void DoublyLinkedList::addAt(int data, int index) {
213     if (index < 0 || index > count) {
```

```
214         throw std::out_of_range("Blad: Nieprawidlowy indeks dla addAt.");
215     }
216     // Uzyj istniejacych metod dla krancow
217     if (index == 0) {
218         addFront(data);
219         return;
220     }
221     if (index == count) {
222         addBack(data);
223         return;
224     }
225     // Dodawanie w srodku
226     Node* prevNode = getNode(index - 1); // Wezel "przed" miejscem wstawienia
227     Node* nextNode = prevNode->next;
228     Node* newNode = new Node(data);
229 
230     // Przepinanie wskaznikow
231     newNode->prev = prevNode;
232     newNode->next = nextNode;
233     prevNode->next = newNode;
234     nextNode->prev = newNode;
235 
236     count++;
237 }
238 
239 /**
240  * @brief Usuwa element o wskazanym indeksie.
241  * @param index Indeks (0-based) elementu do usuniecia, [0, count-1].
242  * @throws std::out_of_range Jesli indeks jest poza zakresem [0, count-1].
243  */
244 void DoublyLinkedList::removeAt(int index) {
245     if (index < 0 || index >= count) {
246         throw std::out_of_range("Blad: Nieprawidlowy indeks dla removeAt.");
247     }
248     // Uzyj istniejacych metod dla krancow
249     if (index == 0) {
250         removeFront();
251         return;
252     }
253     if (index == count - 1) {
```

```
254         removeBack();
255         return;
256     }
257     // Usuwanie ze srodka
258     Node* nodeToRemove = getNode(index);
259     Node* prevNode = nodeToRemove->prev;
260     Node* nextNode = nodeToRemove->next;
261 
262     // "Omin" usuwany wezel
263     prevNode->next = nextNode;
264     nextNode->prev = prevNode;
265 
266     delete nodeToRemove; // Usun wezel z pamieci
267     count--;
268 }
```

Listing 6. DoublyLinkedList.cpp

4.4. Implementacja klasy Iterator

Plik: `Iterator.h`

```
1  /**
2   * @file Iterator.h
3   * @brief Deklaracja klasy Iterator.
4   *
5   * Iterator sluzy do "chodzenia" po liscie.
6   */
7 
8  #pragma once
9  #include "Node.h" // Iterator musi wiedziec, czym jest "Node".
10 
11  /**
12   * @class Iterator
13   * @brief Iterator dwukierunkowy dla DoublyLinkedList.
14   *
15   * Umozliwia przechodzenie (++, --) i odczyt (*) elementow.
16   */
17 class Iterator {
18 private:
19     /**
20      * @brief Wskaznik na biezacy wezel.
21      * Przechowuje adres wezla, na ktory "patrzy" iterator.
22      */
23     Node* currentNode;
```

```
24
25 public:
26     /**
27      * @brief Konstruktor.
28      * Tworzy iterator wskazujacy na podany wezel.
29      * @param node Wezel startowy.
30      */
31     Iterator(Node* node);
32
33     /**
34      * @brief Operator dereferencji (*).
35      * Zwraca referencje do danych (int&) w biezacym wezle.
36      * @return Referencja do danych.
37      */
38     int& operator*();
39
40     /**
41      * @brief Operator pre-inkrementacji (++it).
42      * Przesuwa iterator na nastepny wezel.
43      * @return Referencja do *this (po przesunieciu).
44      */
45     Iterator& operator++();
46
47     /**
48      * @brief Operator pre-dekrementacji (--it).
49      * Przesuwa iterator na poprzedni wezel.
50      * @return Referencja do *this (po przesunieciu).
51      */
52     Iterator& operator--();
53
54     /**
55      * @brief Operator porownania (==).
56      * Sprawdza, czy iteratory wskazuja na ten sam wezel.
57      * @param other Inny iterator do porownania.
58      * @return ‘true‘ jesli rowne, ‘false‘ wpp.
59      */
60     bool operator==(const Iterator& other);
61
62     /**
63      * @brief Operator nierownosci (!=).
64      * Sprawdza, czy iteratory wskazuja na rozne wezly.
65      * @param other Inny iterator do porownania.
66      * @return ‘true‘ jesli rozne, ‘false‘ wpp.
67      */
68     bool operator!=(const Iterator& other);
```

```
};
```

Listing 7. Iterator.h

Plik: `Iterator.cpp`

```
/**
 * @file Iterator.cpp
 * @brief Implementacja Iterator.
 *
 * Definicje operatorow iteratora.
 */

#include "Iterator.h"
#include <stdexcept> // Dla std::runtime_error

 /**
  * @brief Konstruktor.
  * @param node Startowy wezel.
  */
Iterator::Iterator(Node* node) : currentNode(node) {
    // Puste cialo
}

/**
 * @brief Operator dereferencji (*).
 * Zwraca dane (int&).
 * @return Dane (int&).
 * @throws std::runtime_error Jesli nullptr.
 */
int& Iterator::operator*() {
    // Sprawdz, czy nie nullptr
    if (currentNode == nullptr) {
        // Blad
        throw std::runtime_error("Blad: Proba odczytu danych z
   iteratora wskazujacego na nullptr!");
    }
    // Zwroc dane
    return currentNode->data;
}

/**
 * @brief Operator ++it.
 * Przesun na nastepny (next).
 * @return *this.
 */
```

```
40 Iterator& Iterator::operator++() {
41     // Jesli nie null
42     if (currentNode) {
43         // Idz do przodu
44         currentNode = currentNode->next;
45     }
46     // Zwroc *this
47     return *this;
48 }
49 
50 /**
51  * @brief Operator --it.
52  * Przesun na poprzedni (prev).
53  * @return *this.
54  */
55 Iterator& Iterator::operator--() {
56     // Jesli nie null
57     if (currentNode) {
58         // Idz do tylu
59         currentNode = currentNode->prev;
60     }
61     // Zwroc *this
62     return *this;
63 }
64 
65 /**
66  * @brief Operator ==.
67  * Sprawdza rownosc wezlow.
68  * @param other Drugi iterator.
69  * @return true jesli te same, false wpp.
70  */
71 bool Iterator::operator==(const Iterator& other) {
72     // Porownaj wskazniki
73     return currentNode == other.currentNode;
74 }
75 
76 /**
77  * @brief Operator !=.
78  * Sprawdza nierownosc wezlow.
79  * @param other Drugi iterator.
80  * @return true jesli rozne, false wpp.
81  */
82 bool Iterator::operator!=(const Iterator& other) {
83     // Uzyj operatora ==
84     return !(*this == other);
```

```
85 }
```

Listing 8. Iterator.cpp

4.5. Implementacja wzorca Factory

Plik: ListFactory.h

```
1  /**
2   * @file ListFactory.h
3   * @brief Definicja klasy ListFactory (wzorzec Fabryki).
4   *
5   * Plik zawiera prosta Fabryke (Factory)
6   * do tworzenia instancji DoublyLinkedList na stercie.
7   */
8
9  #pragma once
10 #include "DoublyLinkedList.h" // Fabryka musi wiedziec, co tworzy
11
12  /**
13   * @class ListFactory
14   * @brief Implementacja wzorca Fabryki (Factory) dla
     DoublyLinkedList.
15   *
16   * Dostarcza statyczna metode `createList()` do tworzenia obiektow
     .
17   * Ukrywa uzycie `new` przed klientem (np. `main.cpp`).
18   */
19 class ListFactory {
20 public:
21     /**
22      * @brief Statyczna metoda fabryczna tworzaca nowa liste na
    stercie.
23      *
24      * Moze byc wywolana jako `ListFactory::createList()`.
25      *
26      * @return Wskaznik (DoublyLinkedList*) na nowo utworzony
    obiekt
27      * na stercie (przez `new`).
28      */
29     static DoublyLinkedList* createList() {
30
31         // "new" tworzy obiekt na stercie
32         // Fabryka zwraca wskaznik (*) na ten obiekt.
33         return new DoublyLinkedList();
```

```
34      }
35  };
```

Listing 9. ListFactory.h

4.6. Przykład użycia — program główny

Plik: `main.cpp`

```
1  /**
2   * @file main.cpp
3   * @brief Testy dla DoublyLinkedList.
4   */
5
6  #include <iostream>
7  #include "ListFactory.h"
8  #include "DoublyLinkedList.h"
9  using namespace std;
10
11  /**
12   * @brief Funkcja main() testujaca liste.
13   *
14   * Testy:
15   * 1. Dodawanie
16   * 2. Usuwanie
17   * 3. Wyswietlanie
18   * 4. Iterator
19   * 5. Czyszczenie
20   * 6. Obsluga bledow
21   *
22   * @return 0 po zakonczeniu.
23   */
24  int main() {
25      cout << "Testy Listy Dwukierunkowej" << endl;
26
27      // Test 1: Tworzenie
28      DoublyLinkedList* list = ListFactory::createList();
29      cout << "Stworzono liste." << endl;
30
31      // Test 2: Dodawanie
32      cout << "\n Dodawanie (addBack, addFront):" << endl;
33      list->addBack(10);
34      list->addBack(20);
35      list->addFront(5);
36      // Lista: 5 <-> 10 <-> 20
```

```
37      list->display();
38      cout << "Rozmiar: " << list->getSize() << endl;
39
40      // Test 3: Wyswietlanie (odwrotnie)
41      cout << "\n Wyswietlanie (displayReverse):" << endl;
42      list->displayReverse();
43
44      // Test 4: Dodawanie (indeks)
45      cout << "\n Dodawanie (addAt):" << endl;
46      list->addAt(15, 2); // 5 <-> 10 <-> 15 <-> 20
47      list->addAt(0, 0);  // 0 <-> 5 <-> 10 <-> 15 <-> 20
48      list->addAt(25, 5); // 0 <-> 5 <-> 10 <-> 15 <-> 20 <-> 25
49      list->display();
50
51      // Test 5: Usuwanie (kraje)
52      cout << "\n Usuwanie (removeFront, removeBack):" << endl;
53      list->removeFront(); // Usuwa 0
54      list->removeBack();  // Usuwa 25
55      // Lista: 5 <-> 10 <-> 15 <-> 20
56      list->display();
57
58      // Test 6: Usuwanie (indeks)
59      cout << "\n Usuwanie (removeAt):" << endl;
60      list->removeAt(1); // Usuwa 10
61      // Lista: 5 <-> 15 <-> 20
62      list->display();
63
64      // Test 7: Iterator (dla kazdego)
65      cout << "\n Iterator (petla for-each):" << endl;
66      cout << "Zawartosc listy: ";
67      for (int value : *list) {
68          cout << value << " ";
69      }
70      cout << endl;
71
72      // Test 8: Iterator (manualnie)
73      cout << "\n Iterator (manualna obsluga):" << endl;
74
75      Iterator it = list->begin(); // it -> 5
76      cout << "Poczatek: " << *it << endl; // 5
77
78      // Nastepny
79      ++it; // it -> 15
80      cout << "Nastepny (++it): " << *it << endl; // 15
81
```

```
82      // Poprzedni
83      --it; // it -> 5
84      cout << "Poprzedni (--it): " << *it << endl; // 5
85
86      // Test 9: Czyszczenie
87      cout << "\n Czyszczenie (clear):" << endl;
88      list->clear();
89      cout << "Lista po clear() (isEmpty? " << boolalpha << list->
        isEmpty() << "):" << endl;
90      list->display(); // Pusta lista
91
92      // Test 10: Bledu
93      cout << "\n Obsluga bledow:" << endl;
94      try {
95          list->removeFront(); // Usun z pustej
96      }
97      catch (const exception& e) {
98          // Oczekiwany blad
99          cout << "Blad: " << e.what() << endl;
100     }
101
102     // Test 11: Sprzatanie
103     delete list;
104     list = nullptr;
105
106     cout << "\n Zakonczono" << endl;
107     return 0;
108 }
```

Listing 10. main.cpp

4.7. Wyniki działania programu

Przykładowy output programu:

```
Testy Listy Dwukierunkowej
Stworzono liste.

 Dodawanie (addBack, addFront):
Lista (Head -> Tail): 5 <-> 10 <-> 20 <-> NULL
Rozmiar: 3

 Wyswietlanie (displayReverse):
```

```
Lista (Tail -> Head): 20 <-> 10 <-> 5 <-> NULL

 Dodawanie (addAt):
Lista (Head -> Tail): 0 <-> 5 <-> 10 <-> 15 <-> 20 <-> 25 <-> NULL

 Usuwanie (removeFront, removeBack):
Lista (Head -> Tail): 5 <-> 10 <-> 15 <-> 20 <-> NULL

 Usuwanie (removeAt):
Lista (Head -> Tail): 5 <-> 15 <-> 20 <-> NULL

 Iterator (petla for-each):
Zawartosc listy: 5 15 20

 Iterator (manualna obsluga):
Poczatek: 5
Nastepny (++it): 15
Poprzedni (--it): 5

 Czyszczenie (clear):
Lista po clear() (isEmpty? true):
Lista (Head -> Tail): NULL

 Obsluga bledow:
Blad: Proba usuniecia z pustej listy!

 Zakonczono
```

5. Wnioski

5.1. Wnioski techniczne

1. Lista dwukierunkowa jest efektywna przy operacjach na końcach ($O(1)$ dla `addFront`/`removeFront`/`addBack`/`removeBack`).

2. Operacje w środku listy (`addAt`/`removeAt`) wymagają przejścia sekwencyjnego ($O(n)$) do znalezienia węzła, ale optymalizacja w `getNode` skraca to do $O(n/2)$.

3. Zarządzanie pamięcią (poprzez `new` i `delete`) jest kluczowe dla uniknięcia wycieków. `clear()` i destruktor zapewniają zwolnienie wszystkich węzłów.

4. Wskaźnik `prev` umożliwia efektywną dwukierunkową nawigację, w tym implementację dekrementacji iteratora (`--it`).

5.2. Wnioski projektowe

- **Wzorzec Factory** (`ListFactory`) ułatwia centralizację tworzenia obiektu na stercie i zarządzanie jego cyklem życia (usuwanie przez `delete`).

- **Iterator** (`Iterator`) ukrywa szczegóły implementacji listy i daje interfejs podobny do kontenerów STL, umożliwiając wygodne pętle `for-each`.

- Podział na klasy i pliki źródłowe (`.h`/`.cpp`) poprawia czytelność i utrzymanie kodu.

5.3. Zastosowania praktyczne

Lista dwukierunkowa jest odpowiednia gdy:

- często dodajemy/usuwamy elementy z obu końców,

- potrzebujemy przechodzić w obie strony (np. nawigacja w historii),

- rozmiar listy zmienia się dynamicznie.

5.4. Napotkane problemy

Najtrudniejsze: prawidłowe łączenie wskaźników przy wstawianiu i usuwaniu w środku listy. Wymagana jest precyzyjna kolejność aktualizacji wskaźników `next` i `prev` u

czterech węzłów (węzła przed, usuwanego/dodawanego, węzła po, i nowego/usuwanego) oraz obsługa przypadków brzegowych (pusta lista, jeden element, `head`, `tail`).

Spis rysunków

Spis listingów