AKADEMIA NAUK STOSOWANYCH
W NOWYM SĄCZU

Wydział Nauk Inżynieryjnych
Katedra Informatyki

Algorytm listy dwukierunkowej z zastosowaniem GitHub

Autor: Paweł Sobczak

Prowadzący: mgr inż. Dawid Kotlarski

Nowy Sącz 2025

Spis treści

Spis rysunków

Spis tabel

Listings

Rozdział 1

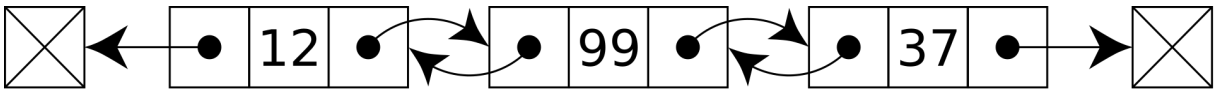
Ogólne określenie wymagań

1.1 Cele

Celem projektu jest napisanie programu działającego na stercie, pozwalającego wykonywać podstawowe operacje na listach dwukierunkowych. Każda klasa ma być w innym pliku wykonawczym natomiast metody mają być wywoływane w `main`. Ponadto projekt ma wykazać w jaki sposób pracować z systemem kontroli wersji jakim jest Git.

1.2 Listy Dwukierunkowe

Lista dwukierunkowa to liniowa struktura, w której każdy węzeł ma dynamiczne połączenie zarówno z następnym jak i poprzednim elementem. Dzięki takiej strukturze w przeciwieństwie do list jednoliniowych można poruszać się po strukturze zarówno do przodu jak i do tyłu.



Rysunek 1.1: Rys. 1.1. Schemat ilustrujący działanie listy dwukierunkowej

Każdy element listy składa się z co najmniej dwóch pól: klucza oraz pola wskazującego na następny element listy. W przypadku list dwukierunkowych każdy element listy zawiera także pole wskazujące na poprzedni element listy. Pole wskazujące poprzedni i następny element listy są najczęściej wskaźnikami.

Na listach dwukierunkowych możemy przeprowadzać między innymi operacje manipulacyjne listą lub przeszukiwania i dostępu.

Operacje manipulacyjne to:

- Wstawianie na początek (Prepend)
- Wstawianie na koniec (Append)

- Usuwanie pierwszego elementu (Remove Head)
- Usuwanie ostatniego elementu (Remove Tail)
- Wstawianie po danym elemencie (Insert After)
- Wstawianie przed danym elementem (Insert Before)
- Usuwanie danego elementu (Remove Specific Node)

Operacje przeszukiwania i dostępu:

- Przeszukiwanie (Search)
- Przechodzenie (Traversal)
- Dostęp do elementu na pozycji i (Access by Index)

Inne operacje:

- Sprawdzenie, czy lista jest pusta (Check if Empty)
- Łączenie list (Concatenation)
- Odwracanie listy (Reverse)
- Sortowanie

1.3 System Kontroli Wersji - Git

Git to rozproszony system kontroli wersji (DVCS - Distributed Version Control System). Kontrola wersji to system, który rejestruje zmiany w pliku lub zestawie plików w czasie, dzięki czemu można w każdej chwili przywołać określone wersje.

Zastosowanie:

- Śledzenie zmian
- Współpraca
- Przywracanie wersji
- Branching (Rozgałęzienia)

1.4 Podstawowe komendy - GIT

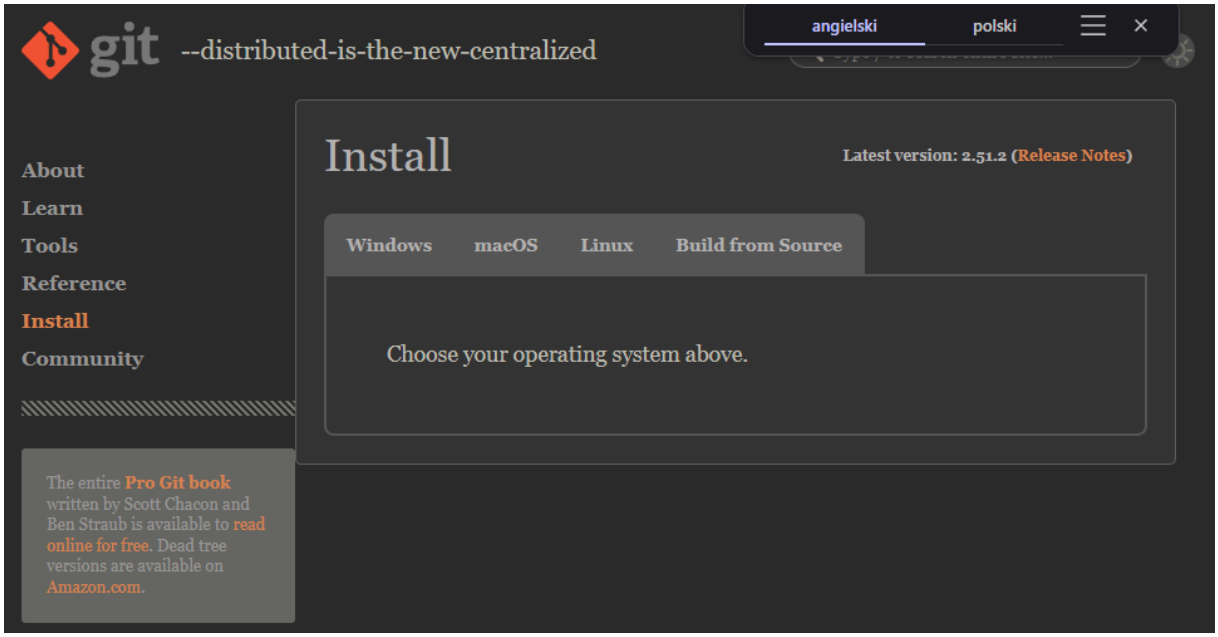
Aby zacząć pracę z gitem, musimy na samym początku dać mu znać „kim jesteśmy”. W tym celu używamy:

```
git config --global user.email "email@example.com"
git config --global user.name "Imie␣Nazwisko"
git init
git add .
git commit -m "Komentarz␣do␣commitu"
git remote add origin https://github.com/uzytkownik/repo.git
git push -u origin master
```

Listing 1.1: Podstawowe Komendy Git

1.5 Instalacja GIT’a

Aby zainstalować GIT’a najlepiej skorzystać z oficjalnego źródła: https://git-scm.com/downloads.



Rysunek 1.2: Rys. 1.2. Wersje do pobrania

Rozdział 2

Analiza problemu

2.1 Wymagane klasy

Aby program spełniał założenia trzeba przemyśleć jakie klasy będą potrzebne. Pierwszą klasą będzie węzeł (node). Nastepnie iterator, klasa list oraz fabryka.

2.2 Wzorzec projektowy

Omówienie wzorców: nazwa, problem, rozwiązanie, konsekwencje. W kontekście C++ użycie polimorfizmu, szablonów i RAII.

2.3 Fabryka

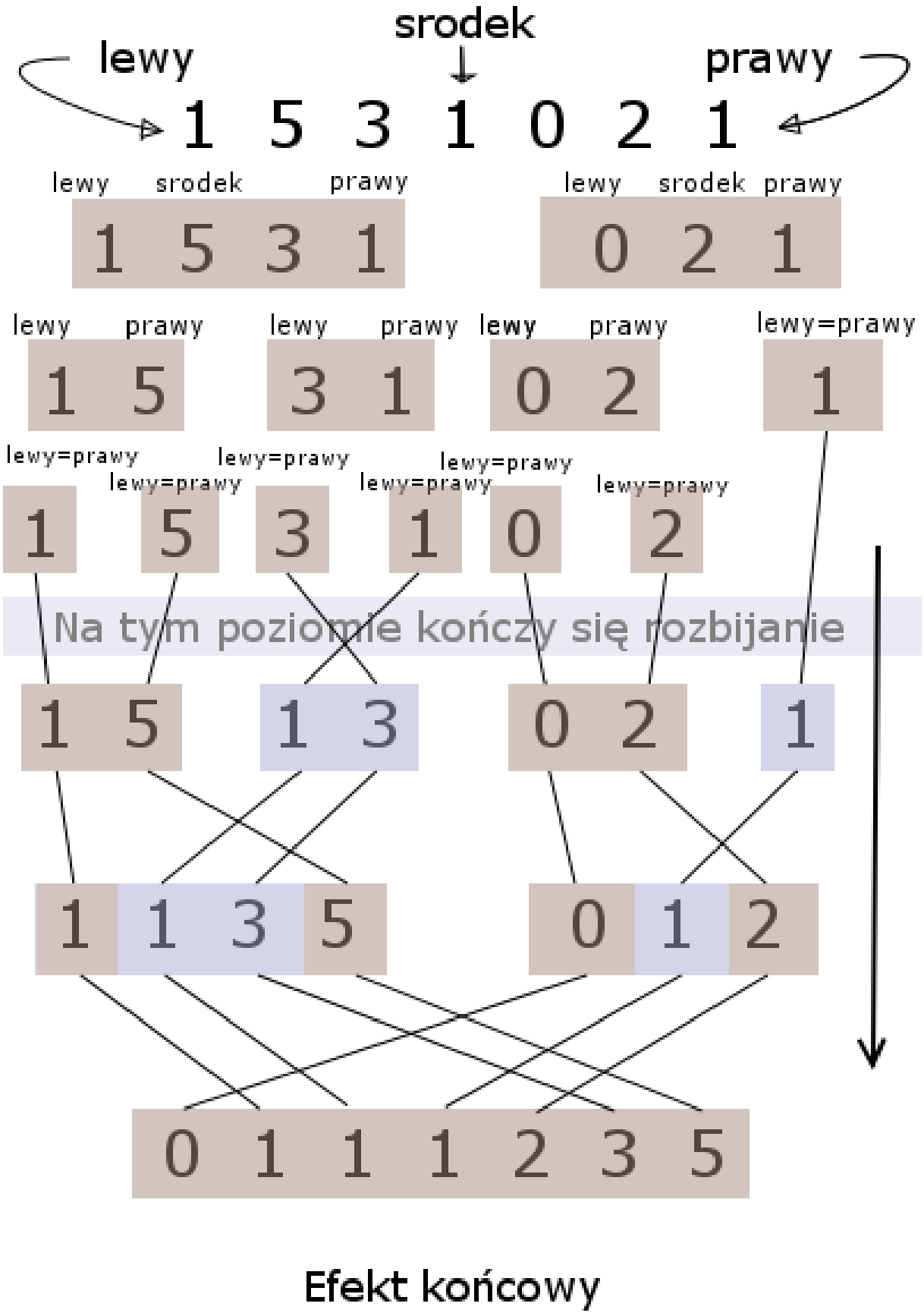
Fabryka to kreacyjny wzorzec projektowy — metoda wytwórcza tworząca obiekt zamiast konstruowania go bezpośrednio.

2.4 Iterator

Wzorzec Iterator — access do elementów kontenera bez ujawniania wnętrza. Implementacja powinna dostarczać metody `hasNext`, `next`, `hasPrevious`, `previous`.

2.5 Algorytmy działające na listach dwukierunkowych

Opis operacji: wstawianie, usuwanie, wyszukiwanie (liniowe, obu-kierunkowe), sortowanie (insertion sort, merge sort), z krótkimi opisami implementacji i złożoności.



Efekt końcowy

Rysunek 2.1: Rys. 2.1. Schemat ilustrujący działanie sortowanie przez scalanie

Rozdział 3

Projektowanie

3.1 Środowisko programistyczne

Wybrano Visual Studio 2022 (MSVC), argumenty: wsparcie C++17/C++20, wygoda IDE, 64-bit.

3.2 Język programowania

C++: kontrola wskaźników, wydajność, kompilowany język odpowiedni dla struktur niskiego poziomu.

3.3 System Kontroli Wersji: Git

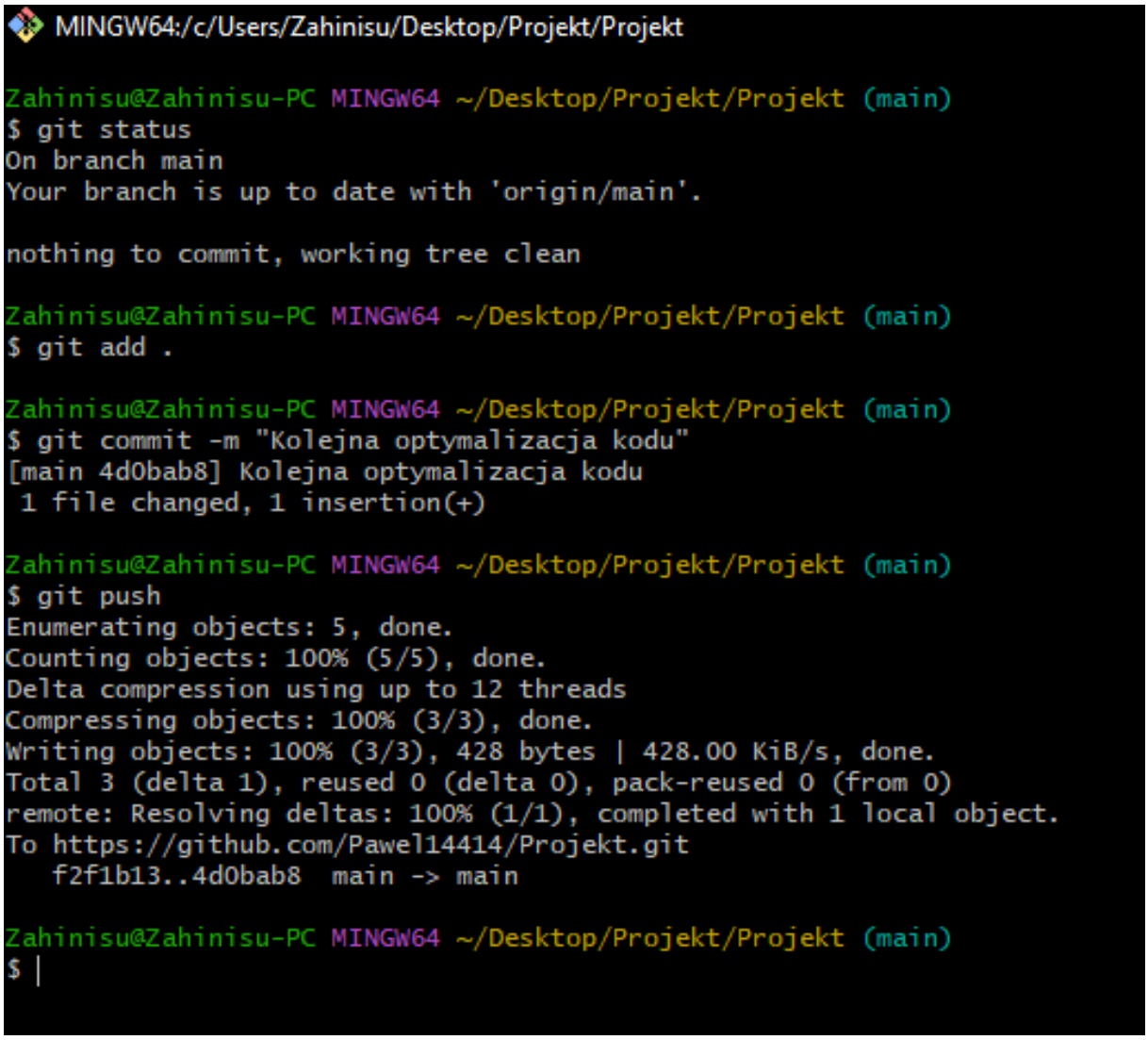
Git jako narzędzie do kontroli wersji, gałęzi i współpracy.

3.4 Platforma Hostingowa i Współpracy: GitHub

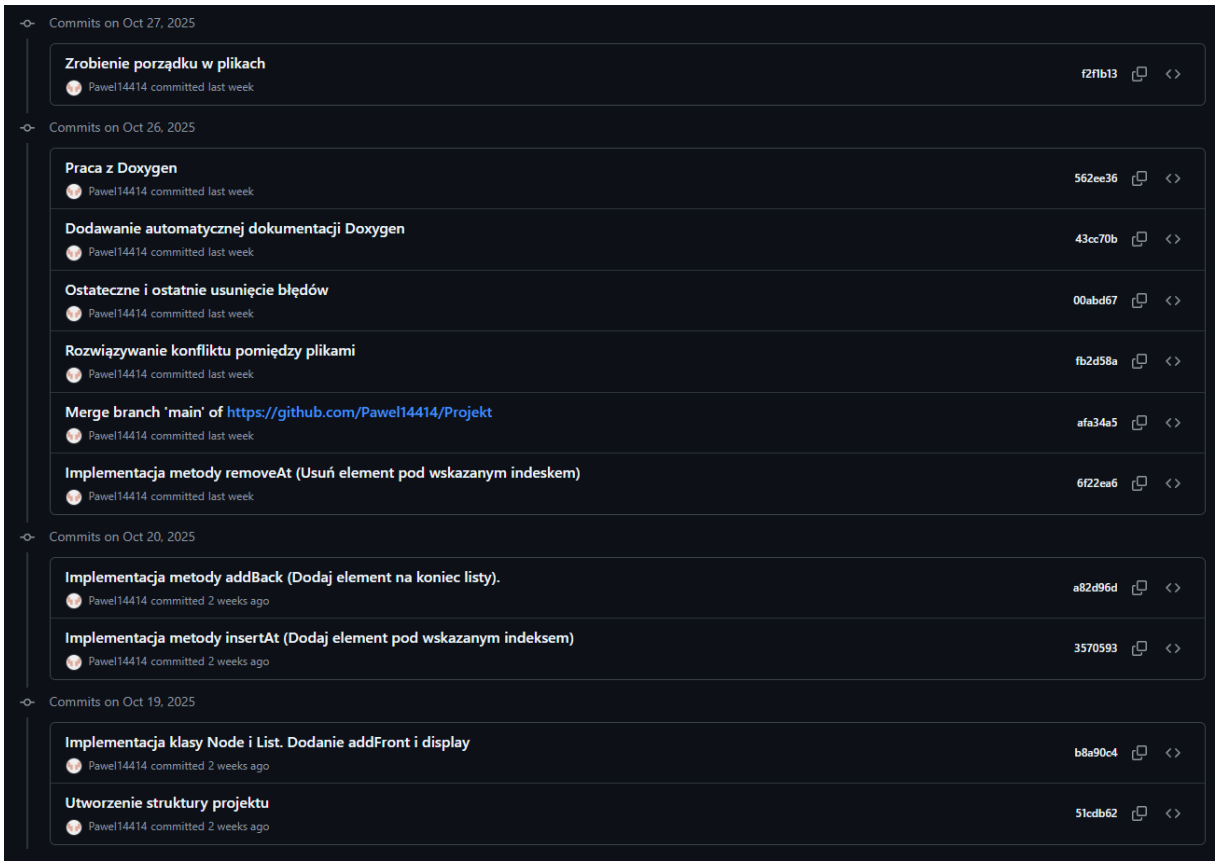
GitHub: zdalne repozytorium, PR, Issues, Actions itd.

3.5 Przebieg pracy z systemem kontroli wersji Git

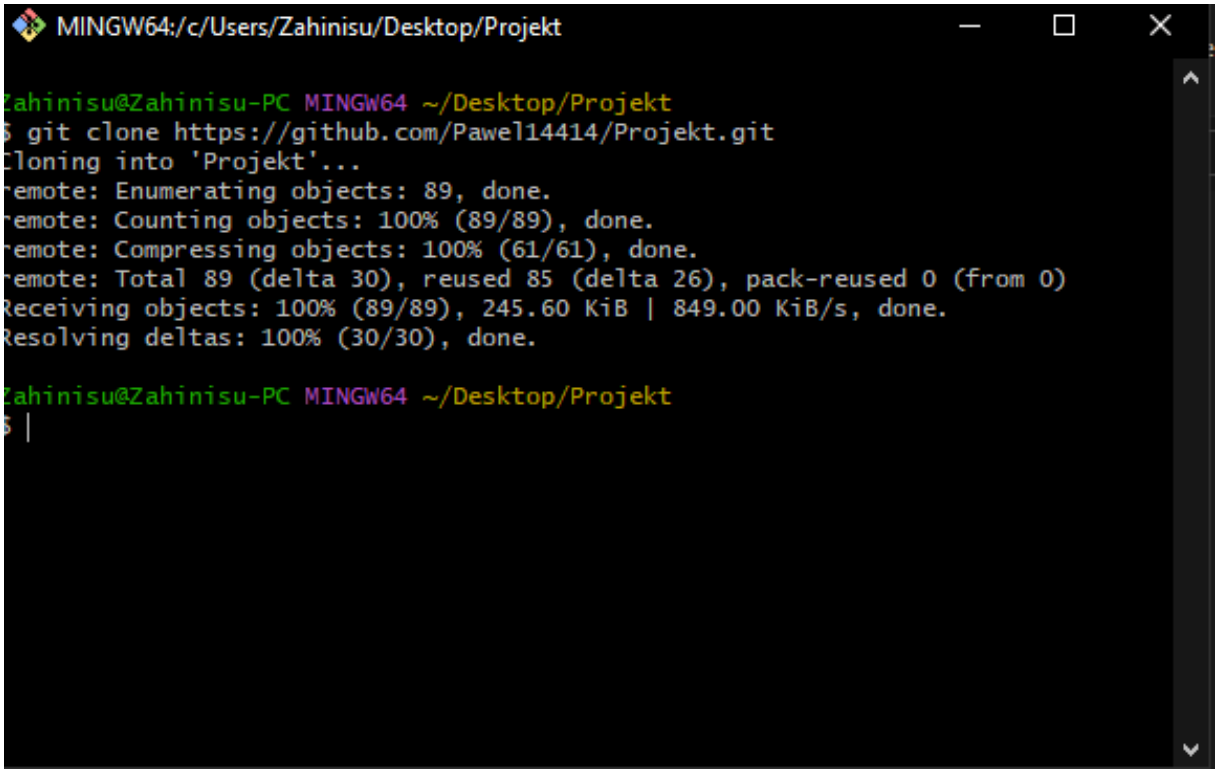
Opis procesu ‘git add‘, ‘git commit‘, ‘git push‘, ‘git pull‘, .gitignore.



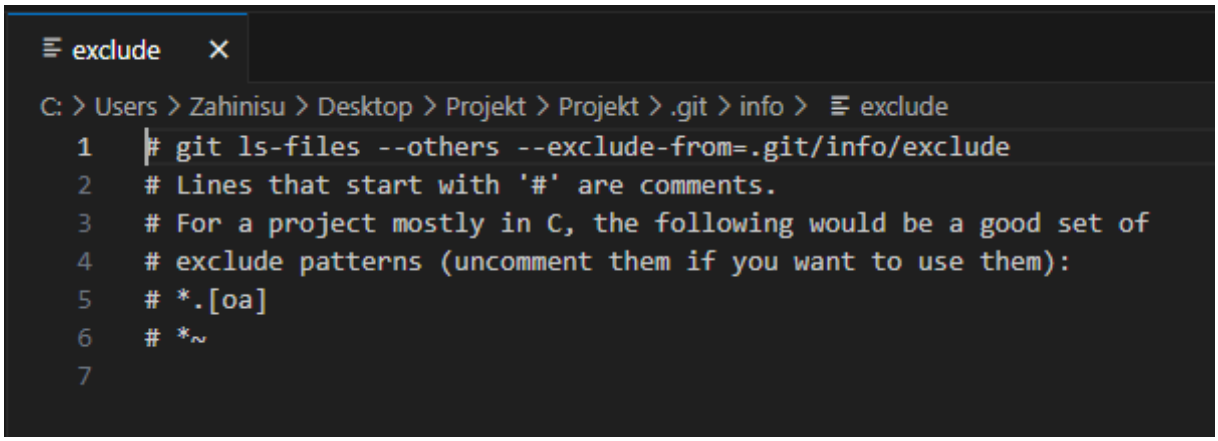
Rysunek 3.1: Rys. 3.1. Cyk pracy git add, git commit i git push



Rysunek 3.2: Rys. 3.2. Szczegółowa historia zmian (commitów) widoczna w panelu repozytorium na platformie GitHub



Rysunek 3.3: Rys. 3.3. Proces pobrania respozytorium z GitHub na repozytorium lokalne



Rysunek 3.4: Rys. 3.5. Zawartość pliku konfiguracyjnego .gitignore używanego w projekcie

Rozdział 4

Implementacja

W implementacji użyto C++ oraz klas: `node`, `list`, `listIterator`, `Factory`. Kod rozdzielony do plików nagłówkowych i cpp.

4.1 Implementacja klas

4.1.1 Klasa Węzła (Node)

Plik `Node.h`:

```
#ifndef NODE_H
#define NODE_H

class node {
public:
    int data;
    node* next;
    node* previous;
    node(int value);
};

#endif
```

Listing 4.1: Node.h

Opis: trzy pola publiczne: `int data`, `node* next`, `node* previous`. Konstruktor inicjalizuje wskaźniki na `nullptr`.

4.1.2 Klasa Listy (list)

Plik `List.h`:

```
class list {
private:
    node* head;
    node* tail;
public:
    list();
    void push_front(int number);
    void push_back(int number);
    void push_at(int number , int index);
    void pop_back ();
    void pop_front ();
    void pop_at(int index);
    void show();
    void showReverse ();
    void show_next(int index);
    void show_previous(int index);
    void clear();
    ~list();
};
```

Listing 4.2: List.h

Fragment implementacji `push_front` (List.cpp):

```
void list:: push_front(int number) {
    node* newEl = new node(number);
    if (head != nullptr) {
        head->previous = newEl;
        newEl->next = head;
    } else {
        tail = newEl;
    }
    head = newEl;
}
```

Listing 4.3: List.cpp - push_front

Metoda `clear`:

```
void list:: clear() {
    node* current = head;
    while (current) {
        node* temp = current->next;
        delete current;
```

```
        current = temp;
    }
    head = nullptr;
    tail = nullptr;
}
```

Listing 4.4: List.cpp - clear

4.2 Implementacja wzorców

4.2.1 Wzorzec Iterator (listIterator)

Plik `Iterator.h`:

```
class listIterator {
private:
    node* current;
    bool reverse;
public:
    listIterator(node* start , bool reverseOrder = false);
    bool hasNext ();
    int next();
    bool hasPrevious ();
    int previous ();
};
```

Listing 4.5: Iterator.h

Użycie w metodzie `show` i `showReverse`:

```
void list::show() {
    listIterator it(head , false);
    if (!it.hasNext()) {
        cout << "Lista␣jest␣pusta\n";
        return;
    }
    while (it.hasNext()) {
        cout << it.next() << "\n";
    }
}

void list:: showReverse () {
    listIterator it(tail , true);
    if (tail == nullptr) {
```

```
15          cout << "Lista jest pusta\n";
16          return;
17      }
18      while (it.hasPrevious()) {
19          cout << it.previous() << "\n";
20      }
21  }
```

Listing 4.6: List.cpp - show / showReverse

4.2.2 Wzorzec Fabryka (Factory)

Plik `Factory.h`:

```
1  class Factory {
2  public:
3      static list* createlist ();
4  };
```

Listing 4.7: Factory.h

Plik `Factory.cpp`:

```
1  #include "Factory.h"
2  list* Factory :: createlist () {
3      return new list();
4  }
```

Listing 4.8: Factory.cpp

Użycie w `main.cpp`:

```
1  list* lista = Factory :: createlist ();
```

Listing 4.9: main.cpp - tworzenie listy

4.3 Prezentacja działania i testy

Scenariusz testowy (fragmenty z `main.cpp`):

```
1  (...)
2  cout << "\n--- Dodawanie elementow (...) ---" << endl;
3  lista->push_front(10);
4  lista->push_back(5);
5  lista->push_at(15, 2);
6  cout << "Lista (show): ";
```

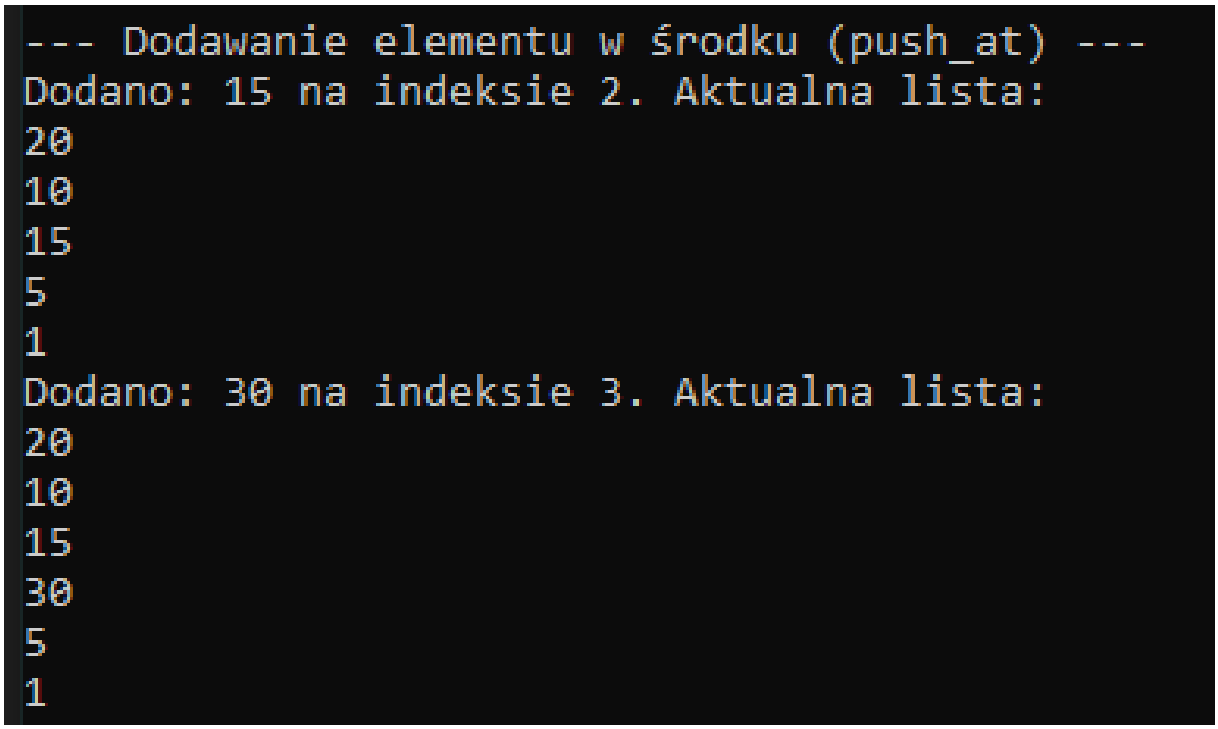
```
lista->show();
cout << "Lista od tylu (showReverse): ";
lista->showReverse();
(...)
cout << "\n--- Usuwanie elementu z konca (pop_back) ---" << endl;
lista->pop_back();
cout << "Aktualna lista: ";
lista->show();
(...)
cout << "\n--- Calkowite czyszczenie listy (clear) ---" << endl;
lista->clear();
lista->show();
(...)
delete lista;
(...)
```

Listing 4.10: main.cpp - scenariusz testowy

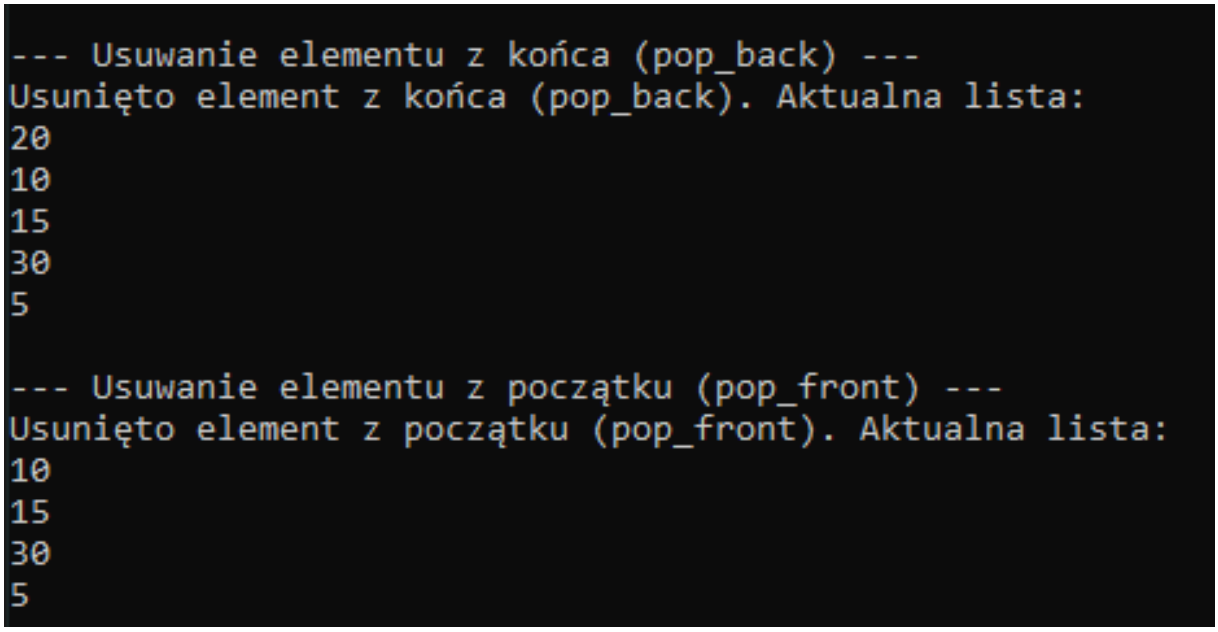
```
--- Inicjalizacja listy ---

--- Dodawanie elementów na początek (push_front) i na koniec (push_back) ---
Dodano: 10 (push_front). Aktualna lista:
10
Dodano: 20 (push_front). Aktualna lista:
20
10
Dodano: 5 (push_back). Aktualna lista:
20
10
5
Dodano: 1 (push_back). Aktualna lista:
20
10
5
1
```

Rysunek 4.1: Rys. 4.1. Dodawanie na początek i koniec



Rysunek 4.2: Rys. 4.2. Dodawanie w środku



Rysunek 4.3: Rys. 4.3. Usuwanie z końca i z początku

```
--- Usuwanie elementu ze środka (pop_at) ---
Usunięto element z indeksu 3 (pop_at(3)). Aktualna lista:
99
10
15
5

--- Próba usunięcia z nieistniejącego indeksu (pop_at) ---
Próba usunięcia elementu z indeksu 10. Aktualna lista:
99
10
15
5
```

Rysunek 4.4: Rys. 4.4. Usuwanie ze środka oraz usuwanie nieistniejącego indeksu

```
--- Wyświetlenie listy w obu kierunkach ---
Lista (show): 20
10
15
30
5
1
Lista od tyłu (showReverse):
1
5
30
15
10
20
```

Rysunek 4.5: Rys. 4.5. Wyświetlanie elementów od początku i od końca

```
--- Wyświetlenie sąsiadów dla elementu na indeksie 1 (show_next, show_previous) ---
Następny element po indeksie 1:
Nastepny po indeksie 1 (10) to: 15
Poprzedni element przed indeksem 1:
Poprzedni dla indeksu 1 (10) to: 99

--- Wyświetlenie listy od tyłu po operacjach (showReverse) ---
Lista od tyłu (showReverse):
5
15
10
99

--- Całkowite czyszczenie listy (clear) ---
Lista została wyczyszczona (clear).
Aktualna lista (show):
Lista jest pusta

--- Dodanie elementu po czyszczeniu ---
Dodano: 500 (push_front). Aktualna lista:
500

--- Usuwanie listy z pamięci (delete) ---
Pamięć zwolniona.
```

Rysunek 4.6: Rys. 4.6. Wyświetlanie sąsiadów, wyświetlanie listy od tyłu, czyszczenie listy, dodanie elementu po wyczyszczeniu listy, usunięcie listy z pamięci

Rozdział 5

Wnioski

5.1 Git oraz GitHub

W trakcie realizacji projektu zdobyto praktyczne umiejętności związane z wykorzystaniem systemu kontroli wersji **Git** oraz platformy **GitHub**. Praca z tymi narzędziami umożliwiła zrozumienie, jak istotne w pracy programisty jest śledzenie zmian w kodzie źródłowym, kontrola wersji oraz możliwość powrotu do wcześniejszych etapów projektu.

Nauka korzystania z poleceń takich jak `git init`, `git add`, `git commit`, `git push`, czy `git pull` pozwoliła na opanowanie pełnego cyklu pracy z repozytorium. Dzięki wykorzystaniu zdalnego repozytorium na GitHub możliwe było utrzymywanie bezpiecznej kopii zapasowej projektu oraz wersjonowanie każdej modyfikacji w sposób przejrzysty i kontrolowany.

Projekt ten unaocznił również, jak duże znaczenie w praktyce ma stosowanie pliku `.gitignore`, który pozwala uniknąć dodawania do repozytorium zbędnych lub prywatnych plików (np. plików binarnych, tymczasowych czy konfiguracyjnych IDE).

Zastosowanie GitHuba jako centralnego miejsca przechowywania kodu okazało się bardzo wygodne — interfejs webowy pozwalał przeglądać historię commitów, śledzić zmiany i wizualizować strukturę projektu. Poznanie podstaw **branchingu** oraz **mergingu** pozwoliło zrozumieć, jak efektywnie prowadzić równoległy rozwój funkcjonalności bez ryzyka utraty danych.

Podsumowując, praca z systemem Git i platformą GitHub w znaczący sposób wpłynęła na zrozumienie zasad profesjonalnego zarządzania projektem programistycznym. Umiejętność wersjonowania kodu i współpracy zdalnej jest dziś nieodzowna w każdym zespole programistycznym, niezależnie od skali projektu.

5.2 Listy Dwukierunkowe

Zaimplementowanie listy dwukierunkowej w języku **C++** pozwoliło lepiej zrozumieć działanie wskaźników, zarządzanie pamięcią oraz powiązania między elementami struktur danych. Realizacja projektu wymagała przemyślenia struktury programu, zaprojektowania klas, a także właściwego podziału odpowiedzialności między poszczególne komponenty.

Wykorzystanie klas `node`, `list`, `listIterator` oraz `Factory` umożliwiło utrzymanie przejrzystości kodu oraz zastosowanie podstawowych wzorców projektowych. Dzięki temu program jest nie tylko funkcjonalny, ale również elastyczny i łatwy w rozbudowie. Wzorzec **Iterator** umożliwił eleganckie przechodzenie po liście bez ujawniania jej wewnętrznej struktury, natomiast wzorzec **Factory** pozwolił na oddzielenie logiki tworzenia obiektów od ich użycia.

Podczas implementacji szczególną uwagę zwrócono na poprawne zarządzanie pamięcią dynamiczną oraz zwalnianie nieużywanych zasobów w destruktorach. Pozwoliło to uniknąć błędów takich jak wycieki pamięci, które są typowe dla projektów w językach niskopoziomowych.

Praktyczna implementacja listy dwukierunkowej pokazała również, jak istotne są testy jednostkowe i etapowe uruchamianie programu po każdej zmianie. Dzięki temu łatwiej było identyfikować błędy i szybko je korygować.

Podsumowując, realizacja projektu była nie tylko ćwiczeniem z zakresu programowania obiektowego, ale również cenną lekcją projektowania i organizacji kodu. Pozwoliła połączyć wiedzę teoretyczną ze struktur danych z praktycznymi umiejętnościami pracy z kodem w środowisku inżynierskim.

5.3 Podsumowanie ogólne

Projekt **„Algorytm listy dwukierunkowej z zastosowaniem GitHub”** pozwolił połączyć elementy teorii informatyki, inżynierii oprogramowania oraz narzędzi deweloperskich w spójną całość. W ramach pracy zrealizowano zarówno część programistyczną (implementacja struktury danych i wzorców), jak i organizacyjną (kontrola wersji, zarządzanie projektem w repozytorium).

Najważniejsze umiejętności zdobyte podczas realizacji projektu to:

- zrozumienie zasad działania list dwukierunkowych i implementacja ich w języku C++;
- umiejętność stosowania wzorców projektowych w praktyce;
- wykorzystanie systemu Git i platformy GitHub w zarządzaniu projektem;
- praca z kodem w sposób iteracyjny i kontrolowany;

- praktyczne doświadczenie z debugowaniem i testowaniem programów;
- tworzenie przejrzystej i czytelnej dokumentacji technicznej w LaTeXu.

Zrealizowany projekt stanowi solidną podstawę do dalszego rozwoju w kierunku bardziej złożonych struktur danych, algorytmów oraz profesjonalnych praktyk w zakresie wytwarzania oprogramowania.

Bibliografia

[1] Wikipedia - Lista. https://pl.wikipedia.org/wiki/Lista (term. wiz. 18.10.2025).

[2] Lasindi - Praca własna, Domena publiczna. https://commons.wikimedia.org/w/index.php?curid=2245165 (term. wiz. 19.10.2025).

[3] Serwis Edukacyjny w I-LO w Tarnowie - Operacje na listach dwukierunkowych. https://eduinf.waw.pl/inf/alg/001_search/0087.php (term. wiz. 18.10.2025).

[4] Wikipedia - Git. https://en.wikipedia.org/wiki/Git (term. wiz. 18.10.2025).

[5] Wikipedia - Code Patterns Design. https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns (term. wiz. 24.10.2025).

[6] Wikipedia - Factory Method Pattern. https://en.wikipedia.org/wiki/Factory_method_pattern (term. wiz. 24.10.2025).

[7] Wikipedia - Iterator Pattern. https://en.wikipedia.org/wiki/Iterator_pattern (term. wiz. 24.10.2025).

[8] Tutorialspoint - Doubly Linked List Data Structure. https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_algorithm.htm (term. wiz. 24.10.2025).

[9] Serwis Edukacyjny - Insertion Sort. https://eduinf.waw.pl/inf/alg/003_sort/0010.php (term. wiz. 24.10.2025).

[10] Serwis Edukacyjny - Merge Sort. https://eduinf.waw.pl/inf/alg/003_sort/0013.php (term. wiz. 24.10.2025).

[11] Wikipedia - C++. https://en.wikipedia.org/wiki/C%2B%2B (term. wiz. 19.10.2025).

[12] Wikipedia - GitHub. https://en.wikipedia.org/wiki/GitHub (term. wiz. 24.10.2025).