Sprawozdanie 04: Implementacja list jedno- i dwukierunkowych oraz cyklicznych w C++

Dmytro Sakharskyi - L02 - 31259 16.04.2025

Metody Programowania

Informatyka 1 rok. 2 semestr

Wstęp

Struktury danych odgrywają kluczową rolę w programowaniu, umożliwiając efektywne zarządzanie informacjami w pamięci komputera. Jednym z podstawowych typów struktur dynamicznych są **listy** – zbiory elementów powiązanych ze sobą za pomocą wskaźników.

Na zajęciach skupiliśmy się na trzech rodzajach list:

- **Lista jednokierunkowa** – w której każdy element wskazuje na kolejny.
- **Lista dwukierunkowa** – gdzie każdy element zna zarówno swojego następnika, jak i poprzednika.
- **Lista cykliczna** – w której ostatni element wskazuje z powrotem na pierwszy, tworząc zamknięty krąg.

Celem ćwiczenia było poznanie sposobu implementacji tych struktur w języku C++, nauczenie się podstawowych operacji na listach (dodawanie, usuwanie, przeszukiwanie, liczenie węzłów), a także przeprowadzenie prostych testów wydajnościowych związanych

z limitem długości tablicy i listy. Wszystkie operacje wykonywane były przy użyciu dynamicznej alokacji pamięci oraz wskaźników, co pozwoliło lepiej zrozumieć sposób działania takich struktur “od środka”.

Lista jednokierunkowa

Zadanie 1

```
struct Node {
    int data;
    Node* next;
};
```

Struktura **Node** reprezentuje pojedynczy węzeł listy jednokierunkowej. Zawiera dwie składowe: data, która przechowuje wartość typu **int** oraz **next**, czyli wskaźnik do następnego elementu listy. Jest to podstawowy budulec dynamicznej listy.

```
void pushFront(Node*& head, int value) {
    Node* newNode = new Node{ value, head };
    head = newNode;
}

void pushBack(Node*& head, int value) {
    Node* newNode = new Node{ value, nullptr };
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) temp = temp->next;
    temp->next = newNode;
}

void popFront(Node*& head) {
    if (!head) return;
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

Tworzymy nowy węzeł typu **Node** z wartością **value**, a wskaźnik **next** ustawiamy na obecny **head**.

Zmieniamy **head**, by wskazywał na nowo dodany węzeł. Dzięki temu nowy węzeł trafia na początek listy.

Zadanie 2

```
int n;
cout << "Podaj liczbe wezlow: ";
cin >> n;
for (int i = 0; i < n; i++) {
    pushBack(head, rand() % 100);
}
printList(head);
```

Program pyta użytkownika o liczbę węzłów. Następnie generuje listę jednokierunkową o zadanej długości, przydzielając każdemu węzłowi losową wartość z zakresu 0–99.

Zadanie 3

```
int countNodes(Node* head) {
    int count = 0;
    while (head) {
        count++;
        head = head->next;
    }
    return count;
}
```

Funkcja **countNodes** przechodzi przez całą listę i zlicza liczbę węzłów, aż wskaźnik head będzie równy **nullptr**. Zwraca całkowitą liczbę elementów w liście.

Zadanie 4

```
int liczba = countNodes(head);
cout << "Liczba wezlow: " << liczba << endl;
if (n == liczba) cout << "Zgadza sie." << endl;
else cout << "Nie zgadza sie." << endl;
```

Porównujemy liczbę podaną przez użytkownika z rzeczywistą liczbą węzłów policzoną przez funkcję **countNodes**. Jeśli wartości się zgadzają, oznacza to, że pętla generująca listę działa poprawnie.

```
20 -> 10 -> 5 -> NULL
10 -> 5 -> NULL
10 -> NULL
Podaj liczbe wezlow: 10
41 -> 67 -> 34 -> 0 -> 69 -> 24 -> 78 -> 58 -> 62 -> 64 -> NULL
Liczba wezlow: 10
Zgadza sie.
```

Podałem wartość **10** jako oczekiwaną liczbę węzłów w liście. Po wygenerowaniu listy, funkcja countNodes również zwróciła **10**. Oznacza to, że liczba zadeklarowana przez użytkownika zgadza się z faktyczną liczbą węzłów w liście. Program poprawnie przeprowadził porównanie i wypisał komunikat "**Zgadza się**." — tym samym potwierdzając zgodność obu wartości.

Zadanie 5

```
void popBack(Node*& head) {
    if (!head) return;
    if (!head->next) {
        delete head;
        head = nullptr;
        return;
    }
    Node* temp = head;
    while (temp->next->next) temp = temp->next;
    delete temp->next;
    temp->next = nullptr;
}
```

Funkcja **popBack** służy do usuwania ostatniego elementu z listy jednokierunkowej. Aby to zrobić:

1. Najpierw sprawdza, czy lista jest pusta **(head == nullptr)**. Jeśli tak, nie robimy nic.
2. Następnie sprawdza, czy lista zawiera tylko jeden węzeł **(head->next == nullptr)**. W takim przypadku usuwa ten pojedynczy węzeł i ustawia wskaźnik

head na nullptr, by wskazywał pustą listę.
3. W przypadku listy z więcej niż jednym węzłem przechodzimy przez listę aż do przedostatniego elementu **(czyli takiego, którego next->next == nullptr).**
4. Usuwa ostatni element za pomocą delete temp->next i ustawiamy wskaźnik next przedostatniego węzła na nullptr, by zamknąć listę.

Zadanie 6

```
Node* findNode(Node* head, int value) {
    while (head) {
        if (head->data == value) return head;
        head = head->next;
    }
    return nullptr;
}
```

Funkcja **findNode** przeszukuje listę w poszukiwaniu węzła o wartości wskazanej przez użytkownika. Jeśli znajdzie pasujący element, zwraca jego adres. W przeciwnym razie zwraca nullptr.

```
Podaj liczbe wezlow: 10
41 -> 67 -> 34 -> 0 -> 69 -> 24 -> 78 -> 58 -> 62 -> 64 -> NULL
Liczba wezlow: 10
Zgadza sie.
Podaj wartosc do znalezienia: 34
Znaleziono wezel z wartoscia: 34
```

Funkcja findNode przeszukała listę i zwróciła adres węzła zawierającego tę wartość. Program następnie wypisał komunikat "**Znaleziono wezel z wartoscia: 34**", co oznacza, że funkcja działa poprawnie i potrafi skutecznie odnaleźć element o zadanej wartości w liście jednokierunkowej.

Lista dwukierunkowa

Zadanie 1

```cpp
struct DNode {
    int data;
    DNode* prev;
    DNode* next;
};

void pushFront(DNode*& head, int value) {
    DNode* newNode = new DNode{ value, nullptr, head };
    if (head) head->prev = newNode;
    head = newNode;
}
```

```cpp
void popBack(DNode*& head) {
    if (!head) return;
    if (!head->next) {
        delete head;
        head = nullptr;
        return;
    }
    DNode* temp = head;
    while (temp->next) temp = temp->next;
    temp->prev->next = nullptr;
    delete temp;
}

void printList(DNode* head) {
    while (head) {
        cout << head->data << " <-> ";
        head = head->next;
    }
    cout << "NULL" << endl;
}
```

DNode — struktura reprezentująca węzeł listy dwukierunkowej z polem danych data oraz wskaźnikami do poprzedniego i następnego węzła.

pushFront — dodaje nowy węzeł na początek listy.

- Jeśli lista nie była pusta, ustawia prev obecnego head'a na nowy węzeł.

popBack — usuwa węzeł z końca listy.

- Przechodzi na koniec i usuwa ostatni element, odłączając go od listy.

printList — wypisuje wszystkie elementy listy w kolejności od początku do końca, oddzielając je znakiem <->.

Testowe działanie:

```
20 <-> 10 <-> NULL
20 <-> NULL
Podaj liczbe wezlow: 4
0 <-> 34 <-> 67 <-> 41 <-> 20 <-> NULL
```

Zadanie 2

```
int n;
cout << "Podaj liczbe wezlow: ";
cin >> n;
for (int i = 0; i < n; ++i) {
    pushFront(head, rand() % 100);
}
printList(head);
```

Użytkownik podaje liczbę węzłów.

Program tworzy pętlę for, która n razy wywołuje **pushFront** z losową liczbą z zakresu 0–99.

Na końcu lista jest wypisana za pomocą **printList**.

Lista cykliczna

Zadanie 1

```cpp
struct CNode {
    int data;
    CNode* next;
};

void addNode(CNode*& head, int value) {
    CNode* newNode = new CNode{ value, nullptr };
    if (!head) {
        newNode->next = newNode;
        head = newNode;
        return;
    }
    CNode* temp = head;
    while (temp->next != head) temp = temp->next;
    temp->next = newNode;
    newNode->next = head;
}
```

```cpp
void removeNode(CNode*& head, int value) {
    if (!head) return;
    CNode* curr = head;
    CNode* prev = nullptr;
    do {
        if (curr->data == value) {
            if (prev) {
                prev->next = curr->next;
                if (curr == head) head = curr->next;
                delete curr;
            }
            else {
                if (curr->next == head) {
                    delete head;
                    head = nullptr;
                }
                else {
                    CNode* tail = head;
                    while (tail->next != head) tail = tail->next;
                    tail->next = head->next;
                    delete head;
                    head = tail->next;
                }
            }
            return;
        }
        prev = curr;
        curr = curr->next;
    } while (curr != head);
}
```

CNode to struktura węzła z wartością data i wskaźnikiem na kolejny element.

addNode dodaje węzeł do końca listy cyklicznej, ustawiając next nowego węzła na początek listy.

removeNode znajduje i usuwa węzeł o określonej wartości. Obsługuje przypadki:

- Usuwanie pierwszego węzła.
- Usuwanie ostatniego.
- Usuwanie jedynego węzła.

Zadanie 2

```
int arraySize, listSize;
cout << "Podaj rozmiar tablicy do testu: ";
cin >> arraySize;
cout << "Podaj rozmiar listy do testu: ";
cin >> listSize;

try {
    int* bigArray = new int[arraySize];
    cout << "Utworzono duza tablice." << endl;
    delete[] bigArray;
}
catch (bad_alloc&) {
    cout << "Nie udalo sie utworzyc duzej tablicy." << endl;
}

CNode* bigList = nullptr;
try {
    for (int i = 0; i < listSize; ++i) {
        addNode(bigList, i);
    }
    cout << "Utworzono duza liste." << endl;
}
catch (bad_alloc&) {
    cout << "Nie udalo sie utworzyc duzej listy." << endl;
}
```

1. Test tworzenia dużej tablicy:

int* bigArray = new int[arraySize];

- Program próbuje utworzyć dynamiczną tablicę o rozmiarze podanym przez użytkownika.
- Jeśli rozmiar jest zbyt duży, new zgłasza wyjątek bad_alloc, co oznacza, że nie udało się przydzielić ciągłego bloku pamięci.
- W takim przypadku program wypisuje komunikat, że tablica nie mogła zostać utworzona.

2. Test tworzenia dużej listy:

```
for (int i = 0; i < listSize; ++i) {
    addNode(bigList, i);
}
```

- Program tworzy listę cykliczną, dodając listSize elementów jeden po drugim.
- Każdy element alokowany jest osobno w pamięci (przez new w funkcji addNode).
- Jeśli pamięć się kończy, new również rzuca bad_alloc, a program wypisuje odpowiedni komunikat.

3. Test max:

Tablica:

- Program działał poprawnie do rozmiaru około **10^10** (dziesięć miliardów).
- Przy **10^11** pojawiał się wyjątek bad_alloc — system nie był w stanie przydzielić tak dużego bloku pamięci.

Lista:

- Tworzenie listy działało bez problemu do rozmiaru około **10^4** (10 tysięcy).
- Powyżej **10^5** program zawieszał się lub zgłaszał bad_alloc.

Maksimum:

```
10 -> 20 -> 30 -> (powrot do poczatku)
10 -> 30 -> (powrot do poczatku)
Podaj rozmiar tablicy do testu: 1000000000
Podaj rozmiar listy do testu: 10000
Utworzono duza tablice.
Utworzono duza liste.
```

Wnioski

Lista jednokierunkowa umożliwia dodawanie elementów na początek i na koniec oraz usuwanie elementu z początku. Wyszukiwanie wartości wymaga przejścia całej listy od początku do końca. Struktura zawiera wskaźnik tylko do następnego elementu.

Lista dwukierunkowa umożliwia przechodzenie w obu kierunkach dzięki wskaźnikom **prev i next**. Dodawanie elementów na początek oraz usuwanie elementów z końca zostało zrealizowane i przetestowane. Struktura wymaga więcej pamięci na każdy węzeł niż lista jednokierunkowa.

Lista cykliczna posiada połączenie ostatniego elementu z pierwszym, co pozwala na ciągłe przechodzenie przez wszystkie węzły bez punktu końcowego. Dodawanie i usuwanie węzłów działa w oparciu o identyfikację wartości i odpowiednie przestawianie wskaźników.

W eksperymencie z dynamiczną **tablicą** zaobserwowano, że alokacja kończy się niepowodzeniem (bad_alloc) przy wartości pomiędzy **10^10 a 10^11**.

W eksperymencie z **listą** dynamiczną (cykliczną) alokacja elementów kończy się niepowodzeniem (bad_alloc) przy liczbie elementów pomiędzy **10^4 a 10^5**.

W przypadku tablicy dynamicznej program próbuje przydzielić ciągły blok pamięci. W przypadku listy dynamicznej alokacja odbywa się pojedynczo dla każdego elementu.

Operacje na listach wymagają przechodzenia przez węzły, co powoduje liniową złożoność czasową w przypadku większości funkcji.

Lista jednokierunkowa, dwukierunkowa i cykliczna zostały zaimplementowane zgodnie z wymaganiami i przetestowane pod kątem podstawowych operacji oraz poprawności działania.