Sprawozdanie 05: Praktyczna implementacja liniowych struktur danych: stosy i kolejki

Dmytro Sakharskyi - L02 - 31259 29.04.2025

Metody Programowania

Informatyka 1 rok. 2 semestr

Wstęp

Ćwiczenie miało na celu praktyczne zastosowanie i implementację dwóch podstawowych liniowych struktur danych: **stosu** oraz **kolejk**, z użyciem statycznej tablicy w języku C++. Obie struktury zostały zrealizowane przy pomocy klasy, zawierającej prywatne pole typu tablica oraz odpowiednie wskaźniki sterujące (top dla stosu, front i rear dla kolejki).

W ramach ćwiczenia wykonano następujące zadania:

- przepisanie i analiza gotowych fragmentów kodu implementujących operacje podstawowe dla stosu i kolejki,
- przetestowanie metod **isEmpty, isFull, push, pop, peek, enqueue, dequeue**,
- porównanie działania różnych wariantów zapisu instrukcji **push**,
- napisanie pętli wprowadzającej do struktur wartości pseudolosowe w ilości określonej przez użytkownika,
- implementacja funkcji zwracających liczbę elementów w strukturze,
- porównanie liczby elementów podanej przez użytkownika z liczbą faktycznie obecną w strukturze,
- implementacja metod **printStack** i **printQueue**, wyświetlających zawartość struktur w określonym porządku,
- implementacja metod **resetStack** i **resetQueue**, umożliwiających logiczne wyzerowanie struktur i ich ponowne wykorzystanie.

Stos

Zadanie 1

```
bool isEmpty() {
    return top == -1;
}
```

isEmpty — Sprawdza, czy stos jest pusty. Jeśli wskaźnik top ma wartość -1, oznacza to, że żaden element nie został dodany.

```
bool isFull() {
    return top == MAX - 1;
}
```

isFull — Sprawdza, czy stos osiągnął maksymalną pojemność.

```
void push(int x) {
    if (isFull()) {
        cout << "Stos pelny!" << endl;
        return;
    }
    arr[++top] = x;
}

int pop() {
    if (isEmpty()) {
        cout << "Stos pusty!" << endl;
        return -1;
    }
    return arr[top--];
}

int peek() {
    if (isEmpty()) {
        cout << "Stos pusty!" << endl;
        return -1;
    }
    return arr[top];
}
```

- **push** — Dodaje nowy element na stos. Najpierw zwiększa top, a potem zapisuje wartość x pod tym indeksem.
- **pop** — Zwraca i usuwa element ze szczytu stosu. Najpierw pobiera go, potem zmniejsza top.
- **peek** — Zwraca wartość szczytowego elementu, ale go nie usuwa.

Test działania:

```
30 20 10
Element z wierzchu: 30
20 10
```

Zadanie 2

```
void push(int x) {
    if (isFull()) {
        cout << "Stos pelny!" << endl;
        return;
    }
    arr[++top] = x;
}
```

arr[++top] = x; — poprawny zapis. Najpierw zwiększa wskaźnik top, potem zapisuje element pod tym indeksem.

arr[top++] = x; — niepoprawny dla pustego stosu (top = -1), bo zapisze do arr[-1] przed zwiększeniem top, co prowadzi do błędu.

Zadanie 3

```
int n;
cout << "Podaj liczbe elementow do dodania: ";
cin >> n;
for (int i = 0; i < n; ++i) {
    s.push(rand() % 100);
}
s.printStack();
```

Test działania:

```
30 20 10
Element z wierzchu: 30
20 10
Podaj liczbe elementow do dodania: 6
24 69 0 34 67 41 20 10
Liczba elementow na stosie: 8
```

Stos został poprawnie zapełniony losowymi wartościami w liczbie podanej przez użytkownika.

Zadanie 4

```
int size() {
    return top + 1;
}
```

Metoda **size()** zwraca ilość elementów znajdujących się aktualnie na stosie. Zmienna top wskazuje indeks ostatniego dodanego elementu. Dla pustego stosu top = -1, czyli stos zawiera 0 elementów. Dodanie każdego nowego elementu zwiększa top o 1. Dlatego faktyczna liczba elementów to top + 1.

Zadanie 5

```
int count = s.size();
cout << "Liczba elementow na stosie: " << count << endl;
if (n == count) cout << "Zgadza sie." << endl;
else cout << "Nie zgadza sie." << endl;
```

Wartość n reprezentuje liczbę elementów, którą użytkownik chciał umieścić na stosie. Wartość **count**, zwrócona przez metodę **size()**, to rzeczywista liczba elementów, jaka znajduje się na stosie po wykonaniu pętli **push().**

Porównanie **n == count** pozwala sprawdzić, czy pętla generująca elementy działała poprawnie i czy wszystkie elementy zostały rzeczywiście odłożone.

Zadanie 6

```
void printStack() {
    if (isEmpty()) {
        cout << "Stos pusty." << endl;
        return;
    }
    for (int i = top; i >= 0; --i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

Metoda **printStack()** przechodzi przez tablicę **arr[]** od indeksu top do 0, wypisując każdy element. Zaczynamy od top, ponieważ stos to struktura **LIFO** — elementy są zdejmowane od góry. Wydruk w tej kolejności odpowiada faktycznemu ułożeniu stosu.

- **isEmpty()** sprawdza, czy stos nie jest pusty — jeśli tak, metoda kończy działanie.
- W przeciwnym razie — wypisuje każdy element od góry do dołu.

Zadanie 7

```
void resetStack() {
    top = -1;
}
```

Metoda **resetStack()** przywraca wartość zmiennej top do **-1**, co odpowiada pustemu stosowi. Tablica arr[] nie jest czyszczona fizycznie, ale zostaje logicznie uznana za pustą — bo top = -1 oznacza brak aktywnych danych.

```
s.resetStack();
s.printStack();

s.push(5);
s.push(15);
s.pop();
s.printStack();
```

```
30 20 10
Element z wierzchu: 30
20 10
Podaj liczbe elementow do dodania: 3
34 67 41 20 10
Liczba elementow na stosie: 5
Nie zgadza sie.
Stos pusty.
5
```

Metoda **resetStack()** skutecznie przywraca strukturę do stanu początkowego. Dalsze operacje po resecie nie generują błędów i zachowują pełną funkcjonalność.

Kolejka

Zadanie 1

```
bool isEmpty() {
    return front > rear;
}

bool isFull() {
    return rear == MAX - 1;
}
```

isEmpty() — Sprawdza, czy kolejka jest pusta. Jeśli indeks front przekroczy rear, oznacza to, że wszystkie elementy zostały usunięte.

isFull() —Sprawdza, czy kolejka osiągnęła maksymalną pojemność. Dalsze dodawanie będzie niemożliwe.

```
void enqueue(int x) {
    if (isFull()) {
        cout << "Kolejka pelna!" << endl;
        return;
    }
    arr[++rear] = x;
}

int dequeue() {
    if (isEmpty()) {
        cout << "Kolejka pusta!" << endl;
        return -1;
    }
    return arr[front++];
}

int peek() {
    if (isEmpty()) {
        cout << "Kolejka pusta!" << endl;
        return -1;
    }
    return arr[front];
}
```

enqueue — Dodaje element x do kolejki. Zwiększa indeks rear, a następnie zapisuje wartość do tablicy.

dequeue — Zwraca pierwszy element z kolejki i zwiększa front, czyli usuwa ten element logicznie.

peek — Zwraca wartość pierwszego elementu bez usuwania.

Zadanie 2

```
int n;
cout << "Podaj liczbe elementow do dodania: ";
cin >> n;
for (int i = 0; i < n; ++i) {
    q.enqueue(rand() % 100);
}
q.printQueue();
```

```
30 20 10
Pierwszy element: 10
30 20
Podaj liczbe elementow do dodania: 4
0 34 67 41 30 20
```

Kolejka została poprawnie wypełniona wartościami losowymi w liczbie podanej przez użytkownika. Elementy są wstawiane zgodnie z zasadą FIFO.

Zadanie 3

```
int size() {
    return isEmpty() ? 0 : rear - front + 1;
}
```

Metoda **size()** oblicza liczbę elementów w kolejce na podstawie różnicy między indeksami rear i front. Gdy rear == front, to jeden element. Gdy front > rear, oznacza pustą kolejkę. Funkcja zwraca dokładną liczbę elementów w czasie O(1).

```
Podaj liczbe elementow do dodania: 3
34 67 41 30 20
Liczba elementow w kolejce: 5
```

Rozmiar został poprawnie policzony.

Zadanie 4

```
int count = q.size();
cout << "Liczba elementow w kolejce: " << count << endl;
if (n == count) cout << "Zgadza sie." << endl;
else cout << "Nie zgadza sie." << endl;
```

Zmienna **n** zawiera liczbę elementów, które użytkownik chciał wprowadzić do kolejki.

Metoda **size()** oblicza aktualną liczbę elementów w kolejce na podstawie pozycji wskaźników front i rear.

Porównanie **n == count** weryfikuje, czy wszystkie elementy zostały poprawnie zapisane do kolejki.

W przypadku, gdy przed tym testem kolejka zawierała już jakieś dane, wynik **size()** będzie większy niż n.

```
30 20 10
Pierwszy element: 10
30 20
Podaj liczbe elementow do dodania: 4
0 34 67 41 30 20
Liczba elementow w kolejce: 6
Nie zgadza sie.
Kolejka pusta.
15
```

Porównanie pozwala sprawdzić poprawność działania metody **enqueue** oraz metody **size()**. Wynik różny od n nie musi oznaczać błędu, jeśli w kolejce były wcześniejsze dane. Kod powinien być uruchamiany w kontekście "**czystej**" struktury, jeśli zależy nam na dokładnym porównaniu z n.

Zadanie 5

```
void printQueue() {
    if (isEmpty()) {
        cout << "Kolejka pusta." << endl;
        return;
    }
    for (int i = rear; i >= front; --i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

Metoda sprawdza, czy kolejka jest pusta.

Jeżeli nie, przechodzi od indeksu **rear** do front w pętli malejącej, wypisując wszystkie wartości.

Taka kolejność wypisywania (od końca do początku) nie zmienia zawartości kolejki, lecz pokazuje ją w odwrotnym porządku niż **dequeue** by ją usuwał.

Zadanie 6

```
void resetQueue() {
    front = 0;
    rear = -1;
}
```

resetQueue polega na ustawieniu wskaźników front i rear do wartości początkowych.

Nie ma potrzeby czyszczenia danych z tablicy **arr[]**, ponieważ przy ponownym dodawaniu będą one nadpisane.

Po resecie kolejka logicznie jest pusta (**isEmpty() zwróci true**).

Operacje **enqueue() i dequeue()** nadal działają poprawnie od indeksu 0.

```
30 20 10
Pierwszy element: 10
30 20
Podaj liczbe elementow do dodania: 5
69 0 34 67 41 30 20
Liczba elementow w kolejce: 7
Nie zgadza sie.
Kolejka pusta.
15
```

Wnioski

Struktura stosu została zrealizowana w oparciu o wskaźnik top, który kontroluje indeks ostatnio dodanego elementu. Operacje **push** i **pop** działają zgodnie z zasadą **LIFO**.

Struktura kolejki została zrealizowana w oparciu o wskaźniki **front** i **rear**, które kontrolują początek i koniec kolejki. Operacje **enqueue i dequeue** działają zgodnie z zasadą **FIFO**.

Metody **isEmpty** oraz **isFull** poprawnie identyfikują stan struktur na podstawie pozycji wskaźników.

Metody **peek** dla obu struktur zwracają poprawną wartość bez zmiany zawartości.

Metoda **size()** dla obu struktur zwraca liczbę elementów na podstawie różnicy wskaźników. Operacja wykonuje się w czasie stałym O(1) i nie wymaga iteracji po tablicy.

Porównanie liczby elementów podanej przez użytkownika z rzeczywistą zawartością struktury pozwala wykryć stan wcześniejszej inicjalizacji lub pozostałości danych z poprzednich operacji.

Metody **printStack i printQueue** pozwalają na wyświetlenie zawartości struktur w formie czytelnej. Porządek wyświetlania został dostosowany do logiki danej struktury: od góry do dołu dla stosu, od końca do początku dla kolejki.

Metody **resetStack i resetQueue** logicznie opróżniają struktury danych poprzez przywrócenie wskaźników do wartości początkowych. Struktury są gotowe do ponownego użycia bez ponownej inicjalizacji.

Testy wykonane w funkcji main() potwierdziły poprawność działania każdej metody. Nie zaobserwowano błędów logicznych ani przekroczenia zakresów.