WYDZIAŁ
INFORMATYKI
I TELEKOMUNIKACJI

Stosy i kolejki

Veronika Tronchuk

I rok Informatyki

Nr albumu: 30019

Stos (LIFO)

Stos to liniowa struktura danych, która działa na zasadzie Last In, First Out — ostatni dodany element jest usuwany jako pierwszy.
Przypomina stos talerzy — zawsze zabieramy ten, który jest na górze.

Podstawowe operacje:
push(x) – dodaje element na górę stosu,
pop() – usuwa element z góry,
peek() – zwraca element z góry bez usuwania.

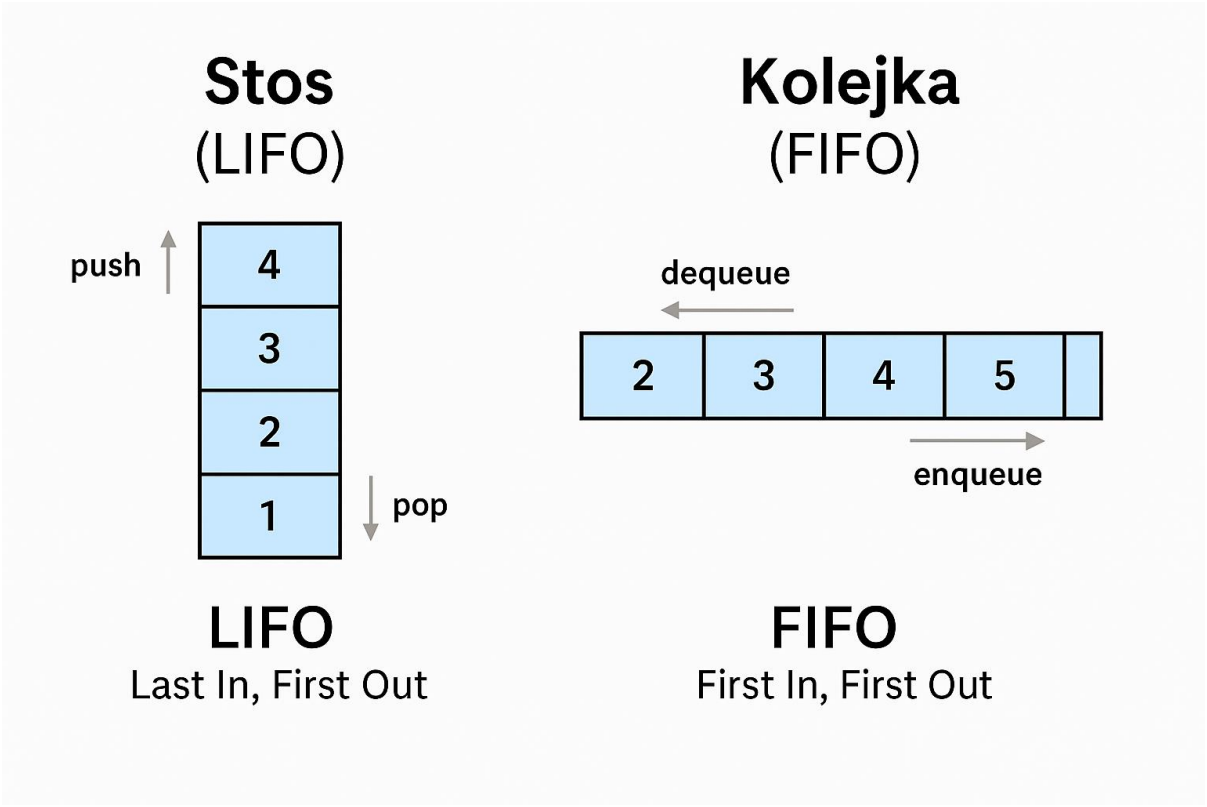
W implementacji tablicowej stos używa zmiennej top, która wskazuje indeks ostatniego elementu.

Kolejka (FIFO)

Kolejka to struktura danych działająca na zasadzie First In, First Out — pierwszy dodany element jest usuwany jako pierwszy.
Przypomina kolejkę w sklepie — osoby są obsługiwane w takiej kolejności, w jakiej przyszły.

Podstawowe operacje:
enqueue(x) – dodaje element na koniec kolejki,
dequeue() – usuwa element z początku,
peek() – zwraca pierwszy element bez usuwania.

W wersji tablicowej kolejka używa dwóch indeksów: first (początek) i last (koniec)


Stos
(LIFO)
push
4
3
2
1
pop
LIFO
Last In, First Out
Kolejka
(FIFO)
dequeue
2
3
4
5
enqueue
FIFO
First In, First Out

Stosy

```
#include <iostream>
#include <string>
```

Biblioteka do wejścia i wyjścia.

Dodaje obsługę napisów (`string`).

```
const int MAX = 100;
```

Maksymalna liczba elementów na stosie.

```
const string stackEmptyInfo = "Stack is empty.";
```

Tekst informujący o pustym stosie.

```
class Stack {
```

Deklaracja klasy stack

```
int arr[MAX];
```

Tablica na elementy stosu.

```
int top = -1;
```

Indeks góry stosu, -1 oznacza pusty stos.

```
public:
```

Zaczyna się blok publicznych metod.

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

Początek metody, która sprawdza, czy stos jest pusty.

Zwraca true jeśli stos jest pusty.

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

Metoda do sprawdzenia, czy stos jest pełny.

Zwraca true jeśli stos jest pełny.

```
void push(int x) {
    if (isFull()) {
        return;
    }
    arr[++top] = x;
}
```

Początek metody push — dodaje element.

Sprawdzenie: jeśli stos jest pełny — nic nie robimy.
Zakończenie metody bez działania.

Zwiększamy top i wkładamy x na górę stosu.

```
int pop() {
    return arr[top--];
}
```

Początek metody, która usuwa górny element i go zwraca.
Zwraca wartość z góry stosu i ją usuwa.

```
int peek() {
    return arr[top];
}
```

Metoda zwraca górny element bez jego usuwania

Zwraca górny element bez usuwania.

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

Metoda liczy, ile elementów jest na stosie.
Liczba elementów = top + 1

```
void printStack() {
    for (int i = top; i >= 0; i--) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

Metoda do wyświetlania zawartości stosu.
Pętla od góry stosu do dołu.

Wyświetla element stosu.

Przechodzi do nowej linii.

```
int main() {
```

Początek głównej funkcji programu.

```
Stack s;
```

Tworzymy obiekt s typu stack

```
cout << "pusty? ";
```

Wyświetlamy pytanie.

```
if (s.isEmpty()) {
    cout << "Yes" << endl;
}
else {
    cout << "No" << endl;
}
```

Sprawdzamy, czy stos jest pusty.
Wyświetlamy "Tak".
Jeśli stos nie jest pusty — inna gałąź.
Wyświetlamy "Nie".

```
s.push(10);
s.push(20);
s.push(30);
```

Dodajemy elementy na stos.

```
cout << "gorny element: " << s.peek() << endl;
```

Wyświetlamy wartość górnego elementu.

```
cout << "usuniety: " << s.pop() << endl;
```

Usuwamy górny element i go wyświetlamy.

```
cout << "gorny: " << s.peek() << endl;
```

Pokazujemy nowy górny element.

```
cout << "generowanie stosu: \n";
```

Tekst dla użytkownika.

```
int n;
cout << "podaj ilisc elementow ";
cin >> n;
```

Tworzymy zmienną n na ilość elementów.
Prosimy użytkownika o wpisanie liczby.
Odczytujemy n z klawiatury.

```
for (int i = 0; i < n; i++) {
    s.push(rand() % 100);
}
```

Generujemy n sowych liczb i dodajemy na stos.

```
cout << "elementow w stosie: " << s.countElements() << endl;
```

Wyświetlamy liczbę elementów.

```
cout << "wartosc stosu: ";
s.printStack();
```

Wyświetlamy zawartość stosu.

```
return 0;
```

Zwracamy 0 — program zakończył się poprawnie.

```
arr[top++] = x;
```

```
pusty? Yes
gorny element: -858993460
usuniety: -858993460
gorny: 30

generowanie stosu:
podaj ilisc elementow 7
elementow w stosie: 9
wartosc stosu: -858993460 78 24 69 0 34 67 41 20
```

```
arr[++top] = x;
```

```
pusty? Yes
gorny element: 30
usuniety: 30
gorny: 20

generowanie stosu:
podaj ilisc elementow 7
elementow w stosie: 9
wartosc stosu: 78 24 69 0 34 67 41 20 10
```

Wnioski:

Zapis **arr[top++]** = x; nie działa poprawnie, ponieważ pierwszy element trafia do **arr[-1]**, czyli poza tablicę.
W wyniku tego w stosie pojawiają się błędne wartości, tzw. „śmieci” — na przykład **-858993460.**
Poprawny zapis to **arr[++top] = x**;, ponieważ najpierw zwiększa indeks top, a dopiero potem zapisuje wartość w tablicy.
Na zrzutach ekranu wyraźnie widać różnicę w działaniu obu wersji — tylko **++top** daje poprawne wyniki.

Kolejki

```
#include <iostream>
#include <string>
```

Biblioteka do wejścia i wyjścia.

Dodaje obsługę napisów (`string`).

```
const int MAX = 100;
```

Maksymalna liczba elementów na stosie.

```
const string queueEmptyInfo = "Queue is empty.";
```

Tekst informujący o pustej kolejce.

```
class Queue {
```

Deklaracja klasy Queue.

```
int arr[MAX];
```

Tablica na elementy kolejki.

```
int first = 0;
```

Indeks początku kolejki (pierwszy element do usunięcia).

```
int last = 0;
```

Indeks końca kolejki (gdzie dodajemy nowe elementy).

```
public:
```

Zaczyna się blok publicznych metod.

```
bool isEmpty() {
    return first == last;
}
```

Początek metody sprawdzającej, czy kolejka jest pusta.
Zwraca true, jeśli indeksy są równe — kolejka jest pusta.

```
bool isFull() {
    return last == MAX;
}
```

Początek metody sprawdzającej, czy kolejka jest pełna.
Zwraca true, jeśli osiągnęliśmy maksymalny rozmiar.

```
void enqueue(int x) {
    arr[last++] = x;
}
```

Początek metody dodającej element do kolejki.
Wkładamy element na koniec i zwiększamy last

```
int dequeue() {
    return arr[first++];
}
```

Początek metody usuwającej pierwszy element z kolejki.
Zwracamy pierwszy element i przesuwamy first do przodu.

```
int dequeue() {
    return arr[first++];
}
```

Metoda pokazuje pierwszy element bez usuwania.

Zwraca pierwszy element w kolejce.

```
int peek() {
    return arr[first];
}
```

Metoda pokazuje pierwszy element bez usuwania.
Zwraca pierwszy element w kolejce.

```
int countElements() {
    return last - first;
}
```

Metoda liczy ilość elementów w kolejce.
Różnica między końcem i początkiem — tyle elementów mamy.

```
void printQueue() {
    for (int i = first; i < last; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

Metoda wyświetla elementy kolejki od pierwszego do ostatniego.
Pętla przechodzi od indeksu first do last.
Wyświetla każdy element w kolejce.
Przechodzi do nowej linii po zakończeniu.

```
int main() {
```

Początek funkcji głównej programu.

```
Queue q;
```

Tworzymy obiekt q typu Queue

```
if (q.isEmpty()) {
    cout << "Yes" << endl;
}
else {
    cout << "No" << endl;
}
```

Pytamy użytkownika, czy kolejka jest pusta.
Sprawdzamy, czy kolejka jest pusta.
Wyświetlamy "Yes".
Wyświetlamy "No".

```
q.enqueue(10);
q.enqueue(20);
q.enqueue(30);
```

Dodajemy trzy elementy do kolejki.

```
cout << "pierwszy element: " << q.peek() << endl;
```

Pokazujemy pierwszy element bez usuwania.

```
 cout << "usuniety: " << q.dequeue() << endl;
```

Usuwamy pierwszy element i pokazujemy go.

```
cout << "pierwszy teraz: " << q.peek() << endl << endl;
```

Po usunięciu — pokazujemy nowy pierwszy element.

```
 cout << "generowanie kolejki: \n";
```

Informujemy, że zaraz utworzymy kolejkę z losowymi liczbami.

```
int n;
cout << "podaj ilosc elementow ";
cin >> n;
```

Tworzymy zmienną n na ilość elementów.
Pytamy użytkownika o ilość elementów do dodania.
Wczytujemy liczbę n z klawiatury.

```
for (int i = 0; i < n; i++) {
    q.enqueue(rand() % 100);
```

Losujemy n liczb i dodajemy je do kolejki.

```
cout << "elementow w kolejce: " << q.countElements() << endl;
```

Wyświetlamy ilość elementów w kolejce.

```
cout << "wartosc kolejki: ";
```

```
q.printQueue();
```

Wyświetlamy zawartość kolejki.

```
return 0;
```

Program kończy się poprawnie — zwracamy 0.

Wnioski

Podczas testowania zapisu arr[top++] = x; pierwszy element trafiał do arr[-1], co prowadziło do błędnych danych w stosie.
Brak sprawdzania, czy stos lub kolejka są puste, przed pop() lub dequeue(), może powodować błędy działania.
W metodzie peek() brak warunku bezpieczeństwa może prowadzić do odczytu nieistniejącego elementu.