Stosy

[std::stack - reference.com]

Stos jest to struktura danych, która działa według zasady "last in, first out" (LIFO), czyli ostatni element dodany do stosu będzie pierwszy do wyjścia. Oznacza to, że ostatni element dodany na stos zostanie jako pierwszy usunięty. Stos jest używany do przechowywania danych w taki sposób, że tylko ostatnio dodany element jest dostępny (na szczycie stosu).

Korzystanie z biblioteki STL

Przykład użycia:

```
include <iostream>
include <stack>

int main() {
    std::stack<int> myStack;

    // Dodawanie elementów na stos
    myStack.push(10);
    myStack.push(20);
    myStack.push(30);

    // Usuwanie elementu ze stosu
    myStack.pop();

    // Podglądanie elementu na szczycie stosu
    std::cout << "Element na szczycie stosu: " << myStack.top() <<
std::endl;

    // Sprawdzanie czy stos jest pusty
    if (myStack.empty()) {
        std::cout << "Stos jest pusty.\n";
    } else {
        std::cout << "Stos nie jest pusty.\n";
    }

    return 0;
}
```

Implementacja obiektowa

```
include <iostream>

// Definicja struktury węzła stosu
```

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

    Node(int d) : data(d), next(nullptr) {} // Konstruktor węzła
};

// Definicja klasy Stosu
class Stack {
private:
    Node* top; // Wskaźnik na szczyt stosu

public:
    Stack(); // Konstruktor stosu
    ~Stack(); // Destruktor stosu
    void push(int data); // Dodawanie elementu na stos
    void pop(); // Usuwanie elementu ze stosu
    int peek(); // Podglądanie szczytu stosu
    bool isEmpty(); // Sprawdzanie, czy stos jest pusty
};

// Konstruktor stosu
Stack::Stack() : top(nullptr) {}

// Destruktor stosu
Stack::~Stack() {
    while (!isEmpty()) {
        pop();
    }
}

// Dodawanie elementu na stos
void Stack::push(int data) {
    Node* newNode = new Node(data);
    newNode->next = top;
    top = newNode;
}

// Usuwanie elementu ze stosu
void Stack::pop() {
    if (!isEmpty()) {
        Node* temp = top;
        top = top->next;
        delete temp;
    }
}

// Podglądanie szczytu stosu
int Stack::peek() {
    if (!isEmpty()) {
        return top->data;
    }
    return -1; // Stos jest pusty
}

// Sprawdzanie, czy stos jest pusty
```

```
bool Stack::isEmpty() {
    return top == nullptr;
}

// Przykładowe użycie stosu
int main() {
    Stack myStack;

    myStack.push(10);
    myStack.push(20);
    myStack.push(30);

    myStack.pop(); // Usuwanie ostatniego elementu

    std::cout << "Element na szczycie stosu: " << myStack.peek() <<
std::endl;

    if (myStack.isEmpty()) {
        std::cout << "Stos jest pusty.\n";
    } else {
        std::cout << "Stos nie jest pusty.\n";
    }

    return 0;
}
```

Różnice między kolejką a sostem są następujące:

1. Struktura danych:

- W przypadku kolejki (Queue) używamy struktury opartej na liście jednokierunkowej, gdzie mamy wskaźniki na początek (front) i koniec (rear) kolejki.

- W przypadku stosu (Stack) używamy struktury opartej na liście jednokierunkowej, gdzie mamy wskaźnik na szczyt (top) stosu.

2. Operacje:

- Kolejka obsługuje operacje dodawania elementu na końcu kolejki (enqueue) i usuwania elementu z początku kolejki (dequeue).

- Stos obsługuje operacje dodawania elementu na szczyt stosu (push) i usuwania elementu ze szczytu stosu (pop).

3. Kierunek operacji:

- Kolejka działa na zasadzie FIFO (First-In-First-Out), czyli pierwszy dodany element jest pierwszy usuwany.

- Stos działa na zasadzie LIFO (Last-In-First-Out), czyli ostatni dodany element jest pierwszy usuwany.

4. Wykorzystanie:

- Kolejka jest często wykorzystywana w scenariuszach, gdzie potrzebujemy kolejności przetwarzania elementów, takich jak zarządzanie zadaniami w systemie operacyjnym, buforowanie danych, czy kolejkowanie żądań w sieci.

- Stos jest często wykorzystywany do rozwiązywania problemów związanych z nawigacją (np. cofanie się w historii), obsługą wywołań funkcji (np. stos wywołań w pamięci programu), czy analizą składniową w analizatorach składniowych.

Zadania:

1. Odwróć kolejność elementów w tablicy z pomocą stosu
2. Napisz program, który będzie sprawdzał, czy nawiasy w danym ciągu są poprawnie sparowane. Program będzie korzystał ze stosu do sprawdzenia równoważności nawiasów.

Instrukcje:

a. Utwórz klasę BracketChecker, która będzie zawierała metodę checkBrackets, która przyjmie jako argument ciąg znaków (np. std::string), a następnie zwróci true, jeśli nawiasy w ciągu są poprawnie sparowane, lub false, jeśli nie.
b. Wewnątrz metody checkBrackets utwórz pusty stos (wykorzystując std::stack<char>).
c. Przejdź po każdym znaku w ciągu wejściowym.
d. Jeśli napotkasz otwierający nawias ((, [, {), dodaj go na stos.
e. Jeśli napotkasz zamykający nawias (),], }), sprawdź, czy stos nie jest pusty. Jeśli jest pusty, zwróć false, ponieważ znaleziono zamykający nawias bez odpowiadającego mu otwierającego nawiasu.
f. Jeśli stos nie jest pusty, pobierz element ze stosu i sprawdź, czy odpowiada on otwierającemu nawiasowi aktualnego zamykającego nawiasu. Jeśli nie odpowiada, zwróć false.
g. Po zakończeniu iteracji po wszystkich znakach, sprawdź, czy stos jest pusty. Jeśli nie jest, zwróć false, ponieważ znaleziono więcej otwierających nawiasów niż zamykających.
h. Jeśli wszystkie nawiasy są poprawnie sparowane i stos jest pusty, zwróć true.