Estruturas de Dados: Pilhas (Stacks)

Uma **Pilha** (ou **Stack** em inglês) é uma das estruturas de dados mais fundamentais e intuitivas na ciência da computação. Ela opera sob um princípio muito simples e específico: **LIFO (Last In, First Out)**, que significa "Último a Entrar, Primeiro a Sair".

Pense em uma pilha de pratos: você sempre adiciona um novo prato no topo da pilha, e quando você quer pegar um prato, você sempre pega o que está no topo. O primeiro prato que você colocou na pilha será o último a ser retirado.

Características Principais da Pilha:

- **LIFO (Last In, First Out):** O último elemento adicionado é o primeiro a ser removido.
- **Acesso Restrito:** As operações são realizadas apenas em uma das extremidades da pilha, geralmente chamada de "topo" (top).

Operações Básicas de uma Pilha:

Existem algumas operações padrão que definem o comportamento de uma pilha:

1. `Push` **(Empilhar):**
   - **O que faz:** Adiciona um novo elemento ao topo da pilha.
   - **Exemplo:** Se você tem uma pilha de livros e adiciona um novo livro, ele vai para o topo.
2. `Pop` **(Desempilhar):**
   - **O que faz:** Remove o elemento que está no topo da pilha.
   - **Exemplo:** Se você tem uma pilha de livros e remove um, você tira o livro que está no topo.
   - **Importante:** Antes de fazer um `pop`, é crucial verificar se a pilha não está vazia para evitar erros.
3. `Peek` **ou** `Top` **(Consultar Topo):**
   - **O que faz:** Retorna o elemento que está no topo da pilha, mas **não o remove**.
   - **Exemplo:** Olhar o título do livro que está no topo da pilha sem tirá-lo.
   - **Importante:** Assim como no `pop`, é bom verificar se a pilha não está vazia antes de tentar consultar o topo.
4. `isEmpty` **(Está Vazia?):**
   - **O que faz:** Verifica se a pilha não contém nenhum elemento.

- **Retorno:** Geralmente retorna `true` (verdadeiro) se a pilha estiver vazia e `false` (falso) caso contrário.

5. `isFull` **(Está Cheia?):**
   - **O que faz:** Verifica se a pilha atingiu sua capacidade máxima (aplicável quando a pilha tem um tamanho fixo, como em implementações baseadas em array).
   - **Retorno:** Geralmente retorna `true` se a pilha estiver cheia e `false` caso contrário.

Aplicações Comuns de Pilhas:

Pilhas são usadas em diversas áreas da computação, incluindo:

- **Validação de Expressões:** Como no seu exercício de parênteses, para verificar a correspondência de delimitadores (parênteses, chaves, colchetes).
- **Desfazer/Refazer (Undo/Redo):** Em editores de texto ou programas gráficos, as ações são empilhadas para permitir que o usuário as desfaça em ordem inversa.
- **Navegação em Páginas Web:** O histórico de navegação de um navegador pode ser visto como uma pilha (a página atual é o topo).
- **Chamadas de Função (Call Stack):** Em programação, as chamadas de função são gerenciadas por uma pilha, onde a última função chamada é a primeira a ser concluída.
- **Avaliação de Expressões:** Converte e avalia expressões matemáticas (notação polonesa reversa).

No próximo passo, demonstrarei como implementar uma pilha básica em C, que será essencial para resolver o seu problema de validação de parênteses.

Implementação Básica de uma Pilha em C (Usando Array)

Uma das formas mais comuns e simples de implementar uma pilha em C é usando um array. Precisaremos de um array para armazenar os elementos da pilha e uma variável para controlar o "topo" da pilha.

Plain Text

```
#include <stdio.h>
#include <stdbool.h> // Para usar 'true' e 'false'

#define MAX_SIZE 100 // Tamanho máximo da pilha

// Estrutura para representar a pilha
typedef struct {
    char items[MAX_SIZE]; // Array para armazenar os elementos (neste caso,
```

```
caracteres)
    int top;              // Índice do topo da pilha
} Stack;

// Inicializa a pilha
void initialize(Stack *s) {
    s->top = -1; // -1 indica que a pilha está vazia
}

// Verifica se a pilha está vazia
bool isEmpty(Stack *s) {
    return s->top == -1;
}

// Verifica se a pilha está cheia
bool isFull(Stack *s) {
    return s->top == MAX_SIZE - 1;
}

// Adiciona um elemento ao topo da pilha (Push)
void push(Stack *s, char value) {
    if (isFull(s)) {
        printf("Erro: Pilha cheia! Nao eh possivel adicionar mais
elementos.\n");
    } else {
        s->items[++(s->top)] = value; // Incrementa o topo e adiciona o valor
    }
}

// Remove e retorna o elemento do topo da pilha (Pop)
char pop(Stack *s) {
    if (isEmpty(s)) {
        printf("Erro: Pilha vazia! Nao ha elementos para remover.\n");
        return '\0'; // Retorna um caractere nulo ou outro valor de erro
    } else {
        return s->items[(s->top)--]; // Retorna o valor e decrementa o topo
    }
}

// Retorna o elemento do topo da pilha sem remove-lo (Peek/Top)
char peek(Stack *s) {
    if (isEmpty(s)) {
        printf("Erro: Pilha vazia! Nao ha elementos para consultar.\n");
        return '\0';
    } else {
        return s->items[s->top];
    }
}
```

```
int main() {
    Stack minhaPilha;
    initialize(&minhaPilha);

    printf("Pilha vazia? %s\n", isEmpty(&minhaPilha) ? "Sim" : "Nao");

    push(&minhaPilha, 'A');
    push(&minhaPilha, 'B');
    push(&minhaPilha, 'C');

    printf("Topo da pilha: %c\n", peek(&minhaPilha));
    printf("Pilha vazia? %s\n", isEmpty(&minhaPilha) ? "Sim" : "Nao");

    printf("Desempilhando: %c\n", pop(&minhaPilha));
    printf("Desempilhando: %c\n", pop(&minhaPilha));

    printf("Topo da pilha: %c\n", peek(&minhaPilha));

    push(&minhaPilha, 'D');
    printf("Topo da pilha: %c\n", peek(&minhaPilha));

    printf("Desempilhando: %c\n", pop(&minhaPilha));
    printf("Desempilhando: %c\n", pop(&minhaPilha));

    printf("Pilha vazia? %s\n", isEmpty(&minhaPilha) ? "Sim" : "Nao");

    // Tentando desempilhar de uma pilha vazia
    pop(&minhaPilha);

    return 0;
}
```

Explicação do Código:

- **`MAX_SIZE`:** Define o tamanho máximo da pilha. Em uma implementação baseada em array, a pilha tem um tamanho fixo.
- **`Stack` struct:** Contém um array `items` para armazenar os elementos e um `int top` que aponta para o índice do elemento no topo da pilha. Quando a pilha está vazia, `top` é geralmente inicializado como `-1`.
- **`initialize(Stack *s)`:** Define `s->top = -1`, indicando que a pilha está vazia.
- **`isEmpty(Stack *s)`:** Retorna `true` se `s->top` for `-1`.
- **`isFull(Stack *s)`:** Retorna `true` se `s->top` for igual ao último índice válido do array (`MAX_SIZE - 1`).

- **`push(Stack *s, char value)`:** Se a pilha não estiver cheia, incrementa `s->top` e armazena `value` na nova posição do topo.
- **`pop(Stack *s)`:** Se a pilha não estiver vazia, retorna o elemento em `s->top` e então decrementa `s->top`.
- **`peek(Stack *s)`:** Se a pilha não estiver vazia, retorna o elemento em `s->top` sem modificar `s->top`.

Esta implementação fornece as operações básicas necessárias para gerenciar uma pilha. No próximo passo, veremos como aplicar essa estrutura de dados para resolver o problema de validação de parênteses.

Relacionando Pilhas com a Validação de Parênteses

O problema de validar a correspondência de parênteses em uma expressão é um caso clássico onde a estrutura de dados de pilha se encaixa perfeitamente. A lógica por trás disso é que, para cada parêntese de abertura, deve haver um parêntese de fechamento correspondente em uma ordem específica.

O Algoritmo de Validação Usando Pilha:

1. **Inicialize uma pilha vazia.** Esta pilha será usada para armazenar os parênteses de abertura que ainda não foram fechados.
2. **Percorra a expressão caractere por caractere.**
3. **Se o caractere atual for um parêntese de abertura `(`:**
   - Adicione-o à pilha (`push`). Isso significa que encontramos um parêntese que precisa ser fechado mais tarde.
4. **Se o caractere atual for um parêntese de fechamento `)`:**
   - **Verifique se a pilha está vazia.** Se estiver, significa que encontramos um parêntese de fechamento sem um parêntese de abertura correspondente. A expressão é **inválida**.
   - **Se a pilha não estiver vazia, remova o elemento do topo (`pop`).** Isso significa que o parêntese de fechamento atual encontrou seu parêntese de abertura correspondente no topo da pilha.
5. **Após percorrer toda a expressão:**
   - **Se a pilha estiver vazia, a expressão é válida.** Isso indica que todos os parênteses de abertura foram corretamente fechados por um parêntese de fechamento correspondente.

- **Se a pilha não estiver vazia, a expressão é inválida.** Isso significa que existem parênteses de abertura que não foram fechados.

Exemplo Passo a Passo:

Vamos validar a expressão `(a+(b*c)-2+a)`:

- **Expressão:** `(a+(b*c)-2+a)`
- **Pilha:** Vazia

1. `(` **:** É um parêntese de abertura. `Push` para a pilha.
   - Pilha: `[`
2. `a` **:** Ignorar (não é parêntese).
3. `+` **:** Ignorar.
4. `(` **:** É um parêntese de abertura. `Push` para a pilha.
   - Pilha: `[((`
5. `b` **:** Ignorar.
6. `*` **:** Ignorar.
7. `c` **:** Ignorar.
8. `)` **:** É um parêntese de fechamento. Pilha não está vazia. `Pop` da pilha.
   - Pilha: `[`
9. `-` **:** Ignorar.
10. `2` **:** Ignorar.
11. `+` **:** Ignorar.
12. `a` **:** Ignorar.
13. `)` **:** É um parêntese de fechamento. Pilha não está vazia. `Pop` da pilha.
    - Pilha: `[]` (Vazia)

Fim da Expressão: A pilha está vazia. Portanto, a expressão `(a+(b*c)-2+a)` é **válida**.

Vamos validar a expressão `(a*b-(2+c)`:

- **Expressão:** `(a*b-(2+c)`
- **Pilha:** Vazia

1. `(` **:** `Push`.
   - Pilha: `[`

2. `a` **:** Ignorar.
3. `*` **:** Ignorar.
4. `b` **:** Ignorar.
5. `-` **:** Ignorar.
6. `(` **:** `Push` .
    - Pilha: `[((`
7. `2` **:** Ignorar.
8. `+` **:** Ignorar.
9. `c` **:** Ignorar.

Fim da Expressão: A pilha **não** está vazia (contém `((`). Portanto, a expressão `(a*b-(2+c)` é **inválida**.

Este método garante que a ordem e a correspondência dos parênteses sejam verificadas de forma eficiente, tornando a pilha a estrutura de dados ideal para este tipo de problema.