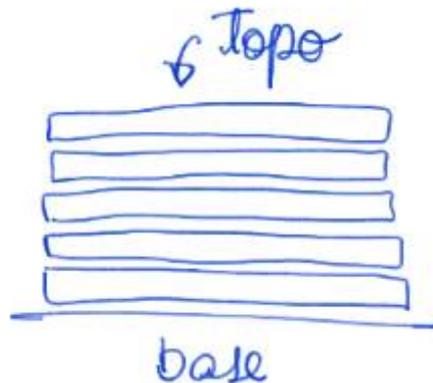


Algoritmos e Estruturas de Dados 1 (AED1)

Pilha implementada em vetor, aplicação com parênteses e colchetes, pilha de execução, relação de pilha com recursão

Pilha



Uma pilha (no inglês stack) é uma lista dinâmica,

- ou seja, uma sequência em que elementos podem ser removidos e inseridos,
- mas que possui regras bem específicas de funcionamento.

Em particular, as seguintes regras devem ser obedecidas:

- Uma operação de remoção sempre remove o elemento do fim da sequência.
- Uma operação de inserção sempre insere o elemento no fim da sequência.

Chamamos a última posição de uma pilha de topo.

- Assim, as operações de inserção, remoção e consulta
 - sempre são feitas no topo da pilha.

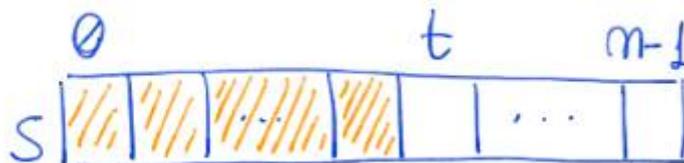
Costumamos resumir o comportamento de uma pilha na frase

- o último a entrar é o primeiro a sair.
- Por isso, pilhas também são conhecidas por LIFO,
 - acrônimo do inglês Last-In-First-Out.

Implementação de pilha usando vetor

Uma pilha s é armazenada em um vetor de tamanho n ,

- alocado estática ou dinamicamente.



Um inteiro t indica o topo da pilha,

- que é a posição do próximo elemento.
 - Por isso, a pilha ocupa o subvetor $s[0 .. t - 1]$.

Note que t corresponde ao número de elementos presentes na pilha,

- pois o vetor começa na posição 0.
- Em particular,
 - se $t = 0$ a pilha está vazia
 - se $t = n$ a pilha está cheia.

Para empilhar um elemento x fazemos

- $s[t++] = x$;
- que corresponde a
 - $s[t] = x; t = t + 1$;
- Note que, esta operação não é segura se a pilha estiver cheia,
 - i.e., se $t = n$.

Para desempilhar um elemento e armazená-lo em x fazemos

- $x = s[--t]$;
- que corresponde a
 - $t = t - 1; x = s[t]$;
- Note que, esta operação não é segura se a pilha estiver vazia,
 - i.e., se $t = 0$.

Para consultar o valor do elemento no topo da pilha fazemos

- $s[t - 1]$;

Note que todas as operações de manipulação da pilha

- levam tempo constante em relação ao número de elementos empilhados,
 - i.e., $O(1)$.

Se o número de elementos crescer muito, a pilha pode ficar cheia.

- Neste caso, uma alternativa é redimensionar a pilha.
 - Por exemplo, alocando um vetor com o dobro do tamanho anterior
 - e copiando todos os elementos do vetor anterior para o novo,
 - preservando a ordem dos elementos.

Estruturas de dados são ferramentas que nos ajudam

- a projetar algoritmos eficientes para resolver problemas.
 - A seguir temos um primeiro exemplo do uso de pilha.

Aplicação de pilha para verificação de parênteses e colchetes

Sequências bem formadas:

-
- $(())[()]$
- $[]()[]$
- $(([])[()])[()]$
- Tudo que for aberto após um $(/ [$
 - é fechado antes do $) /]$ correspondente.

Sequências mal formadas:

- ([])
- ([])
- ([])
- Ter o mesmo número de abres e fechadas de cada tipo
 - é condição necessária, mas não suficiente para ser bem formada.

Definição geral recursiva:

- sendo S uma sequência de parênteses e colchetes bem formada, temos
 - $S = \{ \text{sequência vazia},$
 $(S) S,$
 $[S] S \}$

Simulação: Primeiro veremos uma simulação passo-a-passo

- para entender a ideia do algoritmo.
- Considere a seguinte sequência $(([]) []) ()$

string[0 .. i - 1]	pilha[0 .. t - 1]	caracter
((
((((
(([(([
(([]	(([
(([])	()
(([]) [([
(([]) []	(]]
(([]) []))
(([]) []) ((
(([]) []) ())
(([]) []) ()]	pilha vazia -> mal formada	

Códigos:

```
#define N 100
char pilha[N];
int t;

void criapilha() {
    t = 0;
}

void empilha(char y) {
    pilha[t++] = y;
}
```

```

char desempilha() {
    return pilha[--t];
}

int pilhavazia() {
    return t <= 0;
}

// Esta função devolve 1 se a string ASCII s
// contém uma sequência bem-formada de
// parênteses e colchetes e devolve 0 se
// a sequência é malformada.
int bemFormada(char string[]) {
    criapilha();
    for (int i = 0; string[i] != '\0'; ++i) {
        char c;
        switch (string[i]) {
            case ')': // depois de ler um )
                if (pilhavazia()) // dá erro se não tiver alguém pra
pareá-lo
                    return 0;
                c = desempilha(); // ou se tiver alguém diferente de (
                if (c != '(')
                    return 0;
                break;
            case ']': // depois de ler um ]
                if (pilhavazia()) // dá erro se não tiver alguém pra
pareá-lo
                    return 0;
                c = desempilha(); // ou se tiver alguém diferente de [
                if (c != '[')
                    return 0;
                break;
            default:
                empilha(string[i]); // se leu algum abre empilha pra
esperar o fecha
        }
    }
    return pilhavazia(); // verifica se sobrou alguém sem parear
}

```

Note que, nesta aplicação uma alternativa para a pilha nunca estourar seu tamanho

- seria alocar para ela um vetor do tamanho da string de entrada.

Quizzes:

- Quiz1: Como modificar o algoritmo/código anterior para que ele passe a
 - verificar sequências bem formadas envolvendo { }, além de () e []?
- Quiz2: Como modificar as operações empilha e desempilha anteriores
 - para que indiquem erro caso a pilha esteja cheia ou vazia?
- Quiz3: Como modificar a operação empilha anterior para realocar a pilha
 - num vetor maior caso a pilha esteja cheia?

Pilha de execução de um programa

A pilha de execução de um programa é usada para armazenar:

- Variáveis locais das funções ativas,
- Parâmetros das funções ativas,
- Endereço de retorno para o ponto do código em que a função foi chamada,
- Cálculo de expressões.

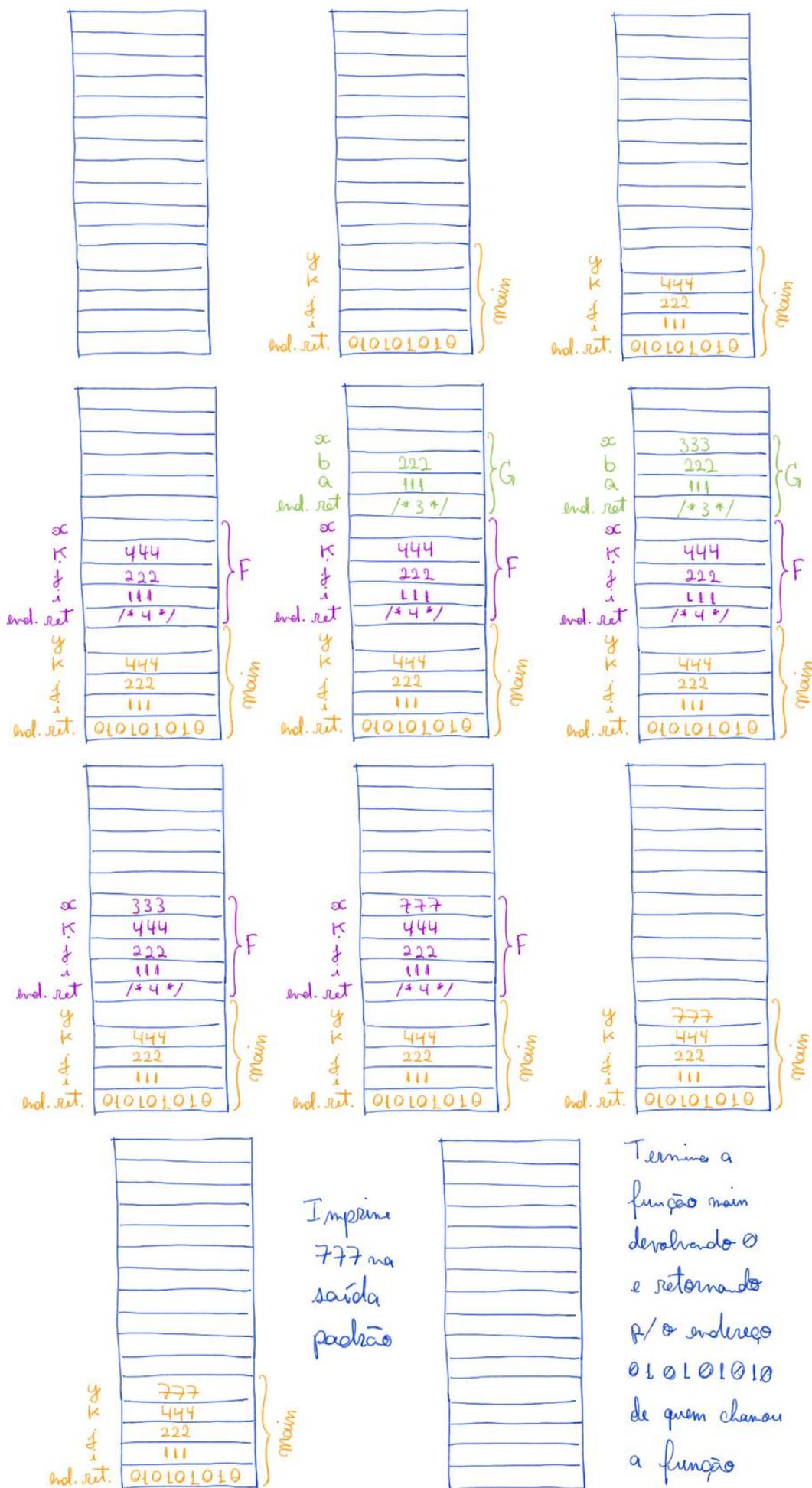
Códigos:

```
int G(int a, int b) {
    int x;
    x = a + b;
    return x;
}

int F(int i, int j, int k) {
    int x;
    x = /*2*/ G(i, j) /*3*/;
    x = x + k;
    return x;
}

int main(void) {
    int i, j, k, y;
    i = 111;
    j = 222;
    k = 444;
    y = /*1*/ F(i, j, k) /*4*/;
    printf("%d\n", y);
    return EXIT_SUCCESS;
}
```

Ilustração da pilha de execução deste código:



Imprime
777 na
saída
padrão

Termina a
função main
devolvendo 0
e retornando
p/ o endereço
010101010
de quem chamou
a função

Quiz4: Onde são armazenadas as variáveis alocadas dinamicamente?

- Resp: na outra ponta do espaço de memória.

Relação entre pilhas e recursão

Existe uma relação muito íntima entre pilhas e recursão.

- De fato, sempre é possível converter sistematicamente
 - um algoritmo recursivo em um algoritmo iterativo,
 - usando uma pilha explícita.
- Olhando por outro ângulo, um algoritmo recursivo
 - se vale da pilha de execução para atacar problemas
 - de maneiras que não seriam viáveis sem uma pilha.

Como exemplo,

- considere o problema de verificar se uma sequência é bem formada.
- Podemos construir um algoritmo recursivo para este problema
 - utilizando a definição recursiva do mesmo
 - $S = \{ \text{sequência vazia},$
 $(S) S,$
 $[S] S \}$

Código:

```
int bemFormadaR(char string[], int *pi) {
    int sol;
    if (string[*pi] == '(') { // S = ( S ) S
        *pi = *pi + 1;
        sol = bemFormadaR(string, pi) && string[*pi] == ')';
        *pi = *pi + 1;
        return sol && bemFormadaR(string, pi);
    }
    if (string[*pi] == '[') { // S = [ S ] S
        *pi = *pi + 1;
        sol = bemFormadaR(string, pi) && string[*pi] == ']';
        *pi = *pi + 1;
        return sol && bemFormadaR(string, pi);
    }
    return 1; // S = sequência vazia
}

int bemFormada2(char string[]) {
    int i = 0;
    return bemFormadaR(string, &i) && string[i] == '\0';
}
```