AED1 - Aula 15
Pilhas, inversão de sequências,
notação infixa para pósfixa

Relembrando operações para manipulação de pilha implementada em vetor:

- empilhar "s[t++] = x;"
- desempilhar "x = s[--t];"
- consultar topo "s[t - 1];"

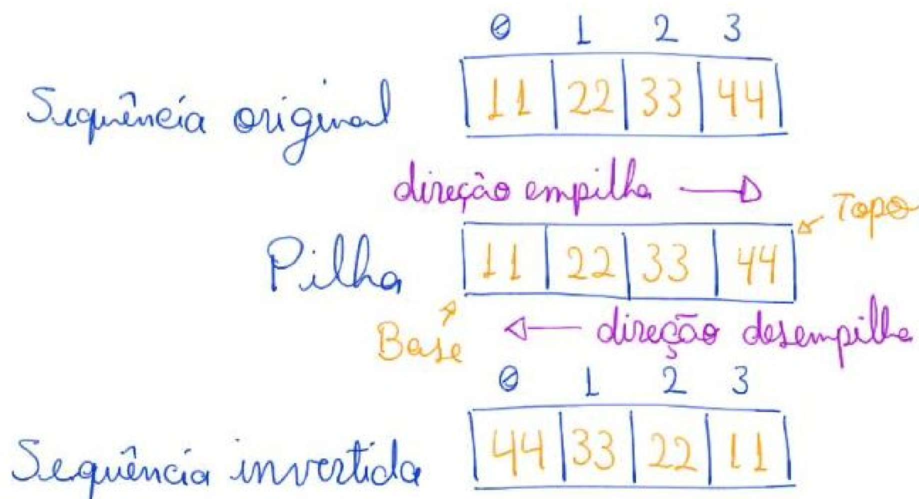
Pilhas e inversão de sequências

Uma aplicação direta e bastante útil de pilhas

- é na inversão de sequências.

Por conta do comportamento LIFO,

- i.e., último a ser inserido é o primeiro a ser removido,
- para inverter uma sequência basta
  - empilhar todos os seus elementos
  - e depois desempilhar todos eles.
- Note que, o primeiro elemento da sequência original
  - ficará no fundo da pilha,
  - sendo o último a ser desempilhado,
    - se tornando o último da nova sequência.



- De modo geral, considerando uma sequência com n elementos,
  - o i-ésimo elemento da sequência original
  - ficará na i-ésima posição da pilha,
    - se contarmos da base para o topo,
  - e na (n - i - 1)-ésima posição da pilha,
    - se contarmos do topo para a base.

        - Portanto, será o elemento (n - i - 1) a ser desempilhado
            - e ocupará a posição (n - i - 1) da nova sequência,
                - que é complementar a sua posição na sequência original.

Convertendo da notação infixa para pósfixa

Entendendo a notação infixa:

- A expressão é lida da esquerda para a direita e,
    - a princípio, os operadores são resolvidos conforme aparecem.
- Os operadores ficam entre os operandos.
- Certos operadores têm maior precedência que outros,
    - i.e., eles devem ser resolvidos antes,
        - ainda que não tenham aparecido antes.
- Deve-se resolver primeiro o que está entre parênteses.

Exemplos:

- 2 + 3 = 5
- 1 + 3 * 4 = 1 + 12 = 13
- (4 - 2) * (3 - 4) = 2 * -1 = -2

Entendendo a notação pósfixa:

- A expressão é lida da esquerda para a direita
    - e os operadores são resolvidos conforme aparecem.
- Os operadores ficam depois dos operandos.
- Cada operador é resolvido assim que encontrado,
    - por isso precedência de operadores e parênteses não são relevantes.

Exemplos:

- 2 3 + = 5
- 1 3 4 * + = 1 12 + = 13
- 4 2 - 3 4 - * = 2 -1 * = -2

Curiosidades:

- notação pósfixa também é chamada de notação polonesa reversa
    - em alusão ao matemático polonês que inventou a notação préfixa
- Vale destacar que, apesar da estranheza inicial,
    - notação pósfixa é mais simples e fácil de processar que notação infixa.

O problema que vamos tratar é:

- converter expressões
    - da notação infixa
    - para a notação pósfixa

Convertendo manualmente:

- (A + B * C) => ABC*+
- (A * (B + C) / D - E) => ABC+*D/E-
- (A+B*(C-D*(E-F)-G*H)-I*J) => ABCDEF-*-GH*-*+IJ*-

Regras da conversão:

- Os operandos aparecem na mesma ordem nas duas notações.
- Os operandos aparecem entre operadores na infixa,
    - e depois dos operandos na pós fixa.
    - Isso sugere que precisaremos de alguma estrutura auxiliar
        - para armazenar um operador enquanto lemos operandos.
- Além disso, os operadores podem mudar de ordem, pois:
    - Na infixa, a ordem em que os operadores serão executados depende
        - da ordem em que eles aparecem,
        - da precedência dos operadores,
        - dos parênteses.
    - Na pósfixa, operadores que aparecem primeiro
        - são sempre executados primeiro
    - Isso sugere que nossa estrutura auxiliar também precisará
        - inverter a ordem de operadores em algumas situações.
- As operações entre parênteses na infixa
    - continuam aparecendo em blocos contínuos na pósfixa.

Como já vimos, a pilha é uma estrutura útil

- para armazenar informações e para inverter a ordem de sequências.

Observe que, como na notação pósfixa o operador é executado assim que é lido

- a ordem dos operadores na pós fixa
    - corresponde à ordem em que os operadores são executados na infixa.

Vamos estudar um algoritmo para realizar esta conversão.

- Trata-se de um algoritmo iterativo
    - que utiliza uma pilha.

Simulação:

- Primeiro veremos uma simulação passo-a-passo
  - para entender a ideia do algoritmo.
- Considere a seguinte string em notação infixa
  - (A*(B*C+D))

inf[0 .. i - 1]	s[0 .. t - 1]	posf[0 .. j - 1]
(	(	
(A	(	A
(A*	(*	A
(A*(	(*(	A
(A*(B	(*(	AB
(A*(B*	(*(*	AB
(A*(B*C	(*(*	ABC
(A*(B*C+	(*(+	ABC*
(A*(B*C+D	(*(+	ABC*D
(A*(B*C+D)	(*	ABC*D+
(A*(B*C+D))		ABC*D+*

Código:

```
// Esta função recebe uma expressão infixa inf
// e devolve a correspondente expressão posfixa.
char *infix2posfix(char *inf)
{
   int n = strlen(inf);
   char *posf; // expressão pósfixa
   posf = malloc((n + 1) * sizeof(char));
   int i;   // percorre infixa
   int j;   // percorre posfixa
   char *s; // pilha
   int t;   // topo da pilha

   // inicializa a pilha
   s = malloc(n * sizeof(char));
   t = 0;

   for (i = j = 0; inf[i] != '\0'; i++)
   {
```

```c
        switch (inf[i])
        {
            char x; // auxiliar para item do topo da pilha
        case '(':
            s[t++] = inf[i]; // empilha
            break;
        case ')':
            x = s[--t]; // desempilha
            while (x != '(')
            {
                posf[j++] = x;
                x = s[--t]; // desempilha
            }
            break;
        case '+':
        case '-':
            // desempilha enquanto pilha não for vazia e não encontrar '('
            while (t > 0 && s[t - 1] != '(')
            {
                posf[j++] = s[--t]; // desempilha
            }
            s[t++] = inf[i]; // empilha
            break;
        case '*':
        case '/':
            // desempilha enquanto não encontrar início do bloco ou operador de menor
precedência na pilha
            while (t > 0 && (x = s[t - 1]) != '(' && x != '+' && x != '-')
            {
                posf[j++] = s[--t]; // desempilha
            }
            s[t++] = inf[i];
            break;
        default:
            if (inf[i] != ' ')
                posf[j++] = inf[i];
        }
    }
    // desempilha o que sobrou na pilha
    while (t > 0)
        posf[j++] = s[--t];
    posf[j] = '\0';
    free(s);
    return posf;
}
```

Eficiência de tempo:

- o algoritmo realiza da ordem de n operações, i.e., O(n),

    - sendo n o número de caracteres na string inf.
- Isto porque o laço principal realiza n iterações,
    - para percorrer a string de entrada,
- e todos os demais laços inserem ou removem elementos da pilha.
    - Sendo que cada operando da entrada
        - é inserido no máximo uma vez na pilha.

Eficiência de espaço:
- o algoritmo utiliza memória extra da ordem de n, i.e., O(n),
    - já que precisa alocar uma string de saída "posf" e uma pilha "s",
        - de tamanho (n + 1) e n, respectivamente.