Pilhas, inversão de sequências, notação infixa para pósfixa, conversão de recursão para iteração

Relembrando operações para manipulação de pilha implementada em vetor:

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

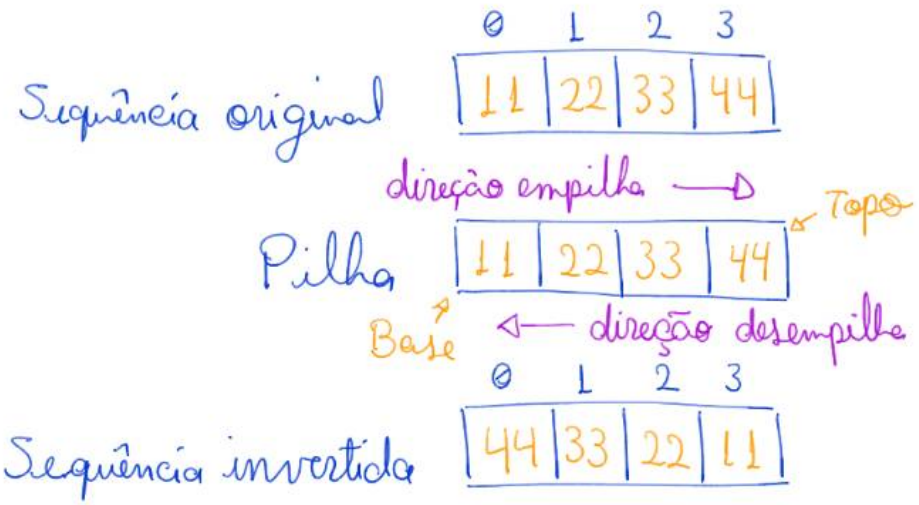
Pilhas e inversão de sequências

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 operadores aparecem entre operandos 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,
            - como quando um + aparece logo antes de um * na infixa.
- 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]	pilha[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:

```c
// 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 *pilha;
    int t; // topo da pilha

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

    for (i = j = 0; inf[i] != '\0'; i++) {
        switch (inf[i]) {
            char x; // auxiliar para item do topo da pilha
        case '(':
            pilha[t++] = inf[i]; // empilha
            break;
```

```
        case ')': // move da pilha pra pósfixa até encontrar '('
            x = pilha[--t]; // desempilha
            while (x != '(') {
                posf[j++] = x;
                x = pilha[--t]; // desempilha
            }
            break;
        case '+':
        case '-':
            // joga na pósfixa conteúdo da pilha até esta ficar
            // vazia ou encontrar o início do bloco '('
            while (t > 0 && pilha[t - 1] != '(') {
                posf[j++] = pilha[--t]; // desempilha
            }
            pilha[t++] = inf[i]; // empilha
            break;
        case '*':
        case '/':
            // joga na pósfixa conteúdo da pilha até esta ficar
            // vazia, encontrar o início do bloco '(', ou
            // encontrar operador de menor precedência '+' ou '-'
            while (t > 0 && (x = pilha[t - 1]) != '(' && x != '+' &&
x != '-') {
                posf[j++] = pilha[--t]; // desempilha
            }
            pilha[t++] = inf[i]; // empilha
            break;
        default:
            if (inf[i] != ' ')      // ignora espaços
                posf[j++] = inf[i]; // copia operandos pra pósfixa
        }
    }
    // desempilha o que sobrou na pilha
    while (t > 0)
        posf[j++] = pilha[--t];
    posf[j] = '\0';
    free(pilha);
    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 operador 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
        - de tamanho (n + 1) e n, respectivamente.

Bônus: conversão de recursão para iteração

Como converter um algoritmo recursivo para um iterativo?

- Primeiro um exemplo do caso simples, quando trata-se de recursão caudal.
    - Depois um exemplo de recursão geral, usando pilha.

Recursão caudal: É o caso em que a chamada recursiva

- é a última coisa a acontecer antes do final da função.

Algoritmo recursivo para busca em vetor:

```
int buscaR(int x, int *v, int n) {
    if (n == 0) return -1;
    if (x == v[n - 1]) return n - 1;
    return buscaR(x, v, n - 1);
}
```

Conversão para iterativo:

```
int buscaI(int x, int *v, int n) {
    while (1) {
        if (n == 0) return -1;
        if (x == v[n - 1]) return n - 1;
        n = n - 1; /* atualiza o valor dos parâmetros que mudam na
chamada recursiva */
    }
}
```

Note que, recursão caudal é facilmente convertida para algoritmo iterativo,

- sem uso de pilha,
- pois quando a chamada recursiva termina,
    - não há mais nada que fazer na função que a chamou.
- É exatamente para tratar o retorno da recursão que a pilha é essencial.

Recursão geral:

Algoritmo recursivo para somar os elementos positivos de um vetor.

```c
int somaPositivosR(int *v, int n) {
    int res; /* variável supérflua que ajuda a entender a conversão */
    if (n == 0) { /* caso base */
        res = 0;
        return res;
    }
    /* 111 - marcador do inicio da função (após caso base) */
    if (v[n - 1] > 0) {
        res = somaPositivosR(v, n - 1); /* 222 - marcador da volta da primeira recursão */
        res += v[n - 1];
        return res;
    }
    else { // v[n - 1] <= 0
        res = somaPositivosR(v, n - 1); /* 333 - marcador da volta da segunda recursão */
        return res;
    }
}
```

Conversão para iterativo com pilha:

```c
int somaPositivosI(int *v, int n) {
    int res = -1;
    int addr = 111; /* variável auxiliar para saber em que ponto voltar na função */
    int *s, t;      /* variáveis para pilha e topo */
    s = malloc((2 * n + 2) * sizeof(int));
    t = 0;
    s[t++] = 0; // inicializando endereço inicial arbitrário
    s[t++] = 0; // e valor original arbitrário pra começar a pilha

    while (t > 0) {
        if (n == 0) { /* caso base */
            res = 0;
            n = s[--t];
            addr = s[--t]; /* corresponde ao return */
        }
```

```
        else { // n > 0
            switch (addr) {
            case 111: /* inicio da função (após caso base) */
                if (v[n - 1] > 0) {
                    s[t++] = 222;
                    s[t++] = n; /* armazena variáveis para volta */
                    addr = 111;
                    n = n - 1; /* atualiza variáveis para chamada
recursiva */
                }
                else { // v[n - 1] <= 0
                    s[t++] = 333;
                    s[t++] = n; /* armazena variáveis para volta */
                    addr = 111;
                    n = n - 1; /* atualiza variáveis para chamada
recursiva */
                }
                break;
            case 222: /* volta da primeira recursão */
                res += v[n - 1];
                n = s[--t];
                addr = s[--t]; /* corresponde ao return */
                break;
            case 333:      /* volta da segunda recursão */
                res = res; /* supérfluo para manter o padrão */
                n = s[--t];
                addr = s[--t]; /* corresponde ao return */
                break;
            }
        }
    }
    free(s);
    return res;
}
```

Notem que, se o valor de res

- não fosse apenas acumulado ao longo das chamadas/iterações,
- ele também teria que ser salvo na pilha e restaurado desta,
  - da mesma forma que fazemos com o endereço de retorno addr
    - e com o valor de n.