Pilhas

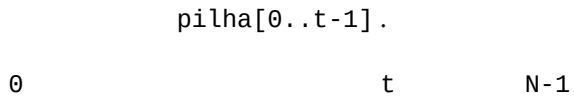
Uma pilha é uma das várias estruturas de dados que admitem remoção de elementos e inserção de novos elementos. Mais especificamente, uma **pilha** (= *stack*) é uma estrutura sujeita à seguinte regra de operação: sempre que houver uma remoção, o elemento removido é o que está na estrutura há menos tempo.

Em outras palavras, o primeiro objeto a ser inserido na pilha é o último a ser removido. Essa política é conhecida pela sigla LIFO (= *Last-In-First-Out*).

Veja o verbete *Stack (data structure)* na Wikipedia.

Implementação em um vetor

Suponha que nossa pilha está armazenada em um vetor `pilha[0..n-1]`. Suponha ainda que os elementos de `pilha` são números inteiros. (Isto é só um exemplo; os elementos de `pilha` poderiam ser objetos de qualquer outro tipo.) A parte do vetor ocupada pela pilha será



O índice `t` indica o **topo** (= *top*) da pilha. Esta é a primeira posição vaga da pilha. A pilha está **vazia** se `t` vale `0` e **cheia** se `t` vale `n`.

Para **remover** um elemento da pilha — esta operação é conhecida como desempilhar (= *to pop*) — faça

```
x = pilha[--t];
```

Isso equivale ao par de instruções `"t -= 1; x = pilha[t];"` nesta ordem. É claro que você só deve desempilhar se tiver certeza de que a pilha não está vazia. Para **consultar** a pilha sem desempilhar faça `x = pilha[t-1]`.

Para **inserir** — ou seja, para empilhar (= *to push*) — um objeto `y` na pilha faça

```
pilha[t++] = y;
```

Isso equivale ao par de instruções `"pilha[t] = y; t += 1;"` nesta ordem. Antes de empilhar, verifique se a pilha já está cheia para evitar que ela *transborde* (ou seja, para evitar um *overflow*). Em geral, a tentativa de inserir em uma pilha cheia é uma situação excepcional, que indica um mau planejamento lógico do seu programa.

Exercícios

1. Suponha que, diferentemente da convenção adotada no texto, a parte do vetor ocupada pela pilha é `pilha[0..t]`. Escreva a instrução que remove um elemento da pilha. Escreva a instrução que insere um objeto `y` na pilha.

2. Escreva funções `empilha` e `desempilha` para manipular uma pilha. Lembre-se de que uma pilha é um pacote com dois objetos: um vetor e um índice. Não use variáveis globais. Quais os parâmetros de suas funções?

Aplicação: parênteses e colchetes

Suponha que queremos decidir se uma dada sequência de parênteses e colchetes está bem-formada (ou seja, parênteses e colchetes são fechados na ordem inversa àquela em que forma abertos). Por exemplo, a primeira das sequências abaixo está bem-formada enquanto a segunda não está.

```
( ( ) [ ( ) ] )          ( [ ) ]
```

Suponha que a sequência de parênteses e colchetes está armazenada em uma cadeia de caracteres (= *string*) `s`. Como é hábito em C, o último caractere da cadeia é o caractere nulo, `'\0'`.

```
// A função devolve 1 se a cadeia s contém uma sequência
// bem-formada de parênteses e colchetes e devolve 0 se
// a sequência está malformada.

int bemFormada (char s[])
{
   char *pilha; int t;
   int n, i;

   n = strlen (s);
   pilha = mallocX (n * sizeof (char));
   t = 0;
   for (i = 0; s[i] != '\0'; ++i) {
      // a pilha está armazenada no vetor pilha[0..t-1]
      switch (s[i]) {
         case ')': if (t != 0 && pilha[t-1] == '(')
                      --t;
                   else return 0;
                   break;
         case ']': if (t != 0 && pilha[t-1] == '[')
                      --t;
                   else return 0;
                   break;
         default:  pilha[t++] = s[i];
      }
   }
   return t == 0;
}
```

(Eu deveria ter invocado `free(pilha)` antes de cada `return`. Só não fiz isso para não obscurecer a lógica da função.) Note que a pilha não transborda porque nunca terá mais elementos que o número de caracteres de `s`.

Exercícios

Eis algumas questões sobre a função `bemFormada`:

3. A função funciona corretamente se `s` tem apenas dois elementos? apenas um? nenhum?

4. Mostre que no início de cada iteração `s` está bem-formada se e somente se a sequência `pilha[0..t-1] s[i...]` estiver bem-formada.

5. Escreva uma versão melhorada da função `bemFormada` que desaloque o vetor `pilha` antes de terminar a execução da função. [Solução]

Outra aplicação: notação polonesa

Na notação usual de expressões aritméticas, os operadores são escritos *entre* os operandos; por isso, a notação é chamada **infixa**. Na notação **polonesa**, ou **posfixa**, os operadores são escritos *depois* dos operandos. (A propósito, veja exercício sobre expressões aritméticas e árvores binárias.) Exemplo:

infixa	posfixa
`(A+B*C)`	`ABC*+`
`(A*(B+C)/D-E)`	`ABC+*D/E-`
`(A+B*(C-D*(E-F)-G*H)-I*3)`	`ABCDEF-*-GH*-*+I3*-`
`(A+B*C/D*E-F)`	`ABC*D/E*+F-`
`(A+(B-(C+(D-(E+F)))))`	`ABCDEF+-+-+`
`(A*(B+(C*(D+(E*(F+G))))))`	`ABCDEFG+*+*+*`

A notação posfixa dispensa parênteses. A ordem dos operandos é a mesma nas expressões infixa e posfixa. Nosso problema:

> Traduzir para notação posfixa a expressão infixa armazenada em uma cadeia de caracteres `inf`.

Para simplificar nossa vida, vamos supor que a expressão infixa está correta e consiste apenas de letras, abre-parêntese, fecha-parêntese e símbolos para as quatro operações aritméticas. Vamos supor também que cada nome de variável tem uma letra apenas. Finalmente, vamos supor que a expressão toda está embrulhada em um par de parênteses, isto é, `inf[0]` vale `'('` e os dois últimos elementos de `inf` são `')'` e `'\0'`.

Usaremos uma pilha para resolver o problema. Como a expressão infixa está embrulhada em um par de parênteses, não precisamos nos preocupar com pilha vazia!

```
// A função abaixo recebe uma expressão infixa inf e
// devolve a correspondente expressão posfixa.

char *infixaParaPosfixa (char inf[]) {
   char *posf;
   char *pi; int t; // pilha
   int n, i, j;

   n = strlen (inf);
   posf = mallocX (n * sizeof (char));
   pi = mallocX (n * sizeof (char));
   t = 0;
   pi[t++] = inf[0];                          // empilha '('
   for (j = 0, i = 1; /*X*/ inf[i] != '\0'; ++i) {
      // a pilha está em pi[0..t-1]
      switch (inf[i]) {
         char x;
         case '(': pi[t++] = inf[i];          // empilha
                   break;
         case ')': while (1) {                // desempilha
                      x = pi[--t];
                      if (x == '(') break;
                      posf[j++] = x;
                   }
                   break;
         case '+':
         case '-': while (1) {
                      x = pi[t-1];
                      if (x == '(') break;
                      --t;                    // desempilha
```

```
                    posf[j++] = x;
                }
                pi[t++] = inf[i];          // empilha
                break;
        case '*':
        case '/': while (1) {
                    x = pi[t-1];
                    if (x == '(' || x == '+' || x == '-')
                       break;
                    --t;
                    posf[j++] = x;
                }
                pi[t++] = inf[i];
                break;
        default:  posf[j++] = inf[i];
      }
   }
   free (pi);
   posf[j] = '\0';
   return posf;
}
```

[Veja outra maneira de escrever a função tirando proveito dos recursos sintáticos da linguagem C.]
Constantes e variáveis vão diretamente de `inf` para `posf`. Abre-parêntese vai para a pilha. Ao encontrar um fecha-parêntese, a função remove tudo da pilha até o abre-parêntese inclusive. Ao encontrar um `+` ou `-` a função desempilha tudo até um abre-parêntese exclusive. Ao encontrar um `*` ou `/` desempilha tudo até um abre-parêntese ou um `+` ou um `-`.

Eis o resultado da aplicação da função à expressão infixa `( A * ( B * C + D ) )`. A tabela abaixo registra os valores das variáveis a cada passagem pelo ponto `X`:

```
inf[0..i-1]  pi[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+*
```

Exercícios

6. Aplique o algoritmo de conversão para a notação posfixa à expressão aritmética

   `(A + B) * D + E / (F + A * D) + C`

7. Considere a função `infixaParaPosfixa`. Suponha que a expressão infixa `inf` tem `n` caracteres (sem contar o `'\0'` final). Que tamanho a pilha `pi` pode atingir, no pior caso? Em outras palavras, qual o valor máximo da variável `t` no pior caso? Que acontece se o número de abre-parênteses na expressão for limitado (menor que 6, por exemplo)?

8. No código da função `infixaParaPosfixa`, alguns casos têm um `while (1)`. Escreva uma nova versão sem esses `while`. (Dica: troque o `for` externo por um `while` apropriado.)

9. Reescreva a função `infixaParaPosfixa` *sem* supor que a expressão infixa está embrulhada em um par de parênteses.

10. Reescreva a função `infixaParaPosfixa` supondo que a expressão pode ter parênteses *e colchetes*.

11. Reescreva a função `infixaParaPosfixa` supondo que a expressão pode não estar bem-formada.

Exercícios

12. VALOR DE EXPRESSÃO POLONESA. Suponha que `posf` é uma string que guarda uma expressão aritmética em notação posfixa. Suponha que `posf` não é vazio e contém somente os operadores `+`, `-`, `*` e `/` (todos exigem *dois* operandos). Suponha também que a expressão não tem constantes e que todos os nomes de variáveis na expressão consistem de uma única letra maiúscula (`A`, . . . , `Z`). Suponha ainda que temos um vetor `tabela` que dá os valores das variáveis (todas as variáveis têm valores inteiros):

    `tabela[0]` é o valor da variável `A`,
    `tabela[1]` é o valor da variável `B` etc.

    Escreva uma função que calcule o valor da expressão `posf`. Cuidado com divisões por zero!

13. Escreva um algoritmo que use uma pilha para *inverter a ordem* das letras de cada palavra de uma string, preservando a ordem das palavras. Por exemplo, dado o texto `ESTE EXERCICIO E MUITO FACIL` a saída deve ser `ETSE OICICREXE E OTIUM LICAF`. (Lembre-se: strings em C terminam com `'\0'`.)

14. Digamos que nosso alfabeto é formado pelas letras `a`, `b` e `c`. Considere o seguinte conjunto de cadeias de caracteres sobre nosso alfabeto:

    `c, aca, bcb, abcba, bacab, aacaa, bbcbb, ...`

    Qualquer cadeia deste conjunto tem a forma *W*`c`*M*, onde *W* é uma sequência de letras que só contém `a` e `b` e *M* é o inverso de *W*, ou seja, *M* é *W* lido de trás para frente. Escreva um programa que determina se uma cadeia *X* pertence ou não ao nosso conjunto, ou seja, determina se *X* é da forma *W*`c`*M*.

15. PERMUTAÇÕES PRODUZIDAS PELO DESEMPILHAR. Suponha que os inteiros 1,2,3,4 são colocados, nesta ordem, numa pilha inicialmente vazia. Depois de empilhar cada inteiro, você pode retirar zero ou mais elementos da pilha. Cada elemento desempilhado é impresso numa folha de papel. Por exemplo, a sequência de operações empilha 1, empilha 2, desempilha, empilha 3, desempilha, desempilha, empilha 4, desempilha, produz a impressão da sequência 2,3,1,4. Quais das 24 permutações de 1,2,3,4 podem ser obtidas desta maneira?

Pilha implementada em uma lista encadeada

Como implementar uma pilha em uma lista encadeada? Digamos que as células da lista são do tipo `celula`:

```
typedef struct cel {
   int         conteudo;
   struct cel *prox;
} celula;
```

Decisões de projeto: Vamos supor que nossa lista tem uma célula-cabeça (ou seja, vamos supor que a primeira célula da lista não faz parte da pilha). Vamos supor que o topo da pilha está na *segunda* célula e não na última (por quê?). A pilha pode ser criada e inicializada assim:

```
celula cabeca;
```

```
celula *tp;
tp = &cabeça;
tp->prox = NULL;
```

Para ter acesso à pilha, só preciso do ponteiro `tp`. De acordo com nossa decisão de projeto, teremos sempre `tp` == `&cabeca`. A pilha está *vazia* se `tp->prox` == `NULL`.

```
// Insere um elemento y na pilha tp.

void empilha (int y, celula *tp) {
   celula *nova;
   nova = mallocX (sizeof (celula));
   nova->conteudo = y;
   nova->prox  = tp->prox;
   tp->prox = nova;
}

// Remove um elemento da pilha tp.
// Supõe que a pilha não está vazia.
// Devolve o elemento removido.

int desempilha (celula *tp) {
   int x;
   celula *p;
   p = tp->prox;
   x = p->conteudo;
   tp->prox = p->prox;
   free (p);
   return x;
}
```

Exercícios

16. Implemente um pilha em uma lista encadeada *sem* célula-cabeça (só pra ver ver que dor de cabeça isso dá!). A pilha será dada pelo endereço da primeira célula da lista (que é também o topo da pilha).

17. Reescreva as funções `bemFormada` e `infixaParaPosfixa` armazenando a pilha em uma lista encadeada.

18. Resolva o problem da intercalação de duas listas ordenadas.

19. Suponha dada uma lista encadeada que armazena números inteiros. Cada célula da lista tema estrutura abaixo.

    ```
    struct cel {
       int         conteudo;
       struct cel *prox;
    };
    ```

    Escreva uma função que transforme a lista em duas: a primeira contendo as células cujo conteúdo é par e a segunda aquelas cujo conteúdo é ímpar.

Apêndice: A pilha de execução de um programa

Para executar um programa, o computador usa uma "pilha de execução". A operação pode ser descrita conceitualmente da seguinte maneira.

Todo programa C é composto por uma ou mais funções (sendo `main` a primeira função a ser executada). Ao encontrar a invocação de uma função, o computador cria um novo "espaço de trabalho", que contém todos os parâmetros e todas as variáveis locais da função. Esse espaço de trabalho é colocado na pilha de execução (sobre o espaço de trabalho que invocou a função) e a execução da função começa (confinada ao seu espaço de trabalho). Quando a execução da função termina, o seu espaço de trabalho é retirado da pilha e descartado. O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido interrompida.

Considere o seguinte exemplo:

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

int F (int i, j, 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;
}
```

A execução do programa prossegue da seguinte maneira:

- Um espaço de trabalho é criado para a função `main` e colocado na pilha de execução. O espaço contém as variáveis locais `i`, `j`, `k` e `y`. A execução de `main` começa.
- No ponto `1`, a execução de `main` é temporariamente interrompida e um espaço de trabalho para a função `F` é colocado na pilha. Esse espaço contém os parâmetros `i`, `j`, `k` da função (com valores `111`, `222` e `444` respectivamente) e a variável local `x`. Começa então a execução de `F`.
- No ponto `2`, a execução de `F` é interrompida e um espaço de trabalho para a função `G` é colocado na pilha. Esse espaço contém os parâmetros `a` e `b` da função (com valores `111` e `222` respectivamente) e a variável local `x`. Em seguida, começa a execução de `G`.
- Quando a execução de `G` termina, a função devolve `333`. O espaço de trabalho de `G` é removido da pilha e descartado. O espaço de trabalho de `F` (que agora está no topo da pilha de execução) é reativado e a execução é retomada no ponto `3`. A primeira instrução executada é "`x = 333;`".
- Quando a execução de `F` termina, a função devolve `777`. O espaço de trabalho de `F` é removido da pilha e descartado. O espaço de trabalho de `main` (que agora está no topo da pilha) é reativado e a execução é retomada no ponto `4`. A primeira instrução executada é "`y = 777;`".

No nosso exemplo, `F` e `G` são funções distintas. Mas tudo funcionaria da mesma maneira se `F` e `G` fossem idênticas, ou seja, se `F` fosse uma função recursiva.

Exercícios

20. Considere a função recursiva abaixo. Escreva uma versão iterativa da função que simule o comportamento da versão recursiva. Use uma pilha.

```
int TTT (int x[], int n) {
   if (n == 0) return 0;
   if (x[n] > 0) return x[n] + TTT (x, n-1);
   else return TTT (x, n-1);
}
```

URL of this site: www.ime.usp.br/~pf/algoritmos/
1998 | Last modified: Mon Oct 18 08:58:35 BRT 2010
Paulo Feofiloff
IME-USP