Pilhas

Uma pilha é uma estrutura de dados que admite remoção de elementos e inserção de novos objetos. 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*).

Sumário:

Implementação em um vetor

Suponha que nossa pilha está armazenada em um vetor `pilha[0..N-1]`. (A natureza dos elementos do vetor é irrelevante: eles podem ser inteiros, bytes, ponteiros, etc.) Digamos que a parte do vetor ocupada pela pilha é

`pilha[0..t-1]` .

O índice `t` indica a primeira posição vaga do vetor e `t-1` é o índice do *topo* da pilha. A pilha está *vazia* se `t` vale `0` e *cheia* se `t` vale `N`. No exemplo da figura, os caracteres A, B, ... , H foram inseridos na pilha nessa ordem:

Para *remover*, ou *tirar*, um elemento da pilha — essa operação é conhecida como *desempilhar* (= *to pop*) — faça

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

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

Para *inserir*, ou *colocar*, um objeto `y` na pilha — a operação é conhecida como *empilhar* (= *to push*) — faça

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

Isso equivale ao par de instruções "`pilha[t] = y; t += 1;`". Antes de empilhar, certifique-se de que a pilha não está cheia, para evitar um *transbordamento* (= *overflow*).

Para facilitar a leitura do código, é conveniente embalar essas operações em duas pequenas funções. Se os objetos com que estamos lidando são do tipo `char`, podemos escrever

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

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

Estamos supondo aqui que as variáveis `pilha` e `t` são globais, ou seja, foram definidas fora do código das funções. Para completar o pacote, precisaríamos de mais três funções: uma que crie uma pilha, uma que verifique se a pilha está vazia e uma que verifique se a pilha está cheia. (Veja exercício abaixo.)

Exercícios 1

1. Módulo de implementação de pilha (versão 1). Escreva um módulo `pilhadechars.c` que implemente uma pilha de caracteres ASCII. O módulo deve conter as funções `criapilha`, `empilha`, `desempilha`, `pilhavazia`, `pilhacheia`. Trate os parâmetros da pilha (o vetor `pilha` e o índice `t`) como variáveis globais do módulo. Escreva também uma interface `pilhadechars.h` para o módulo. [Solução: ./solucoes/pilha3.html]

2. Tome uma decisão de projeto diferente da adotada acima: suponha que a parte do vetor ocupada pela pilha é `pilha[1..t]`. Escreva o código das funções `empilha`, `desempilha`, `pilhavazia` e `pilhacheia`.

3. Escreva um algoritmo que use uma pilha para inverter a ordem das letras de cada palavra de uma string ASCII, preservando a ordem das palavras. Por exemplo, para a string `ESTE EXERCICIO E MUITO FACIL` o resultado deve ser `ETSE OICICREXE E OTIUM LICAF`.

4. Permutações produzidas pelo desempilhar. Suponha que objetos 1, 2, 3, 4 são colocados, nessa ordem, numa pilha inicialmente vazia. Depois de empilhar um objeto, você pode tirar 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 dessa maneira?

5. [Sedgewick] O fragmento de programa abaixo opera uma pilha de objetos do tipo `char`. (A função `espia` devolve uma cópia do topo da pilha, mas não tira esse elemento da pilha.) Diga, em português, *o que* o fragmento faz. Escreva um fragmento de código equivalente que seja bem mais curto e mais simples.

```
if (pilhavazia ())  empilha ('B');
else {
   if (espia () != 'A')  empilha ('B');
   else {
      while (!pilhavazia () && espia () == 'A')
         desempilha ();
      empilha ('B'); } }
```

Aplicação: parênteses e colchetes

Considere o problema de 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 foram abertos). Por exemplo, a sequência

(( ) [( )])

está bem-formada, enquanto ([ )] está malformada. Suponha que a sequência de parênteses e colchetes está armazenada em uma string ASCII s. (Como é hábito em C, o último caractere da string é \0.)

Usaremos uma pilha para resolver o problema. O algoritmo é simples: examine a string da esquerda para a direita e empilhe os parênteses e colchetes esquerdos à espera de que apareçam os correspondentes parênteses e colchetes direitos. (Veja pseudocódigo.)

Para simplificar, as variáveis `pilha` e `t` serão globais. Suporemos também que o tamanho `N` do vetor que abriga a pilha é maior que o tamanho da string e portanto a pilha jamais transborda.

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

// Esta função devolve 1 se a string ASCII s
// contém uma sequência bem-formada de
// parênteses e colchetes; devolve 0 se
// a sequência é malformada.

int bemFormada (char s[])
{
   criapilha ();
   for (int i = 0; s[i] != '\0'; ++i) {
      char c;
      switch (s[i]) {
         case ')': if (pilhavazia ()) return 0;
                   c = desempilha ();
                   if (c != '(') return 0;
                   break;
         case ']': if (pilhavazia ()) return 0;
                   c = desempilha ();
                   if (c != '[') return 0;
                   break;
         default:  empilha (s[i]);
      }
   }
   return pilhavazia ();
}

void criapilha (void) {
   t = 0;
}

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

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

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

(Poderíamos operar a pilha diretamente, sem invocar as funções de manipulação da pilha. O resultado seria bem mais curto e compacto, mas um pouco menos legível.)

Exercícios 2

1. Dê uma definição formal de *sequência bem-formada* de parênteses e colchetes. Sugestão: definição recursiva.
2. A função `bemFormada` funciona corretamente se a sequência `s` tem apenas dois elementos? apenas um? nenhum?
3. Mostre que, no início de cada iteração da função `bemFormada`, `s` está bem-formada se e somente se a sequência `pilha[0..t-1]s[i...]` está bem-formada.
4. Escreva uma versão da função `bemFormada` que aloque a pilha dinamicamente.
5. Digamos que nosso alfabeto contém apenas as letras `a`, `b` e `c`. Considere o seguinte conjunto de strings: `c`, `aca`, `bcb`, `abcba`, `bacab`, `aacaa`, `bbcbb`, … Qualquer string desse conjunto tem a forma *W*c*M*, sendo *W* uma sequência de letras que só contém `a` e `b` e *M* o inverso de *W* (ou seja, *M* é *W* lido de trás para frente). Escreva um programa que decida se uma string *x* pertence ou não ao nosso conjunto, ou seja, decida se *x* é da forma *W*c*M*.

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 *posfixa*, ou *polonesa*, os operadores são escritos *depois* dos operandos. Eis alguns exemplos de expressões infixas e correspondentes expressões posfixas:

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*G)`	`AB+CD*+EF*G*-`
`(A+(B-(C+(D-(E+F)))))`	`ABCDEF+-+-+`
`(A*(B+(C*(D+(E*(F+G))))))`	`ABCDEFG+*+*+*`

Note que os operandos (`A`, `B`, `C`, etc.) aparecem na mesma ordem na expressão infixa e na correspondente expressão posfixa. Note também que a notação posfixa *dispensa parênteses* e *regras de precedência* entre operadores (como a precedência de * sobre + por exemplo), que são indispensáveis na notação infixa.

Nosso problema: traduzir para notação posfixa a expressão infixa armazenada em uma string `inf`. Para simplificar nossa vida, vamos supor que

- a expressão `inf` é válida e contém apenas letras ASCII, parênteses, e os símbolos das quatro operações aritméticas,
- os nomes das variáveis têm apenas uma letra cada,
- todas as operações (em particular `-` e `+`) têm *dois* operandos,

- a expressão `inf` está embrulhada em um par de parênteses (ou seja, o primeiro caractere da string é `'('` e os dois últimos são `')'` e `'\0'`).

O algoritmo lê a expressão `inf` caractere-a-caractere e usa uma pilha para fazer a tradução. Todo parêntese esquerdo é colocado na pilha. Ao encontrar um parêntese direito, o algoritmo desempilha tudo até encontrar um parêntese esquerdo, que também é desempilhado. Ao encontrar um `+` ou um `-`, o algoritmo desempilha tudo até encontrar um parêntese esquerdo, que não é desempilhado. Ao encontrar um `*` ou um `/`, o algoritmo desempilha tudo até encontrar um parêntese esquerdo ou um `+` ou um `-`. Constantes e variáveis são transferidos diretamente de `inf` para a expressão posfixa. (Veja um rascunho em pseudocódigo.)

As variáveis `pilha` e `t` são globais. Vamos supor que `N` é maior que o tamanho da string `inf`, e portanto não precisamos nos preocupar com pilha cheia. Como a expressão `inf` está embrulhada em parênteses, não precisamos nos preocupar com pilha vazia.

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

// Esta função recebe uma expressão infixa inf
// e devolve a correspondente expressão posfixa.

char *infixaParaPosfixa (char *inf) {
   int n = strlen (inf);
   char *posf;
   posf = malloc ((n+1) * sizeof (char));
   criapilha ();
   empilha (inf[0]);       // empilha '('

   int j = 0;
   for (int i = 1; inf[i] != '\0'; ++i) {
      switch (inf[i]) {
         char x;
         case '(': empilha (inf[i]);
                   break;
         case ')': x = desempilha ();
                   while (x != '(') {
                      posf[j++] = x;
                      x = desempilha ();
                   }
                   break;
         case '+':
         case '-': x = desempilha ();
                   while (x != '(') {
                      posf[j++] = x;
                      x = desempilha ();
                   }
                   empilha (x);
                   empilha (inf[i])
                   break;
         case '*':
         case '/': x = desempilha ();
                   while (x != '(' && x != '+' && x != '-') {
                      posf[j++] = x;
                      x = desempilha ();
                   }
                   empilha (x);
                   empilha (inf[i]);
                   break;
         default:  posf[j++] = inf[i];
      }
```

```
    }
    posf[j] = '\0';
    return posf;
}
```

(Poderíamos operar a pilha diretamente, sem invocar as funções de manipulação da pilha. O resultado seria mais curto e compacto, mas um pouco menos legível.)

Veja o resultado da aplicação da função `infixaParaPosfixa` à expressão infixa `(A*(B*C+D))`. A tabela registra os valores das variáveis no início de cada iteração:

`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+*`

Exercícios 3

1. Use a função `infixaParaPosfixa` para converter `(A+B)*D+E/(F+A*D)+C` na expressão posfixa equivalente.
2. TAMANHO DA PILHA. Na função `infixaParaPosfixa`, que tamanho a `pilha` pode atingir no pior caso, em função de `n`? Em outras palavras, qual o valor máximo da variável `t` no pior caso? Que acontece se o número de parênteses esquerdos na expressão for limitado (menor que 6, por exemplo)?
3. Reescreva a função `infixaParaPosfixa` sem supor que a expressão infixa está embrulhada em um par de parênteses.
4. ALOCAÇÃO DINÂMICA. Reescreva a função `infixaParaPosfixa` e as funções de manipulação da pilha de modo que o vetor `pilha` seja alocado dinamicamente.
5. COLCHETES E PARÊNTESES. Reescreva a função `infixaParaPosfixa` supondo que a expressão infixa pode ter colchetes além de parênteses.
6. Reescreva a função `infixaParaPosfixa` supondo que a expressão pode não ser válida.
7. VALOR DE EXPRESSÃO POLONESA. Suponha que `posf` é uma string ASCII não vazia que armazena uma expressão aritmética em notação posfixa. Suponha que `posf` 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 em uma única letra maiúscula. Suponha ainda que temos um vetor `valor` que dá os valores das variáveis (todos inteiros):

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

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

Implementação em vetor com redimensionamento

Nem sempre é possível prever a quantidade de espaço que uma pilha vai usar. Se a pilha residir num vetor, podemos recorrer ao redimensionamento sempre que a pilha ficar cheia, como já fizemos ao implementar uma fila.

Exercícios 4

1. Módulo de implementação de pilha (versão 2). Escreva um módulo `pilhadechars.c` que implemente uma pilha de caracteres ASCII num vetor com redimensionamento. O módulo deve conter as funções `criapilha`, `empilha`, `desempilha`, `pilhavazia` e `liberapilha`. Trate os parâmetros da pilha como variáveis globais do módulo. Escreva também uma interface `pilhadechars.h` para o módulo.

Pilha implementada em uma lista encadeada

Como implementar uma pilha em uma lista encadeada? Digamos que os elementos da pilha são caracteres ASCII e que as células da lista são do tipo `celula` (veja *Estrutura de uma list ligada*):

```
typedef struct celula {
   char           conteudo;
   struct celula *prox;
} celula;
```

Decisões de projeto: 1. Nossa lista terá uma célula-cabeça (e portanto a primeira célula da lista não fará parte da pilha). 2. O topo da pilha ficará na *segunda* célula e não na última (por quê?). 3. Uma variável global `pi` apontará a cabeça da lista.

As funções de criação e manipulação da pilha podem então ser escritas assim:

```
celula *pi;

void criapilha (void) {
   pi = malloc (sizeof (celula)); // cabeça
   pi->prox = NULL;
}

void empilha (char y) {
   celula *nova;
   nova = malloc (sizeof (celula));
   nova->conteudo = y;
   nova->prox  = pi->prox;
   pi->prox = nova;
}

char desempilha (void) {
   celula *p;
   p = pi->prox;
   char x = p->conteudo;
   pi->prox = p->prox;
   free (p);
   return x;
}
```

(Como de hábito, a função `desempilha` não deve ser invocada se a pilha estiver vazia.)

Exercícios 5

1. Implemente um pilha em uma lista encadeada *sem* célula-cabeça. A pilha será dada pelo endereço da primeira célula da lista (que é o topo da pilha).
2. Reescreva as funções `bemFormada` e `infixaParaPosfixa` armazenando a pilha em uma lista encadeada.

Apêndice: A pilha de execução de um programa

Todo programa C consiste em uma ou mais funções (sendo `main` a primeira função a ser executada). Para administrar as invocações das funções, o computador (ou melhor, o sistema operacional) usa uma *pilha de execução*. (Veja o verbete *Call stack* na Wikipedia.) A operação pode ser descrita conceitualmente da seguinte maneira.

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 (por cima do 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 removido 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. (Veja Julia's drawings: What's the stack?)

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 6

1. Escreva uma função iterativa que simule o comportamento da seguinte função 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);
}
```

2. PILHA DE EXECUÇÃO DE PROGRAMAS. (Este exercício não só simula o funcionamento da pilha de execução como também o processamento das diretivas `#include` e `#define` do pré-processador do compilador C.) Escreva um programa que receba um arquivo de texto ASCII e grave outro arquivo de texto ASCII como especificado a seguir. Cada linha do arquivo de entrada contém palavras separadas por espaços; as palavras que começam com `#` são *especiais* e as outras são *normais*. O arquivo de saída conterá as palavras normais, em uma só linha, numa certa ordem. Suponha, por exemplo, que o arquivo de entrada contém as seguintes linhas:

```
0   #4 aaa #2
1   bbb
2   CC #4 DDD #1 ee
3   FF #2 #4
4   GG hhh
```

   (Os números no início das linha servem apenas de referência e não fazem parte do arquivo.) Então o arquivo de saída deverá conter

```
GG hhh aaa CC GG hhh DDD bbb ee
```

   Como o exemplo sugere, as palavras especiais são substituídas pela linha do arquivo de entrada cujo número é dado depois de "`#`", e isso deve ser feito *recursivamente*. Para tornar o exercício mais interessante, não use funções recursivas.

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

Atualizado em 2018-08-30
https://www.ime.usp.br/~pf/algoritmos/

DCC-IME-USP