11. Pilhas

W. Celes e J. L. Rangel

Uma das estruturas de dados mais simples é a pilha. Possivelmente por essa razão, é a estrutura de dados mais utilizada em programação, sendo inclusive implementada diretamente pelo *hardware* da maioria das máquinas modernas. A idéia fundamental da pilha é que todo o acesso a seus elementos é feito através do seu topo. Assim, quando um elemento novo é introduzido na pilha, passa a ser o elemento do topo, e o único elemento que pode ser removido da pilha é o do topo. Isto faz com que os elementos da pilha sejam retirados na ordem inversa à ordem em que foram introduzidos: o primeiro que sai é o último que entrou (a sigla LIFO – *last in, first out* – é usada para descrever esta estratégia).

Para entendermos o funcionamento de uma estrutura de pilha, podemos fazer uma analogia com uma pilha de pratos. Se quisermos adicionar um prato na pilha, o colocamos no topo. Para pegar um prato da pilha, retiramos o do topo. Assim, temos que retirar o prato do topo para ter acesso ao próximo prato. A estrutura de pilha funciona de maneira análoga. Cada novo elemento é inserido no topo e só temos acesso ao elemento do topo da pilha.

Existem duas operações básicas que devem ser implementadas numa estrutura de pilha: a operação para empilhar um novo elemento, inserindo-o no topo, e a operação para desempilhar um elemento, removendo-o do topo. É comum nos referirmos a essas duas operações pelos termos em inglês *push* (empilhar) e *pop* (desempilhar). A Figura 10.1 ilustra o funcionamento conceitual de uma pilha.

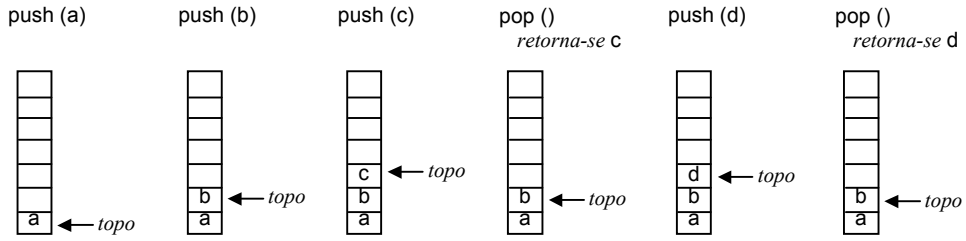

Figura 10.1: Funcionamento da pilha.

O exemplo de utilização de pilha mais próximo é a própria pilha de execução da linguagem C. As variáveis locais das funções são dispostas numa pilha e uma função só tem acesso às variáveis que estão no topo (não é possível acessar as variáveis da função locais às outras funções).

Há várias implementações possíveis de uma pilha, que se distinguem pela natureza dos seus elementos, pela maneira como os elementos são armazenados e pelas operações disponíveis para o tratamento da pilha.

11.1. Interface do tipo pilha

Neste capítulo, consideraremos duas implementações de pilha: usando vetor e usando lista encadeada. Para simplificar a exposição, consideraremos uma pilha que armazena valores reais. Independente da estratégia de implementação, podemos definir a interface do tipo abstrato que representa uma estrutura de pilha. A interface é composta pelas operações que estarão disponibilizadas para manipular e acessar as informações da pilha. Neste exemplo, vamos considerar a implementação de cinco operações:

- criar uma estrutura de pilha;
- inserir um elemento no topo (*push*);
- remover o elemento do topo (*pop*);
- verificar se a pilha está vazia;
- liberar a estrutura de pilha.

O arquivo `pilha.h`, que representa a interface do tipo, pode conter o seguinte código:

```
typedef struct pilha Pilha;

Pilha* cria (void);
void push (Pilha* p, float v);
float pop (Pilha* p);
int vazia (Pilha* p);
void libera (Pilha* p);
```

A função `cria` aloca dinamicamente a estrutura da pilha, inicializa seus campos e retorna seu ponteiro; as funções `push` e `pop` inserem e retiram, respectivamente, um valor real na pilha; a função `vazia` informa se a pilha está ou não vazia; e a função `libera` destrói a pilha, liberando toda a memória usada pela estrutura.

11.2. Implementação de pilha com vetor

Em aplicações computacionais que precisam de uma estrutura de pilha, é comum sabermos de antemão o número máximo de elementos que podem estar armazenados simultaneamente na pilha, isto é, a estrutura da pilha tem um limite conhecido. Nestes casos, a implementação da pilha pode ser feita usando um vetor. A implementação com vetor é bastante simples. Devemos ter um vetor (`vet`) para armazenar os elementos da pilha. Os elementos inseridos ocupam as primeiras posições do vetor. Desta forma, se temos `n` elementos armazenados na pilha, o elemento `vet[n-1]` representa o elemento do topo.

A estrutura que representa o tipo pilha deve, portanto, ser composta pelo vetor e pelo número de elementos armazenados.

```
#define MAX  50

struct pilha {
   int n;
   float vet[MAX];
};
```

A função para criar a pilha aloca dinamicamente essa estrutura e inicializa a pilha como sendo vazia, isto é, com o número de elementos igual a zero.

```
Pilha* cria (void)
{
   Pilha* p = (Pilha*) malloc(sizeof(Pilha));
   p->n = 0;   /* inicializa com zero elementos */
   return p;
}
```

Para inserir um elemento na pilha, usamos a próxima posição livre do vetor. Devemos ainda assegurar que exista espaço para a inserção do novo elemento, tendo em vista que trata-se de um vetor com dimensão fixa.

```
void push (Pilha* p, float v)
{
   if (p->n == MAX) {   /* capacidade esgotada */
      printf("Capacidade da pilha estourou.\n");
      exit(1);          /* aborta programa */
   }
   /* insere elemento na próxima posição livre */
   p->vet[p->n] = v;
   p->n++;
}
```

A função `pop` retira o elemento do topo da pilha, fornecendo seu valor como retorno. Podemos também verificar se a pilha está ou não vazia.

```
float pop (Pilha* p)
{
   float v;
   if (vazia(p)) {
      printf("Pilha vazia.\n");
      exit(1);          /* aborta programa */
   }
   /* retira elemento do topo */
   v = p->vet[p->n-1];
   p->n--;
   return v;
}
```

A função que verifica se a pilha está vazia pode ser dada por:

```
int vazia (Pilha* p)
{
   return (p->n == 0);
}
```

Finalmente, a função para liberar a memória alocada pela pilha pode ser:

```
void libera (Pilha* p)
{
   free(p);
}
```

11.3. Implementação de pilha com lista

Quando o número máximo de elementos que serão armazenados na pilha não é conhecido, devemos implementar a pilha usando uma estrutura de dados dinâmica, no caso, empregando uma lista encadeada. Os elementos são armazenados na lista e a pilha pode ser representada simplesmente por um ponteiro para o primeiro nó da lista.

O nó da lista para armazenar valores reais pode ser dado por:

```
struct no {
   float info;
   struct no* prox;
};
typedef struct no No;
```

A estrutura da pilha é então simplesmente:

```
struct pilha {
   No* prim;
};
```

A função `cria` aloca a estrutura da pilha e inicializa a lista como sendo vazia.

```
Pilha* cria (void)
{
   Pilha* p = (Pilha*) malloc(sizeof(Pilha));
   p->prim = NULL;
   return p;
}
```

O primeiro elemento da lista representa o topo da pilha. Cada novo elemento é inserido no início da lista e, conseqüentemente, sempre que solicitado, retiramos o elemento também do início da lista. Desta forma, precisamos de duas funções auxiliares da lista: para inserir no início e para remover do início. Ambas as funções retornam o novo primeiro nó da lista.

```
/* função auxiliar: insere no início */
No* ins_ini (No* l, float v)
{
   No* p = (No*) malloc(sizeof(No));
   p->info = v;
   p->prox = l;
   return p;
}


/* função auxiliar: retira do início */
No* ret_ini (No* l)
{
   No* p = l->prox;
   free(l);
   return p;
}
```

As funções que manipulam a pilha fazem uso dessas funções de lista:

```
void push (Pilha* p, float v)
{
   p->prim = ins_ini(p->prim,v);
}

float pop (Pilha* p)
{
   float v;
   if (vazia(p)) {
      printf("Pilha vazia.\n");
      exit(1);         /* aborta programa */
   }
   v = p->prim->info;
   p->prim = ret_ini(p->prim);
   return v;
}
```

A pilha estará vazia se a lista estiver vazia:

```
int vazia (Pilha* p)
{
   return (p->prim==NULL);
}
```

Por fim, a função que libera a pilha deve antes liberar todos os elementos da lista.

```
void libera (Pilha* p)
{
   No* q = p->prim;
   while (q!=NULL) {
      No* t = q->prox;
      free(q);
      q = t;
   }
   free(p);
}
```

A rigor, pela definição da estrutura de pilha, só temos acesso ao elemento do topo. No entanto, para testar o código, pode ser útil implementarmos uma função que imprima os valores armazenados na pilha. Os códigos abaixo ilustram a implementação dessa função nas duas versões de pilha (vetor e lista). A ordem de impressão adotada é do topo para a base.

```
/* imprime: versão com vetor */
void imprime (Pilha* p)
{
   int i;
   for (i=p->n-1; i>=0; i--)
      printf("%f\n",p->vet[i]);
}

/* imprime: versão com lista */
void imprime (Pilha* p)
{
   No* q;
   for (q=p->prim; q!=NULL; q=q->prox)
      printf("%f\n",q->info);
}
```

11.4. Exemplo de uso: calculadora pós-fixada

Um bom exemplo de aplicação de pilha é o funcionamento das calculadoras da HP (Hewlett-Packard). Elas trabalham com expressões pós-fixadas, então para avaliarmos uma expressão como `(1-2)*(4+5)` podemos digitar `1 2 - 4 5 + *`. O funcionamento dessas calculadoras é muito simples. Cada operando é empilhado numa pilha de valores. Quando se encontra um operador, desempilha-se o número apropriado de operandos (dois para operadores binários e um para operadores unários), realiza-se a operação devida e empilha-se o resultado. Deste modo, na expressão acima, são empilhados os valores `1` e `2`. Quando aparece o operador `-`, `1` e `2` são desempilhados e o resultado da operação, no caso `-1` (`= 1 - 2`), é colocado no topo da pilha. A seguir, `4` e `5` são empilhados. O operador seguinte, `+`, desempilha o `4` e o `5` e empilha o resultado da soma, `9`. Nesta hora, estão na pilha os dois resultados parciais, `-1` na base e `9` no topo. O operador `*`, então, desempilha os dois e coloca `-9` (`= -1 * 9`) no topo da pilha.

Como exemplo de aplicação de uma estrutura de pilha, vamos implementar uma calculadora pós-fixada. Ela deve ter uma pilha de valores reais para representar os operandos. Para enriquecer a implementação, vamos considerar que o formato com que os valores da pilha são impressos seja um dado adicional associado à calculadora. Esse formato pode, por exemplo, ser passado quando da criação da calculadora.

Para representar a interface exportada pela calculadora, podemos criar o arquivo `calc.h`:

```
/* Arquivo que define a interface da calculadora */

typedef struct calc Calc;

/* funções exportadas */
Calc* cria_calc (char* f);
void operando (Calc* c, float v);
void operador (Calc* c, char op);
void libera_calc (Calc* c);
```

Essas funções utilizam as funções mostradas acima, independente da implementação usada na pilha (vetor ou lista). O tipo que representa a calculadora pode ser dado por:

```
struct calc {
   char f[21];   /* formato para impressão */
   Pilha* p;     /* pilha de operandos */
};
```

A função `cria` recebe como parâmetro de entrada uma cadeia de caracteres com o formato que será utilizado pela calculadora para imprimir os valores. Essa função cria uma calculadora inicialmente sem operandos na pilha.

```
Calc* cria_calc (char* formato)
{
 Calc* c = (Calc*) malloc(sizeof(Calc));
 strcpy(c->f,formato);
 c->p = cria();           /* cria pilha vazia */
 return c;
}
```

A função `operando` coloca no topo da pilha o valor passado como parâmetro. A função `operador` retira os dois valores do topo da pilha (só consideraremos operadores binários), efetua a operação correspondente e coloca o resultado no topo da pilha. As operações válidas são: `'+'` para somar, `'-'` para subtrair, `'*'` para multiplicar e `'/'` para dividir. Se não existirem operandos na pilha, consideraremos que seus valores são zero. Tanto a função `operando` quanto a função `operador` imprimem, utilizando o formato especificado na função `cria`, o novo valor do topo da pilha.

```
void operando (Calc* c, float v)
{
 /* empilha operando */
 push(c->p,v);

 /* imprime topo da pilha */
 printf(c->f,v);
}
```

```
void operador (Calc* c, char op)
{
 float v1, v2, v;

 /* desempilha operandos */
 if (vazia(c->p))
    v2 = 0.0;
 else
    v2 = pop(c->p);
 if (vazia(c->p))
    v1 = 0.0;
 else
    v1 = pop(c->p);

 /* faz operação */
 switch (op) {
    case '+':  v = v1+v2;  break;
    case '-':  v = v1-v2;  break;
    case '*':  v = v1*v2;  break;
    case '/':  v = v1/v2;  break;
 }

 /* empilha resultado */
 push(c->p,v);

 /* imprime topo da pilha */
 printf(c->f,v);
}
```

Por fim, a função para liberar a memória usada pela calculadora libera a pilha de operandos e a estrutura da calculadora.

```
void libera_calc (Calc* c)
{
   libera(c->p);
   free(c);
}
```

Um programa cliente que faça uso da calculadora é mostrado abaixo:

```
/* Programa para ler expressão e chamar funções da calculadora */

#include <stdio.h>
#include "calc.h"

int main (void)
{
 char c;
 float v;
 Calc* calc;

 /* cria calculadora com precisão de impressão de duas casas decimais
*/
 calc = cria_calc("%.2f\n");

 do {
  /* le proximo caractere nao branco */
  scanf(" %c",&c);
  /* verifica se e' operador valido */
  if (c=='+' || c=='-' || c=='*' || c=='/') {
     operador(calc,c);
  }
  /* devolve caractere lido e tenta ler número */
  else {
     ungetc(c,stdin);
     if (scanf("%f",&v) == 1)
        operando(calc,v);
```

```
    }
   } while (c!='q');
   libera_calc(calc);
   return 0;
  }
```

Esse programa cliente lê os dados fornecidos pelo usuário e opera a calculadora. Para tanto, o programa lê um caractere e verifica se é um operador válido. Em caso negativo, o programa "devolve" o caractere lido para o *buffer* de leitura, através da função `ungetc`, e tenta ler um operando. O usuário finaliza a execução do programa digitando `q`.

Se executado, e considerando-se as expressões digitadas pelo usuário mostradas abaixo, esse programa teria como saída:

```
3 5 8 * +              ← digitado pelo usuário
3.00
5.00
8.00
40.00
43.00
7 /                    ← digitado pelo usuário
7.00
6.14
q                      ← digitado pelo usuário
```

Exercício: Estenda a funcionalidade da calculadora incluindo novos operadores unários e binários (sugestão: `~` como menos unário, `#` como raiz quadrada, `^` como exponenciação).