AULA 19

PILHAS

Quando tratamos de listas lineares, o armazenamento seqüencial é usado, em geral, quando essas estruturas sofrem poucas modificações ao longo do tempo de execução em uma aplicação, com poucas inserções e remoções realizadas durante sua existência. Isso implica claramente em poucas movimentações de células. Por outro lado, a alocação encadeada é mais usada em situações inversas, ou seja, quando modificações nas listas lineares são mais freqüentes.

Caso os elementos a serem inseridos ou removidos de uma lista linear se localizem em posições especiais nessas estruturas, como por exemplo a primeira ou a última posição, então temos uma situação favorável para o uso de listas lineares em alocação seqüencial. Este é o caso da estrutura de dados denominada pilha. O que torna essa lista linear especial é a adoção de uma política bem definida de inserções e remoções, sempre em um dos extremos da lista. Além da alocação seqüencial, a implementação de uma pilha em alocação encadeada tem muitas aplicações importantes e também será discutida nesta aula.

Esta aula é baseada nas referências [2, 13].

19.1 Definição

Uma **pilha** é uma lista linear tal que as operações de inserção e remoção são realizadas em um único extremo dessa estrutura de dados.

O funcionamento dessa lista linear pode ser comparado a qualquer pilha de objetos que usamos com freqüência como, por exemplo, uma pilha de pratos de um restaurante. Em geral, os clientes do restaurante retiram pratos do início da pilha, isto é, do primeiro prato mais alto na pilha. Os funcionários colocam pratos limpos também nesse mesmo ponto da pilha. Seria estranho ter de movimentar pratos, por exemplo no meio ou no final da pilha, sempre que uma dessas dessas operações fosse realizada.

O indicador do extremo onde ocorrem as operações de inserção e remoção é chamado de **topo** da pilha. Essas duas operações são também chamadas de empilhamento e desempilhamento de elementos. Nenhuma outra operação é realizada sobre uma pilha, a não ser em casos específicos. Observe, por exemplo, que a operação de busca não foi mencionada e não faz parte do conjunto de operações básicas de uma pilha.

19.2 Operações básicas em alocação seqüencial

Uma pilha em alocação seqüencial é descrita através de duas variáveis: um vetor e um índice para o vetor. Supomos então que a pilha esteja armazenada no vetor `P[0..MAX-1]` e que a parte do vetor efetivamente ocupada pela pilha seja `P[0..t]`, onde `t` é o índice que define o topo da pilha. Os compartimentos desse vetor são convencionalmente do tipo `int`, mas podemos defini-los de qualquer tipo. Uma ilustração de uma pilha em alocação seqüencial é dada na figura 19.1.

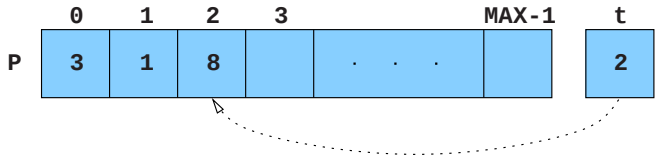
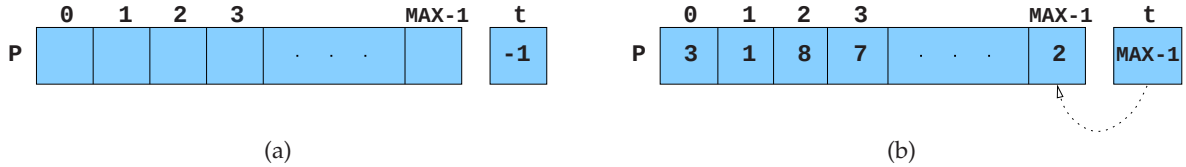


Figura 19.1: Representação de uma pilha `P` com topo `t` em alocação seqüencial.

Convencionamos que uma pilha está **vazia** se seu topo `t` vale -1 e **cheia** se seu topo `t` vale `MAX-1`. As representações de pilhas vazia e cheia são mostradas na figura 19.2.



(a) (b)

Figura 19.2: (a) Pilha vazia. (b) Pilha cheia.

A declaração de uma pilha e sua inicialização são mostradas abaixo:

```
int t, P[MAX];
t = -1;
```

Suponha agora que queremos inserir um elemento de valor 7 na pilha `P` da figura 19.1. Como resultado desta operação, esperamos que a pilha tenha a configuração mostrada na figura 19.3 após sua execução.

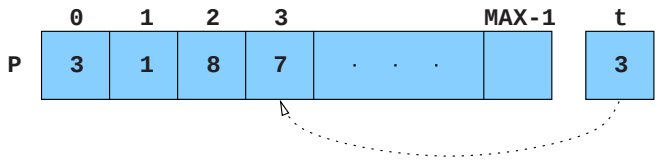


Figura 19.3: Inserção da chave 7 na pilha `P` da figura 19.1.

A operação de inserir um objeto em uma pilha, ou empilhar, é descrita na função `empilha_seq` a seguir.

```
/* Recebe um ponteiro t para o topo de uma pilha P
   e um elemento y e insere y no topo da pilha P */
void empilha_seq(int *t, int P[MAX], int y)
{
   if (*t != MAX - 1) {
      (*t)++;
      P[*t] = y;
   }
   else
      printf("Pilha cheia!\n");
}
```

Suponha agora que queremos remover um elemento de uma pilha. Observe que, assim como na inserção, a remoção também deve ser feita em um dos extremos da pilha, isto é, no topo da pilha. Assim, a remoção de um elemento da pilha `P` que tem sua configuração ilustrada na figura 19.1 tem como resultado a pilha com a configuração mostrada na figura 19.4 após a sua execução.

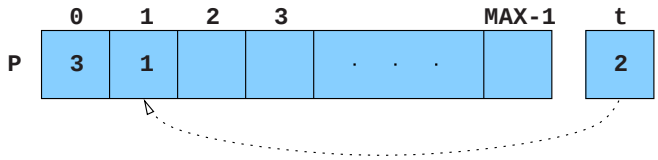


Figura 19.4: Remoção de um elemento da pilha `P` da figura 19.1.

A operação de remover, ou desempilhar, um objeto de uma pilha é descrita na função `desempilha_seq` a seguir.

```
/* Recebe um ponteiro t para o topo de uma pilha
   P e remove um elemento do topo da pilha P */
int desempilha_seq(int *t, int P[MAX])
{
   int r;

   if (*t != -1) {
      r = P[*t];
      (*t)--;
      } else {
         r = INT_MIN;
         printf("Pilha vazia!\n");
      }
      return r;

   }
```

Diversas são as aplicações que necessitam de uma ou mais pilhas nas suas soluções. Dentre elas, podemos citar a conversão de notação de expressões aritméticas (notação *prefixa*, *infixa* e *posfixa*), a implementação da pilha de execução de um programa no sistema operacional, a análise léxica de um programa realizada pelo compilador, etc. Algumas dessas aplicações serão vistas nos exercícios desta aula.

Como exemplo, suponha que queremos saber se uma seqüência de caracteres é da forma aZb, onde a é uma seqüência de caracteres e b é o reverso da a. Observe que se a = ABCDEFG e b = GFEDCBA então a cadeia aZb satisfaz a propriedade. Suponh ainda que as seqüências a e b possuem a mesma quantidade de caracteres. A função **aZb_seq** abaixo verifica essa propriedade para uma seqüência de caracteres usando uma pilha em alocação seqüencial.

```
/* Recebe, via entrada padrão, uma seqüência de caracteres e ve-
   rifica se a mesma é da forma aZb, onde b é o reverso de a */
int aZb_seq(void)
{
   char c, P[MAX];
   int t;

   t = -1;
   c = getchar();
   while (c != 'Z') {
      t++;
      P[t] = c;
      c = getchar();
   }
   c = getchar();
   while (t != -1 && c == P[t]) {
      t--;
      c = getchar();
   }
   if (t == -1)
      return 1;
   while (t > -1) {
      c = getchar();
      t--;
   }
   return 0;
}
```

19.3 Operações básicas em alocação encadeada

Existem muitas aplicações práticas onde as pilhas implementadas em alocação encadeada são importantes e bastante usadas. Consideramos que as células da pilha são do tipo abaixo:

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

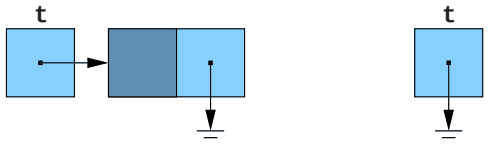
Uma pilha vazia com cabeça pode ser criada da seguinte forma:

```
   celula *t;
   t = (celula *) malloc(sizeof (celula));
   t->prox = NULL;
```

Uma pilha vazia sem cabeça pode ser criada da seguinte forma:

```
celula *t;
t = NULL;
```

Uma ilustração de uma pilha em alocação encadeada vazia é mostrada na figura 19.5.



(a) (b)

Figura 19.5: Pilha vazia em alocação encadeada (a) com cabeça e (b) sem cabeça.

Considere uma pilha em alocação encadeada com cabeça. A operação de empilhar uma nova chave `y` nessa pilha é descrita na função `empilha_enc_C` a seguir.

```
/* Recebe um elemento y e uma pilha t e insere y no topo de t */
void empilha_enc_C(int y, celula *t)
{
   celula *nova;

   nova = (celula *) malloc(sizeof (celula));
   nova->conteudo = y;
   nova->prox = t->prox;
   t->prox = nova;
}
```

A operação de desempilhar uma célula da pilha é descrita na função `desempilha_enc_C`.

```
/* Recebe uma pilha t e remove um elemento de seu topo */
int desempilha_enc_C(celula *t)
{
   int x;
   celula *p;

   if (t->prox != NULL) {
      p = t->prox;
      x = p->conteudo;
      t->prox = p->prox;
      free(p);
      return x;
   }
   else {
      printf("Pilha vazia!\n");
      return INT_MIN;
   }
}
```

A função `aZb_enc` é a versão que usa uma pilha em alocação encadeada para solucionar o mesmo problema que a função `aZb_seq` soluciona usando uma pilha em alocação seqüencial.

```
/* Recebe, via entrada padrão, uma seqüência de caracteres e ve-
   rifica se a mesma é da forma aZb, onde b é o reverso de a */
int aZb_enc(void)
{
   char c;
   celula *t, *p;

   t = (celula *) malloc(sizeof (celula));
   t->prox = NULL;
   c = getchar();
   while (c != 'Z') {
      p = (celula *) malloc(sizeof (celula));
      p->chave = c;
      p->prox = t->prox;
      t->prox = p;
      c = getchar();
   }
   c = getchar();
   while (t->prox != NULL && c == t->chave) {
      p = t->prox;
      t->prox = p->prox;
      free(p);
      c = getchar();
   }
   if (t->prox == NULL)
      return 1;
   while (t->prox != NULL) {
      p = t->prox;
      t->prox = p->prox;
      free(p);
      c = getchar();
   }
   return 0;
}
```

Exercícios

19.1 Considere uma pilha em alocação encadeada sem cabeça. Escreva as funções para empilhar um elemento na pilha e desempilhar um elemento da pilha.

19.2 Uma palavra é um **palíndromo** se a seqüência de caracteres que a constitui é a mesma quer seja lida da esquerda para a direita ou da direita para a esquerda. Por exemplo, as palavras RADAR e MIRIM são palíndromos. Escreva um programa eficiente para reconhecer se uma dada palavra é palíndromo.

19.3 Um estacionamento possui um único corredor que permite dispor 10 carros. Existe somente uma única entrada/saída do estacionamento em um dos extremos do corredor. Se um cliente quer retirar um carro que não está próximo à saída, todos os carros impedindo sua passagem são retirados, o cliente retira seu carro e os outros carros são recolocados na mesma ordem que estavam originalmente. Escreva um algoritmo que processa o fluxo

de chegada/saída deste estacionamento. Cada entrada para o algoritmo contém uma letra `E` para entrada ou `S` para saída, e o número da placa do carro. Considere que os carros chegam e saem pela ordem especificada na entrada. O algoritmo deve imprimir uma mensagem sempre que um carro chega ou sai. Quando um carro chega, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não existe vaga, o carro não entra no estacionamento e vai embora. Quando um carro sai do estacionamento, a mensagem deve incluir o número de vezes que o carro foi movimentado para fora da garagem, para permitir que outros carros pudessem sair.

19.4 Considere o problema de decidir se uma dada seqüência de parênteses e chaves é bem-formada. Por exemplo, a seqüência abaixo:

```
( ( ) { ( ) } )
```

é bem-formada, enquanto que a seqüência

```
( { ) }
```

é malformada.

Suponha que a seqüência de parênteses e chaves está armazenada em uma cadeia de caracteres `s`. Escreva uma função `bem_formada` que receba a cadeia de caracteres `s` e devolva `1` se `s` contém uma seqüência bem-formada de parênteses e chaves e devolva `0` se a seqüência está malformada.

19.5 Como mencionado na seção 19.2, as expressões aritméticas podem ser representadas usando notações diferentes. Costumeiramente, trabalhamos com expressões aritméticas em notação *infixa*, isto é, na notação em que os operadores binários aparecem entre os operandos. Outra notação freqüentemente usada em compiladores é a notação *posfixa*, aquela em que os operadores aparecem depois dos operandos. Essa notação é mais econômica por dispensar o uso de parênteses para alteração da prioridade das operações. Exemplos de expressões nas duas notações são apresentados abaixo.

notação *infixa*	notação *posfixa*
`(A + B * C)`	`A B C * +`
`(A * (B + C) / D + E)`	`A B C + * D / E -`
`(A + B * C / D * E - F)`	`A B C * D / E * + F -`

Escreva uma função que receba um cadeia de caracteres contendo uma expressão aritmética em notação *infixa* e devolva uma cadeia de caracteres contendo a mesma expressão aritmética em notação *posfixa*. Considere que a cadeia de caracteres da expressão *infixa* contém apenas letras, parênteses e os símbolos `+`, `-`, `*` e `/`. Considere também que cada variável tem apenas uma letra e que a cadeia de caracteres *infixa* sempre está envolvida por um par de parênteses.

19.6 Suponha que exista um único vetor M de céluas de um tipo pilha pré-definido, com um total de `MAX` posições. Este vetor fará o papel da memória do computador. Este vetor M será compartilhado por duas pilhas em alocação seqüencial. Implemente eficientemente as operações de empilhamento e desempilhamento para as duas pilhas de modo que nenhuma das pilhas estoure sua capacidade de armazenamento, a menos que o total de elementos em ambas as pilhas seja `MAX`.