Alocação Encadeada

Como vimos, uma fila nada mais é do que uma lista com uma disciplina de acesso. Logo, podemos nos utilizar de todos os conceitos vistos em listas para implementarmos filas. Por exemplo, podemos utilizar uma lista circular para armazenar uma fila, como exercício de fixação, implemente um TAD com este conceito.

Pilhas

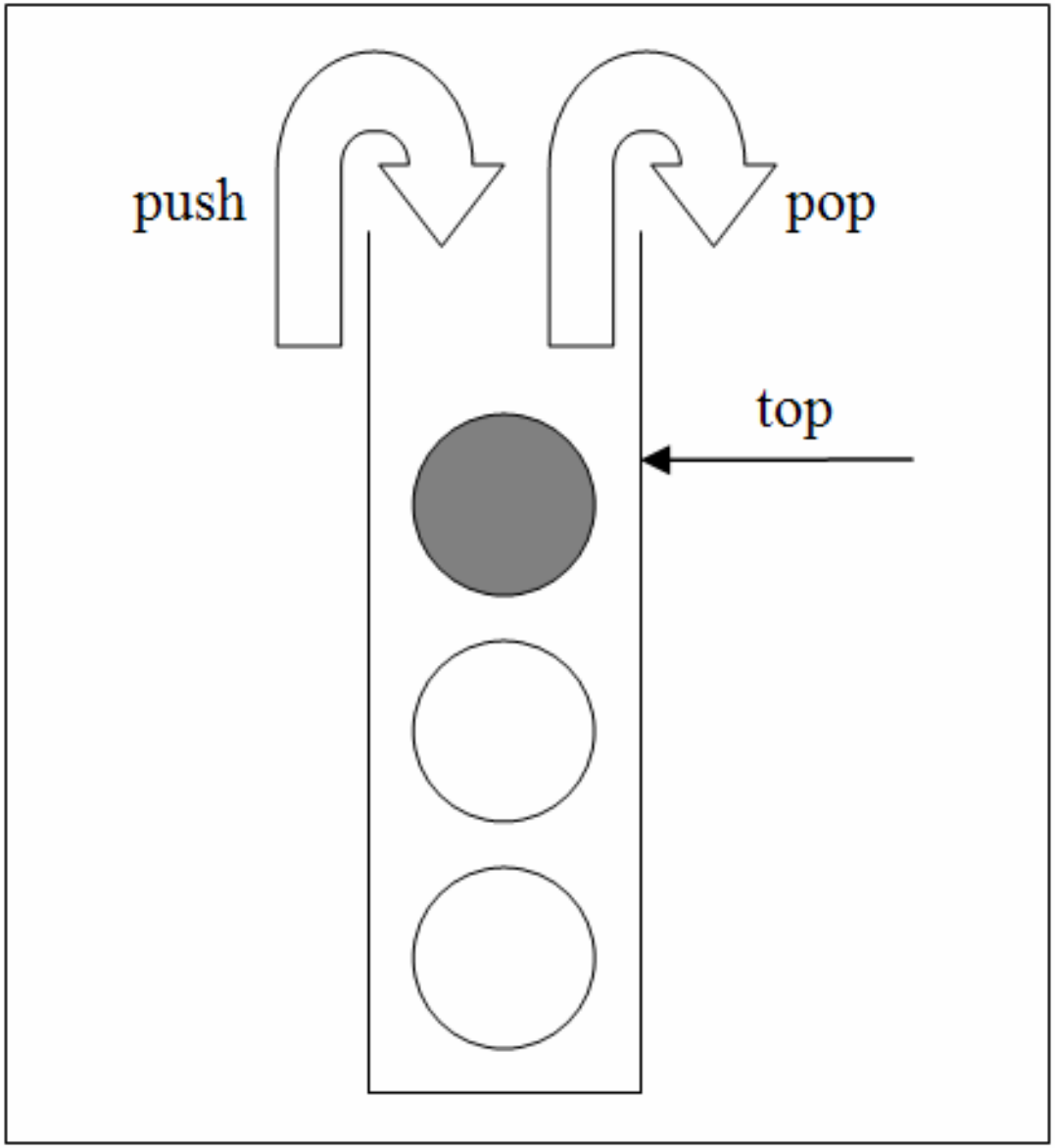
Caracterização

Uma pilha é uma lista com restrições de acesso, onde todas as operações só podem ser aplicadas sobre uma das extremidades da lista, denominada topo da pilha.

Com isso estabelece-se o critério LIFO (last in, first out), que indica que o último item que entra é o primeiro a sair.

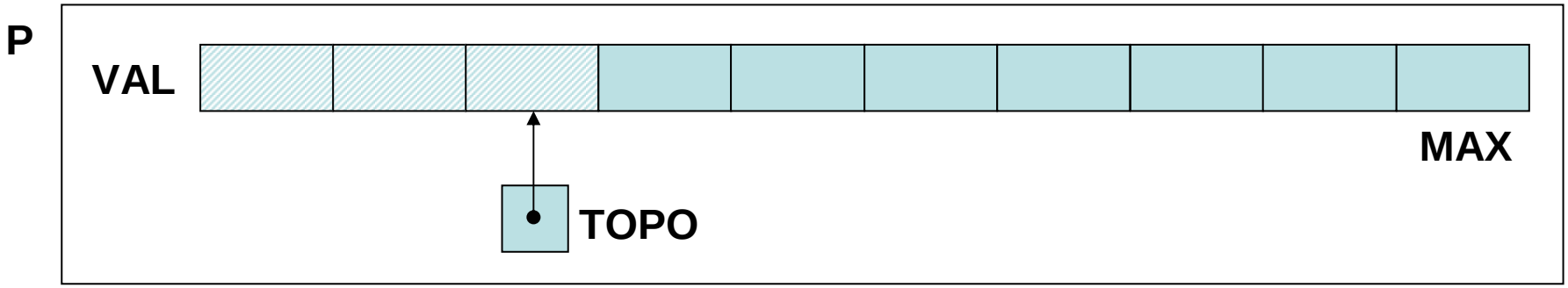
O modelo intuitivo para isto é o de uma pilha de pratos, ou livros, etc, na qual só se pode visualizar (consultar) o último empilhado e este é o único que pode ser retirado. E também qualquer novo empilhamento (inserção) se fará sobre o último da pilha.

Caracterização



Alocação Seqüencial

Uma forma de se implementar uma pilha é armazená-la num vetor VAL de MAX valores associado com um cursor inteiro TOPO que indica onde está o topo da pilha, como se vê abaixo:



A Implementação das operações é trivial: a pilha cresce do começo para o fim do vetor; uma pilha vazia tem o

Alocação Seqüencial

cursor TOPO igual a -1; a cada inserção, o cursor topo é incrementado para apontar para a próxima posição livre do vetor, onde é armazenado o novo elemento; uma retirada decrementa o cursor; uma consulta devolve o valor do elemento indexado por TOPO.

De pose destas informações definiremos e implementaremos agora o TAD PILHA_SEQ:

```
typedef struct
{
   int TOPO; /*índice do topo da pilha*/
   int VAL[MAX];    /*vetor de elementos*/
}PILHA_SEQ;
void cria_pilha (PILHA_SEQ *);
int eh_vazia (PILHA_SEQ *);
void push (PILHA_SEQ *, int);
int top (PILHA_SEQ *);
void pop (PILHA_SEQ *);
int top_pop (PILHA_SEQ *);
```

```
void cria_pilha (PILHA_SEQ *p)
{
   p->TOPO = -1;
}

int eh_vazia (PILHA_SEQ *p)
{
   return (p->TOPO == -1);
}
```

```
void push (PILHA_SEQ *p, int v)
{
   if (p->TOPO == MAX-1)
   {
      printf ("\nERRO! Estouro na pilha.\n");
      exit (1);
   }
   p->VAL[++(p->TOPO)]=v;
}
```

```
int top (PILHA_SEQ *p)
{
   if (eh_vazia(p))
   {
      printf ("\nERRO! Consulta na pilha vazia.\n");
      exit (2);
   }
   else
      return (p->VAL[p->TOPO]);
}
```

```
void pop (PILHA_SEQ *p)
{
   if (eh_vazia(p))
   {
      printf ("\nERRO! Retirada na pilha vazia.\n");
      exit (3);
   }
   else
      p->TOPO--;
}
```

```
int top_pop (PILHA_SEQ *p)
{
   if (eh_vazia(p))
   {
      printf ("\nERRO! Consulta e retirada na pilha
   vazia.\n");
      exit (4);
   }
   else
      return (p->VAL[p->TOPO--]);
}
```

Alocação Seqüencial - Exercício

Utilizando-se dos TAD´s FILA_SEQ e PILHA_SEQ, vistos anteriormente, implemente a seguinte operação:

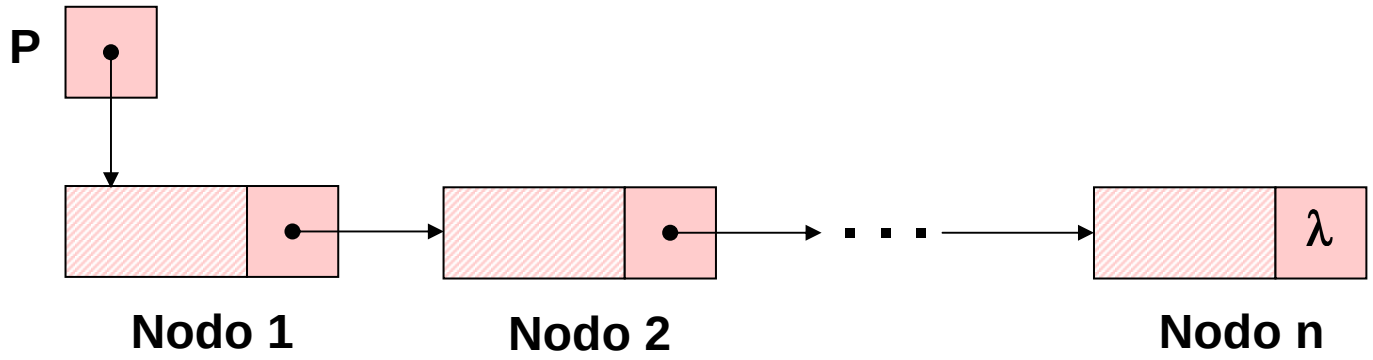
void inverte_fila (FILA_SEQ *f);

a qual recebe uma referência para uma fila seqüencial de inteiros e inverte a ordem de seus elementos, utilizando-se para isto de uma pilha seqüencial.

Alocação Encadeada

Com já discutimos, a alocação seqüência apresenta algumas desvantagens. Em virtude disso, podemos nós utilizar de uma lista encadeada para armazenarmos uma pilha, assim como fizemos com as filas. Como todas as operações ocorrem numa das extremidades da lista, a representação da pilha se reduz a um único ponteiro para o primeiro nodo da lista.

Alocação Encadeada



A implementação das operações é trivial. Para fazer uma inserção, basta alocar um nodo para o novo valor, ligá-lo ao primeiro nodo da lista e fazer o ponteiro apontar para o novo nodo. Uma retirada exige apenas que o ponteiro passe a apontar para o segundo nodo da lista (ou ser anulado, se houver

Alocação Encadeada

apenas um nodo). Uma consulta exige apenas a recuperação do valor do primeiro nodo. OBS. : em uma retirada o espaço de memória ocupado pelo primeiro nodo deve ser liberado.

Desta forma, definiremos e implementaremos, agora, o TAD PILHA_ENC (de valores inteiros).

```
typedef struct nodo
{
   int inf;
   struct nodo * next;
}NODO;
typedef NODO * PILHA_ENC;
void cria_pilha (PILHA_ENC *);
int eh_vazia (PILHA_ENC);
void push (PILHA_ENC *, int);
int top (PILHA_ENC);
void pop (PILHA_ENC *);
int top_pop (PILHA_ENC *);
```

```
void cria_pilha (PILHA_ENC *pp)
{
   *pp=NULL;
}

int eh_vazia (PILHA_ENC p)
{
   return (!p);
}
```

```
void push (PILHA_ENC *pp, int v)
{
   NODO *novo;
   novo = (NODO *) malloc (sizeof(NODO));
   if (!novo)
   {
      printf ("\nERRO! Memoria insuficiente!\n");
      exit (1);
   }
   novo->inf = v;
   novo->next = *pp;
   *pp=novo;
}
```

```
int top (PILHA_ENC p)
{
   if (eh_vazia(p))
   {
      printf ("\nERRO! Consulta em pilha vazia!\n");
      exit (2);
   }
   else
      return (p->inf);
}
```

```
void pop (PILHA_ENC *pp)
{
   if (eh_vazia(*pp))
   {
      printf ("\nERRO! Retirada em pilha vazia!\n");
      exit (3);
   }
   else
   {
      NODO *aux=*pp;
      *pp=(*pp)->next;
      free (aux);
   }
}
```

```
int top_pop (PILHA_ENC *pp)
{
   if (eh_vazia(*pp))
   {
      printf ("\nERRO! Consulta e retirada em pilha vazia!\n");
      exit (4);
   }
   else
   {
      int v=(*pp)->inf;
      NODO *aux=*pp;
      *pp=(*pp)->next;
      free (aux);
      return (v);
   } }
```

Notações: in, pré e posfixada

Examinaremos agora uma importante aplicação que ilustra os diferentes tipos de pilhas e as diversas operações definidas a partir delas. O exemplo é, em si mesmo, um relevante tópico da computação.

Considerando a soma de A mais B. Imaginamos a aplicação do **operador** "+" sobre os **operandos** A e B, e escrevemos a soma como A + B. Essa representação particular é chamada infixada. Existem duas notações alternativas para expressar a soma de A e B usando os símbolos A, B e +.

UNIVASF

Notações: in, pré e posfixada

São elas:

+ A B prefixada

A B + posfixada

Analisando expressões infixadas um pouco mais complexas, como, por exemplo, A + B * C. Notamos a necessidade da definição de precedência entre os operadores (em casos em que é preciso alterar a ordem de precedência pré estabelecida se utilizam parênteses) visando eliminar a ambigüidade, tornando a tarefa computacional menos simples.

Notações: in, pré e posfixada

Notamos que a representação pré e posfixada para expressões aritméticas é mais conveniente do ponto de vista computacional.

Para ilustrarmos os diferentes tipos de representações, utilizaremos em nosso exemplos cinco operações binárias: adição, subtração, multiplicação, divisão e exponenciação.

Vamos fazer algumas conversões da forma infixada para a prefixada e posfixada.

Notações: in, pré e posfixada

Forma Infixada	Forma Prefixada
A + B - C	- + ABC
(A + B) * (C - D)	* + AB - CD
A ^ B * C - D + E / F / (G + H)	+ - * ^ ABCD / / EF + GH
((A + B) * C - (D - E)) ^ (F + G)	^ -* + ABC - DE + FG
A - B / (C * D ^ E)	- A / B * C ^ DE

Notações: in, pré e posfixada

Forma Infixada	Forma Posfixada
A + B - C	AB + C -
(A + B) * (C - D)	AB + CD - *
A ^ B * C - D + E / F / (G + H)	AB ^ C * D - EF / GH + / +
((A + B) * C - (D - E)) ^ (F + G)	AB + C * DE - - FG + ^
A - B / (C * D ^ E)	ABCDE ^ * / -