# 1. A Pilha

## 1.1 Introdução

É bastante natural conceber uma estrutura de dados como um dispositivo, o qual, de forma semelhante ao ATM visto na , tem funcionalidade disponibilizada em uma interface e cuja implementação e estado interno é oculta ao usuário. Uma pilha, por exemplo, poderia ser vista como um dispositivo como o exibido na. Tomando esta proposição como verdadeira e adotando o modelo de implementação anteriormente descrito, será possível construir estruturas de dados na forma de dispositivos lógicos do tipo TDA (Figura 2.1).

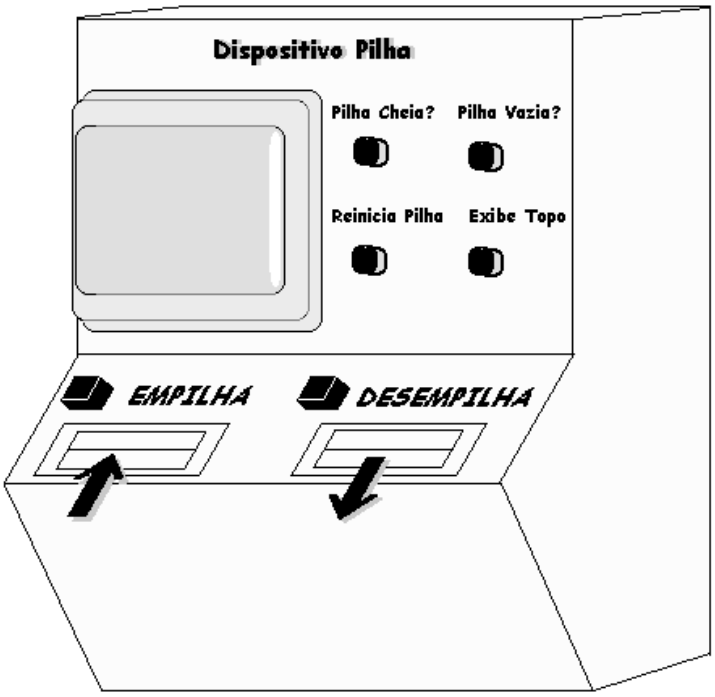


**Figura 1.1-A pilha como dispositivo acionado através de uma interface.**

Uma pilha pode ser operada − sofrer inserções, remoções, ou buscas − apenas pelo seu *topo*, assim, dentre as estruturas de dados aqui analisadas, a pilha será a mais simples. Supondo a necessidade de acessar um elemento empilhado, que não seja aquele no *topo*, nesse caso seria necessário desempilhar o elemento que ocupa o *topo* da pilha e, caso existam, todos os demais elementos entre o *topo* e o item procurado. A pilha, portanto, é uma estrutura de acesso seqüencial.

Apesar de sua simplicidade a pilha é uma estrutura extremamente útil, especialmente quando uma determinada seqüência de eventos deve ser tratada na ordem inversa das suas

ocorrências. Por conta dessas características as pilhas são conhecidas como estruturas LIFO - *Last In First Out*, ou seja, o último item que entra é o primeiro que sai.

Em outras palavras: o mecanismo da pilha funciona como se cada item inserido tivesse um atributo especial que registrasse o instante de cada inserção, segundo um relógio global. A partir desse atributo seria possível determinar uma relação cronológica entre os itens inseridos, de forma a dar-se prioridade de remoção (ou acesso) àquele que, segundo o valor do atributo hipotético, tivesse menor tempo de estadia dentro da pilha (Figura 2.2).

Neste capítulo se inicia o estudo das Estruturas de Dados como Tipos de Dados Abstratos, porém será necessária a apresentação de mais alguns conceitos importantes, tais como o conceito de implementação estática e o conceito de implementação dinâmica. Vencida esta etapa será apresentada a implementação da pilha ou aglomerados de pilhas como um TDA.

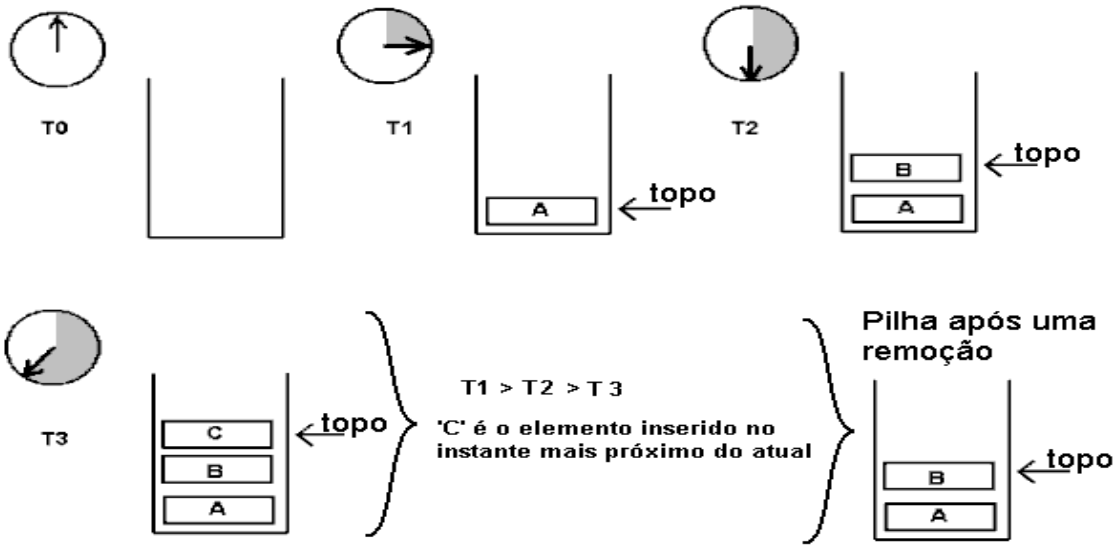


**Figura 1.2- O mecanismos de pilha – LIFO.**

## 1.2 Implementação Estática ou Dinâmica?

As estruturas de dados são implementadas sobre regiões de memória e tais regiões podem ser delimitadas de forma estática ou não estática (dinâmica). Neste texto, considera-se como implementação estática àquela em que um Tipo de Dados Abstrato é criado para se desenvolver dentro dos limites de um *container* (normalmente um vetor), sendo que o tamanho máximo desse *container*, será um dos parâmetros para a criação do TDA. Portanto, durante este texto, o termo "implementação estática" se referirá àquela onde o tamanho máximo do TDA implementado, permanecerá imutável (estático), enquanto o mesmo existir na memória do computador. Na implementação não estática ou dinâmica, por outro lado, não há um vetor construído para dar suporte ao Tipo de Dados Abstrato, nesse caso, o TDA se desenvolverá em função de inserções (alocando memória) ou remoções (liberando memória), porém, sem a limitação de tamanho, pré-definida como parâmetro para a sua criação.

Cabem aqui, as seguintes observações:

- Independente da estratégia adotada, a disponibilidade de memória no sistema é um limitador à implementação.
- Aqui não será tratada a possibilidade de remanejar o tamanho do TDA-estático através da função *realloc*. Para fins de criação dos TDA's será utilizada preferencialmente a função *malloc*, apesar da existência da função alternativa *calloc*.
- No decorrer deste estudo, independentemente do modelo utilizado vir a ser dinâmico ou estático, um TDA será sempre criado através da manipulação de uma região de memória conhecida como *heap* (região de memória disponível para alocação através de funções de alocação, tais como *calloc* e *malloc*).
- É importante frisar que, devido à ausência de um *garbage collector* (coletor de lixo: tecnicamente, um programa que libera automaticamente as regiões de memória que estão em desuso), toda a memória alocada pelo TDA ou até mesmo o módulo de aplicação, deve ser explicitamente liberada (*freed*) quando em desuso, devolvendo-se esta área ao gerenciador de memória do sistema operacional.

### 1.2.1 Arquitetura de Arquivos

A partir do esquema apresentado na Figura 1.3, é possível implementar um sistema baseado no modelo de abstração apresentado no capítulo anterior. Por este esquema, os arquivos "*APLICA.**" referem-se à aplicação da pilha na solução de algum problema.

No arquivo de interface (painel do dispositivo TDA) define-se basicamente a prototipagem das operações públicas (funções) do TDA, não havendo a revelação das suas implementações nem da estrutura interna do TDA. Esta estrutura interna é privativa ao TDA e está definida em um arquivo que não é incluído no módulo de aplicação (arquivo "privativo" do TDA), portanto, em termos de programação, a estrutura interna do TDA está oculta, pois não consta na interface do mesmo. Fato semelhante ocorre com a implementação das operações do TDA. Tem-se então o conceito de uma cápsula, com a ocultação de dados, implementação e a definição de uma interface.

Este esquema/modelo será ampliado mais adiante quando, além da necessidade de ocultar a estrutura interna privativa do TDA, surgirem certas operações privativas que também necessitarão de ocultação em relação ao mundo exterior ao TDA.

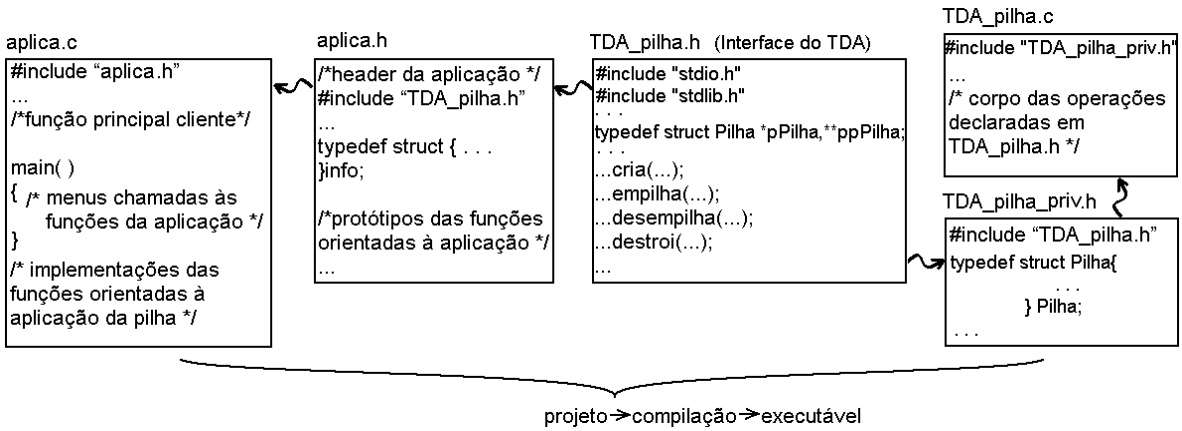


**Figura 1.3 - Arquitetura de arquivos em *C* para um sistema baseado na abstração de dados.**

### 1.2.1.1 Implementando a Pilha por Alocação Estática - PE

## Arquivo TDA_PE.H - Interface com o mundo

Neste arquivo são definidos vários rótulos (macros[1]) de utilidade geral para o sistema em implementação, bem como, são incluídas várias bibliotecas de uso comum, tais como *stdio, stdlib*, etc.... Além disto, é neste arquivo que é feita a prototipagem das operações e suas respectivas pré e pós-condições (uma espécie de "contrato de prestação de serviço" do TDA).

Perceba, na linha em destaque no fragmento de código exibido abaixo, que a estrutura *descPE* não está totalmente especificada, isso será feito em outro arquivo, mais especificamente no TDA_PE_PRIV.H, já que a descrição de *descPE* é privativa ao TDA.

```
...
#define FRACASSO 0
#define SUCESSO 1
#define SIM 1
#define NAO 0
#define VAZIA -1

typedef struct descPE PE;
...

/*protótipos das operações (funções) acompanhadas de suas pré e pós-condições */
```

## Arquivo TDA_PE.H – Continuação

Continuando a descrição do arquivo TDA_PE.H, agora serão exibidos os protótipos e as respectivas descrições das operações de interação com o TDA. Para melhor organizar a codificação aplicou-se uma certa padronização nos protótipos e uma descrição textual simples para as pré e pós-condições. Quanto ao corpo das funções, fez-se um foco maior na funcionalidade básica das operações. Os usuários do TDA deverão seguir a risca as recomendações das pré-condições, buscando evitar certos casos de exceções através dos avisos constantes nas pré-condições. Aqui se optou por um tratamento relativamente pobre para exceções, ficando como exercício o enriquecimento deste tratamento.

```
 /*Operação de criação
Pré-condição: existência de um apontador adequadamente declarado (tipo TDA) no escopo do módulo de
aplicação e passado por referência para a função de criação juntamente com a capacidade máxima (> 0) do
TDA e o tamanho (a quantidade de bytes) do tipo de informação a ser manipulada.
Pós-condição: retorna a macro FRACASSO caso tamInfo seja inválida ou ocorra erro de alocação de
memória , caso contrário retorna SUCESSO com a atribuição do endereço do TDA ao ponteiro passado por
referencia., */
int cria(PE **pp, int capMax, int tamInfo);
```

[1] Macro: palavra que identifica uma ou mais instruções e que em *C* são definidas com o auxílio do comando *define*.

/*Operação destruidora
Pré-condição: existência de um apontador adequadamente declarado (tipo TDA) no escopo do módulo de aplicação e passado por referência para a função de destruição.
Pós-condição: a liberação de qualquer região de memória utilizada pelo TDA, seguida da anulação do ponteiro passado por referência.*/
void destroi(PE **pp);

/*Operação de acesso
Pré-condição: a existência do TDA instanciado cujo endereço é passado para a função.
Pós-condição: se o TDA estiver vazio retorna a macro SIM, caso contrário retorna a macro NAO. */
int testaVazia(PE *p);

/*Operação de acesso
Pré-condição: a existência do TDA instanciado cujo endereço é passado para a função.
Pós-condição: se o TDA estiver cheio retorna a macro SIM, caso contrário retorna a macro NAO.*/
int testaCheia(PE *p);

/*Operação de manipulação
Pré-condição: pré-existência do TDA instanciado cujo endereço é passado para a função
Pós-condição: esvaziamento dos dados, a aplicação de *testaVazia* resultará em SIM.*/
void reinicia(PE *p);

/*Operação de manipulação
Pré-condição: pré-existência do TDA instanciado cujo endereço é passado para a função juntamente com uma referência para o novo registro de informação a ser empilhado. O novo item a ser inserido deve ter exatamente o tamanho da informação para a qual a pilha foi parametrizada na criação.
Pós-condição: se houver espaço na pilha será feita inserção no topo e em seguida o retorno de SUCESSO. Caso contrário retornar-se-á a macro FRACASSO.*/
int empilha(PE *p, void * novo);

/*Operação de manipulação
Pré-condição: pré-existência do TDA cujo endereço é passado para a função.
Pós-condição: se a pilha não estiver vazia será removido o item no seu topo e em seguida o retorno de SUCESSO, caso contrário retornar-se-á FRACASSO.*/
int desempilha(PE *p);

/*Operação de acesso
Pré-condição: pré-existência do TDA instanciado cujo endereço é passado para a função juntamente com o endereço de um destino para a informação a ser buscada. A área de destino deve ter sido previamente alocada e deve suportar exatamente o tamanho da informação a ser buscada.
Pós-condição: se a pilha estiver vazia retorna-se a macro FRACASSO. Caso contrário a informação buscada no topo da pilha é copiada para um destino na memória, em seguida retorna-se a macro SUCESSO.*/
int buscaNoTopo(PE *p, void *destino);

## TDA_PE_PRIV.H - Macros e Descritor - Estado interno do TDA

Uma das estruturas mais importantes do TDA_PE e que será utilizada em outros TDA's é o descritor. Um descritor é um elemento estrutural que contém uma referência para a área de dados do TDA e, normalmente, outras informações úteis ao seu gerenciamento. O descritor do TDA_PE corresponderá à estrutura vista abaixo.

```
#include "TDA_PE.H"

/* Descritor do TDA: */
struct descPE
{       void **vet;  /* ponteiro para o vetor de ponteiros */
        int topo;  /* topo da pilha */
        int tamVet;  /*tamanho do vetor/tamanho máximo da pilha */
        int tamInfo; /* tamanho do pacote de informação a ser guardada  */
};
```

## Arquivo TDA_PE.C – Codificação dos protótipos

As operações cujos protótipos foram definidos no arquivo TDA_PE.H devem ser implementadas no arquivo TDA_PE.C, partindo do pré suposto que as pré-condições acima descritas serão sempre garantidas. Perceba que esse TDA não apresenta operações privadas, ou seja, todas as operações a serem implementadas são de uso público – fazem parte da interface.

### A criação  da Pilha Estática

Consiste na instanciação do TDA na memória, neste caso da PE, utilizando uma estratégia estática. Tal instanciação constitui-se basicamente:

a) Na criação do descritor e iniciação dos seus campos (fique atento à iniciação do *topo* com um valor que não indexa o vetor, indicando pilha vazia);
b) Na criação de um vetor com os endereços das regiões que irão guardar elementos de dados (também conhecidos como *nós* ou *nodos*). Algum campo do descritor criado em *a* deverá referenciar este vetor;

c) Na criação das regiões de memória que serão referenciadas por cada célula do vetor criado em *b*.

```
#include “tda_pe.h”
int cria(PE **pp, int capMax, int tamInfo)
{
	int i=0, ret = SUCESSO;
	PE *desc = NULL;
	desc = (PE*) malloc( sizeof(PE) ); /* 1 */

	if( desc != NULL ) {
		desc->topo = -1;		/* 2 */
		desc->tamVet = capMax;	/* 3 */
		desc->tamInfo = tamInfo; /* 4 */
		desc->vet = (void**) malloc( capMax * sizeof(void*) );	/* 5 */

		if( desc->vet != NULL ) {
			for(i=0; i < capMax; i++) {
				desc->vet[i] = (void*) malloc( tamInfo ); /* 6 */

			if( desc->vet[i] == NULL ) { /* 7 */
				for(i=i-1; i >= 0; i--) {
					free(desc->vet[i]);
				}

				free(desc->vet);
				free(desc);
				desc = NULL;
				ret = FRACASSO;
				break;
				}
			}
		}
		else {	/* 8 */
			free(desc);
			desc = NULL;
			ret = FRACASSO;
		}
	}
	else {
		ret = FRACASSO;
	}

	*pp = desc; /* 9 */

	return ret;
}
/*Comentários:
1. Instanciação do descritor da pilha;
2. Iniciação do descritor: iniciando o topo com valor que não indexa o vetor (há uma macro que define VAZIA como -1) o que implica em pilha vazia;
3. Iniciação do descritor: armazenando o comprimento do vetor;
4. Iniciação do descritor: armazenando o tamanho dos registros de informação;
5. Iniciação do descritor: criação do vetor, recipiente da pilha.
6. Alocação de memória para iniciar os “slots” de dados (apontados pelos ponteiros void);
7. Atomicidade: uma alocação de “slot” falhou, deve-se desfazer todo o TDA;
8. Atomicidade: a alocação do vetor falhou; deve-se desfazer o TDA;
9. Sucesso na criação: após alocações e incializações bem-sucedidas, o endereço do descritor é “conectado” ao ponteiro passado por referência e retorna-se ao cliente */
```

### Visualização da Pilha Estática recém criada

É possível desenhar um modelo gráfico para o TDA_PE recém criado, um exemplo de ilustração pode ser visto na Figura 1.4 abaixo.

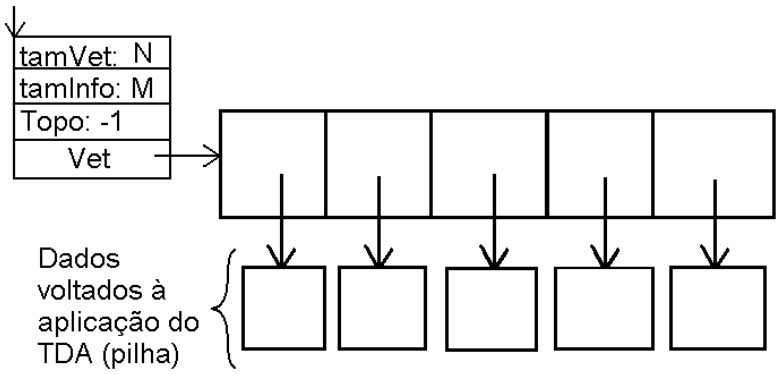


**Figura 1.4 - Pilha Estática recém criada.**

### A destruição da Pilha Estática

A operação de destruição de um TDA consiste em liberar qualquer região de memória (para dados ou descritor) que foram alocadas durante a criação do mesmo. Além disso deve-se anular o ponteiro passado por referência. Ao final desta operação, no que se refere à pilha, o sistema assume um estado igual àquele anterior à sua criação (da pilha).

```
void destroi(PE **pp)
{ int i;
  for(i = 0; i< (*pp)->tamVet; i++)  /*libera memória que inicializa os ponteiros void*/
          free( (*pp)->vet[i]);          /* alternativa:  free(*(p->vet + i));        */
 free((*pp)->vet);
 free(*pp);
 *pp = NULL;
}
```

### Utilização da indireção múltipla através de Ponteiro para Ponteiro

Nas duas operações acima descritas (*cria* e *destroi*) foi utilizada uma prototipagem constituída com ponteiros para ponteiros, para um melhor entendimento sobre o uso e função destes apontadores faz-se necessária uma discussão prévia sobre o conceito de parâmetro de um subprograma.

No protótipo de uma função em *C* pode constar uma lista com uma ou mais declarações de variáveis. Conceitualmente, esta lista define os chamados *parâmetros formais* da função. Ao ocorrer a chamada à função, os *parâmetros formais* são vinculados a uma lista de *parâmetros reais*. O resultado prático desse processo é a passagem de dados (ou computações) para a função. Neste texto, os *parâmetros formais* serão designados simplesmente como *parâmetros*. Os chamados *parâmetros reais*, por sua vez, serão designados como *argumentos*.

Outro conceito importante é o de *escopo*. O *escopo* de uma variável corresponde ao bloco de instruções no qual a variável é visível e diretamente acessível, podendo ser diretamente manipulada por uma instrução dentro deste bloco de código. Por sua vez, os chamados parâmetros de uma função, são de escopo local à respectiva função.

Considerando as definições propostas nos parágrafos anteriores, daremos seguimento à discussão do uso de apontadores, aqui proposto. Para tanto será feita uma simulação de execução da função de criação da pilha, cujo protótipo é: *int cria(PE **pp, int capMax, int tamInfo)* conforme ilustrado na 1.5. Na mesma figura, dentro da função *main*, *pode*-se observar a declaração *PE *p = NULL*, ou seja, o ponteiro *p* pertence ao escopo da função *main*, tratando-se, portanto, de uma variável dita invisível para qualquer função que não seja a *main*, ou seja, o apontador *p* está inacessível à função de criação da pilha.

Novamente na 1.5, no 3º ponto no código fonte, imediatamente após a chamada à função de criação, perceba que o ponteiro *p* encontra-se modificado e que isso ocorreu apenas pela interferência da função de criação, a qual, de alguma maneira indireta, conseguiu manipular este apontador. Da forma como foi definido o protótipo da função de criação, tal manipulação só é possível através do uso de ponteiro para ponteiro. Abaixo se analisa o papel do ponteiro para ponteiro *pp* nesse processo.

Ainda pela 1.5, no 2º passo são feitas todas as alocações e inicializações, se forem bem-sucedidas teremos a estrutura de dados pronta para uso, porém, em termos de endereço de memória, ainda "invisível" ao módulo cliente. Para estabelecer essa visibilidade se utiliza do ponteiro-para-ponteiro *pp*. Vê-se que o endereço do ponteiro *p* serve como parâmetro real para o parâmetro formal *pp*, de forma que o endereço de *p* é copiado na variável *pp*, a qual, pertence ao escopo local à função de criação e, por conta disso, pode ser diretamente acessada pelo corpo de instruções dessa função. A partir desse ponto (*pp* aponta para *p*) é possível através do ponteiro para ponteiro manipular o apontador *p*, bem como qualquer objeto por ele apontado. É o que ocorre no final do código de criação, onde o endereço do descritor da pilha é indiretamente atribuído a *p* através da sintaxe **pp*. Também é possível atribuir valores a um objeto apontado por *p* através da sintaxe ***pp*.

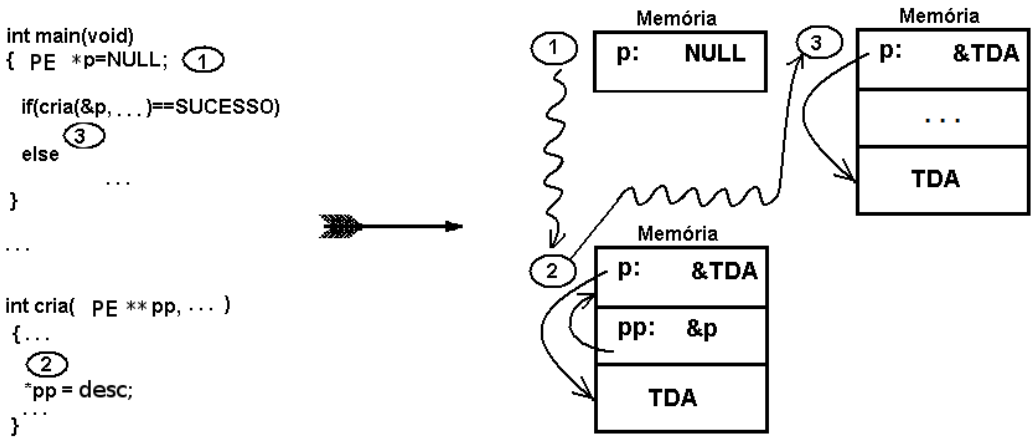


**Figura 1.5: O apontador "p" tem escopo local à função "main" enquanto "pp" tem escopo local à função de criação. No 4o passo, após a criação do TDA, "pp" fica inacessível, enquanto "p" apresenta-se devidamente iniciado com o endereço do TDA**

Esse tipo de passagem de parâmetros – passagem do endereço de um ponteiro por referência – só é possível através da utilização de ponteiro para ponteiro.

No caso da operação de destruição ocorre algo semelhante, porém no sentido contrário, liberando a memória alocada na criação e anulando o ponteiro *p*.

Abaixo segue um exemplo da declaração de variável ponteiro para referenciar a pilha bem como a sintaxe para sua criação e destruição, tais códigos fazem parte do arquivo APLICA.C e apesar de estarem sendo exibidos neste capítulo, basicamente a mesma estratégia será utilizada no desenvolvimento dos sistemas dos capítulos seguintes, quando da abordagem de outros TDA's.

```
#include "aplica.h"
...
main( )
{ PE *p=NULL;    /* declarando o ponteiro que referenciará o TDA */
. . .
  if(cria(&p, x, y) ==SUCESSO) /* criando o TDA*/
  {  ...
     destroi(&p);
     exit(1);
   }
  exit(0);
}
```

### Reiniciando a Pilha Estática

O processo de reinicializar a pilha (ou *reset*) corresponde a esvaziá-la de dados restabelecendo o seu estado inicial, imediatamente após a sua criação. No caso da PE, aqui proposta, a maneira mais simples de fazer isso é reiniciar o topo da pilha, atribuindo-lhe o valor *–1*. A partir daí qualquer informação na pilha passa a ser considerada "sujeira".

```
void reinicia(pPE p)
{ int i;
  p->topo = VAZIA;
}
```

### A operação de busca na Pilha Estática

Consiste apenas na busca do elemento que se encontra referenciado pela célula indexada pelo *topo* da pilha. Este elemento, caso exista, é copiado para uma região de destino cujo endereço é passado por referência (Figura 1.6).

```
int buscaNoTopo(PE *p, void *destino)
{ int ret;
  if (p->topo == VAZIA) /* acesso direto ao topo */
        ret = FRACASSO;
 else
 {      memcpy(destino, p->vet[p->topo],p->tamInfo);
        ret = SUCESSO;
 }
   return ret;
}
```

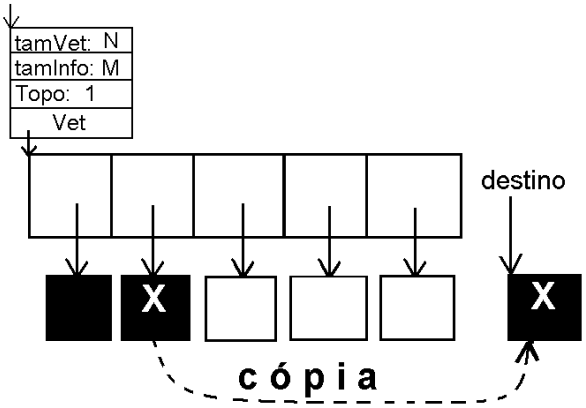


**Figura 1.6 - Operação de busca na PE.**

### A inserção na  Pilha Estática

O empilhamento na PE é quase uma operação inversa à *buscaNoTopo*, a diferença está no sentido do fluxo da cópia (Figura 1.7). Agora deve-se copiar uma quantidade de bytes para uma posição livre na pilha. Uma posição livre é obtida pelo incremento do campo *topo*. Se o vetor não possui posições na condição "livre", a operação fracassará.

```
int empilha(PE *p, void * novo)
{ if (p->topo < p->tamVet-1)
  {   p->topo++;
      /*cópia um bloco de tamanho definido deste o endereço novo até o endereço
           definido por p->vet[p->topo] */
      memcpy(p->vet[p->topo],novo,p->tamInfo);
      return SUCESSO;
   }
   else
       return FRACASSO;
}
```

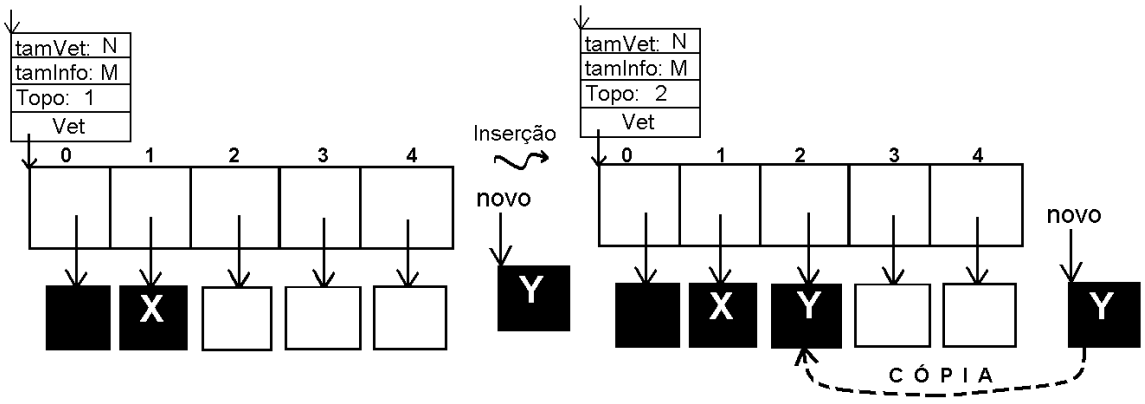


**Figura 1.7 - Inserindo na P.E .**

### A remoção (desempilhamento) da Pilha Estática

Caso a pilha não esteja vazia, esta operação consistirá apenas em um decremento do campo *topo* (Figura 1.8). Pelo conceito de transação atômica (cada operação do TDA executando uma única tarefa) isso já seria é suficiente, porém, há autores que relaxam a operação *desempilha* permitindo que o seu chamador também tenha acesso ao elemento que foi removido (uma combinação da remoção e busca).

```
int desempilha(PE *p)
{ if (p->topo == VAZIA)
      return FRACASSO;
  else
  {  p->topo--;
     return SUCESSO;
   }
}
```

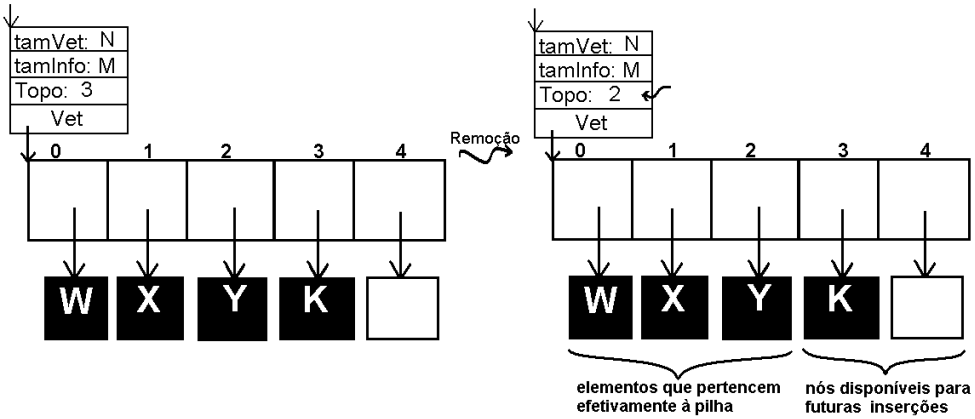


**Figura 1.8 -Removendo da PE.**

### Verificando o estado da Pilha Estática

As operações de teste de pilha vazia ou cheia são triviais. A primeira apenas testa o campo *topo* verificando se o mesmo vale *–1* (pilha vazia), enquanto a segunda verifica se o campo *topo* já alcançou a última célula do vetor (pilha cheia).

```
int testaVazia(PE *p)
{  int ret;
   if (p->topo == VAZIA) /* VAZIA deve ser definido como -1  */
     ret = SIM;
   else
     ret = NAO;
   return ret;
}

int testaCheia(PE *p) {
  int ret;
  if (p->topo >= p->tamVet-1)
    ret = SIM;
  else
    ret = NAO;
return ret;
}
```

## 1.2.1.2 Pilha por Alocação Dinâmica – PD

Uma pilha implementada por alocação dinâmica é aquela em que é a alocação/liberação de memória ocorre respectivamente a cada inserção/remoção de um item de dados. Portanto, a pilha dinâmica diferentemente da estática, não tem o seu tamanho máximo definido no momento da sua criação, podendo armazenar tantos elementos quantos o sistema operacional permitir. Em uma PD os *nós* de dados não fazem parte de um vetor e por conta disso, cada item de dados em uma Pilha Dinâmica deve possuir elo(s) de ligação para o(s) vizinho(s) de forma a encadear todos os itens inseridos. Tais *nós* são tão importantes que designam os dois tipos básicos de Pilhas Dinâmicas: a Simplesmente Encadeada PDSE e a Duplamente Encadeada PDDE, conforme Figura 1.9 abaixo. Esses conceitos serão aplicados em outros TDA's dinamicamente construídos.

A PDDE não é particularmente vantajosa. Esta estrutura será abordada nesse momento com a finalidade precípua de familiarizar o leitor com encadeamento duplo, pois esse tipo de manipulação será bastante utilizada em outras estruturas (TDAs) de manipulação mais complexa. Por ser a estrutura mais simples, a PDDE permite uma introdução ao duplo encadeamento sem a complexidade encontrada, por exemplo, na manipulação da fila de prioridade ou da lista encadeada.

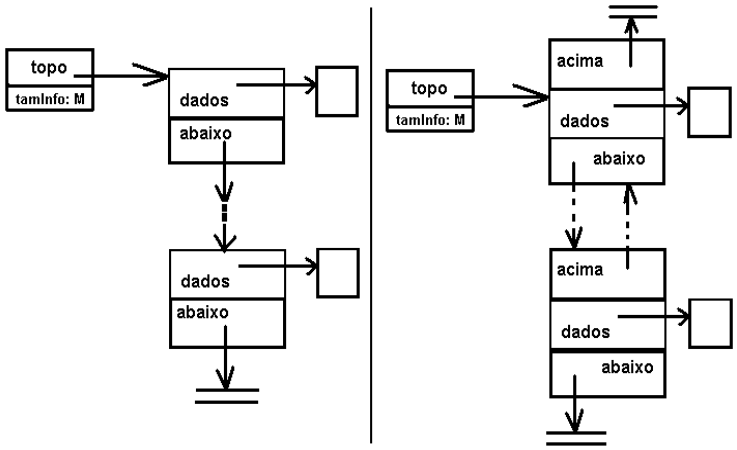


**Figura 1.9 -(esq) PDSE e (dir) PDDE.**

## Implementando a PDSE

A Seguir descreve-se a implementação das operações básicas para uma PDSE considerando-se as pré e pós-condições adequadas (as suas descrições ficam como exercício). Para a implementação de uma PDDE a partir da PDSE, deve-se alterar, entre outros, o descritor e o nó que referencia os dados da pilha, além disso, deve-se modificar as operações que manipulam os ponteiros de ligação dos *nós* de dados (operações: empilha e desempilha), é também possível tirar proveito do duplo encadeamento na nova implementação das operações de reiniciação e destruição.

Arquivo TDA_PDSE.H

```
typedef struct pdse PDSE;

/* operações da interface: */
...
```

Arquivo TDA_PDSE_PRIV.H

```
#include "PDSE.h"

typedef struct noPDSE {
        void *dados;
        struct noPDSE *abaixo;
} NoPDSE;

typedef struct pdse {
        int tamInfo;
        NoPDSE *topo;
} PDSE;
```

## Arquivo TDA_PDSE.C

### Criando a Pilha Dinâmica Simplesmente Encadeada

A criação da PDSE implica na criação do nó descritor e sua iniciação: anula-se o campo *topo* e atribuí-se o tamanho da informação ao campo *tamInfo*, além disso deve-se atribuir o endereço do descritor ao ponteiro passado por referência (Figura 1.10).

```
int cria(PDSE **pp, int tamInfo)
{
        int ret = FRACASSO;
        PDSE* desc = NULL;

        desc = (PDSE*) malloc( sizeof(PDSE) );

        if( desc != NULL ) {
                desc->topo = NULL;
                desc->tamInfo = tamInfo;
                ret = SUCESSO;
        }

        *pp = desc;

        return ret;
}
```

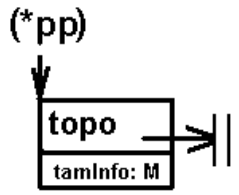


**Figura 1.10 – PDSE recém criada**

### A busca na Pilha Dinâmica Simplesmente Encadeada

É a cópia do conteúdo de dados, que se encontra referenciado pelo item no topo da pilha, para uma região de memória cujo endereço é passado. Se a pilha estiver vazia, obviamente não haverá dados a serem copiados (Figura 1.11).

```
int busca(PDSE *p, void *destino)
{ int ret;
  if (p->topo == NULL)
                ret = FRACASSO;
  else
    { memcpy(destino, p->topo->dados,p->tamInfo);
      ret = SUCESSO;
    }
  return ret;
}
```

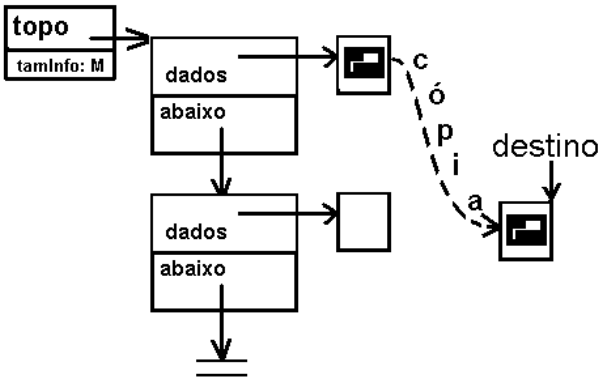


**Figura 1.11 - Buscando um item no topo da PDSE.**

### A inserção na Pilha Dinâmica Simplesmente Encadeada

Para melhor explicar este processo o mesmo será dividido em duas partes, cada uma com duas etapas. A seguir têm-se as duas primeiras etapas da parte I:

(A) Realizam-se duas alocações de memória: uma para o nó que referenciará os dados e outra para os dados propriamente ditos. Isso é realizado através de um ponteiro de auxílio (*temp* no caso).

(B) Considerando o sucesso das alocações no passo anterior, deve-se agora copiar os novos dados de um local de memória devidamente referenciado (aqui se utiliza o ponteiro *novo*) para a área de dados já alocada na pilha. Figura 1.12.

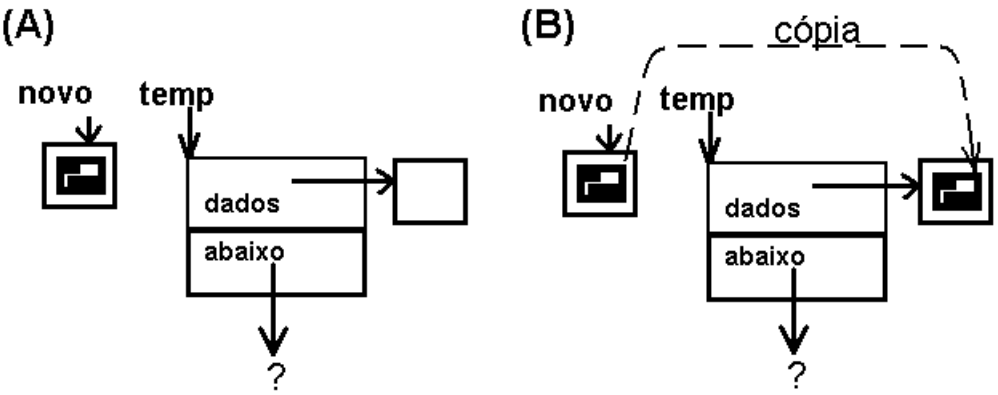


**Figura 1.12 - Empilhando um novo elemento – Parte I.**

Com os dados inseridos no novo elemento da pilha, deve-se ligá-lo ao resto da estrutura, o que consistirá na Parte II, com as seguintes etapas:

(C) Como o novo elemento entrará como o novo topo da pilha, todos os outros elementos passarão a estar abaixo dele, portanto deve-se ligá-lo acima de todos os outros, após isso...

(D) Atualiza-se o descritor para o mesmo referenciar o novo elemento de topo (Figura 1.13).

```
int empilha(PDSE *p, void *novo)
{ int ret;
  NoPDSE *temp;
  if ((temp = (NoPDSE *) malloc(sizeof(NoPDSE))) == NULL)
       ret = FRACASSO;
  else
     { if ( (temp->dados = (void *)malloc(p->tamInfo))==NULL)
        {free(temp);
         ret = FRACASSO;
        }
       else
        { memcpy(temp->dados, novo, p->tamInfo);
          temp->abaixo = p->topo;
          p->topo = temp;
          ret = SUCESSO;
        }
     }
  return ret;
}
```

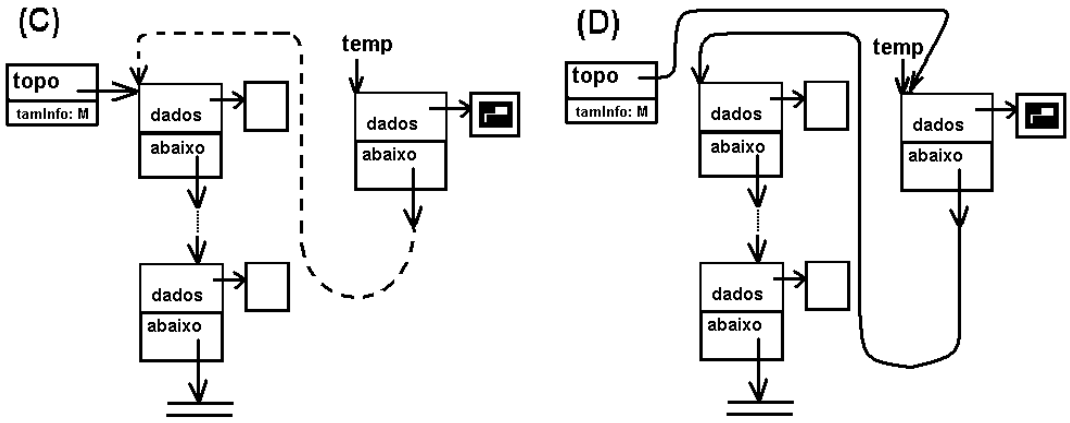


**Figura 1.13 – Empilhando um novo elemento – Parte II.**

## A remoção da Pilha Dinâmica Simplesmente Encadeada

Esta consiste basicamente na liberação da região de memória que foi alocada durante a última inserção, ou seja, aquela inserção que incluiu o item que atualmente se encontra no topo da pilha. Tal operação é realizada por uma função nativa do *C*, trata-se da função *free* a qual executa o papel inverso do *malloc*.

Perceba que o fato de executar a chamada *free(p)* não implica em anular o ponteiro *p*, nem mesmo implica em uma imediata "limpeza" da região de memória referenciada por *p*. O que *free* efetivamente faz é informar ao gerenciador de memória (sistema operacional) que a região apontada por *p*, que foi previamente alocada por *malloc* (ou outra função de alocação) está disponível para outro uso. Além da função *free* a operação de remoção necessita do auxílio de um ponteiro auxiliar que permita a correta atualização do *topo* da pilha ao final do processo (Figura 1.14). Obviamente, se a pilha estiver vazia não há o que remover.

```
int desempilha(PDSE *p)
{ NoPDSE  *aux;
  int ret;
  if (p->topo == NULL)
                    ret = FRACASSO;
  else
     { aux = p->topo->abaixo;
       free(p->topo->dados);
       free(p->topo);
       p->topo = aux;
       ret = SUCESSO;
     }
  return ret;
}
```

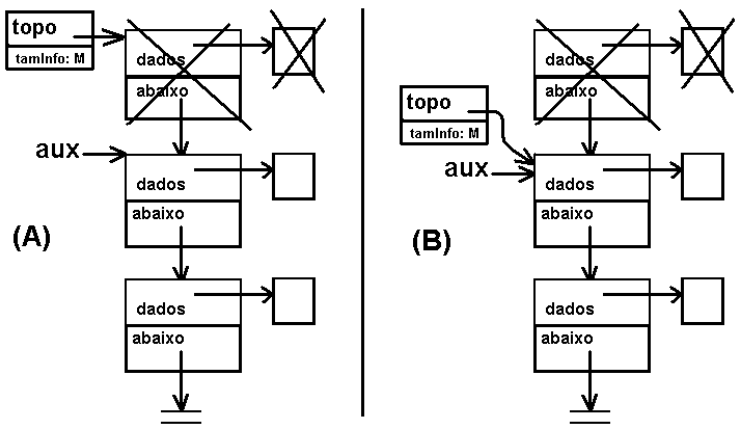


**Figura 1.14 - Removendo da PDSE.**

## Reiniciando a Pilha Dinâmica Simplesmente Encadeada

Similar à congênere na PE, a operação de reinício da PD corresponde a um retorno deste TDA ao estado imediatamente após a sua criação, ou seja, há uma liberação dos dados e dos *nós* que os referenciavam. Quaisquer regiões de memória que tiverem sido alocadas durante as inserções, deverão ser liberadas. Isso é implementado como se ocorresse uma repetição das ações de remoção, até que a pilha esteja vazia.

```
void reinicia( PDSE *p)
{ NoPDSE *temp;
  while(p->topo != NULL)
  { temp = p->topo->abaixo;
    free(p->topo->dados);
    free(p->topo);
    p->topo = temp;
  }
}
```

## A destruição da Pilha Dinâmica Simplesmente Encadeada

O processo de destruição é um complemento ao reinício da pilha, o objetivo desta operação é eliminar, não somente os dados, mas qualquer traço de memória que tenha sido alocada para criação ou inserção na pilha, dessa forma, além da liberação de memória realizada pela operação *reinicia*, faz-se uma liberação da região de memória alocada para o descritor da pilha e por fim anula-se o ponteiro passado por referência (verifique no código abaixo). Ao final desta operação, no que se refere à pilha,  o sistema assume um estado igual àquele anterior à sua criação (da pilha).

```
void destroi(PDSE **pp)
{ reinicia(*pp);
  free(*pp);
  (*pp) = NULL;

}
```

## 1.3 Pilha Estática Múltipla

Nesta estrutura, várias pilhas estáticas são implementadas sobre um único vetor particionado. Individualmente, cada partição contém uma pilha e cada pilha é associada ao respectivo descritor.

### 1.3.1 Implementação da Multipilha Estática – MpE

Cada célula do vetor que dá suporte à MpE compartilha diferentes tipos de dados em uma mesma região de memória, as células servem tanto à representação dos nós que referenciam dados quanto à representação de descritores. Em linguagem *C* é possível realizar esse tipo de compartilhamento através da estrutura do tipo união (*union*). Encare a MpE como uma boa oportunidade para exercitar esse recurso em *C*.

#### 1.3.1.1 Utilizando a União - *Union*

Na implementação aqui proposta, são necessários os parâmetros N (correspondente ao número de pilhas) e L (relativo ao comprimento máximo de cada pilha) para a criação do vetor com N+N*L células do tipo união (espaço para os N descritores e suas respectivas pilhas de tamanho L). A fronteira "virtual" entre as regiões de descritores e a região das respectivas pilhas, é delimitada da seguinte forma: as N primeiras células do vetor servirão aos descritores enquanto as demais servirão aos nós que referenciam os dados das pilhas, conforme a Figura 1.15. Os valores N e L constarão em um descritor geral da MpE.

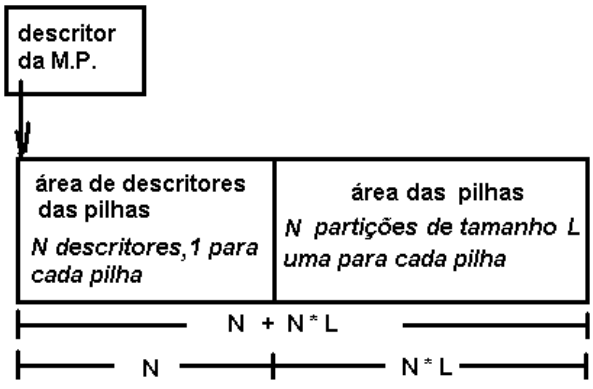


**Figura 1.15 - Multipilha Estática.**

#### 1.3.1.2 Indexando uma Pilha na MpE

O módulo de aplicação do TDA se abstrai do fato dos vetores em *C* serem *zero-indexados*, portanto, parte-se do princípio que o módulo de aplicação entende uma *multipilha* com N

pilhas, como uma seqüência começando da 1ª pilha, seguida da 2ª, até a *N-ésima* pilha. Não existe uma *pilha zero*.

Considerando-se a organização do vetor conforme descrito anteriormente e exibida na 1.16, para acessar o *i*-ésimo descritor (1≤ i ≤ N) basta acessar ou indexar a célula *i-1* do vetor. Esse descritor conterá dados sobre a pilha (*i-ésima )* a ele associada.

Além de identificar o seu descritor, para realizar uma operação sobre uma pilha é preciso localizar as células do vetor correspondentes à mesma, ou seja, quais as células que delimitam o início da partição (*inicioPartição*) e o final da sua partição (*finalPartição*) da pilha-alvo em questão. A determinação desses valores é realizada com base nas informações no descritor dessa pilha, *inicioPartição* consta explicitamente no descritor. Quanto à segunda informação – *finalPartição* – poderá ser calculada facilmente pela expressão: *inicioPartição* + (*L* – 1).

Caso as informações de partição não existissem no descritor, ainda assim seria possível calcular o *inicioPartição* através da expressão (N+ L(i-1)), onde i corresponde à pilha-alvo e $1 \leq i \leq N$.

A partir do valor *inicioPartição* se calcula *finalPartição* como sendo *inicioPartição*+L-1.

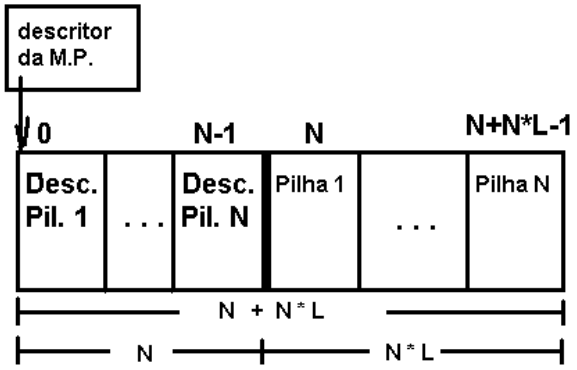


**Figura 1.16 - Indexação dos descritores e respectivas pilhas.**

### 1.3.1.3 Implementando a MpE.

Para a MpE a arquitetura de arquivos ainda é semelhante àquela utilizada para a implementação da PE e da PD, aqui, no entanto, eles serão nomeados como TDA_MpE.H, TDA_MpE_PRIV.H e TDA_MpE.C. Aqui não será exibido todo o código dos arquivos, apenas alguns fragmentos importantes para uma posterior implementação.

Diferentemente da *struct*, que ao ser instanciada reserva memória para todos os seus campos, ao instanciar uma *union* (ou união), reserva-se um espaço de memória que corresponde ao tamanho do maior campo definido, sendo que este espaço será compartilhado por todos os campos envolvidos na definição da *union*. Quando um desses campos é muito menor que os demais membros e é requisitado de maneira mais freqüente, tem-se um desperdício de memória. Dessa forma, em termos de um bom uso da memória, o

uso de *union* só se justifica se houver um equilíbrio entre o tamanho (em *bytes*) dos elementos que constituirão a unidade da *union*. Isto está contemplado na implementação aqui proposta, onde a *union* definida como *NoMP* é constituída basicamente de um ponteiro genérico (*void*) e dois campos do tipo inteiro curto (*short int*). Considerando que um ponteiro *void* equivale a duas variáveis do tipo *short int*, haverá uma utilização equilibrada de memória.

Na 1.17 é vista uma representação gráfica do resultado da criação da MpE. Observe que, por estarem vazias, as pilhas exibidas na figura possuem valores de *topo* que retratam tal estado. Para a Pilha Estática simples a representação do estado "pilha vazia" foi feita pela inicialização do *topo* com um valor que não indexa o vetor (utilizou-se a macro VAZIA que foi definida com o valor menos um "-1"). Para a MpE pretende-se utilizar a mesma concepção para iniciar os topos das N pilhas envolvidas. Outra possibilidade seria cada *topo* ser iniciado com um valor inteiro que não indexe a respectiva partição da pilha esse valor corresponderia ao início da respectiva partição menos um.

## Fragmento do arquivo TDA_MPE.H

```
...
typedef struct mp MP;
...
```

## Fragmento do arquivo TDA_MPE_PRIV.H

```
/* Nó descritor da uma pilha */
typedef struct {
   short int topo;
   short int inicioParticao;
} DescPilha;

/* Nó da Multi-Pilha */
typedef union {
   DescPilha descritor;
   void* dados;
} NoMP;

/* Descritor da Multi-Pilha */
typedef struct mp {
   int N; /* N = Número de Pilhas*/
   int L; /* L = Tamanho máximo da partição de cada Pilha*/
   int tamInfo;
   NoMP *vet;
} MP;
```

# Fragmento do arquivo TDA_MpE.C

```
int cria( ppMP pp, int N, int L, int tamInfo)
{  int i, M, ret = SUCESSO;
   NoMP  *aux;
   MP *desc=NULL;
  if (N > 0 && L > 0 && tamInfo > 0)
   {    M = N*L;
        if((desc = (MP *) malloc(sizeof(MP)))==NULL) /* 1 */
           ret = FRACASSO;
        else
           if( ( desc->vet = (NoMP *) malloc((M+N)* sizeof(NoMP)) )==NULL) /* 2 */
           {  free (desc);
              (desc) = NULL;
              ret = FRACASSO;
           }
          else
           { desc->N = N;     /* 3 */
             desc->L = L;
             desc->tamInfo = tamInfo;
             aux= desc->vet;          /* 4 */
             for(i = 0; i < N ; i++)            /* 5 */
              {  (aux+ i)->descritor.topo = -1;        /* 6 */
                 (aux+ i)->descritor.inicioPartição = N +  i*L;
              }
             aux = aux + N;         /* 7 */
             for(i = 0; i < M ;i++)     /* 8 */
             if ((aux+i)->dados = (void*) malloc(tamInfo)) == NULL )
               {  for(--i; i >=0;--i)
                             free(desc->vet[i].dados);     /* 9 */
                  free(desc->vet);
                  free(desc);
                  desc=NULL;
                  i = M;
                  ret = FRACASSO;
               }
     }
   }
   else
               ret = FRACASSO;
  *pp = desc;     /* 10 */
  return ret;
}
/*Comentários:
1. criação do descritor geral da MpE;
2. Alocação de memória para a multipilha (vetor de unions) constituida  de M+N nós de
dados e os N nós descritores das respectivas pilhas;
3. Iniciando os atributos gerais da multi-pilha;
4. Ponteiro auxiliar aux apontando para a área de descritores;
5. Formatando a MpE: iniciando os descritores das pilhas;
6. Os topos são iniciados com valores que não indexam as partições das respectivas pilhas,
indicando que a considerada pilha está vazia. Perceba que também é possível utilizar a
sintaxe (*pp)->vet[i].descritor ou ((*pp)->vet + i).descritor para acesso ao i-ésimo
descritor.
7. Ponteiro auxiliar aux apontando para o início da área de dados;
```

8. Criando as áreas de dados;
9. Houve falha na alocação de pelo menos uma área de dados, a MpE deve ser destruída.
10. Sucesso na criação: após alocações e incializações bem-sucedidas, o endereço do descritor é "conectado" ao ponteiro passado por referência e retorna-se ao cliente */

### 1.3.1.4 Acessando a MpE

Os fragmentos descritos a seguir (1) baseiam-se na formatação exibida na 1.17, a qual representa uma MPE recém criada segundo a função de criação anteriormente descrita. Considera-se a i-ésima pilha $P_i$ onde $1 \leq i \leq N$ e que o apontador *ptr* contenha o endereço do descritor geral da MpE.

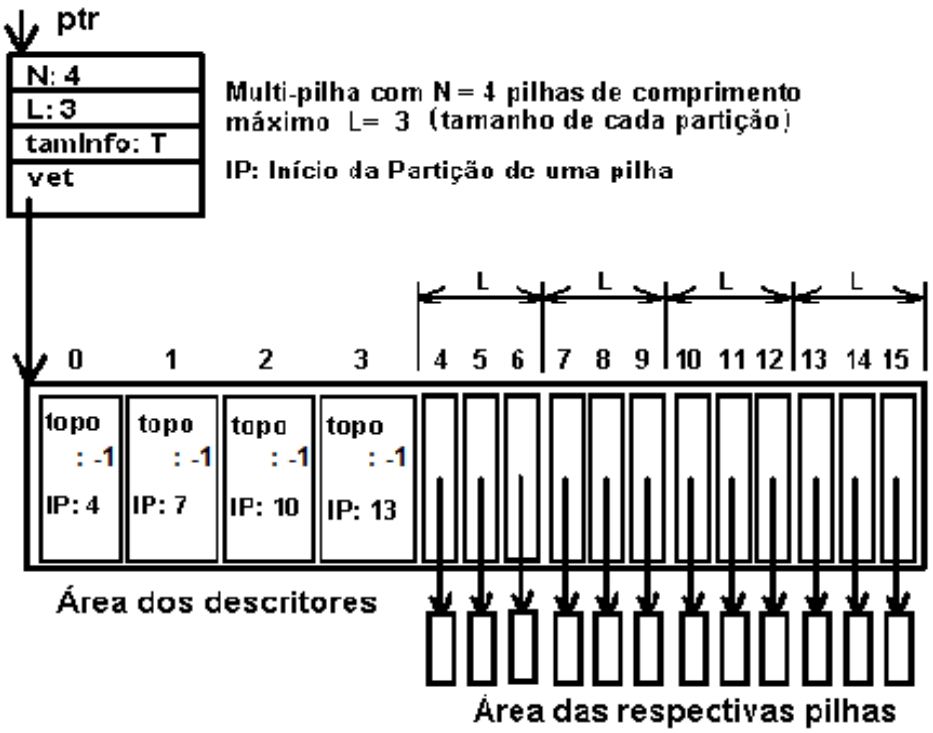


Figura 1.17- MPE recém criada para 4 pilhas de tamanho máximo 3.

| Descrição | Fragmento |
|---|---|
| Pi corresponde à i-ésima pilha, onde 1 ≤ i ≤ N: 1ª pilha = P1, 2ª pilha = P2, ... última pilha = Pn<br>Assim, o acesso ao i-ésimo descritor pode ser realizado por indexação utilizando i-1 como índice. Alternativa, por apontadores diretamente. | ptr->vet[i-1].descritor; |
| Acesso ao início da partição da i-ésima pilha. | IP_i = ptr->vet[i-1].descritor.inicioParticao; |
| Acesso ao final da partição da i-ésima pilha. | FP_i = IP_i + (ptr->L) -1; |
| Acesso ao topo da i-ésima pilha. | topo_i= ptr->vet[i-1].descritor.topo; |
| Índice referente ao topo da i-ésima pilha | IP_i+topo_i |
| Para buscar a informação no topo da i-ésima pilha (não vazia) basta indexar o vetor pelo valor do seu início de partição somado ao valor do seu topo. O topo é um deslocamento a partir do início da partição (-1 ≤ topo ≤ L-1). | enderecoOrigem = ptr->vet[IP_i+topo_i].dados;<br>memcpy(enderecoDestino, enderecoOrigem , p->tamInfo); |
| Para empilhar na i-ésima pilha (não cheia) basta incrementar o seu topo e copiar os novos dados para a célula do vetor indexada pelo valor do seu início de partição somado ao valor do seu topo. O topo é um deslocamento a partir do início da partição (-1 ≤ topo ≤ L-1). | ptr->vet[i-1].descritor.topo += 1;<br>topo_i=ptr->vet[i-1].descritor.topo;<br>enderecoDestino = ptr->vet[IP_i+topo_i].dados;<br>memcpy(enderecoDestino, novo, p->tamInfo); |
| Pata determinar o número de elementos empilhados na *i-ésima* pilha basta somar 1 ao valor atual do seu topo (-1 ≤ topo ≤ L-1). | *numElementosPilha_i=ptr->vet[i-1].descritor.topo+1;* |
| Sabendo que *-1 ≤ topo_i ≤ L-1*, para testar se a *i-ésima* pilha está vazia é necessário saber se seu topo é igual ao valor de inicialização: -1 | SE (*topo_i == -1*)<br>a pilha_i está vazia.<br>SENÃO<br>a pilha_i não está vazia |
| Sabendo que *-1 ≤ topo_i ≤ L-1*, o maior valor de topo da *i-ésima* pilha corresponde a *L-1* e ocorre quando a *i-ésima* pilha está cheia.<br><br>Outras formas de teste dessa condição (cheia) consistem em saber, a partir do início da partição somado ao topo, se a pilha alcançou o final de sua partição.<br><br>Outra forma, ainda, é verificar se o número de elementos empilhados é igual ao comprimento máximo da partição. | SE (*topo_i == (ptr->L) -1*)<br>a pilha_i está cheia.<br>SENÃO<br>a pilha_i não está cheia<br>Alternativas:<br>SE (*IP_i+topo_i == FP_i*)<br>a pilha_i está cheia.<br>SENÃO<br>a pilha_i não está cheia<br><br>SE (*numElementosPilha_i == ptr->L*)<br>a pilha_i está cheia.<br>SENÃO<br>a pilha_i não está cheia |
| Para apenas desempilhar a i-*ésima* pilha (não vazia) basta decrementar o seu topo. | *--(ptr->vet[i-1].descritor.topo);* |
| Para reinicializar a i-*ésima* pilha basta atribuir menos um "-1" ao seu topo. | *ptr->vet[i-1].descritor.topo* = -1; |

**Tabela 1: Fragmentos de código para a MpE.**

# 2. Bibliografia


[1] Backus, John. Evolução das Principais Linguagens de Programação. In: Sebesta, R. W. Conceitos de Linguagens de Programação. 4ª ed. Porto Alegre: Editora Bookman, 2000. p. 49-110.

[2] Barr, Michael. Programming embedded Systems in C and C++. First Edition. Sebastopol: O'Reilly and Associates, 1999. 174p. cap 1.: Introduction: C: The least Common Denominator.

[3] Fishwick, P. A. Computer Simulation Potentials, IEEE , Volume: 15 , Issue: 1 , P:24–27.Feb.-March.1996.

[4] Gersting, Judith L. "Fundamentos Matemáticos para a Ciência da computação". Ed. LTC.

[5] Goldberg, Adele. Suporte para Programação Orientada a Objeto. In: Sebesta, R. W. Conceitos de Linguagens de Programação. 4ª ed. Porto Alegre: Editora Bookman, 2000. p. 417-466.

[6] Horowitz, E. & Sahni, S. "Fundamentos de Estruturas de Dados". Ed. Campus. 1984.

[7] Pereira, Sílvio L. Estruturas de Dados Fundamentais – Conceitos e Aplicações. 7a. ed. Érica, 2003,p. 73

[8] Parnas, D. L. On the Criteria to be Used in Decomposing systems into Modules Communications of the ACM**,** Volume 15 Issue 12. December 1972.

[9] Ritchie, Dennis. Subprogramas. In: Sebesta, R. W. Conceitos de Linguagens de Programação. 4ª ed. Porto Alegre: Editora Bookman, 2000. p. 315-359.

[10] Szwarcfiter, Jayme L. & Markenzon L. "Estruturas de Dados e Seus Algoritmos". Rio de Janeiro. Ed. LTC. 1994.

[11] Tenembaum, Aaron M. e outros. "Estruturas de Dados Usando C". São Paulo. Ed.Makron Books. 1995.

[12] Tremblay, J.; Bunt, R..Ciência dos Computadores. Uma abordagem algorítmica, McGraw Hill, 1983.

[13] Van Gelder, Allen. A Discipline of Data Abstraction using ANSI C. Documento disponível em http://www.cse.ucsc.edu/~avg/. 02/2005.

[14] Veloso, Paulo et al. "Estruturas de Dados". Ed. Campus. 1984.

[15] Wirth, Niklaus. “Algorthms + Data structures = Programs”, Ed. Prentice Hall, 1976.