Algoritmos e estruturas de Dados A07

Filas e Pilhas

Prof. Mauri Ferrandin

Lista

- É uma **sequência de elementos**, geralmente do mesmo tipo: L_1, L_2, …, L_N
- **Operações comuns**:
  - criar uma lista vazia
  - adicionar/remover um elemento a uma lista
  - determinar a posição de um elemento na lista
  - determinar o comprimento (nº de elementos) de uma lista
  - concatenar duas listas

Fila / Queue

São estruturas de dados que possuem comportamento similar à uma fila de caixa do supermercado:

- A primeira pessoa que chegou na fila é atendida primeiro
- Os demais clientes entram na fila e aguardam atendimento

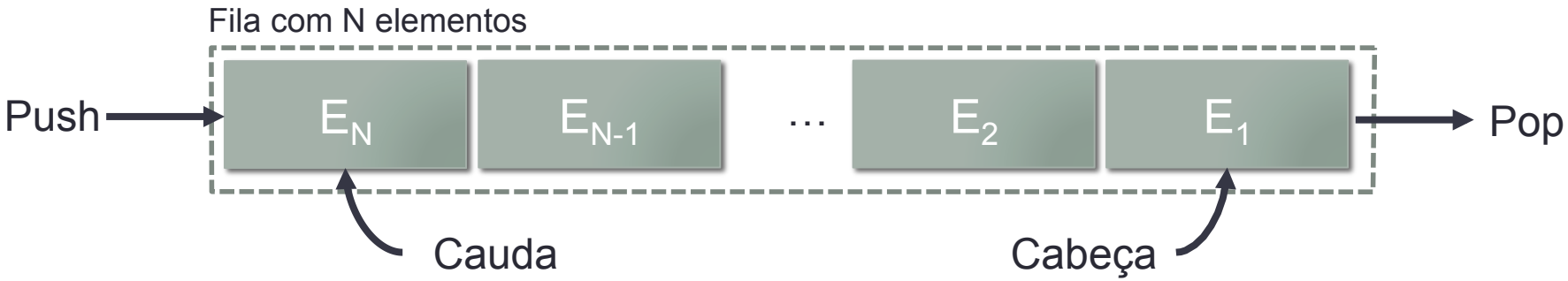
Uma fila pode ser considerada como uma **restrição de lista** (regras especiais para acesso dos elementos):

- Todo elemento que entra na lista, entra no fim
- Todo elemento que sai da lista, sai do início

Essa disciplina de acesso é conhecida como FIFO (First-In-First-Out)

Fila / Queue

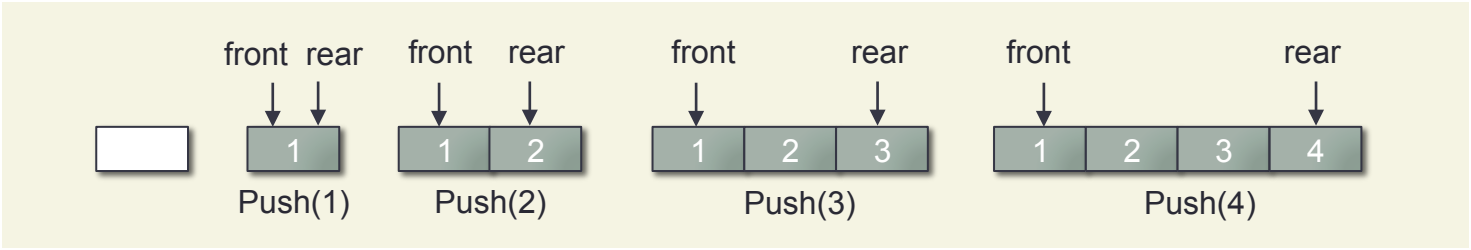
- Uma fila tem início (cabeça) e fim (cauda)
- A inserção de elementos se faz após o ultimo elemento (cauda).
- O elemento eliminado é sempre o que se encontrava na fila a mais tempo (cabeça)

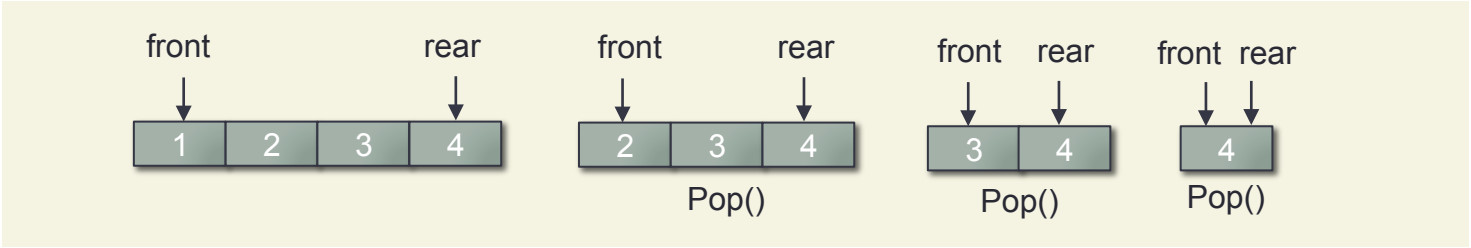


Fila / Queue

Operações comuns:

- criar uma fila vazia
- Adicionar (push)/remover(pop) um elemento a uma fila
- verificar qual o elemento da cabeça/cauda da fila (mais antigo/ recente)





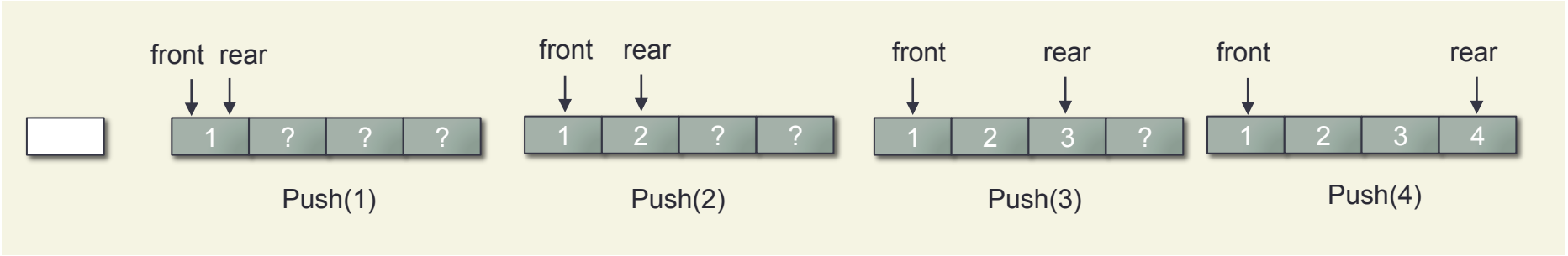
Implementação da Fila / Queue

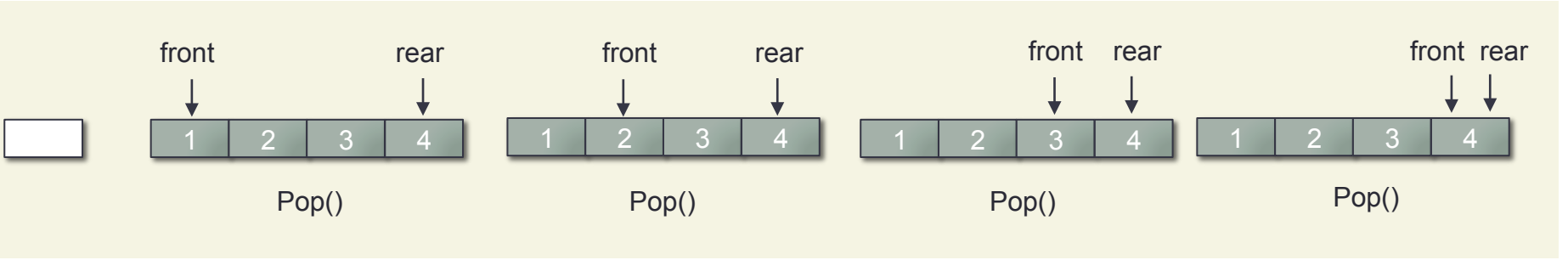
A implementação da fila pode ser feita com:

- Um **vetor**
  - Capacidade do vetor é pré-definida. É necessário saber o tamanho da fila e o índice dos elementos que estão na cauda e na cabeça da fila.
  - **Push**: Se a fila não estiver cheia, insere novo elemento na cauda e incrementa o índice respectivo. O índice volta a zero se se exceder o tamanho do vector (fila circular).
  - **Pop**: Se a fila não estiver vazia, remove o elemento da cabeça da fila e incrementa o índice respectivo. O índice volta a zero se se exceder o tamanho do vector (fila circular).
- Uma **lista ligada**
  - **Push**: Insere novo elemento no fim da lista.
  - **Pop**: Remove o elemento do início da lista.

Fila (Implementação baseada em Vetor)

- O tamanho do vetor não se altera durante a execução do programa
- Somente são atualizados os indices da cabeça e da cauda

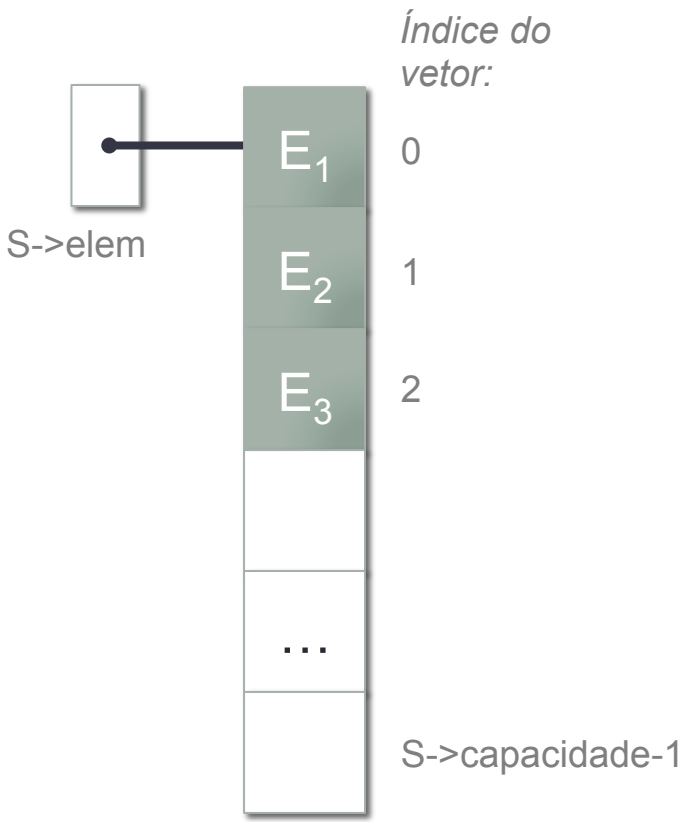




Fila (Implementação baseada em Vetor)

```
struct filaItem
{
   int capacidade; /* capacidade da fila */
   int cabeca; /* índice da cabeça da fila */
   int cauda; /* índice da cauda da fila */
   int tam; /* tamanho da fila */
   int *elem; /* vetor com os elementos */
};
typedef struct filaItem* Fila;
```

```
/* cria uma nova fila */
Fila criaFila( int maxSize );

/* insere um novo elemento na cauda*/
void Push( int X, Fila F ) ;

/* remove o elemento da frente */
void Pop(Fila F );

/* obtém o valor do elemento da frente */
int Front(Fila F );
```



Fila (Implementação baseada em Vetor)

```
Fila criaFila( int maxSize )
{
   Fila F1;
   if( maxSize < MIN_QUEUE_SIZE )
      printf( "Fila size is too small\n" );

   F1 = ( struct filaItem * ) malloc( sizeof( struct filaItem ) );
   if( F1 == NULL ) {
      printf( "Out of space!\n" ); exit(EXIT_FAILURE);
   }
   F1->elem = ( int * ) malloc( sizeof( int ) * maxSize );
   if( F1->elem == NULL ) {
      printf( "Out of space!\n" ); exit(EXIT_FAILURE);
   }

   F1->capacidade = maxSize;
   F1->tam = 0;
   F1->cabeca = 0;
   F1->cauda = 0;
   return F1;
}
```

Fila (Implementação baseada em Vetor)

```
void Push( int X, Fila F )
{
   if( F->tam == F->capacidade ) printf( "Full queue\n" );
   else {
      F->tam++;
      F->elem[ F->cauda ] = X;
      if ( ++F->cauda == F->capacidade )
         F->cauda = 0;
   }
}

void Pop( Fila F )
{
   if( F->tam == 0 ) printf( "Empty queue\n" );
   else
   {
      F->tam--;
      if ( ++F->cabeca == F->capacidade )
         F->cabeca = 0;
   }
}
```

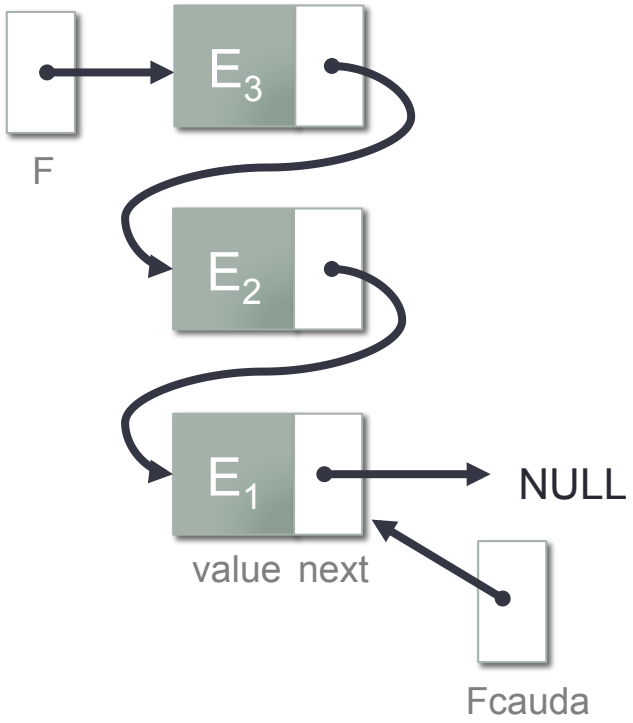
Fila (Implementação baseada em Vetor)

```
int Front( Fila F )
{
   if( F->tam != 0 )
      return F->elem[ F->cabeca ];
   printf( "Empty queue\n" ); return 0;
}

int main( )
{
   Fila F; int i;

   F = criaFila(15);
   for( i = 0; i < 10; i++ )
      Push( i, F );

   while( F->tam != 0 )
   {
      printf( "Cabeca: %d\tTamanho: %d\n", Front( F ), F->tam );
      Pop( F );
   }

   free( F->elem ); free( F);
}
```

Fila (Implementação baseada em Lista Ligada)

```
typedef struct filaItem {
    /* valor do elemento da fila */
    int valor;
    /* apontador para o elemento seguinte */
    struct filaItem *prox;
} Filaitem;
typedef struct {
    Filaitem *inicio;
    Filaitem *fim;
    int nelem;
} Fila;
```

```
/* cria uma nova fila */
Fila *criaFila();

/* insere um novo elemento na fila */
void Push( int X, Fila *F );

/* remove o elemento da frente */
void Pop( Fila *F );

/* obtém o valor do elemento da frente */
int Peek( Fila *F );
```



Fila (Implementação baseada em Lista Ligada)

```
Fila *criaFila( void ) {
    Fila *F = (Fila *) malloc( sizeof( Fila ) );
    if( F == NULL ) {
        printf("Pop: Sem espaco na memoria!!!\n"); exit(EXIT_FAILURE);
    }
    F->inicio = NULL; F->fim = NULL;
    F->nelem = 0; return F;
}
void Push( int X, Fila *F ) {
    Filaitem *no = (Filaitem *) malloc( sizeof(Filaitem ) );
    if( no == NULL ) {
        printf( "Push: sem espaco na memoria!!!\n" ); exit(EXIT_FAILURE);
    }
    no->valor = X; no->prox = NULL;

    if( F->nelem == 0 ){
        //if (F->inicio == NULL){
                F->inicio = no; F->fim = no;
        } else {
                F->fim->prox = no; F->fim = no;
        }
        F->nelem++;

}
```

Fila (Implementação baseada em Lista Ligada)

```
void Pop( Fila *F )
{
    if( F->inicio == NULL )
        printf( "Fila vazia\n" );
    else
    {
        Filaitem *aux = F->inicio;
        F->inicio = F->inicio->prox;
        free(aux);
        F->nelem--;
    }
}

int Peek( Fila *F )
{
    if( F->inicio != NULL )
        return F->inicio->valor;
    printf( "Fila vazia\n" );
    return -1;
}
```

Fila (Implementação baseada em Lista Ligada)

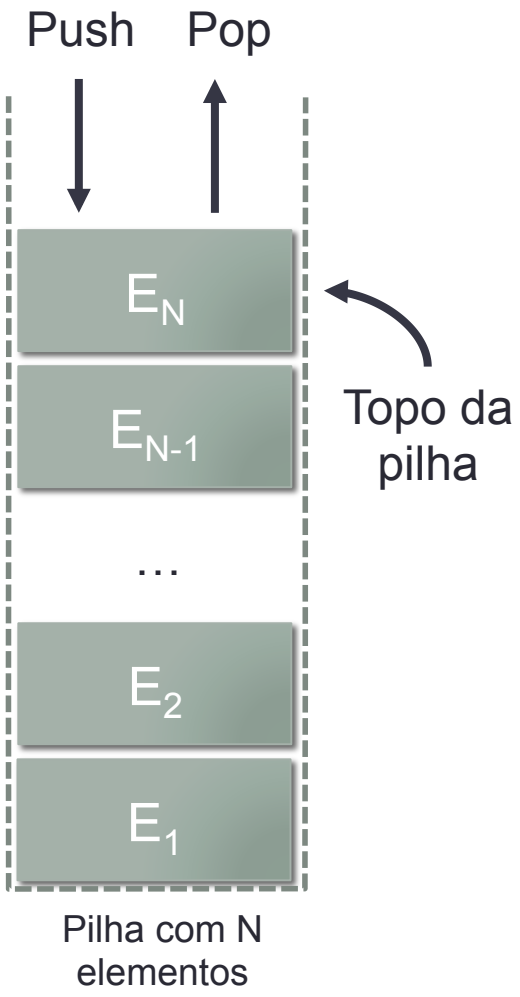
```
int main( )
{
    Fila *F = criaFila();
    int i;

    for( i = 0; i < 10; i++ )
        Push( i, F );

    while( F->inicio != NULL )
    {
        printf( "Cabeca: %d\n", Peek( F ) );
        Pop( F );
    }

    free( F );
}
```

Pilha / Stack

- Uma pilha também pode ser considerada como uma **lista com restrição de acesso**.
- A inserção e remoção de elementos se faz pela mesma extremidade (topo da pilha)
- O elemento eliminado é sempre o mais recente (cabeça)
- Os elementos da pilha são retirados na ordem inversa à que foram introduzidos: LIFO (Last-In-First-Out)

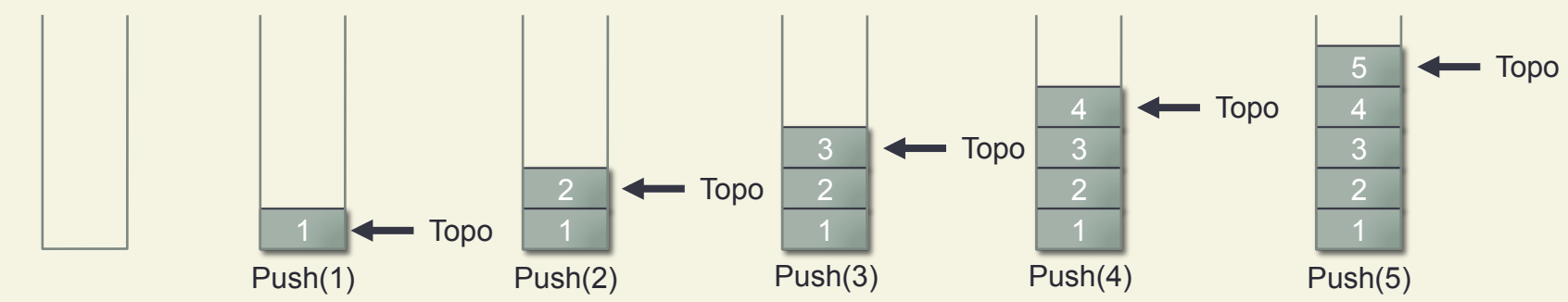
Exemplos :

- Pilha de pratos em um restaurante
- Vagões de um trem
- Retirada de mercadorias em um caminhão de entregas



Pilha / Stack

- **Operações comuns:**
  - criar uma pilha vazia
  - Adicionar (push)/remover(pop) um elemento à pilha
  - verificar qual o topo da pilha (último elemento adicionado)



Implementação da Pilha / Stack

A implementação da pilha pode ser feita com:

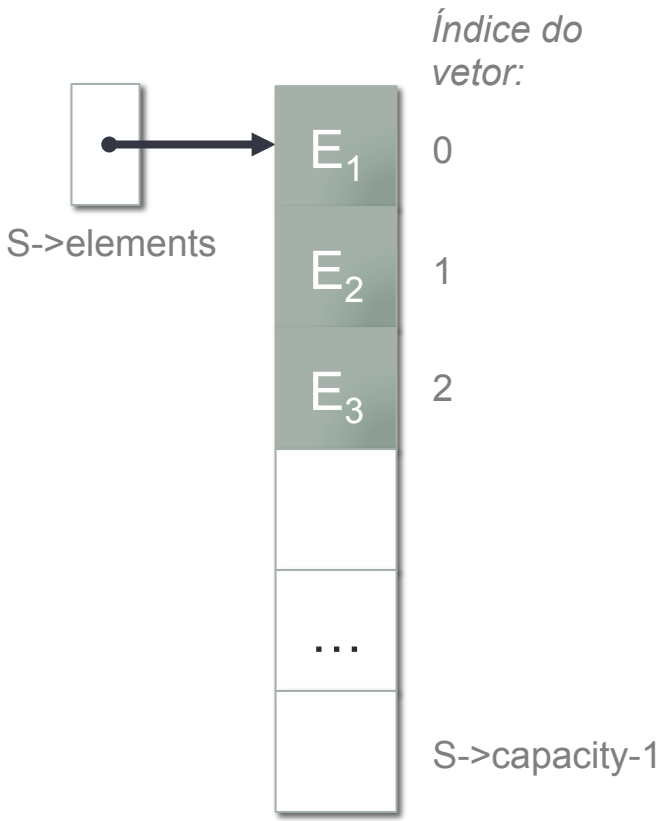
- Um **vetor**
  - Capacidade do vetor é pré-definida e é necessário saber o índice do elemento que está no topo da pilha.
  - **Push**: Se o vetor não estiver cheio, insere novo elemento na primeira posição vazia e incrementa o índice do topo.
  - **Pop**: Se o vetor não estiver vazio, remove o elemento que cujo índice corresponde ao topo da pilha e decrementa o índice.

- Uma **lista ligada**
  - **Push**: Insere novo elemento no início da lista.
  - **Pop**: Remove o elemento do início da lista.

Pilha (Implementação baseada em Vetor)

```
#define PILHA_VAZIA -1
#define MIN_TAM_PILHA 5
struct pilhaItem
{
    int capacidade; /* capacidade da pilha*/
    int topo; /* índice do topo da pilha*/
    int *elem; /* vetor elementos */

};
typedef struct pilhaItem* Pilha;
```

```
/* cria uma nova pilha (stack) */
Pilha criaPilha( int maxSize );

/* insere um novo elemento no topo */
void Push( int X, Pilha S ) ;

/* remove o elemento do topo */
void Pop( Pilha S );

/* obtém o valor do elemento do topo */
int Top( Pilha S );
```



Pilha (Implementação baseada em Vetor)

```
Pilha criaPilha( int maxSize )
{
    Pilha S;
    if( maxSize < MIN_TAM_PILHA ) {
        printf("Pilha size is too small\n" ); exit(EXIT_FAILURE);
    }

    S = (struct pilhaItem *) malloc( sizeof(struct pilhaItem) );
    if( S == NULL ) {
        printf( "Out of space!\n" ); exit(EXIT_FAILURE);
    }
    S->elem = (int *) malloc( sizeof(int) * maxSize );
    if( S->elem == NULL ) {
        printf( "Out of space!\n" );
        exit(EXIT_FAILURE);
    }
    S->capacidade = maxSize;
    S->topo = PILHA_VAZIA;
    return S;
}
```

Pilha (Implementação baseada em Vetor)

```
void Push( int X, Pilha S )
{
    if( S->topo == S->capacidade - 1 ) {
        printf( “Pilha Cheia\n" ); exit(EXIT_FAILURE);
    }
    else S->elem[ ++S->topo ] = X;
}

void Pop( Pilha S )
{
    if( S->topo == PILHA_VAZIA )
        printf( “Pilha Vazia\n" );
    else S->topo--;
}

int Top( Pilha S )
{
    if( S->topo != PILHA_VAZIA )
        return S->elem[ S->topo ];
    printf( “Pilha cheia" ); exit(EXIT_FAILURE);
    return 0;
}
```

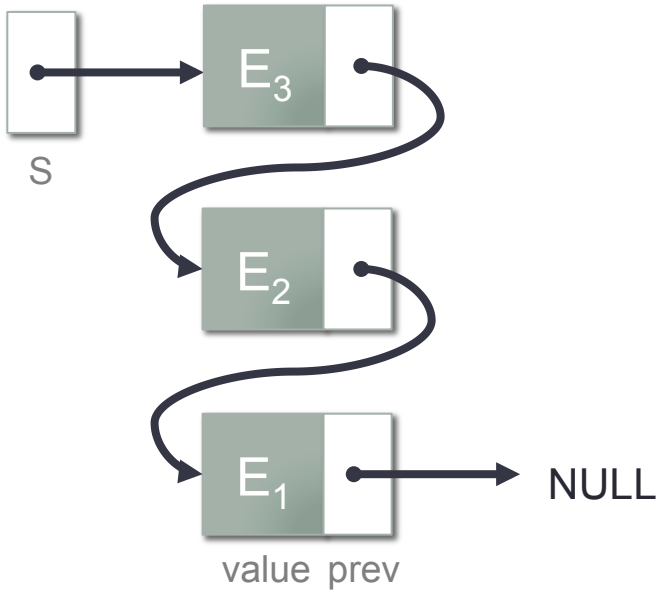
Pilha (Implementação baseada em Vetor)

```
int main( )
{
   Pilha S;
   int i;

   S = criaPilha( 15 );
   for( i = 0; i < 10; i++ )
      Push( i, S );

   while( S->topo != PILHA_VAZIA )
   {
      printf("Topo: %d\n", Top( S ) );
      Pop( S );
   }

   if( S != NULL ) {
      free( S->elem ); free( S );
   }
}
```

Pilha (Implementação baseada em Lista Ligada)

```
typedef struct pilhaItem
{
    int valor; /* valor do elemento da stack */
    struct pilhaItem *prev; /* apontador para o elemento anterior
*/
} Pilhaitem;

typedef struct {
    Pilhaitem *inicio;
    int nelem;
} Pilha;
```

```
/* cria uma nova pilha (stack) */
Pilha *criaPilha( );

/* insere um novo elemento no topo */
void Push( int X, Pilha *P ) ;

/* remove o elemento do topo */
void Pop( Pilha *P );

/* obtém o valor do elemento do topo */
int Peek( Pilha *P );
```



Pilha (Implementação baseada em Lista)

```
Pilha *criaPilha( )
{
        Pilha *pilha = (Pilha *) malloc( sizeof( Pilha ) );
        if( pilha == NULL ) {
                printf("Sem espaço na memória!!!\n"); exit(EXIT_FAILURE);
        }
        pilha->inicio = NULL;
    pilha->nelem = 0;
        return pilha;
}

void Push( int X, Pilha *P )
{
        Pilhaitem *tmp = (Pilhaitem *) malloc( sizeof(Pilhaitem ) );
        if( tmp == NULL ) {
                printf( "Push: Sem espaço na memória!!!\n" );
        exit(EXIT_FAILURE);
        }
        else {
                tmp->valor = X;
                tmp->prev = P->inicio;
                P->inicio = tmp;
         P->nelem++;

        }
}
```

Pilha (Implementação baseada em Lista)

```
void Pop( Pilha *P )
{
        Pilhaitem *firstElem;
        if( P->inicio == NULL )
                printf( "Pilha vazia\n" );
        else
        {
                firstElem = P->inicio;
                P->inicio = P->inicio->prev;
                free( firstElem );
         P->nelem--;
        }
}

int Peek( Pilha *P )
{
   if( P->inicio != NULL )
                return P->inicio->valor;
        printf( "Pilha vazia\n" );
                return -1;
}
```

Pilha (Implementação baseada em Lista)

```
int main( )
{
        Pilha *P = criaPilha( );
        int i;

    for( i = 0; i < 10; i++ )
                Push( i, P );

        while( P->inicio != NULL )
        {
                printf( "%d\n", Peek( P ) );
                Pop( P );
        }
        free( P );
}
```