ASSOCIAÇÃO PARAIBANA DE ENSINO RENOVADO
FACULDADE PARAIBANA DE PROCESSAMENTO DE DADOS

ESTRUTURA DE DADOS

INTRODUÇÃO

O que é uma Estrutura de Dados (ED)?

- ? Tipos de Dados
- ? Estruturas de Dados e
- ? Tipos Abstratos de Dados

Embora estes termos sejam parecidos, eles têm significados diferentes. Em linguagens de programação, o **tipo de dados** de uma variável define o conjunto de valores que a variável pode assumir. Por exemplo, uma variável do tipo *lógico* pode assumir o valor *verdadeiro* ou *falso*.

Uma declaração de variável em uma linguagem como *C* ou *Pascal* especifica:

1. O conjunto de valores que pode assumir.
2. O conjunto de operações que podemos efetuar.
3. A quantidade de bytes que deve ser reservada para ela.
4. Como o dado representado por esses bytes deve ser interpretado (por exemplo, uma cadeia de bits pode ser interpretada como um inteiro ou real...).

Então, tipos de dados podem ser vistos como métodos para interpretar o conteúdo da memória do computador.

Mas podemos ver o conceito de **Tipo de Dados** de uma outra perspectiva: não em termos do que um **computador** pode fazer (interpretar os bits...) mas em termos do que os **usuários** desejam fazer (somar dois inteiros...)

Este conceito de **Tipo de Dado** divorciado do hardware é chamado **Tipo Abstrato de Dado - TAD.**

Estrutura de Dados é um método particular de se implementar um **TAD**.

A **implementação** de um **TAD** escolhe uma **ED** para representá-lo. Cada **ED** é construída dos tipos primitivos (inteiro, real, char,...) ou dos tipos compostos (array, registro,...) de uma linguagem de programação.

Não importa que tipo de dados estaremos trabalhando, a primeira operação a ser efetuada em um TAD é a **criação**. Depois, podemos realizar **inclusões** e **remoções** de dados. A operação que varre todos os dados armazenados num TAD é o **percurso**, podendo também ser realizada uma **busca** por algum valor dentro da estrutura.

Exemplos de TAD:

Lineares:

- Listas Ordenadas
- Pilhas
- Filas
- Deques

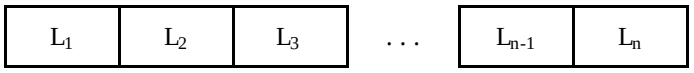
Não Lineares:

- Árvores
- Grafos

LISTAS

São estruturas formadas por um conjunto de dados de forma a preservar a relação de ordem linear entre eles. Uma lista é composta por nós, os quais podem conter, cada um deles, um dado primitivo ou composto.

Representação:



Sendo:

L_1 ✍ 1º elemento da lista
L_2 ✍ Sucessor de L_1
L_{n-1} ✍ Antecessor de L_{n-1}
L_n ✍ Último elemento da lista.

Exemplos de Lista:

- Lista Telefônica
- Lista de clientes de uma agência bancária
- Lista de setores de disco a serem acessados por um sistema operacional
- Lista de pacotes a serem transmitidos em um nó de uma rede de computação de pacotes.

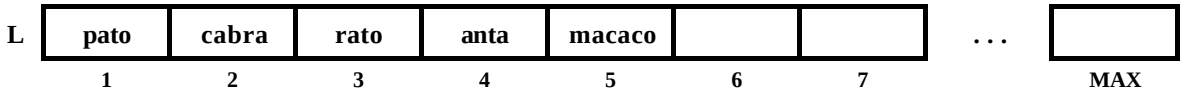
Operações Realizadas com Listas:

- Criar uma lista vazia
- Verificar se uma lista está vazia
- Obter o tamanho da uma lista
- Obter/modificar o valor do elemento de uma determinada posição na lista
- Obter a posição de elemento cujo valor é dado
- Inserir um novo elemento após (ou antes) de uma determinada posição na lista
- Remover um elemento de uma determinada posição na lista
- Exibir os elementos de uma lista
- Concatenar duas listas

FORMAS DE REPRESENTAÇÃO:

a) Seqüencial:

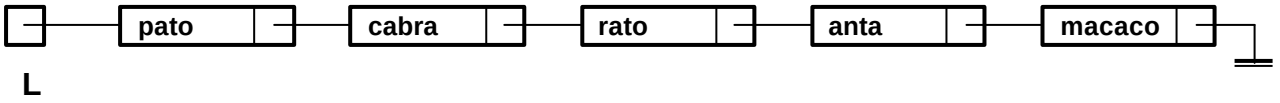
Explora a sequencialidade da memória do computador, de tal forma que os nós de uma lista sejam armazenados em endereços sequenciais, ou igualmente distanciados um do outro. Pode ser representado por um vetor na memória principal ou um arquivo seqüencial em disco.



b) Encadeada:

Esta estrutura é tida como uma seqüência de elementos encadeados por ponteiros, ou seja, cada elemento deve conter, além do dado propriamente dito, uma referência para o próximo elemento da lista.

Ex: L = pato, cabra, rato, anta, macaco



LISTA SEQÜENCIAL

Uma lista representada de forma seqüencial é um conjunto de registros onde estão estabelecidas regras de precedência entre seus elementos. O sucessor de um elemento ocupa posição física subsequente.

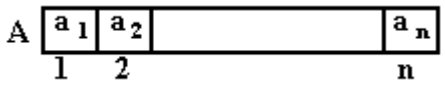
A implementação de operações pode ser feita utilizando *array* e *registro*, associando o elemento a(i) com o índice i (mapeamento seqüencial).

CARACTERÍSTICAS

- os elementos na lista estão armazenados fisicamente em posições consecutivas;
- a inserção de um elemento na posição a(i) causa o deslocamento a direita do elemento de a(i) ao último;
- a eliminação do elemento a(i) requer o deslocamento à esquerda do a(i+1) ao último;

Mas, absolutamente, uma lista seqüencial ou é vazia ou pode ser escrita como (a(1), a(2), a(3), ... a(n)) onde a(i) são átomos de um mesmo conjunto A.

Além disso, a(1) é o primeiro elemento, a(i) precede a(i+1), e a(n) é o último elemento.



Assim as propriedades estruturadas da lista permitem responder a questões tais como:

- se uma lista está vazia
- se uma lista está cheia
- quantos elementos existem na lista
- qual é o elemento de uma determinada posição
- qual a posição de um determinado elemento
- inserir um elemento na lista
- eliminar um elemento da lista

Conseqüência:
As quatro primeiras operações são feitas em tempo constante. As demais, porém, requererão mais cuidados.

Vantagem:

- acesso direto indexado a qualquer elemento da lista
- tempo constante para acessar o elemento *i* - dependerá somente do índice.

Desvantagem:

- movimentação quando eliminado/inserido elemento
- tamanho máximo pré-estimado

Quando usar:

- listas pequenas
- inserção/remoção no fim da lista
- tamanho máximo bem definido

Vamos tentar evitar as desvantagens anteriores ao usar endereços não consecutivos (Lista Encadeada).

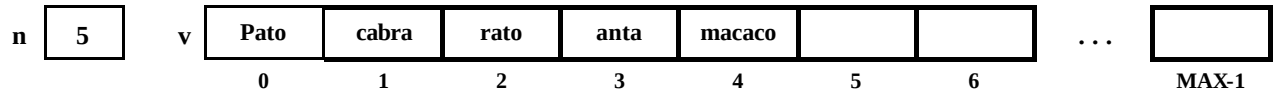
OPERAÇÕES BÁSICAS

A seguir, apresentaremos a estrutura de dados e as operações básicas sobre listas, implementadas na linguagem C.

Definição da ED:

```
#define MAX _________     /* tamanho máximo da lista */

typedef ______ telem;    /* tipo base dos elementos da lista */
typedef struct {
           telem v[MAX]; /* vetor que contém a lista */
           int n;        /* posição do último elemento da lista */
       } tlista;         /* tipo lista */
```

tlista



Operações simples utilizando lista seqüencial:

1) Criar uma lista vazia

```
void criar (tlista *L) {
   L->n = 0;
}
```

2) Verificar se uma lista está vazia

```
int vazia (tlista L) {
   return (L.n == 0);
}
```

3) Verificar se uma lista está cheia

```
int cheia (tlista L) {
   return (L.n == MAX);
}
```

4) Obter o tamanho de uma lista

```
int tamanho (tlista L) {
   return (L.n);
}
```

5) Obter o i-ésimo elemento de uma lista

```
int elemento (tlista L, int pos, telem *dado) {

   /* O parâmetro dado irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   if ( (pos > L.n) || (pos <= 0) ) return (0);

   *dado = L.v[pos-1];
   return (1);
}
```

6) Pesquisar um dado elemento, retornando a sua posição

```
int posicao (tlista L, telem dado) {

  /* Retorna a posição do elemento ou 0 caso não seja encontrado */

  int i;

  for (i=1; i<=L.n; i++)
     if (L.v[i-1] == dado)
        return (i);

  return (0);
}
```

7) Inserção de um elemento em uma determinada posição

Requer o deslocamento à direita dos elementos v(i+1)...v(n)

```
int inserir (tlista *L, int pos, telem dado) {

  /* Retorna 0 se a posição for inválida ou se a lista estiver cheia */
  /* Caso contrário, retorna 1 */

  int i;

  if ( (L->n == MAX)) || (pos > L->n + 1) )  return (0);

  for (i=L->n; i>=pos; i--)
     L->v[i] = L->v[i-1];

  L->v[pos-1] = dado;
  (L->n)++;
  return (1);
}
```

8) Remoção do elemento de uma determinada posição

Requer o deslocamento à esquerda dos elementos v(p+1)...v(n)

```
int remover (tlista *L, int pos, telem *dado) {

  /* O parâmetro dado irá receber o elemento encontrado */
  /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

  int i;

  if ( (pos > L->n) || (pos <= 0) ) return (0);

  *dado = L->v[pos-1];

  for (i=pos; i<=(L->n)-1; i++)
     L->v[i-1] = L->v[i];

  (L->n)--;
  return (1);
}
```

PRÁTICA DE LABORATÓRIO

01. Faça um programa que:
    a) crie uma lista L;
    b) exiba o seguinte menu de opções:

```
EDITOR DE LISTAS

1 – EXIBIR LISTA
2 – INSERIR
3 – REMOVER
4 – EXIBIR ELEMENTO
5 – EXIBIR POSIÇÃO
6 – ESVAZIAR
ESC – SAIR

DIGITE SUA OPÇÃO:
```

    c) leia a opção do usuário;
    d) execute a opção escolhida pelo usuário;
    e) implemente a estrutura de dados LISTA em uma biblioteca chamada L_SEQ (com implementação seqüencial e usando o **inteiro** como tipo base), contendo apenas as operações básicas de listas (citadas anteriormente);
    f) na opção de exibir lista, devem ser exibidos o tamanho da lista e os seus elementos;
    g) na opção de inserção, deve ser lido o valor do elemento a ser inserido e a posição onde será efetuada a inserção;
    h) na opção de remoção, deve ser lido a posição do elemento a ser removido;
    i) na opção de exibir elemento, deve ser lido a posição do elemento;
    j) na opção de exibir posição, deve ser lido o valor do elemento;
    k) as operações de exibir e esvaziar a lista devem estar inseridas no programa principal;
    l) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

EXERCÍCIOS – LISTA SEQUENCIAL

01. Inclua, na biblioteca L_SEQ, as funções abaixo especificadas (obs: não faça uso das funções já implementadas).
    a) inserir um dado elemento na primeira posição de uma lista;
    b) inserir um dado elemento na última posição de uma lista;
    c) modificar um elemento de uma lista, sendo dado a sua posição e o novo valor;
    d) remover o primeiro elemento de uma lista;
    e) remover o último elemento de uma lista;
    f) remover um elemento dado o seu valor.

02. Modifique o seu programa EDITOR DE LISTAS, adicionando todas as operações relacionadas na questão anterior.

LISTAS ORDENADAS

São listas lineares onde os elementos estão ordenados segundo um critério pré-estabelecido. Na realidade, as listas ordenadas diferem das listas genéricas pelo fato de que cada novo elemento a ser inserido ocupará uma posição específica, obedecendo à ordenação dos valores já existentes.

LISTA ORDENADA SEQÜENCIAL

DEFINIÇÃO DA ED

```
#define MAX _______                 /* tamanho máximo da lista */

typedef _____ telem;                /* tipo base dos elementos da lista */

typedef struct {
                  telem v[MAX];     /* vetor que contém a lista */
                  int n;            /* posição do último elemento da lista */
} tlistaord;                        /* tipo lista ordenada */
```

OPERAÇÕES BÁSICAS

1) Criar uma lista vazia

```
void criar (tlistaord *L)
{
   L->n = 0;
}
```

2) Verificar se uma lista está vazia

```
int vazia (tlistaord L)
{
   return (L.n == 0);
}
```

3) Verificar se uma lista está cheia

```
int cheia (tlistaord L)
{
   return (L.n == MAX);
}
```

4) Obter o tamanho de uma lista

```
int tamanho (tlistaord L)
{
   return (L.n);
}
```

5) Obter o i-ésimo elemento de uma lista

```
int elemento (tlistaord L, int pos, telem *dado)
{
   /* O parâmetro dado irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   if ( (pos > L.n) || (pos <= 0) ) return (0);
   *dado = L.v[pos-1];
   return (1);
}
```

6) Pesquisar um dado elemento, retornando a sua posição

```
int posicao (tlistaord L, telem dado)
{
   /* Retorna a posição do elemento ou 0 caso não seja encontrado */

   int i;

   for (i=1; i<=L.n; i++)
      if (L.v[i-1] == dado)
         return (i);

   return (0);
}
```

7) Inserção de um elemento dado

```
int inserir (tlistaord *L, telem dado)
{
   /* Retorna 0 se a posição for inválida ou se a lista estiver cheia */
   /* Caso contrário, retorna 1 */

   int i, pos;

   if (L->n == MAX) return (0); /* erro: lista cheia */

   for (i=0; i<L->n; i++) {
      if (L->v[i] == dado) return (0); /* erro: dado já existente */
      if (L->v[i] > dado) break;
   }

   pos = i;

   for (i=L->n; i>pos; i--)
      L->v[i] = L->v[i-1];

   L->v[i] = dado;
   (L->n)++;

   return (1);
}
```

8) Remoção do elemento de uma determinada posição

```
int remover (tlistaord *L, int pos, telem *dado)
{
   /* O parâmetro dado irá receber o elemento encontrado */
   /* Retorna 0 se a posição for inválida. Caso contrário, retorna 1 */

   int i;

   if ( (pos > L->n)) || (pos <= 0) ) return (0); /* erro: posição inválida */

   *dado = L->v[pos-1];

   for (i=pos; i<=(L->n)-1; i++)
      L->v[i-1] = L->v[i];

   (L->n)--;

   return (1);
}
```

EXERCÍCIOS – LISTAS ORDENADAS

01. Implemente a TAD Lista Ordenada com representação seqüencial em uma biblioteca chamada LORD_SEQ (usando como base o tipo string), contendo apenas a estrutura de dados e as operações básicas de listas ordenadas (descritas anteriormente).

02. Faça um programa que, utilizando a biblioteca criada no item anterior:
    a) crie uma lista ordenada L;
    b) exiba o seguinte menu de opções:

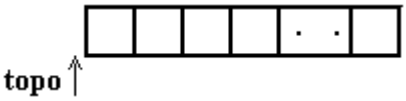
```
EDITOR DE LISTAS ORDENADAS

   1 – EXIBIR LISTA
   2 – INSERIR
   3 – REMOVER
   4 – EXIBIR ELEMENTO
   5 – EXIBIR POSIÇÃO
   6 – ESVAZIAR
ESC – SAIR

DIGITE SUA OPÇÃO:
```

    c) leia a opção do usuário;
    d) execute a opção escolhida pelo usuário;
    e) na opção de exibir lista, devem ser exibidos o tamanho da lista e os seus elementos;
    f) na opção de inserção, deve ser lido o valor do elemento a ser inserido;
    g) na opção de remoção, deve ser lido a posição do elemento a ser removido;
    h) na opção de exibir elemento, deve ser lido a posição do elemento;
    i) na opção de exibir posição, deve ser lido o valor do elemento;
    j) as operações de exibir e esvaziar a lista devem estar inseridas no programa principal;
    k) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

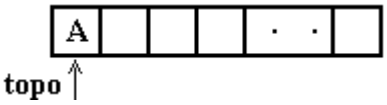
PILHAS

Pilhas são listas onde a inserção de um novo item ou a remoção de um item já existente se dá em uma única extremidade, no topo.

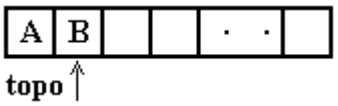
Pilha vazia



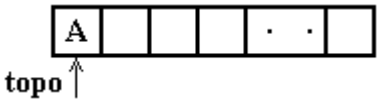
Insere (A)



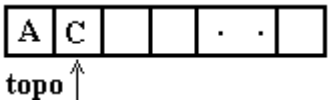
Insere (B)



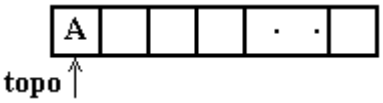
Retira (B)



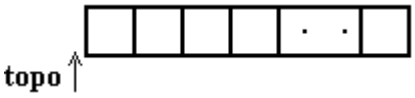
Insere (C)



Retira (C)



Retira (A)



Definição:

Dada uma pilha P = (a(1), a(2), ..., a(n)), dizemos que a(1) é o elemento da base da pilha; a(n) é o elemento topo da pilha; e a(i+1) está acima de a(i).

Pilhas são também conhecidas como listas **LIFO** (last in first out).

Operações Associadas:

1. Criar uma pilha P vazia
2. Testar se P está vazia
3. Obter o elemento do topo da pilha (sem eliminar)
4. Inserir um novo elemento no topo de P (empilhar)
5. Remover o elemento do topo de P (desempilhar)

Implementação de Pilhas

Como lista **Seqüencial** ou **Encadeada**?

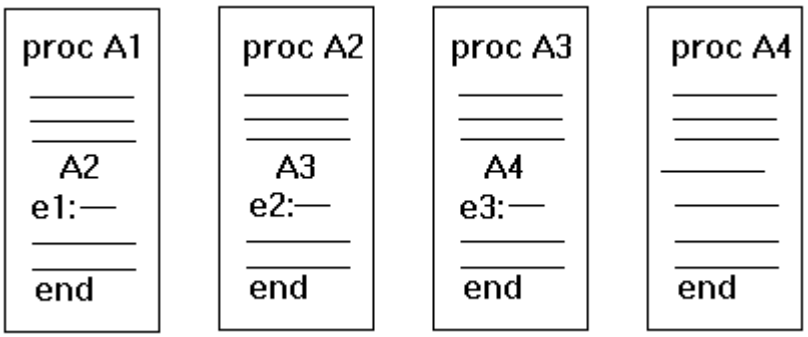
No caso geral de listas ordenadas, a maior vantagem da alocação encadeada sobre a seqüencial - se a memória não for problema - é a eliminação de deslocamentos na inserção ou eliminação dos elementos. No caso das pilhas, essas operações de deslocamento não ocorrem.

Portanto, podemos dizer que a alocação sequencial é mais vantajosa na maioria das vezes.

Exemplo do Uso de Pilhas

Chamadas de procedimentos

Suponha a seguinte situação:



Quando o procedimento A1 é executado, ele efetua uma chamada a A2, que deve carregar consigo o endereço de retorno e1. Ao término de A2, o processamento deve retornar ao A1, no devido endereço. Situação idêntica ocorre em A2 e A3.

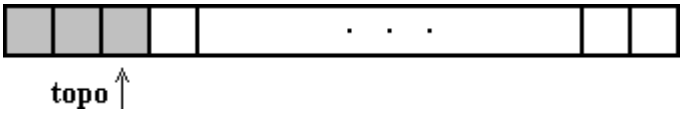
Assim, quando um procedimento termina, é o seu endereço de retorno que deve ser consultado. Portanto, há uma lista **implícita** de endereços (e0, e1, e2, e3) que deve ser manipulada como uma **pilha** pelo sistema, onde e0 é o endereço de retorno de A1.

No caso de processamento recursivo - por exemplo uma chamada a A2 dentro de A4 - o gerenciamento da lista como uma **pilha** resolve automaticamente a obtenção dos endereços de retorno na ordem apropriada (e0, e1, e2, e3, e4).

ALOCAÇÃO SEQÜENCIAL DE PILHAS

Definição da Estrutura de Dados:

```
#define MAX 10

typedef int telem;

typedef struct {
             telem v[MAX];
             int topo;
        } tpilha;
```



OPERAÇÕES:

1) Criar uma pilha vazia

```
void criar (tpilha *p)
{
   p->topo = -1;
}
```

2) Testar se a pilha está vazia

```
int vazia (tpilha p)
{
   return (p.topo == -1);
}
```

3) Obter o elemento do topo da pilha (sem eliminar)

```
int elemtopo (tpilha p, telem *valor)
{
   if (vazia(p)) return 0;
   *valor = p.v[p.topo];
   return 1;
}
```

4) Inserir um novo elemento no topo da pilha (empilhar)

```
int push (tpilha *p, telem valor)
{
   if (p->topo == MAX-1) return 0;
   p->v[>topo] = valor;
   return 1;
}
```

5) Remove o elemento do topo da pilha (desempilhar), retornando o elemento removido

```
int pop (tpilha *p, telem *valor)
{
   if (vazia(*p)) return 0;
   *valor = p->v[p->topo--];
   return 1;
}
```

EXERCÍCIOS – PILHAS

1. Implemente a TAD **Pilha** com **representação seqüencial** em uma biblioteca chamada **P_SEQ** (usando como tipo base o **char**), contendo apenas a estrutura de dados e as operações básicas de pilhas (descritas anteriormente).

2. Faça um programa que, utilizando a biblioteca criada no item anterior:
   a) crie uma pilha P;
   b) exiba o seguinte menu de opções:

> **EDITOR DE PILHA**
>
> 1 – EMPILHAR
> 2 – DESEMPILHAR
> 3 – EXIBIR ELEMENTO DO TOPO
> 4 – EXIBIR A PILHA
> 5 – ESVAZIAR A PILHA
>
> DIGITE SUA OPÇÃO:

   c) leia a opção do usuário;
   d) execute a opção escolhida pelo usuário;
   e) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).

FILAS

É uma lista linear em que a inserção é feita numa extremidade e a eliminação na outra. Conhecida com estrutura FIFO (*First In, First Out*).

```
  ( a1, a2  , ... , an )
```
eliminações no início — inserções no final

Exemplos:

- ✍ Escalonamento de "Jobs": fila de processos aguardando os recursos do sistema operacional.
- ✍ Fila de pacotes a serem transmitidos numa rede de comutação de pacotes.
- ✍ Simulação: fila de caixa em banco.

Operações associadas:

1. **Criar** - cria uma fila vazia
2. **Vazia** - testa se um fila está vazia
3. **Primeiro** - obtém o elemento do início de uma fila
4. **Inserir** - insere um elemento no fim de uma fila
5. **Remover** - remove o elemento do início de uma fila, retornando o elemento removido.

Implementação de Filas

Como lista **Seqüencial** ou **Encadeada** ?

Pelas suas características, as filas têm as eliminações feitas no seu início e as inserções feitas no seu final. A implementação **encadeada dinâmica** torna mais simples as operações (usando uma lista de duas cabeças). Já a implementação **seqüencial** é um pouco mais complexa (teremos que usar o conceito de fila circular), mas pode ser usada quando há previsão do tamanho máximo da fila.

IMPLEMENTAÇÃO SEQÜENCIAL DE FILA

Definição da Estrutura de Dados:

Devido a sua estrutura, será necessária a utilização de dois campos que armazenarão os índices do início e do final da fila e um vetor de elementos (onde serão armazenados os dados) com tamanho pré-estabelecido.

```
#define MAX ______      /* tamanho máximo da fila */

typedef _____ telem;    /* tipo base dos elementos da fila */

typedef struct{

         telem v[MAX];
         int inicio;    /* posição do primeiro elemento */
         int final;     /* posição do último elemento  */

      } tfila;
```

OPERAÇÕES COM FILAS:

1. Criar - cria uma fila vazia

```
void criar (tfila *F)
{
   F->inicio = 0;
   F->final = -1;
}
```

2. Vazia - testa se uma fila está vazia

```
int vazia (tfila F)
{
   return (F.inicio > F.final);
}
```

3. Primeiro - obtém o elemento do início da fila

```
int primeiro (tfila F, telem *dado)
{
   if (vazia(F)) return 0;  /* Erro: fila vazia */

   *dado = F.v[F.inicio];

   return (1);
}
```

4. Insere - insere um elemento no fim de uma fila

```
int inserir (tfila *F, telem valor)
{
   if (F->final == MAX-1) return 0;

   (F->final)++;
   F->v[F->final] = valor;

   return(1);
}
```

5. Remove - remove o elemento do início de uma fila, retornando o elemento removido

```
int remover (tfila *F, telem *valor)
{
   if (vazia(*F)) return 0;  /* Erro: fila vazia */

   primeiro(*F,valor);
   (F->inicio)++;

   return(1);
}
```

Problema na implementação seqüencial

O que acontece com a fila considerando a seguinte seqüência de operações sobre um fila:

I E I E I E I E I E ...

(I - inserção e E - eliminação)

Note que a fila vai se deslocando da esquerda para a direita do vetor. Chegará a condição de "overflow" (cheia), porém estando vazia, ou seja, sem nenhum elemento.

Alternativa:

No algoritmo de remoção, após a atualização de **inicio**, verificar se a fila ficou vazia. Se este for o caso, reinicializar **inicio = 0** e **final = -1**

Portanto, ficaria:

```
int remover (tfila *F, telem *valor) {

   if (vazia(*F)) return 0;  /* Erro: fila vazia */

   primeiro(*F,valor);
   (F->inicio)++;

   if (vazia(*F)) {
      F->inicio = 0;
      F->final = -1;
   }

   return(1);
}
```

O que aconteceria se a seqüência fosse:

I I E I E I E I E I ...

A lista estaria com no máximo dois elementos, mas ainda ocorreria overflow com a lista quase vazia.

Alternativa:

Forçar **final** a usar o espaço liberado por **inicio** (**Fila Circular**)

FILA CIRCULAR

Para permitir a reutilização das posições já ocupadas, usa-se o conceito de "Fila Circular". Precisamos de um novo componente para indicar quantos elementos existem na fila, no momento.

A estrutura de dados com o novo componente ficaria assim representada:

```
#define MAX ______      /* tamanho máximo da fila */

typedef _____ telem;    /* tipo base dos elementos da fila */

typedef struct{
         telem v[MAX];
         int inicio;    /* posição do primeiro elemento */
         int final;     /* posição do último elemento  */
         int tam;       /* número de elementos da fila */
      } tfila;
```

As operações são agora executadas assim:

1. Criar - cria uma fila vazia

```
void criar (tfila *F)
{
   F->inicio = 0;
   F->final = -1;
   F->tam = 0;
}
```

2. Vazia - testa se uma fila está vazia

```
int vazia (tfila F)
{
   return (F.tam == 0);
}
```

3. Primeiro - obtém o elemento do início da fila

```
int primeiro (tfila F, telem *dado)
{
   if (vazia(F)) return 0;  /* Erro: fila vazia */

   *dado = F.v[F.inicio];

   return (1);
}
```

4. Inserir - insere um elemento no fim de uma fila

```
int inserir (tfila *F, telem valor)
{
   if (F->tam == MAX) return 0;

   (F->tam)++;
   F->final = (F->final + 1) % MAX;
   F->v[F->final] = valor;

   return(1);
}
```

5. Remover - remove o elemento do início de uma fila, retornando o elemento removido

```
int remover (tfila *F, telem *valor)
{
   if (vazia(*F)) return 0;  /* Erro: fila vazia */

   primeiro(*F,valor);
   (F->tam)--;
   F->inicio = (F->inicio + 1) % MAX;

   return(1);
}
```

EXERCÍCIOS – FILAS

1) Implemente o TAD **Fila** com **representação seqüencial** em uma biblioteca chamada **F_SEQ** (usando como tipo base o tipo **inteiro**), contendo apenas a estrutura de dados e as operações básicas de filas (descritas anteriormente).

2) Faça um programa que, utilizando a biblioteca criada no item anterior:
   a) crie uma fila F;
   b) exiba o seguinte menu de opções:

> **EDITOR DE FILA**
>
> 1 – INSERIR
> 2 – REMOVER
> 3 – EXIBIR PRIMEIRO ELEMENTO
> 4 – EXIBIR A FILA
> 5 – ESVAZIAR A FILA
>
> DIGITE SUA OPÇÃO:

   c) leia a opção do usuário;
   d) execute a opção escolhida pelo usuário;
   e) após a execução de cada opção, o programa deve retornar ao menu para nova opção do usuário ou o encerramento do programa (através da tecla ESC).