7. ESTRUTURAS DE DADOS ESTÁTICAS E DINÂMICAS

Até .....

7.1 Alocação Estática

Alocação de Espaço em Memória

Como já foi visto anteriormente, a memória de um computador compõe-se de uma sequência de palavras, ou bytes, endereçada. Nesta memória são colocados todos os programas executados pelo computador.

Os programas são compostos por código (instruções) e dados (variáveis). Estes programas são carregados na memória quando forem chamados para execução e são retirados da memória quando for finalizada a sua execução. Assim, em determinado momento, a memória está com partes ocupadas e livres.

A utilização da memória é controlada pelo Sistema Operacional. O Sistema Operacional mantém uma tabela que informa quais partes da memória estão livres (memória disponível) e quais partes da memória estão ocupadas (memória ocupada).

Toda vez que um programa for executado, o Sistema Operacional aloca (reserva), da memória disponível, o espaço suficiente para as variáveis deste programa. Esta memória passa a ser ocupada e não será mais alocada para outras variáveis, até que o programa termine. Desta forma, as memórias disponível e ocupada crescem e diminuem a medida que diferentes programas são executados.

Existem 2 (dois) tipos de variáveis: estática e dinâmica. Veremos aqui somente as variáveis estáticas, pois as variáveis dinâmicas serão objetos de estudo posterior.

As variáveis estáticas são alocadas antes que o programa entre em execução. O programa solicita ao Sistema Operacional que aloque espaço da memória disponível para as variáveis estáticas. Então, o Sistema Operacional retira da memória disponível o espaço necessário para as variáveis e coloca na memória ocupada.

IMPORTANTE!!! As variáveis geralmente são alocadas na mesma ordem em que elas aparecem na declaração.

O Sistema Operacional garante que o espaço de memória utilizado por uma Var estática jamais poderá ser utilizado por uma outra. Esta garantia não é dada às variáveis dinâmicas, como veremos.

Para cada Var será alocada uma quantidade de espaço diferente, dependendo do tipo da Var e da linguagem de programação utilizada. A seguir descreveremos o tamanho de memória utilizada pelas variáveis de tipos comumente conhecidos.

tipo **Integer** = 2 bytes
tipo **Real** = 6 bytes
tipo **String** [n]= n bytes
tipo **Char** = 1 byte
tipo **Boolean** = 1 byte
tipo **Record** = somatória dos tamanhos dos campos

ex: **Record**
nome: **String** [30] **;**
idade: **Integer ;**
sexo: **Char ;**
salário**: Real ;**
End;

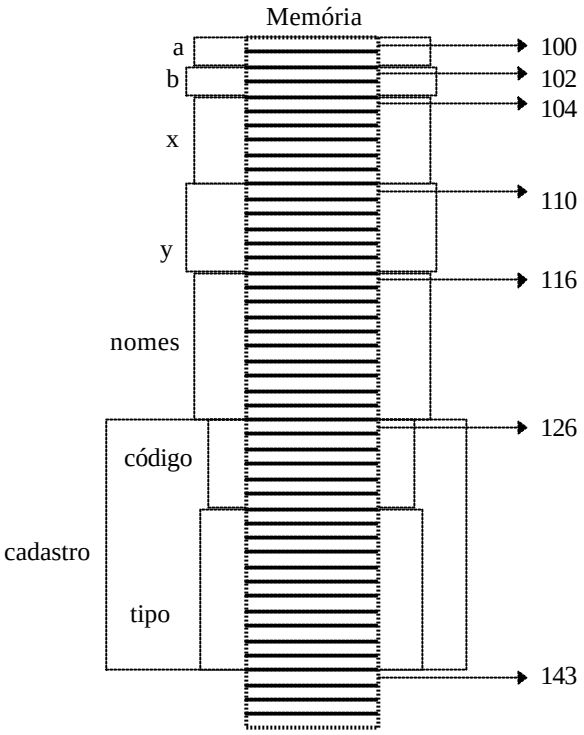
somatória = 30 + 1 + 1 + 6 = 38 bytes

A memória alocada para uma variável vetorial depende da quantidade de campos e do tipo do campo. Alguns exemplos podem ser dados:

V: **Array**[1..100] **of Real ;** total de memória = 100*6 = 600 bytes
A: **Array**[1..20,1..50] **of Integer ;** total de memória = 20*50*2 = 2000 bytes ≈ 2Kbytes
Nomes: **Array**[1..2000] **of Record** total = 2000*38 = 76000 bytes ≈ 76Kbytes
nome**: String** [30] **;**
idade**: Integer ;**
sexo**: Char ;**
salário**: Real ;**
End;

Exercício: Faça um mapa de memória para as variáveis declaradas abaixo. Suponha que a memória disponível seja contínua a partir da posição 100.

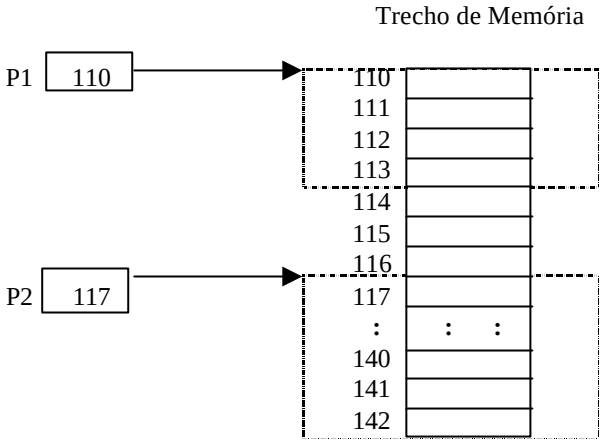
Var a, b**: Integer ;**
x, y**: Real ;**
nomes**: String** [10] **;**
cadastro : **Record**
código**: Real ;**
tipo**: String** [11] **;**
End;



Variáveis Dinâmicas (Ponteiros)

Ponteiros são variáveis que **"apontam para dados na memória",** ou seja, contém o endereço inicial de uma região da memória que pode armazenar um dado. O tamanho desta região depende do tipo do dado **"apontado"** pelo ponteiro.

A variável ponteiro, ou simplesmente ponteiro, pode ter 2 ou mais bytes de tamanho, suficiente para armazenar um endereço de memória. Em nossos estudos, utilizaremos ponteiros de 2 bytes. Sua representação pode ser vista na figura abaixo, onde é representado 2 ponteiros: P1 e P2.



O ponteiro P1, contém o valor 110, indicando que ele está "apontando" para uma região de memória com 4 bytes de tamanho, a partir da posição 110. Já o ponteiro P2 contém o valor 117, indicando que ele está "apontando" para uma região da memória com 26 bytes de tamanho, a partir da posição 117.

A declaração destas variáveis deve ser feita da seguinte forma:

var <nome> **:** ^ <tipo> **;**

Exemplo:

```
var   nomes : ^ string[20] ;     (* ponteiro para literal de 20 bytes *)
      fichas : ^ record      (* ponteiro para registros de 22 bytes *)
                    nome: string[20] ;
                     idade:  integer ;
                end;
```

O símbolo ^ é chamado de ponteiro (também "chapô"). Assim, na declaração de variáveis dinâmicas dizemos que "declaramos um ponteiro para algo", onde o nome utilizado na declaração indica o nome do ponteiro.

No exemplo acima, a variável "nomes" é um ponteiro que **poderá** "apontar" para uma informação (literal) de 20 bytes de tamanho. Ja a variável "fichas" é um ponteiro que **poderá** "apontar" para uma informação (registro) de 22 bytes de tamanho.

A declaração de ponteiros, por si só, **NÃO ALOCA ESPAÇO DE MEMÓRIA** para os dados que eles apontam. A alocação destes espaços deve ser feita em tempo de execução e de forma explícita, ou seja, através dos comandos **New** e **Dispose**. O único espaço alocado pelo Sistema Operacional, durante a declaração de ponteiros, é o espaço ocupado por eles próprios (2 bytes para cada um).

Após a utilização do comando **New**, será reservado um espaço de memória de tamanho suficiente para acomodar dados do tipo definido na declaração do ponteiro. O nome do ponteiro conterá o endereço inicial da memória alocada.

Sempre que não for mais necessário o espaço alocado para uma variável dinâmica, o mesmo deve ser liberado através do comando **Dispose**, para que possa ser utilizado por outras variáveis dinâmicas.

Exemplo: Utilizando a declaração feita anteriormente, poderíamos ter os seguintes comandos:

```
  :
new (nomes) ;
new (fichas) ;
   :    :
(* usa as variáveis *)
   :    :
dispose (nomes) ;
dispose (fichas);
  :
```

No exemplo acima, o comando "**new** (nomes)" fará com que o Sistema Operacional reserve uma região da memória que estava disponível para armazenar um literal de 20 bytes. O endereço inicial desta região será retornado para o ponteiro "nomes". Já o comando **dispose** (nomes) fará com que a região da memória apontada pelo ponteiro "nomes" seja colocada novamente a disposição pelo Sistema Operacional.

IMPORTANTE!!! A manipulação das variáveis dinâmicas deverá ser feita somente após as mesmas terem sido alocadas. A não observação desta condição poderá implicar no uso de espaço de memória que já está sendo utilizado por outras variáveis.

Após a alocação de espaço de memória para um ponteiro, deveremos manipular os dados por ele apontado utilizando o símbolo ^ agregado ao nome do ponteiro. De acordo com a declaração anterior, a manipulação das variáveis dinâmicas após a sua criação poderia ser:

```
  :
read (nomes^) ;
write (nomes^) ;
fichas^. nome := nomes^ ;
read (fichas^. Idade) ;
write (fichas ^. Idade) ;
  :
```

IMPORTANTE!!! O uso somente do nome do ponteiro implica na manipulação apenas de um endereço de memória e não dos dados armazenados.

IMPORTANTE!!! O uso mal elaborado de ponteiros pode acarretar espaço de memória perdido.

Perguntas:

1) Qual é o valor de um ponteiro que foi somente declarado?

2) O que acontece se utilizarmos um ponteiro que foi somente declarado?

3) O que acontece quando um ponteiro recebe outro ponteiro?

4) Quando temos dois ponteiros para a mesma posição da memória, o que acontece quando liberamos um deles?

5) O que significa a posição de memória chamada **Nil** (Terra) ?

Estruturas de Dados Avançadas

Entre as principais estruturas de dados avançadas temos: pilhas, filas, listas e árvores. Estas estruturas armazenam dados e são manipuladas por funções básicas do tipo: cria, insere, elimina, consulta e altera.

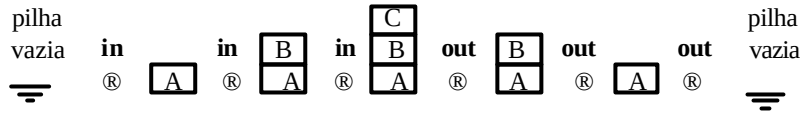
Estas estruturas podem ser implementadas tanto da forma estática quanto da forma dinâmica. Implementaremos somente as estruturas que devem ser declaradas, bem como os procedimentos e funções de manipulação. O algoritmo de utilização das estruturas, procedimentos e funções ficará para desenvolvimento posterior.

Pilha:

Uma pilha pode ser vista como um local onde pode ser inserido dados um sobre o outro, inserindo sempre sobre o último da pilha (topo da pilha). Para retirarmos os dados, devemos também respeitar a ordem pelo qual eles foram inseridos, ou seja, retirando somente o último elemento do topo. Este tipo de estrutura é também conhecida como **LIFO** ("Last In First Out" = "Último a Entrar é o Primeiro a Sair") ou **FILO** ("First In Last Out" = "Primeiro a Entrar é o Último a Sair").

As estruturas de pilha são comumente usadas em algoritmos de gerenciamento de memória (escopo de variáveis e retorno de procedimentos), compiladores e em outras aplicações que serão vistas em disciplinas futuras.

Quando uma pilha é criada, não existe nenhum elemento inserido e dizemos que ela está vazia. A figura abaixo mostra algumas situações em que uma pilha pode estar.



Implementação Estática:

Usamos uma estrutura vetorial para representar a pilha. Como a declaração de um vetor requer um tamanho máximo da estrutura, não podemos inserir elementos quando a estrutura já estiver cheia. Usamos uma variável "topo" para indicar a posição do último elemento.

```
Const   max_pilha = 100 ;
Type    tipo_dado = Integer;
        tipo_pilha = Record
                       pilha: array[1..max_pilha] of tipo_dado ;
                       topo: Integer ;
                     End ;

Procedure Cria_Pilha(var p : tipo_pilha);
begin
  p.topo := 0 ;
end ;

Function Pilha_Cheia(p: tipo_pilha): boolean ;
begin
  Pilha_Cheia := (p.topo = max_pilha) ;
end;

Function Pilha_Vazia (p: tipo_pilha) : boolean;
begin
  Pilha_Vazia := (p.topo = 0) ;
end ;
```

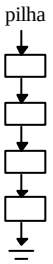
```
Function Empilha(var p: tipo_pilha ; x: tipo_dado): boolean ;
begin
  if  Pilha_Cheia(p)
     then Empilha := false
```

```
        Else begin
                Empilha := True;
                Inc (p.topo);
                p.pilha[p.topo] := x ;
             end;
end ;

Function Desempilha(var p: tipo_pilha ; var x : tipo_dado): Boolean ;
Begin
  If Vazia(p)
     Then Desempilha := false
     Else begin
             Desempilha := True;
             x := p.pilha[p.topo];
             dec (p.topo);
          end;
end ;

Function Topo(p: tipo_pilha): tipo_dado ;
begin
  Topo := p.pilha[p.topo] ;
end ;
```

Implementação Dinâmica:

Para implementarmos esta estrutura usando ponteiros, devemos estabelecer um módulo básico que possa ser alocado a cada nova inserção, e que permita a formação da pilha de elementos em tempo de execução.

Como estes módulos serão estruturas criadas em tempo de execução, para não perdermos o controle das diversas posições que estiverem sendo alocadas, devemos fazer com que cada módulo aponte para o seu próximo, formando uma "**pilha encadeada**". Neste sentido, para controlarmos a pilha, precisaríamos apenas do ponteiro inicial da cadeia, onde o último elemento apontaria para a posição Terra, conforme mostra a figura abaixo:



Devemos observar que não precisamos nos preocupar se a estrutura está cheia ou não, pois ela não é limitada como os vetores, a menos da quantidade de memória real disponível. Esta estrutura pode ser declarada da seguinte forma:

```
type    tipo_dado = integer ;
        tipo_pilha = ^tipo_no;
        tipo_no = record
                    dado: tipo_dado ;
                    prox: tipo_pilha ;
                  end ;
```

Os procedimentos/funções podem ser implementados da seguinte forma:

```
Procedure Cria_Pilha(var p : tipo_pilha);
begin
  p := Nil ;
end ;

Function Vazia (p: tipo_pilha) : boolean;
begin
  Vazia := (p = Nil) ;
end ;

procedure Empilha(var p: tipo_pilha ; x: tipo_dado) ;
var aux : tipo_pilha ;
Begin
  new(aux);
  aux^.dado := x ;
  aux^.prox := p ;
  p := aux ;
end ;

function Desempilha(var p: tipo_pilha ; var x : tipo_dado): boolean ;
var  aux : tipo_pilha ;
begin
  If Vazia(p)
     Then Desempilha := false
     Else begin
            Desempilha := True;
            x := p^.dado;
            aux := p ;
            p := p^.prox ;
            dispose(aux) ;
          end;
end ;



Function Topo(p: tipo_pilha): tipo_dado ;
Begin
```

```
    Topo := p^.dado ;
end ;
```

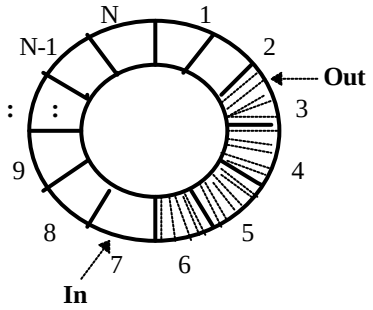
Fila:

Uma fila é uma estrutura na qual os dados são inseridos em um extremo e retirados no outro extremo. São também chamadas de **FIFO** ("First In First Out" = "Primeiro a Entrar e o Primeiro a Sair") ou **LILO** ("Last In Last Out" = "Último a Entrar e o Último a Sair"). Sua representação pode ser vista abaixo:

A implementação desta estrutura requer a utilização de um vetor, onde um elemento é inserido no final da fila (a direita) e retirado do início da fila (a esquerda). Esta implementação é dificultada devido ao fato da fila se locomover dentro do vetor. Se nada for feito para controlar esta locomoção, a fila atingirá o final do vetor (lado direito) e a próxima inserção não poderá ser feita, mesmo que haja espaço do lado esquerdo do vetor. Uma forma de resolver este problema é: sempre que um elemento for removido (lado esquerdo) deslocar os demais elementos da fila uma casa a esquerda. Esta solução implica em muito esforço computacional desnecessário. Três outras soluções podem ser utilizadas de forma mais elaborada: Fila Estática Circular, Fila Estática Encadeada e Fila Dinâmica Encadeada.

Fila Estática Circular:

Usamos uma estrutura vetorial para representar a fila. Como a fila **"se movimenta",** permitindo a manipulação dos dados nos dois extremos, e o vetor é uma estrutura estática, devemos implementar um mecanismo circular (fila circular) para aproveitar o máximo do vetor. A figura abaixo mostra a representação de uma fila circular, para um vetor de **N** posições.



Os elementos são retirados na posição **Out** e são inseridos na posição **In**. Como a estrutura é circular, a posição seguinte a posição **N** é a posição **1**. Assim, o incremento das posições **In** e **Out** deve ser feita da seguinte forma:

In := (In Mod N) + 1 e
Out := (Out Mod N) + 1

Como a declaração de um vetor requer um tamanho máximo da estrutura, não podemos inserir elementos quando a estrutura já estiver cheia. A condição de fila cheia pode ser obtida através de uma **var ("n_elem")** que contenha o número de elementos inseridos. Assim, teríamos:

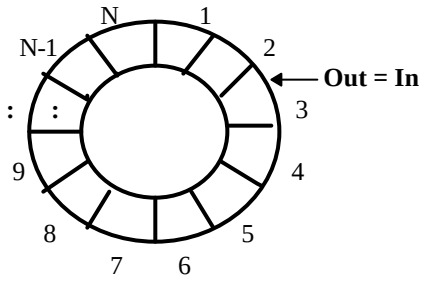
cheia := (n_elem = N) ;

A cada nova inserção ou eliminação, esta variável deverá ser incrementada ou decrementada, respectivamente. Neste caso, a condição de fila vazia seria:

vazia := (n_elem = 0) ;

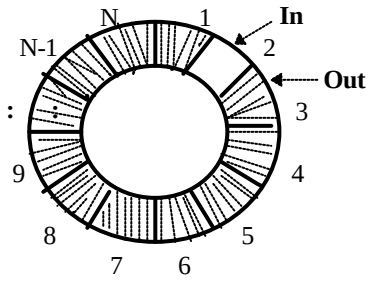
Uma outra forma de verificar as condições de fila cheia e fila vazia é através de uma comparação entre os ponteiros **In** e **Out**. Assim, poderíamos ter:

vazia := (In = Out) ;



cheia := Out =Posterior(In) ; quando o ponteiro **Out** etiver imediatamente posterior ao ponteiro **In**.

Neste caso, sempre teremos um campo inutilizado, pois se for permitido a inserção de elementos em todos os campos do vetor, o ponteiro **In** também estaria junto com o ponteiro **Out**, e não daria para saber se a fila está vazia ou cheia. A figura abaixo ilustra a condição de fila circular cheia.



```
Const   tam_max = 100 ;
type    tipo_dado = integer;
        tipo_fila = record
                        fila: array[ 1..tam_max] of tipo_dado ;
                        in,out: integer ;
                    end ;

Procedure Cria_Fila(var f : tipo_fila);
Begin
  f.in := 1 ; f.out := 1 ;
end ;

Procedure Incrementa(var pos : integer);
Begin
  pos := (pos MOD n) + 1 ;
end;

function Cheia( f : tipo_fila ): boolean ;
var aux : integer;
begin
  aux := f.in ; Incrementa(aux);
  Cheia := (aux = f.out) ;
end;

function Vazia (f : tipo_fila) : boolean;
begin
  Vazia := (f.in = f.out) ;
end ;

function Enfileira(var f : tipo_fila ; x: tipo_dado): boolean ;
begin
  If Cheia(f)
     Then Enfileira := false
     Else  begin
              Enfileira := True;
              f.fila[f.in] := x ; Incrementa(f.in) ;
           end;
end ;

function Desenfileira(var p: tipo_fila ; var x : tipo_dado): boolean ;
begin
  If Vazia(f)
     then Desenfileira := false
     else begin
             Desenfileira := True;
             x := f.fila[f.out];
             Incrementa(f.out) ;
```

```
            end;
    end ;
```

Implementação Estática Encadeada

Uma forma para facilitar o controle da movimentação da fila dentro do vetor é utilizar encadeamento nos campos do vetor. Neste caso, a fila pode utilizar qualquer campo do vetor e os elementos da fila não precisam estar dispostos em ordem. A idéia é ligar um elemento da fila em seu próximo através de um ponteiro lógico (campo que contém o número de uma posição do vetor que representa o próximo elemento da fila). Para controlar quais campos do vetor contém elementos pertencentes a fila e quais campos do vetor possui campos livres (que não estão sendo utilizados pelos elementos da fila) nós utilizaremos duas filas sobrepostas: uma fila F contendo a fila propriamente dita e uma fila L contendo os elementos do vetor que estão livres.

Para inserir um elemento na fila F nós deveremos alocar (reservar) uma entrada vazia do vetor (primeiro elemento da fila L), colocar o elemento nesta entrada e encadear esta entrada na fila F. Quando quisermos remover um elemento da fila F deveremos pegar o primeiro elemento que foi inserido, removê-lo do encadeamento e inseri-lo na fila L para que aquela posição possa ser posteriormente usada para inserir outro elemento.

Definição das Estruturas

```
Const  max = 100;
        Terra = -1;
Type tipo-dado =  integer;
       tipo-reg = record
                     dado: tipo-dado;
                      prox: interger;
                  end;
        tipo-fila = record
                      fila: array[1..max] of tipo-dado;
                      f,        { aponta para o inicio da fila de elementos válidos}
                       l: integer   {aponta para o inicio da fila de campos livres }
                    end;

procedure cria-fila(var F: tipo-fila);
var i: integer;
begin
   F.f := terra;
   F.l := 1; { no início todos são livres }
   for i:=1 to (max-1) do
       F.fila[i].prox := (i +1);
   F.fila[max].prox := terra;
end;

function vazia(F: tipo-fila): boolean;
begin
```

```
   vazia := (F.f = terra);
end;

function cheia(F: tipo-fila): boolean;
begin
   vazia := (F.l = terra);
end;

procedure insere(var F: tipo-fila; x: tipo-dado);
var aux:integer;
begin
  if not cheia(F)
     then begin
               aux := F.l ; { aloca o primeiro campo livre }
               F.l := F.fila[F.l].prox;
               F.fila[aux].dado := x;
               F.fila[aux].prox := F.f;
               F.f := aux;
            end
     else writeln('acabou memoria.....');
end;

procedure remove(var F: tipo-fila; var x: tipo-dado);
var ant, pos: integer;
begin
  if not vazia(F)
     then begin
               ant := F.f;
               if F.fila[ant].prox = terra
                  then begin
                            F.f := terra;
                            x := F.fila[ant].dado;
                            F.fila[ant].prox := F.l;
                            F.l := ant;
                       end
                  else begin
                            pos := F.fila[ant].prox;
                            while (F.fila[pos].prox <> terra) do
                               begin
                                  ant := pos;
                                  pos := F.fila[pos].prox;
                               end;
                            F.fila[ant].prox := terra;
                            x := F.fila[pos].dado;
                            F.fila[pos].prox := F.l;
                            F.l := pos;
                       end;
```

```
            end
     else writeln('pilha vazia .....');
end;
```

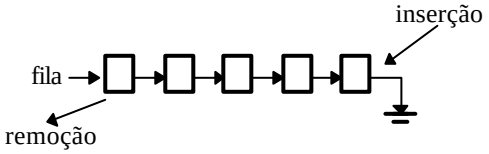
O procedimento remove anterior pode ser simplificado:

```
procedure remove(var F: tipo-fila; var x: tipo-dado);
var ant, pos: integer;
begin
  if not vazia(F)
     then begin
             pos := F.f;
             if F.fila[pos].prox = terra
                then F.f := terra;
                else begin
                        repeat
                           ant := pos;
                           pos := F.fila[pos].prox;
                        until F.fila[pos].prox = terra;
                        F.fila[ant].prox := terra;
                     end;
             x := F.fila[pos].dado;
             F.fila[pos].prox := F.l;
             F.l := pos;
          end
     else writeln('pilha vazia .....');
end;
```

Implementação Dinâmica Encadeada

A implementação dinâmica de uma fila é semelhante a implementação dinâmica de uma pilha. A única diferença é que a inserção e a eliminação ocorre em lados opostos.

Para implementar esta estrutura, utilizaremos uma estrutura encadeada como na pilha, conforme a figura abaixo. Como na pilha, o controle de toda a "**fila encadeada**" deverá ser feito por um ponteiro "fila" que apontará para o primeiro elemento.



A inserção será feita no final da estrutura (após o elemento que "aponta" para Terra) e a remoção será feita no início da estrutura (o elemento "apontado" pelo ponteiro "fila"). O módulo básico usado em cada nó é o mesmo usado para a pilha dinâmica.

```
type   tipo_dado = integer;
       tipo_fila: ^tipo_no ;
        tipo_no = record
```

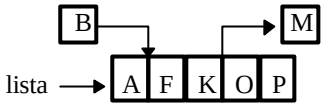
```
        dado: tipo_dado ;
        prox: tipo_fila ;
    end ;
```

Os procedimentos/funções podem ser implementados da seguinte forma:

```
Procedure Cria_Fila(var f : tipo_fila);
Begin
  f := Nil ;
end;

function Vazia (f: tipo_fila) : boolean;
begin
  Vazia := (f = Nil) ;
end;

procedure Enfileira(var f: tipo_fila; x: tipo_dado) ;
var aux1, aux2 : tipo_fila ;
begin
  new(aux1);
  aux1^.dado := x ;
  aux1^.prox := Nil ;
  If (f <> Nil)
    then begin
           aux2 := f
           while (aux2^.prox <> Nil) do
                 aux2 := aux2^.prox ;
           aux2^.prox := aux1 ;
         end
    else f := aux1 ;
end ;

function Desenfileira(var f : tipo_fila; var x : tipo_dado): boolean ;
var  aux : tipo_fila ;
begin
  if Vazia(f)
     then Desenfileira := false
     else begin
            Desenfileira := True;
            x := f^.dado ;
            aux := f ;
            f := f^.prox ;
            dispose(aux) ;
          end;
end ;
```

Lista

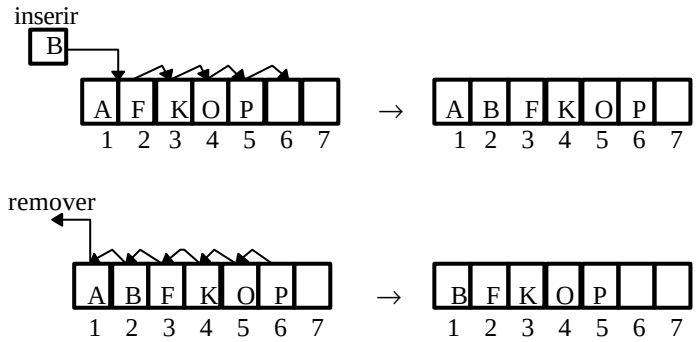
Lista é uma estrutura onde os dados são inseridos em qualquer posição, em função de uma política qualquer. Geralmente, os dados são inseridos por ordenação, conforme a figura abaixo. for eliminar um determinado elemento é necessário uma informação qualquer que especifique qual elemento deverá ser eliminado (nome, código, tipo, etc...).



Implementação Estática

Existem pelo menos 2 possibilidades de implementação utilizando estruturas estáticas: Lista Estática com Remanejamento e Lista Estática com Encadeamento

Na Lista Estática com Remanejamento, a inserção ou remoção de um elemento em uma posição que não seja no final da lista, requer um remanejamento de todos os elementos que estão localizados nas posições seguintes, como mostra a figura abaixo.




```
const n = 100 ;
type  tipo_dado = integer ;
      tipo_lista = record
                     lista: array[1..n] of tipo_dado ;
                     tam : integer ;
                   end ;

procedure Cria_Lista( var L : tipo_lista) ;
begin
  L.tam := 0 ;
end ;
```

```
function Cheia(L: tipo_lista) : boolean;
begin
  Cheia := (L.tam = n) ;
end;

function Vazia(L) : boolean;
var  L : tipo_lista ;
begin
  Vazia  := (L.tam = 0) ;
end;
```

(* A função Busca poderá ser utilizada tanto para a inserção quanto para a remoção de um elemento. Seu funcionamento será o seguinte: A função deverá verificar se existe o elemento X na lista L, devendo retornar o seguinte:

Se X existe, então flag retorna verdadeiro e Busca retorna a posição em que o elemento se encontra

Se X não existe, então flag retorna falso e Busca retorna a posição em que o elemento deve ser inserido *)

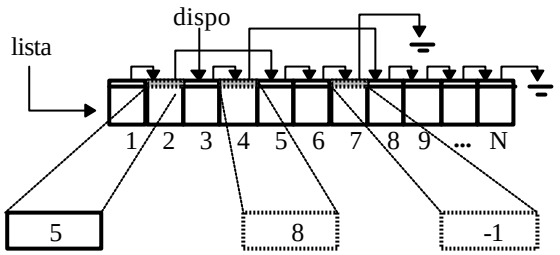
```
function Busca (var L: tipo_lista ; var X : tipo_dado; var flag : boolean) :
var    i : integer ;
begin
  i := 1 ;
  while (X < L.lista[i]) and (i ≤ L.tam) do
      Inc(i) ;
  if  X = L.lista[i]
     then begin
                Busca := i ;
                flag := true ;
             end
     else begin
                flag := false ;
                if X < L.lista[i]
                   then Busca := i ;
                   else Busca := L.tam + 1 ;
           end ;
end ;

procedure Desloca (var L : tipo_lista ; var pos : integer) ;
var  i, d : integer ;
begin
  d := L.tam - pos + 1;
  for i := 0 to (d-1) do
     L.lista[L.tam+1-i] := L.lista[L.tam-i] ;
  Inc(L.tam);
end ;
```

```
function Insere_Lista (var L : tipo_lista ; X) : boolean ;
var     X : tipo_dado ;
        pos : integer;   flag : boolean ;
begin
  if  Cheia (L)
      then Insere := false
      else if Vazia (L)
                then begin
                          L.lista[1] := X;
                          L.tam := 1;
                          Insere := true ;
                     end
                else  begin
                          pos := Busca (L,X,flag);
                          if flag  (* eleme nto já existe *)
                            then Insere := false
                             else begin
                                      Desloca (L,pos);
                                      L.lista[pos] := X ;
                                      Insere := true ;
                                  end;
                     end;
end ;

procedure Reloca (Var L : tipo_lista ; pos : integer) ;
var  i : integer ;
begin
  for i := pos to (L.tam-1) do
       L.lista[i] := L.lista[pos+1] ;
  L.tam := L.tam  - 1 ;
end ;

function Remove_Lista (var L : tipo_lista ; var X : tipo_dado) : boolean ;
var   i : integer ;  flag : boolean ;
begin
  if Vazia (L)
      then Remove_Lista := false
      else begin
                pos := Busca (L, X, flag);
                if  not flag
                    then  Remove_Lista :=  false
                    else   begin
                                X := L.lista[pos] ;
                               Reloca (L,pos);
                               Remove_Lista := true ;
```

```
                    end
            end;
    end ;
```

Na Lista Estática com Encadeamento, os elementos não precisam necessariamente estar situados seqüencialmente um após o outro, no vetor. Assim, para controlar a lista, cada elemento da lista "aponta" para o próximo, ou seja, contém a posição onde se encontra o próximo elemento da lista.

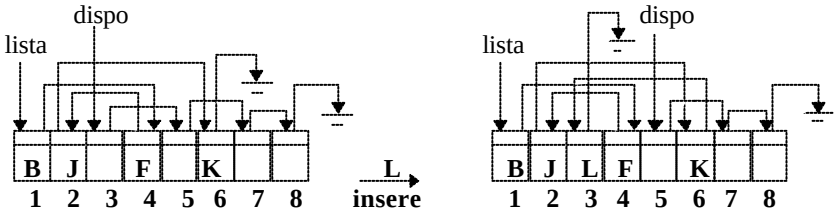
Como o primeiro elemento da lista não precisa necessariamente estar na primeira posição do vetor, deve existir uma var para conter a posição do primeiro elemento. for permitir a inserção e remoção de elementos sem remanejamento, o vetor comporta uma segunda lista, chamada "lista de disponíveis". A figura abaixo mostra o exemplo de um vetor com ambas as listas.



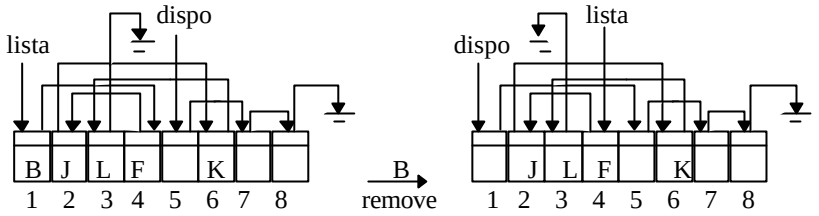
Na figura acima, as posições 1, 2, 5, 6 e 7 representam a lista propriamente dita (lista de ocupados). As posições 3, 4, 8, 9 ...e N representam os campos do vetor que estão vazios (disponíveis) para as novas inserções de elementos. No final de cada lista, o ultimo elemento aponta para uma posição inexistente (-1), representando a posição terra.

Em cada lista, uma posição "aponta" para a seguinte. Assim, na lista de posições ocupadas, cada registro do vetor contém um campo com o número da próxima posição ocupada, e na lista de disponíveis, cada registro do vetor contém um campo com o número da próxima posição livre.

Na inserção de um novo elemento, deve ser utilizado o primeiro campo da lista de disponíveis, para inserir fisicamente o elemento. then, deve-se procurar a posição de inserção lógica deste elemento na lista de ocupados, e acertar os "apontadores". Não é necessário o remanejamento dos elementos. A figura abaixo mostra a inserção do elemento "L".



Na remoção de um elemento, ocorre somente uma remoção lógica. A posição do vetor que contém o elemento a ser removido deve ser retirada da lista de ocupados e inserida na lista de disponíveis, através do acerto do "apontadores". Não é necessário o remanejamento dos elementos. A figura abaixo mostra a remoção do elemento "B".



Inicialmente, a lista de disponíveis deve ocupar todo o vetor e a lista de ocupados deve conter o código de terra (-1).

```
const   n = 100 ;
        fim = -1 ;  (* para indicar o final da lista *)

type  tipo_dado = record
                      dado,
                      prox : integer ;
                  end ;

      tipo_lista = record
                     lista:  array [1..n] of  tipo_dado ;
                     head,       (* apontador para a lista de ocupados *)
                     dispo : integer ;  (*apontador para a lista de disponíveis *)
                   end ;

procedure Cria_Lista(var L : tipo_lista) ;
var   i : integer ;
begin
   L.head := fim ;
   L.dispo := 1 ;
   for i := 1 to (n-1) do
      L.lista[i].prox  := i +1 ;
   L.lista[n].prox  := fim ;
end ;

function Cheia(L: tipo_lista) : boolean;
begin
  Cheia  := (L.dispo = fim) ;
end;

function Vazia(L: tipo_lista) : boolean;
begin
  Vazia  := (L.head = fim) ;
end;
```

(* A função Busca poderá ser utilizada tanto para a inserção quanto para a remoção de um elemento. Seu funcionamento será o seguinte: A função deverá verificar se existe o elemento X na lista L, devendo retornar o seguinte:

Se L está vazia, então flag retorna falso e Busca retorna fim, senão:

Se X existe, then:

- flag retorna verdadeiro e,
- Se X é o primeiro da lista, então Busca retorna fim, senão Busca retorna a posição do elemento anterior ao elemento X.

Se X não existe, então:

- flag retorna falso e,
- Se X é menor que o primeiro, então Busca retorna fim, senão Busca retorna a posição do elemento anterior a posição de inserção *)

```
function Busca (L : tipo_lista; X : tipo_dado; flag : boolean) : integer ;
var      ant, pos : integer ;
begin
   ant := fim ;
   pos := L.head ;
   while (L.lista[pos].dado < x) and (pos <> fim) do
      begin
         ant := pos ;
         pos := L.lista[ant].prox ;
      end ;
   Busca := ant ;
   flag := false ;
   if (pos <> fim)
      then if (L.lista[pos].dado = X)
                  then flag := true ;
end ;

function Insere_Lista (var L : tipo_lista ; X: tipo_dado) : boolean ;
var  aux, aux2, pos : integer;   flag : boolean ;
begin
  if  Cheia (L)
      then Insere_Lista := false
      else  begin
                pos := Busca (L, X, flag); (* se L vazia entao Busca := fim *)
               if flag  (* elemento já existe *)
                  then Insere_Lista := false
                  else  begin
                           aux := L.dispo ;
                           L.dispo := L.lista[aux].prox ;
                           L.lista[aux].dado := X ;
```

```
                        Insere_Lista := true ;
                        if (pos = fim) (* deve ser inserido no begin *)
                           then begin
                                   L.lista[aux].prox := L.head ;
                                   L.head := aux ;
                                end
                           else begin
                                   aux2 := L.lista[pos].prox;
                                   L.lista[aux].prox := aux2 ;
                                   L.lista[pos].prox := aux ;
                                end;
                        end;
                end;
end ;

function Remove_Lista (L : tipo_lista ; X : tipo_dado) : boolean ;
var   aux, pos : integer ;  flag : boolean ;
begin
  if Vazia (L)
      then Remove_Lista := false
      else begin
              pos := Busca (L, X, flag);
             if   not flag
                 then  Remove_Lista :=  false
                 else  begin
                         if (pos=fim)  (* deve-se remover o primeiro  *)
                            then begin
                                    aux := L.head ;
                                    L.head := L.lista[aux].prox ;
                                 end
                            else begin
                                    aux := L.lista[pos].prox ;
                                    L.lista[pos].prox := L.lista[aux].prox ;
                                    X := L.lista[aux].dado ;
                                 end;
                         L.lista[aux].prox := L.dispo ;
                         L.dispo := aux ;
                       end;
           end;
end ;
```

3° TRABALHO INDIVIDUAL:

Fazer um algoritmo completo, que implemente as funções de manipulação de:

a) Pilha Estática	b) Pilha Dinâmica
- Insere_Pilha_E	- Insere_Pilha_D
- Remove_Pilha_E	- Remove_Pilha_D
- topo_Pilha_E	- topo_Pilha_D
c) Fila Estática	d) Fila Dinâmica
- Insere_Fila_E	- Insere_Fila_D
- Remove_Fila_E	- Remove_Fila_D
e) Lista Estática c/ Reman/	f) Lista Estática Encadeada
- Insere_Lista_E_R	- Insere_Lista_E_E
- Remove_Lista_E_R	- Remove_Lista_E_E
- Busca_Lista_E_R	- Busca_Lista_E_E

O algoritmo deverá Simular um Estacionamento. Inicialmente, o algoritmo deverá solicitar o tipo de técnica a ser utilizada (a, b, c, d, e, f). Após a escolha da técnica, o algoritmo deverá solicitar opções do tipo:

- estacionar o carro
- retirar o carro
- olhar carro (verificar se o carro está no local)

A estrutura a ser utilizada (pilha, fila ou Lista) representará o estacionamento.

No caso de se escolher a estrutura de pilha, deve-se supor que o estacionamento possui somente uma entrada e só cabe um carro de largura, não permitindo manobra dentro do estacionamento. Assim, deverá ser utilizado 2 pilhas: uma pilha para o estacionamento principal e outra para o estacionamento secundário, pois assim, para remover um carro deve-se retirar todos os carros que estão no topo do estacionamento principal, um a um, to encontrar o carro desejado. Os carros retirados devem ser colocados na pilha secundária, e depois devolvidos para a pilha primária. for se estacionar um carro, deve-se colocá-lo no final do estacionamento primário.

No caso de se escolher a estrutura de fila, subentense-se que o estacionamento possui duas entradas, uma em cada extremo, mas possui a largura de um único carro, não permitindo a manobra. Assim, para retirar um determinado carro, deve-se retirar da frente e colocar atrás, to encontra o carro desejado. for se estacionar um carro deve-se sempre colocar atrás da fila.

No caso de se escolher a estrutura de Lista, o carro deverá ser colocado em uma posição qualquer de acordo com a ordem das placas. O carro pode ser retirado de qualquer posição.

Listas Duplamente Encadeadas