Estruturas de Dados Básicas*

Última alteração: 26 de Março de 2004

*Transparências elaboradas por Charles Ornelas Almeida e Nivio Ziviani

Listas Lineares

- Uma das formas mais simples de interligar os elementos de um conjunto.
- Estrutura em que as operações inserir, retirar e localizar são definidas.
- Podem crescer ou diminuir de tamanho durante a execução de um programa, de acordo com a demanda.
- Itens podem ser acessados, inseridos ou retirados de uma lista.
- Duas listas podem ser concatenadas para formar uma lista única, ou uma pode ser partida em duas ou mais listas.
- Adequadas quando não é possível prever a demanda por memória, permitindo a manipulação de quantidades imprevisíveis de dados, de formato também imprevisível.
- São úteis em aplicações tais como manipulação simbólica, gerência de memória, simulação e compiladores.

Definição de Listas Lineares

- Seqüência de zero ou mais itens $x_1, x_2, \cdots, x_n$, na qual x_i é de um determinado tipo e n representa o tamanho da lista linear.
- Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão.
  - Assumindo $n \geq 1$, x_1 é o primeiro item da lista e x_n é o último item da lista.
  - x_i precede x_{i+1} para $i = 1, 2, \cdots, n-1$
  - x_i sucede x_{i-1} para $i = 2, 3, \cdots, n$
  - o elemento x_i é dito estar na i-ésima posição da lista.

TAD Listas Lineares

- O conjunto de operações a ser definido depende de cada aplicação.
- Um conjunto de operações necessário a uma maioria de aplicações é:
  1. Criar uma lista linear vazia.
  2. Inserir um novo item imediatamente após o *i*-ésimo item.
  3. Retirar o *i*-ésimo item.
  4. Localizar o *i*-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
  5. Combinar duas ou mais listas lineares em uma lista única.
  6. Partir uma lista linear em duas ou mais listas.
  7. Fazer uma cópia da lista linear.
  8. Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.
  9. Pesquisar a ocorrência de um item com um valor particular em algum componente.

Implementações de Listas Lineares

- Várias estruturas de dados podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares.
- As duas representações mais utilizadas são as implementações por meio de arranjos e de apontadores.
- Exemplo de Conjunto de Operações:
  1. FLVazia(Lista). Faz a lista ficar vazia.
  2. Insere(x, Lista). Insere x após o último item da lista.
  3. Retira(p, Lista, x). Retorna o item x que está na posição p da lista, retirando-o da lista e deslocando os itens a partir da posição p+1 para as posições anteriores.
  4. Vazia(Lista). Esta função retorna *true* se lista vazia; senão retorna *false*.
  5. Imprime(Lista). Imprime os itens da lista na ordem de ocorrência.

Implementação de Listas por meio de Arranjos

- Os itens da lista são armazenados em posições contíguas de memória.
- A lista pode ser percorrida em qualquer direção.
- A inserção de um novo item pode ser realizada após o último item com custo constante.
- A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção.
- Retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.

	Itens
Primeiro $= 1$	x_1
2	x_2
	$\vdots$
Último-1	x_n
	$\vdots$
MaxTam	

Estrutura da Lista Usando Arranjo

- Os itens são armazenados em um **array** de tamanho suficiente para armazenar a lista.
- O campo Último aponta para a posição seguinte a do último elemento da lista.
- O *i*-ésimo item da lista está armazenado na *i*-ésima posição do **array**, $1 \leq i <$Último.
- A constante MaxTam define o tamanho máximo permitido para a lista.

```
const
  InicioArranjo = 1;
  MaxTam        = 1000;
type
  TipoChave = integer;
  Apontador = integer;
  TipoItem  = record
                Chave: TipoChave;
                { outros componentes }
              end;
  TipoLista = record
                Item     : array [1..MaxTam] of TipoItem;
                Primeiro : Apontador;
                Ultimo   : Apontador
              end;
```

Operações sobre Lista Usando Arranjo

```
procedure FLVazia (var Lista: TipoLista);
begin
  Lista.Primeiro := InicioArranjo;
  Lista.Ultimo := Lista.Primeiro;
end;

function Vazia (var Lista: TipoLista): boolean;
begin
  Vazia := Lista.Primeiro = Lista.Ultimo;
end;

procedure Insere (x: TipoItem; var Lista: TipoLista);
begin
  if Lista.Ultimo > MaxTam
  then writeln('Lista esta cheia')
  else begin
       Lista.Item[Lista.Ultimo] := x;
       Lista.Ultimo := Lista.Ultimo + 1;
       end;
end;
```

Operações sobre Lista Usando Arranjo

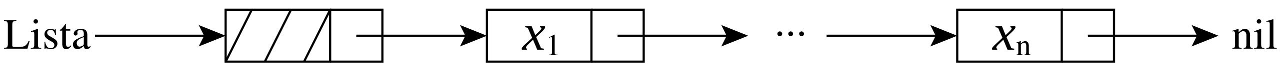
```
procedure Retira (p:Apontador; var Lista:TipoLista;
                              var Item:TipoItem);
var Aux: integer;
begin
  if Vazia (Lista) or (p >= Lista.Ultimo)
  then writeln ('Erro: Posicao nao existe')
  else begin
        Item := Lista.Item[p];
        Lista.Ultimo := Lista.Ultimo - 1;
        for Aux := p to Lista.Ultimo - 1 do
          Lista.Item[Aux] := Lista.Item[Aux+1];
        end;
end;

procedure Imprime (var Lista: TipoLista);
var Aux: integer;
begin
  for Aux := Lista.Primeiro to Lista.Ultimo - 1 do
    writeln (Lista.Item[Aux].Chave);
end;
```

Lista Usando Arranjo - Vantagens e Desvantagens

- Vantagem: economia de memória (os apontadores são implícitos nesta estrutura).
- Desvantagens:
  - custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens, no pior caso;
  - em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos em linguagens como o Pascal pode ser problemática porque neste caso o tamanho máximo da lista tem de ser definido em tempo de compilação.

Implementação de Listas por meio de Apontadores

- Cada item é encadeado com o seguinte mediante uma variável do tipo Apontador.
- Permite utilizar posições não contíguas de memória.
- É possível inserir e retirar elementos sem necessidade de deslocar os itens seguintes da lista.
- Há uma **célula cabeça** para simplificar as operações sobre a lista.



Estrutura da Lista Usando Apontadores

- A lista é constituída de células.
- Cada célula contém um item da lista e um apontador para a célula seguinte.
- O registro TipoLista contém um apontador para a célula cabeça e um apontador para a última célula da lista.

```
type
  Apontador = ^Celula;
  TipoItem  = record
                Chave: TipoChave;
                { outros componentes }
              end;
  Celula    = record
                Item: TipoItem;
                Prox: Apontador;
              end;
  TipoLista = record
                Primeiro: Apontador;
                Ultimo  : Apontador;
              end;
```

Operações sobre Lista Usando Apontadores

```
procedure FLVazia (var Lista: TipoLista);
begin
  new (Lista.Primeiro);
  Lista.Ultimo := Lista.Primeiro;
  Lista.Primeiro^.Prox := nil;
end;

function Vazia (Lista: TipoLista): boolean;
begin
  Vazia := Lista.Primeiro = Lista.Ultimo;
end;

procedure Insere (x: TipoItem; var Lista: TipoLista);
begin
  new(Lista.Ultimo^.Prox);
  Lista.Ultimo := Lista.Ultimo^.Prox;
  Lista.Ultimo^.Item := x;
  Lista.Ultimo^.Prox := nil
end;
```

Operações sobre Lista Usando Apontadores

```
procedure Retira (p:Apontador; var Lista:TipoLista;
                          var Item:TipoItem);
{—— Obs.: o item a ser retirado e
    o seguinte ao apontado por p ——}
var q: Apontador;
begin
  if Vazia (Lista) or (p = nil) or (p^.Prox = nil)
  then writeln ('Erro: Lista vazia ou posicao nao existe')
  else begin
         q := p^.Prox; Item := q^.Item; p^.Prox := q^.Prox;
         if p^.Prox = nil then Lista.Ultimo := p;
         dispose (q);
         end;
end;

procedure Imprime (Lista: TipoLista);
var Aux: Apontador;
begin
  Aux := Lista.Primeiro^.Prox;
  while Aux <> nil do
    begin
    writeln (Aux^.Item.Chave); Aux := Aux^.Prox;
    end;
end;
```

Listas Usando Apontadores - Vantagens e Desvantagens

- Vantagens:
  - Permite inserir ou retirar itens do meio da lista a um custo constante (importante quando a lista tem de ser mantida em ordem).
  - Bom para aplicações em que não existe previsão sobre o crescimento da lista (o tamanho máximo da lista não precisa ser definido *a priori*).
- Desvantagem: utilização de memória extra para armazenar os apontadores.

Exemplo de Uso Listas - Vestibular

- Num vestibular, cada candidato tem direito a três opções para tentar uma vaga em um dos sete cursos oferecidos.
- Para cada candidato é lido um registro:
  - Chave: número de inscrição do candidato.
  - NotaFinal: média das notas do candidato.
  - Opção: vetor contendo a primeira, a segunda e a terceira opções de curso do candidato.

```
Chave     : 1..999;
NotaFinal : 0..10;
Opcao     : array[1..3] of 1..7;
```

- Problema: distribuir os candidatos entre os cursos, segundo a nota final e as opções apresentadas por candidato.
- Em caso de empate, os candidatos serão atendidos na ordem de inscrição para os exames.

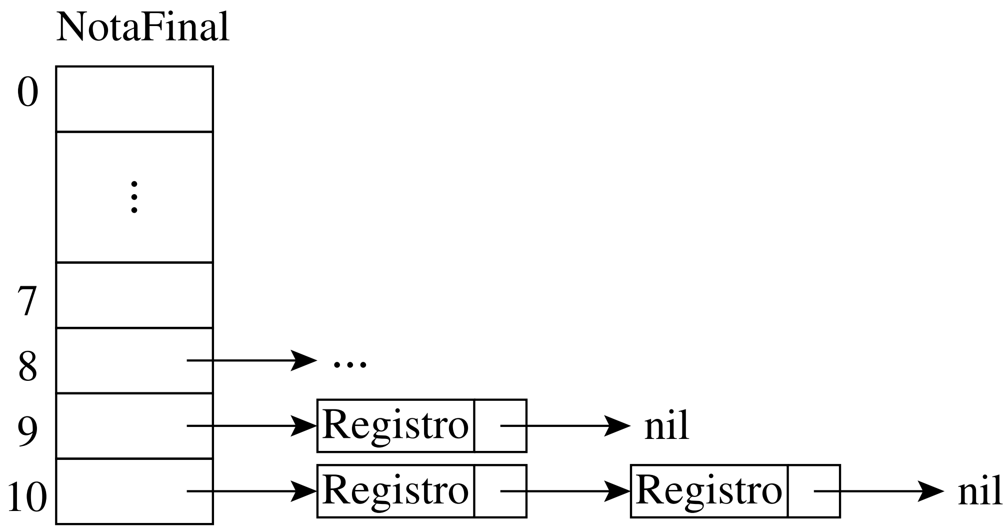
Vestibular - Possível Solução

- ordenar registros pelo campo NotaFinal, respeitando a ordem de inscrição;
- percorrer cada conjunto de registros com mesma NotaFinal, começando pelo conjunto de NotaFinal 10, seguido pelo de NotaFinal 9, e assim por diante.
  - Para um conjunto de mesma NotaFinal tenta-se encaixar cada registro desse conjunto em um dos cursos, na primeira das três opções em que houver vaga (se houver).
- Primeiro refinamento:

```
program Vestibular;
begin
  ordena os registros pelo campo NotaFinal;
  for Nota := 10 downto 0 do
    while houver registro com mesma nota do
      if existe vaga em um dos cursos de opcao do candidato
      then insere registro no conjunto de aprovados
      else insere registro no conjunto de reprovados;
  imprime aprovados por curso;  imprime reprovados;
end.
```

Vestibular - Classificação dos Alunos

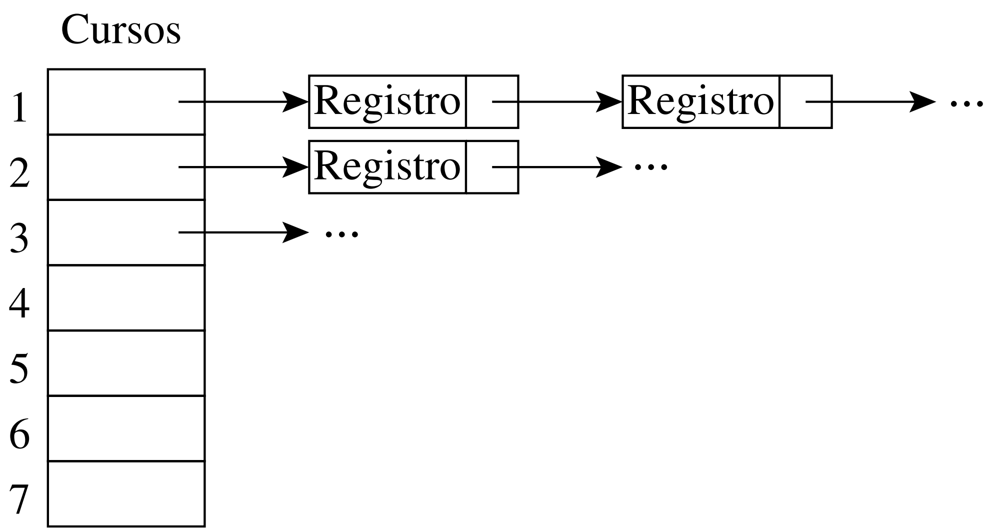
- Uma boa maneira de representar um conjunto de registros é com o uso de listas.
- Ao serem lidos, os registros são armazenados em listas para cada nota.
- Após a leitura do último registro os candidatos estão automaticamente ordenados por NotaFinal.
- Dentro de cada lista, os registros estão ordenados por ordem de inscrição, desde que os registros sejam lidos na ordem de inscrição de cada candidato e inseridos nesta ordem.



Vestibular - Classificação dos Alunos

- As listas de registros são percorridas, iniciando-se pela de NotaFinal 10, seguida pela de NotaFinal 9, e assim sucessivamente.

- Cada registro é retirado e colocado em uma das listas da abaixo, na primeira das três opções em que houver vaga.



- Se não houver vaga, o registro é colocado em uma lista de reprovados.

- Ao final a estrutura acima conterá a relação de candidatos aprovados em cada curso.

Vestibular - Segundo Refinamento

program Vestibular;
begin
  lê número de vagas para cada curso;
  inicializa listas de classificação de aprovados e reprovados;
  lê registro;
  while Chave $<>$ 0 **do**
    begin
    insere registro nas listas de classificação, conforme nota final;
    lê registro;
    end;
  for Nota := 10 **downto** 0 **do**
    while houver próximo registro com mesma NotaFinal **do**
      begin
      retira registro da lista;
      if existe vaga em um dos cursos de opção do candidato
      then begin
              insere registro na lista de aprovados;
              decrementa o número de vagas para aquele curso;
              end
      else insere registro na lista de reprovados;
      obtém próximo registro;
      end;
  imprime aprovados por curso;
  imprime reprovados;
end.

Vestibular - Estrutura Final da Lista

```
const NOpcoes = 3; NCursos = 7;
type
  TipoChave = 1..999;
  TipoItem  = record
                Chave: TipoChave;
                NotaFinal: 0..10;
                Opcao: array [1..NOpcoes] of 1..NCursos;
              end;
  Apontador = ^Celula;
  Celula    = record
                Item: TipoItem;
                Prox: Apontador;
              end;
  TipoLista = record
                Primeiro: Apontador;
                Ultimo: Apontador;
              end;
procedure LeRegistro (var Registro: TipoItem);
{—os valores lidos devem estar separados por brancos—}
var i: integer;
begin
  read (Registro.Chave, Registro.NotaFinal);
  for i := 1 to NOpcoes do read (Registro.Opcao[i]);
  readln;
end;
```

Vestibular - Refinamento Final

- Observe que o programa é completamente independente da implementação do tipo abstrato de dados Lista.

```
program Vestibular;
{-- Entram aqui os tipos da transparência 20 --}
var Registro       : TipoItem;
    Classificacao  : array [0..10] of TipoLista;
    Aprovados      : array [1..NCursos] of TipoLista;
    Reprovados     : TipoLista;
    Vagas          : array [1..NCursos] of integer;
    Passou         : boolean;
    i, Nota        : integer;
{-- Entram aqui as operações sobre listas usando
    apontadores das transparências 12 e 13 --}
{-- Entra aqui o procedimento LeRegistro (transp. 20) --}
begin {-- Programa principal --}
  for i := 1 to NCursos do read (Vagas[i]); readln;
  for i := 0 to 10 do FLVazia (Classificacao[i]);
  for i := 1 to NCursos do FLVazia (Aprovados[i]);
  FLVazia (Reprovados); LeRegistro (Registro);
  while Registro.Chave <> 0 do
    begin
    Insere (Registro, Classificacao[Registro.NotaFinal]);
    LeRegistro (Registro);
    end;
```

Vestibular - Refinamento Final (Cont.)

```
for Nota := 10 downto 0 do
  while not Vazia (Classificacao[Nota]) do
    begin
    Retira(Classificacao[Nota].Primeiro,
           Classificacao[Nota],Registro);
    i := 1; Passou := false;
    while (i <= NOpcoes) and not Passou do
      begin
      if Vagas[Registro.Opcao[i]] > 0
      then begin
           Insere (Registro,
                   Aprovados[Registro.Opcao[i]]);
           Vagas[Registro.Opcao[i]] :=
             Vagas[Registro.Opcao[i]] - 1;
           Passou := true;
           end;
      i := i + 1;
      end;
    if not Passou then Insere (Registro, Reprovados);
    end;
```

Vestibular - Refinamento Final (Cont.)

```
  for i := 1 to NCursos do
    begin
    writeln ('Relacao dos aprovados no Curso', i:2);
    Imprime (Aprovados[i]);
    end;
  writeln ('Relacao dos reprovados');
  Imprime (Reprovados);
end.
```

- O exemplo mostra a importância de utilizar **tipos abstratos de dados** para escrever programas, em vez de utilizar detalhes particulares de implementação.
- Altera-se a implementação rapidamente. Não é necessário procurar as referências diretas às estruturas de dados por todo o código.
- Este aspecto é particularmente importante em programas de grande porte.

Pilha

- É uma lista linear em que todas as inserções, retiradas e, geralmente, todos os acessos são feitos em apenas um extremo da lista.
- Os itens são colocados um sobre o outro. O item inserido mais recentemente está no topo e o inserido menos recentemente no fundo.
- O modelo intuitivo é o de um monte de pratos em uma prateleira, sendo conveniente retirar ou adicionar pratos na parte superior.
- Esta imagem está freqüentemente associada com a teoria de autômato, na qual o topo de uma pilha é considerado como o receptáculo de uma cabeça de leitura/gravação que pode empilhar e desempilhar itens da pilha.

Propriedade e Aplicações das Pilhas

- Propriedade: o último item inserido é o primeiro item que pode ser retirado da lista. São chamadas listas **lifo** ("last-in, first-out").
- Existe uma ordem linear para pilhas, do "mais recente para o menos recente".
- É ideal para processamento de estruturas aninhadas de profundidade imprevisível.
- Uma pilha contém uma seqüência de obrigações adiadas. A ordem de remoção garante que as estruturas mais internas serão processadas antes das mais externas.
- Aplicações em estruturas aninhadas:
  - Quando é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente.
  - O controle de seqüências de chamadas de subprogramas.
  - A sintaxe de expressões aritméticas.
- As pilhas ocorrem em estruturas de natureza recursiva (como árvores). Elas são utilizadas para implementar a **recursividade**.

TAD Pilhas

- Conjunto de operações:
  1. FPVazia(Pilha). Faz a pilha ficar vazia.
  2. Vazia(Pilha). Retorna *true* se a pilha está vazia; caso contrário, retorna *false*.
  3. Empilha(x, Pilha). Insere o item x no topo da pilha.
  4. Desempilha(Pilha, x). Retorna o item x no topo da pilha, retirando-o da pilha.
  5. Tamanho(Pilha). Esta função retorna o número de itens da pilha.
- Existem várias opções de estruturas de dados que podem ser usadas para representar pilhas.
- As duas representações mais utilizadas são as implementações por meio de *arranjos* e de *apontadores*.

Implementação de Pilhas por meio de Arranjos

- Os itens da pilha são armazenados em posições contíguas de memória.
- Como as inserções e as retiradas ocorrem no topo da pilha, um cursor chamado Topo é utilizado para controlar a posição do item no topo da pilha.

	Itens
Primeiro = 1	x_1
2	x_2
	$\vdots$
Topo	x_n
	$\vdots$
MaxTam	

Estrutura da Pilha Usando Arranjo

- Os itens são armazenados em um **array** de tamanho suficiente para conter a pilha.
- O outro campo do mesmo registro contém um apontador para o item no topo da pilha.
- A constante MaxTam define o tamanho máximo permitido para a pilha.

```
const MaxTam = 1000;
type
  TipoChave = integer;
  Apontador = integer;
  TipoItem  = record
                Chave: TipoChave;
                { outros componentes }
              end;
  TipoPilha = record
                Item: array [1..MaxTam] of TipoItem;
                Topo: Apontador;
              end;
```

Operações sobre Pilhas Usando Arranjos

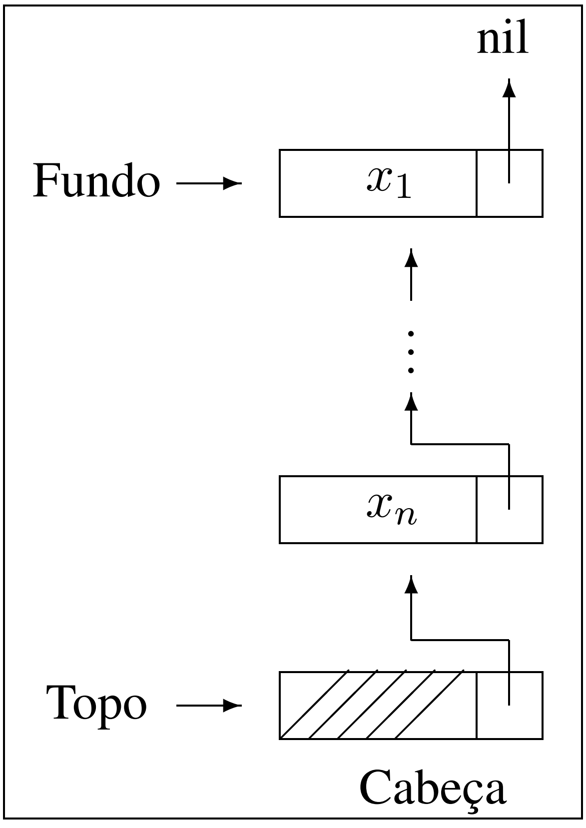
```
procedure FPVazia (var Pilha: TipoPilha);
begin
  Pilha.Topo := 0;
end;

function Vazia (var Pilha: TipoPilha): boolean;
begin
  Vazia := Pilha.Topo = 0;
end;

procedure Empilha (x: TipoItem; var Pilha: TipoPilha);
begin
  if Pilha.Topo = MaxTam
  then writeln ('Erro: pilha esta cheia')
  else begin
         Pilha.Topo := Pilha.Topo + 1;
         Pilha.Item[Pilha.Topo] := x;
       end;
end;
```

Operações sobre Pilhas Usando Arranjos

```
procedure Desempilha (var Pilha: TipoPilha;
                      var Item: TipoItem);
begin
  if Vazia (Pilha)
  then writeln ('Erro: pilha esta vazia')
  else begin
         Item := Pilha.Item[Pilha.Topo];
         Pilha.Topo := Pilha.Topo - 1;
       end;
end;

function Tamanho (Pilha: TipoPilha): integer;
begin
  Tamanho := Pilha.Topo;
end;
```

Implementação de Pilhas por meio de Apontadores

- Há uma célula cabeça é no topo para facilitar a implementação das operações empilha e desempilha quando a pilha está vazia.
- Para desempilhar o item x_n basta desligar a célula cabeça da lista e a célula que contém x_n passa a ser a célula cabeça.
- Para empilhar um novo item, basta fazer a operação contrária, criando uma nova célula cabeça e colocando o novo item na antiga.



Estrutura da Pilha Usando Apontadores

- O campo Tamanho evita a contagem do número de itens na função Tamanho.
- Cada célula de uma pilha contém um item da pilha e um apontador para outra célula.
- O registro TipoPilha contém um apontador para o topo da pilha (célula cabeça) e um apontador para o fundo da pilha.

```
type Apontador = ^Celula;
     TipoItem   = record
                    Chave: TipoChave;
                    { outros componentes }
                  end;
     Celula     = record
                    Item: TipoItem;
                    Prox: Apontador;
                  end;
     TipoPilha  = record
                    Fundo   : Apontador;
                    Topo    : Apontador;
                    Tamanho : integer;
                  end;
```

Operações sobre Pilhas Usando Apontadores

```
procedure FPVazia (var Pilha: TipoPilha);
begin
  new (Pilha.Topo);
  Pilha.Fundo := Pilha.Topo;
  Pilha.Topo^.Prox := nil;
  Pilha.Tamanho := 0;
end;

function Vazia (var Pilha: TipoPilha): boolean;
begin
  Vazia := Pilha.Topo = Pilha.Fundo;
end;

procedure Empilha (x: TipoItem; var Pilha: TipoPilha);
var Aux: Apontador;
begin
  new (Aux);
  Pilha.Topo^.Item := x;
  Aux^.Prox := Pilha.Topo;
  Pilha.Topo := Aux;
  Pilha.Tamanho := Pilha.Tamanho + 1;
end;
```

Operações sobre Pilhas Usando Apontadores

```
procedure Desempilha (var Pilha: TipoPilha;
                      var Item: TipoItem);
var q: Apontador;
begin
  if Vazia (Pilha)
  then writeln ('Erro: lista vazia')
  else begin
       q := Pilha.Topo;
       Pilha.Topo := q^.Prox;
       Item := q^.Prox^.Item;
       dispose (q);
       Pilha.Tamanho := Pilha.Tamanho – 1;
       end;
end;

function Tamanho (Pilha: TipoPilha): integer;
begin
  Tamanho := Pilha.Tamanho;
end;
```

Exemplo de Uso Pilhas - Editor de Textos (ET)

- "#": cancelar caractere anterior na linha sendo editada. Ex.: UEM##FMB#G → UFMG.
- "\": cancela todos os caracteres anteriores na linha sendo editada.
- "*": salta a linha. Imprime os caracteres que pertencem à linha sendo editada, iniciando uma nova linha de impressão a partir do caractere imediatamente seguinte ao caractere salta-linha. Ex: DCC*UFMG.* →
  DCC
  UFMG.
- Vamos escrever um Editor de Texto (ET) que aceite os três comandos descritos acima.
- O ET deverá ler um caractere de cada vez do texto de entrada e produzir a impressão linha a linha, cada linha contendo no máximo 70 caracteres de impressão.
- O ET deverá utilizar o **tipo abstrato de dados** Pilha definido anteriormente, implementado por meio de arranjo.

Sugestão de Texto para Testar o ET

Este et# um teste para o ET, o extraterrestre em

PASCAL.*Acabamos de testar a capacidade de o ET saltar de linha,

utilizando seus poderes extras (cuidado, pois agora vamos estourar

a capacidade máxima da linha de impressão, que é de 70

caracteres.)*O k#cut#rso dh#e Estruturas de Dados et# h#um

cuu#rsh#o #x# x?*!#?!#+.* Como et# bom

n#nt#ao### r#ess#tt#ar mb#aa#triz#cull#ado nn#x#ele!\ Sera

que este funciona\\\? O sinal? não### deve ficar! ~

ET - Implementação

- Este programa utiliza um tipo abstrato de dados sem conhecer detalhes de sua implementação.
- A implementação do TAD Pilha que utiliza arranjo pode ser substituída pela implementação que utiliza apontadores sem causar impacto no programa.

```
program ET;
const MaxTam          = 70;
      CancelaCarater  = '#';
      CancelaLinha    = '\';
      SaltaLinha      = '\*';
      MarcaEof        = '~';
type TipoChave = char;
{-- Entram aqui os tipos da transparência 28--}
var
  Pilha: TipoPilha;
  x    : TipoItem;
{-- Entram aqui os operadores das transparências
   29 e 30--}
{-- Entra aqui o procedimento Imprime (transp. 39)--}
```

ET - Implementação

```
begin {——Programa principal——}
  FPVazia (Pilha); read (x.Chave);
  while x.Chave <> MarcaEof do
    begin
    if x.Chave = CancelaCarater
    then begin
          if not Vazia (Pilha)
          then Desempilha (Pilha, x);
          end
    else if x.Chave = CancelaLinha
          then FPVazia (Pilha)
          else if x.Chave = SaltaLinha
                then Imprime (Pilha)
                else begin
                      if Tamanho (Pilha) = MaxTam
                      then Imprime (Pilha);
                      Empilha (x, Pilha);
                      end;
    read (x.Chave);
    end;
  if not Vazia (Pilha) then Imprime (Pilha);
end.
```

ET - Implementação (Procedimento Imprime)

```
procedure Imprime (var Pilha: TipoPilha);
var Pilhaux: TipoPilha; x: TipoItem;
begin
  FPVazia (Pilhaux);
  while not Vazia (Pilha) do
    begin
    Desempilha (Pilha, x);
    Empilha (x, Pilhaux);
    end;
  while not Vazia (Pilhaux) do
    begin
    Desempilha (Pilhaux, x);
    write (x.Chave);
    end;
  writeln;
end;
```

Fila

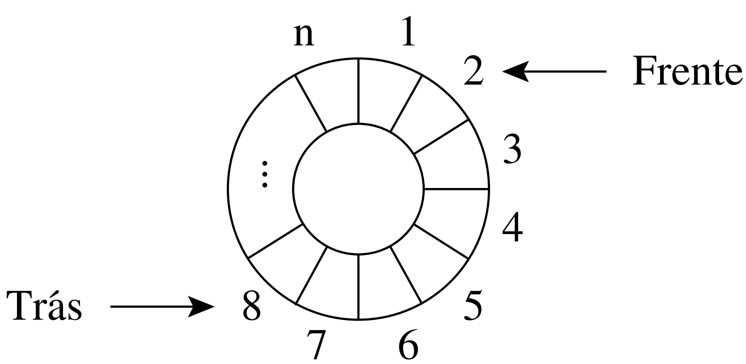
- É uma lista linear em que todas as inserções são realizadas em um extremo da lista, e todas as retiradas e, geralmente, os acessos são realizados no outro extremo da lista.
- O modelo intuitivo de uma fila é o de uma fila de espera em que as pessoas no início da fila são servidas primeiro e as pessoas que chegam entram no fim da fila.
- São chamadas listas **fifo** ("first-in", "first-out").
- Existe uma ordem linear para filas que é a "ordem de chegada".
- São utilizadas quando desejamos processar itens de acordo com a ordem "primeiro-que-chega, primeiro-atendido".
- Sistemas operacionais utilizam filas para regular a ordem na qual tarefas devem receber processamento e recursos devem ser alocados a processos.

TAD Filas

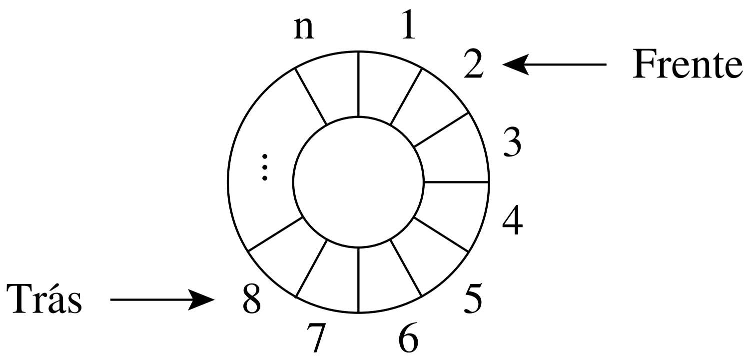
- Conjunto de operações:
  1. FFVazia(Fila). Faz a fila ficar vazia.
  2. Enfileira(x, Fila). Insere o item x no final da fila.
  3. Desenfileira(Fila, x). Retorna o item x no início da fila, retirando-o da fila.
  4. Vazia(Fila). Esta função retorna *true* se a fila está vazia; senão retorna *false*.

Implementação de Filas por meio de Arranjos

- Os itens são armazenados em posições contíguas de memória.
- A operação Enfileira faz a parte de trás da fila expandir-se.
- A operação Desenfileira faz a parte da frente da fila contrair-se.
- A fila tende a caminhar pela memória do computador, ocupando espaço na parte de trás e descartando espaço na parte da frente.
- Com poucas inserções e retiradas, a fila vai ao encontro do limite do espaço da memória alocado para ela.
- Solução: imaginar o **array** como um círculo. A primeira posição segue a última.



Implementação de Filas por meio de Arranjos



- A fila se encontra em posições contíguas de memória, em alguma posição do círculo, delimitada pelos apontadores Frente e Trás.
- Para enfileirar, basta mover o apontador Trás uma posição no sentido horário.
- Para desenfileirar, basta mover o apontador Frente uma posição no sentido horário.

Estrutura da Fila Usando Arranjo

- O tamanho máximo do **array** circular é definido pela constante MaxTam.
- Os outros campos do registro TipoPilha contêm apontadores para a parte da frente e de trás da fila.

```
const MaxTam = 1000;
type
  TipoChave = integer;
  Apontador = integer;
  TipoItem  = record
                Chave: TipoChave;
                { outros componentes }
              end;
  TipoFila  = record
                Item   : array [1..MaxTam] of TipoItem;
                Frente : Apontador;
                Tras   : Apontador;
              end;
```

Operações sobre Filas Usando Posições Contíguas de Memória

- Nos casos de fila cheia e fila vazia, os apontadores Frente e Trás apontam para a mesma posição do círculo.
- Uma saída para distinguir as duas situações é deixar uma posição vazia no **array**.
- Neste caso, a fila está cheia quando Trás$+1$ for igual a Frente.

```
procedure FFVazia (var Fila: TipoFila);
begin
  Fila.Frente := 1;
  Fila.Tras   := Fila.Frente;
end; { FFVazia }

function Vazia (var Fila: TipoFila): boolean;
begin
  Vazia := Fila.Frente = Fila.Tras;
end;
```

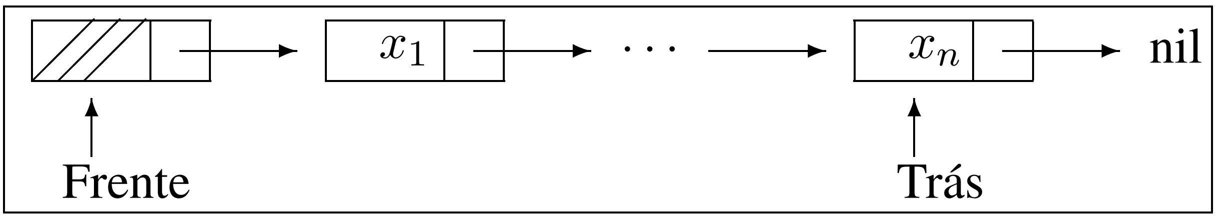
Operações sobre Filas Usando Posições Contíguas de Memória

- A implementação utiliza aritmética modular nos procedimentos Enfileira e Desenfileira (função **mod** do Pascal).

```
procedure Enfileira (x: TipoItem; var Fila: TipoFila);
begin
  if Fila.Tras mod MaxTam + 1 = Fila.Frente
  then writeln ('Erro: fila esta  cheia')
  else begin
         Fila.Item[Fila.Tras] := x;
         Fila.Tras := Fila.Tras mod MaxTam + 1;
       end;
end;

procedure Desenfileira (var Fila: TipoFila;
                        var Item: TipoItem);
begin
  if Vazia (Fila)
  then writeln ('Erro: fila esta vazia')
  else begin
         Item := Fila.Item[Fila.Frente];
         Fila.Frente := Fila.Frente mod MaxTam + 1;
       end;
end;
```

Implementação de Filas por meio de Apontadores

- Há uma célula cabeça é para facilitar a implementação das operações Enfileira e Desenfileira quando a fila está vazia.
- Quando a fila está vazia, os apontadores Frente e Trás apontam para a célula cabeça.
- Para enfileirar um novo item, basta criar uma célula nova, ligá-la após a célula que contém x_n e colocar nela o novo item.
- Para desenfileirar o item x_1, basta desligar a célula cabeça da lista e a célula que contém x_1 passa a ser a célula cabeça.



Estrutura da Fila Usando Apontadores

- A fila é implementada por meio de células.
- Cada célula contém um item da fila e um apontador para outra célula.
- O registro TipoFila contém um apontador para a frente da fila (célula cabeça) e um apontador para a parte de trás da fila.

```
type
  Apontador = ^Celula;
  TipoItem  = record
                Chave: TipoChave;
                { outros componentes }
              end;
  Celula    = record
                Item: TipoItem;
                Prox: Apontador;
              end;
  TipoFila =  record
                Frente: Apontador;
                Tras  : Apontador;
              end;
```

Operações sobre Filas Usando Apontadores

```
procedure FFVazia (var Fila: TipoFila);
begin
  new (Fila.Frente);
  Fila.Tras := Fila.Frente;
  Fila.Frente^.Prox := nil;
end;

function Vazia (var Fila: TipoFila): boolean;
begin
  Vazia := Fila.Frente = Fila.Tras;
end;

procedure Enfileira (x: TipoItem; var Fila: TipoFila);
begin
  new (Fila.Tras^.Prox);
  Fila.Tras := Fila.Tras^.Prox;
  Fila.Tras^.Item := x;
  Fila.Tras^.Prox := nil;
end;
```

Operações sobre Filas Usando Apontadores

```
procedure Desenfileira (var Fila: TipoFila;
                        var Item: TipoItem);
var q: Apontador;
begin
  if Vazia (Fila)
  then writeln ('Erro: fila esta vazia')
  else begin
       q := Fila.Frente;
       Fila.Frente := Fila.Frente^.Prox;
       Item := Fila.Frente^.Item;
       dispose (q);
       end;
end;
```