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.

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.

Conteúdo do Capítulo

3.1 Listas Lineares
- 3.1.1 Implementação de Listas por meio de Arranjos
- 3.1.2 Implementação de Listas por meio de Apontadores

3.2 Pilhas
- 3.2.1 Implementação de Pilhas por meio de Arranjos
- 3.2.2 Implementação de Pilhas por meio de Apontadores

3.3 Filas
- 3.3.1 Implementação de Filas por meio de Arranjos
- 3.3.2 Implementação de Filas por meio de Apontadores

Estruturas de Dados Básicas*

Última alteração: 16 de Setembro de 2010

*Slides elaborados por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

Estrutura da Lista Usando Arranjo

```
#define INICIOARRANJO 1
#define MAXTAM 1000

typedef int TipoApontador;
typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  /* --- outros componentes --- */
} TipoItem;
typedef struct {
  TipoItem Item[MAXTAM];
  TipoApontador Primeiro, Ultimo;
} TipoLista;
```

- 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.

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.

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

- 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.

Implementações de Listas Lineares

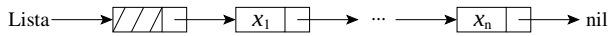
- Várias estruturas de dados podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares.
- As duas representações mais utilizadas são as implementações por meio de arranjos e de apontadores.
- Exemplo de Conjunto de Operações:
  1. FLVazia(Lista). Faz a lista 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.

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.

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.



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 nesse caso o tamanho máximo da lista tem de ser definido em tempo de compilação.

Operações sobre Lista Usando Arranjo

```
void Retira(TipoApontador p, TipoLista *Lista, TipoItem *Item)
{ int Aux;
  if (Vazia(*Lista) || p >= Lista -> Ultimo)
  { printf("Erro: Posicao nao existe\n");
    return;
  }
  *Item = Lista -> Item[p - 1];
  Lista -> Ultimo--;
  for (Aux = p; Aux < Lista -> Ultimo; Aux++)
    Lista -> Item[Aux - 1] = Lista -> Item[Aux];
}  /* Retira */
void Imprime(TipoLista Lista)
{ int Aux;
  for (Aux = Lista.Primeiro - 1; Aux <= (Lista.Ultimo - 2); Aux++)
    printf("%d\n", Lista.Item[Aux].Chave);
}  /* Imprime */
```


Operações sobre Lista Usando Arranjo

```
void FLVazia(TipoLista *Lista)
{ Lista->Primeiro = INICIOARRANJO; Lista->Ultimo = Lista->Primeiro; }

int Vazia(TipoLista Lista)
{ return (Lista.Primeiro == Lista.Ultimo); }  /* Vazia */

void Insere(TipoItem x, TipoLista *Lista)
{ if (Lista -> Ultimo > MAXTAM)
    printf("Lista esta cheia\n");
  else { Lista -> Item[Lista -> Ultimo - 1] = x;
         Lista -> Ultimo++;
       }
}  /* Insere */
```

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.

Operações sobre Lista Usando Apontadores

```
void Retira(TipoApontador p, TipoLista *Lista, TipoItem *Item)
{ /*--O item a ser retirado e o seguinte ao apontado por p--*/
  TipoApontador q;
  if (Vazia(*Lista) || p == NULL || p->Prox == NULL)
  { printf(" Erro: Lista vazia ou posicao nao existe\n");
    return;
  }
  q = p->Prox; *Item = q->Item;   p->Prox = q->Prox;
  if (p->Prox == NULL) Lista->Ultimo = p;
  free(q);
}
void Imprime(TipoLista Lista)
{ TipoApontador Aux;
  Aux = Lista.Primeiro->Prox;
  while (Aux != NULL)
    { printf("%d\n", Aux->Item.Chave); Aux = Aux->Prox; }
}
```


Operações sobre Lista Usando Apontadores

```
void FLVazia(TipoLista *Lista)
{ Lista->Primeiro = (TipoApontador) malloc(sizeof(TipoCelula));
  Lista->Ultimo = Lista->Primeiro; Lista->Primeiro->Prox = NULL;
}

int Vazia(TipoLista Lista)
{ return (Lista.Primeiro == Lista.Ultimo); }

void Insere(TipoItem x, TipoLista *Lista)
{ Lista->Ultimo->Prox = (TipoApontador) malloc(sizeof(TipoCelula));
  Lista->Ultimo = Lista->Ultimo->Prox; Lista->Ultimo->Item = x;
  Lista->Ultimo->Prox = NULL;
}
```

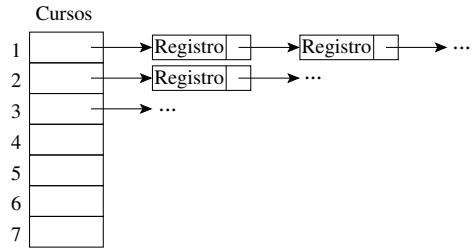

Estrutura da Lista Usando Apontadores

```
typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;
typedef struct TipoCelula *TipoApontador;
typedef struct TipoCelula {
  TipoItem Item;
  TipoApontador Prox;
} TipoCelula;

typedef struct {
  TipoApontador Primeiro, Ultimo;
} TipoLista;
```

- 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.

Vestibular - Classificação dos Alunos

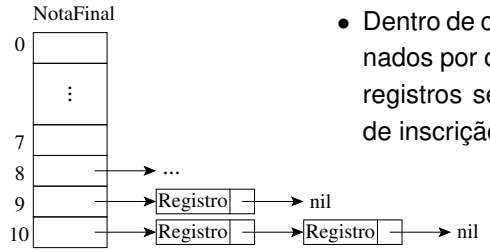
- As listas de registros são percorridas, iniciando-se pela de NotaFinal 10, seguida pela de NotaFinal 9, e assim sucessivamente.
- Cada registro é retirado e colocado em uma das listas da abaixo, na primeira das três opções em que houver vaga.



- 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 - Classificação dos Alunos

- Uma boa maneira de representar um conjunto de registros é com o uso de listas.
- 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 e inseridos na ordem de inscrição de cada candidato.

Vestibular - Possível Solução

- Ordenar registros por NotaFinal, respeitando a ordem de inscrição.
- Percorrer registros com mesma NotaFinal, começando pelo conjunto de NotaFinal 10, depois NotaFinal 9, e assim por diante.
  - Para um conjunto de mesma NotaFinal encaixar cada registro em um dos cursos, na primeira opção em que houver vaga (se houver).
- Primeiro refinamento:

```
int Nota; ordena os registros pelo campo NotaFinal;
for (Nota = 10; Nota >= 0; Nota --)
  { while houver registro com mesma nota
      { if  existe vaga em um dos cursos de opção do candidato
        {insere registro no conjunto de aprovados}
        else insere registro no conjunto de reprovados;
      }
  }
imprime aprovados por curso; imprime reprovados;
```


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 as três 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 - Refinamento Final (2)

```
{ /*——Programa principal——*/
  for (i = 1; i <= NCURSOS; i++) scanf("%d", &Vagas[i-1]);
  scanf("%*[^\n]");
  getchar();
  for (i = 0; i <= 10; i++) FLVazia(&Classificacao[i]);
  for (i = 1; i <= NCURSOS; i++) FLVazia(&Aprovados[i-1]);
  FLVazia(&Reprovados);
  LeRegistro(&Registro);
  while (Registro.Chave != 0)
    {
      Insere(Registro, &Classificacao[Registro.NotaFinal]);
      LeRegistro(&Registro);
    }
```


Vestibular - Refinamento Final (1)

```
#define NOPCOES      3
#define NCURSOS      7
#define FALSE        0
#define TRUE         1
/*—— Entram aqui os tipos do Slide 21——*/
TipoItem Registro;
TipoLista Classificacao[11];
TipoLista Aprovados[NCURSOS];
TipoLista Reprovados;
long Vagas[NCURSOS]; short Passou; long i, Nota;
/*—— Entram aqui os operadores sobre listas dos Slides 13 e 14——*/
void LeRegistro(TipoItem *Registro)
{ /*——os valores lidos devem estar separados por brancos——*/
  long i;
  scanf("%hd%d", &Registro->Chave, &Registro->NotaFinal);
  for (i = 0; i < NOPCOES; i++) scanf("%d", &Registro->Opcao[i]);
}
```


Vestibular - Estrutura Final da Lista

```
#define NOPCOES 3
#define NCURSOS 7
typedef short TipoChave;
typedef struct TipoItem {
  TipoChave Chave;
  int NotaFinal;
  int Opcao[NOPCOES];
} TipoItem;
typedef struct TipoCelula* TipoApontador;
typedef struct TipoCelula {
  TipoItem Item;
  TipoApontador Prox;
} TipoCelula;
typedef struct TipoLista {
  TipoApontador Primeiro, Ultimo;
} TipoLista;
```


Vestibular - Segundo Refinamento

```
int Nota; TipoChave Chave; lê número de vagas para cada curso;
inicializa listas de classificação, de aprovados e de reprovados;
lê registro;
while ( Chave != 0 )
  { insere registro nas listas de classificação, conforme nota final; lê registro; }
for ( Nota= 10; Nota>= 0; Nota--)
  { while ( houver próximo registro com mesma NotaFinal )
    { retira registro da lista;
      if existe vaga em um dos cursos de opção do candidato
      { insere registro na lista de aprovados;
        decrementa o número de vagas para aquele curso;
      }
      else { insere registro na lista de reprovados; }
      obtém próximo registro;
    }
  }
  imprime aprovados por curso; imprime reprovados;
```

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 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**.

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.

Vestibular - Refinamento Final

- Observe que o programa é completamente independente da implementação do tipo abstrato de dados Lista.
- 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.
- Esse aspecto é importante em programas de grande porte.

Vestibular - Refinamento Final (3)

```
  for (Nota = 10; Nota >= 0; Nota--)
    { while (!Vazia(Classificacao[Nota]))
        { Retira(Classificacao[Nota].Primeiro, &Classificacao[Nota], &Registro);
          i = 1;  Passou = FALSE;
        while (i <= NOPCOES && !Passou)
          { if (Vagas[Registro.Opcao[i-1] - 1] > 0)
            { Insere(Registro, &Aprovados[Registro.Opcao[i-1] - 1]);
              Vagas[Registro.Opcao[i-1] - 1]--;  Passou = TRUE;
            }
            i++;
          }
        if (!Passou) Insere(Registro, &Reprovados);
        }
    }
  for (i = 1; i <= NCURSOS; i++)
   { printf("Relacao dos aprovados no Curso%d\n",i); Imprime(Aprovados[i-1]); }
  printf("Relacao dos reprovados\n");  Imprime(Reprovados);
  return 0;
}
```

Operações sobre Pilhas Usando Arranjos

```
void FPVazia(TipoPilha *Pilha)
{ Pilha->Topo = 0; }

int Vazia(TipoPilha Pilha)
{ return (Pilha.Topo == 0); }

void Empilha(TipoItem x, TipoPilha *Pilha)
{ if (Pilha->Topo == MaxTam) printf("Erro: pilha esta cheia\n");
  else { Pilha->Topo++; Pilha->Item[Pilha->Topo - 1] = x; }
}
void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{ if (Vazia(*Pilha)) printf("Erro: pilha esta vazia\n");
  else { *Item = Pilha->Item[Pilha->Topo - 1]; Pilha->Topo --; }
}
int Tamanho(TipoPilha Pilha)
{ return (Pilha.Topo); }
```

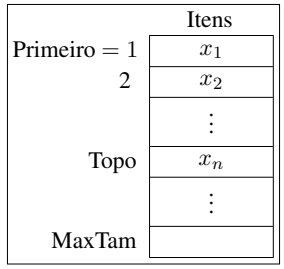

Estrutura da Pilha Usando Arranjo

- Os itens são armazenados em um **array** do tamanho da 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.

```
#define MAXTAM 1000
typedef int TipoApontador;
typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  /* --- outros componentes --- */
} TipoItem;
typedef struct {
  TipoItem Item[MAXTAM];
  TipoApontador Topo;
} TipoPilha;
```


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.



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*.

Operações sobre Pilhas Usando Apontadores

```
void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{ TipoApontador q;
  if (Vazia(*Pilha)) { printf("Erro: lista vazia\n"); return; }
  q = Pilha->Topo;
  Pilha->Topo = q->Prox;
  *Item = q->Prox->Item;
  free(q); Pilha->Tamanho--;
}

int Tamanho(TipoPilha Pilha)
{ return (Pilha.Tamanho); }
```


Operações sobre Pilhas Usando Apontadores

```
void FPVazia(TipoPilha *Pilha)
{ Pilha->Topo = (TipoApontador) malloc(sizeof(TipoCelula));
  Pilha->Fundo = Pilha->Topo;
  Pilha->Topo->Prox = NULL;
  Pilha->Tamanho = 0;
}
int Vazia(TipoPilha Pilha)
{ return (Pilha.Topo == Pilha.Fundo); }

void Empilha(TipoItem x, TipoPilha *Pilha)
{ TipoApontador Aux;
  Aux = (TipoApontador) malloc(sizeof(TipoCelula));
  Pilha->Topo->Item = x;
  Aux->Prox = Pilha->Topo;
  Pilha->Topo = Aux;
  Pilha->Tamanho++;
}
```

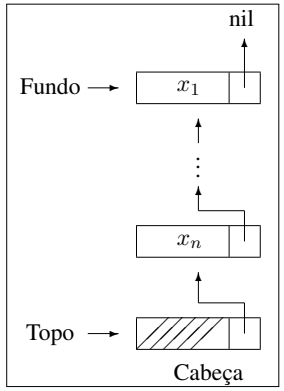

Estrutura da Pilha Usando Apontadores

- 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.

```
typedef int TipoChave;
typedef struct {
  int Chave;
  /* outros componentes */
} TipoItem;
typedef struct TipoCelula *TipoApontador;
typedef struct TipoCelula {
  TipoItem Item;
  TipoApontador Prox;
} TipoCelula;
typedef struct {
  TipoApontador Fundo, Topo;
  int Tamanho;
} TipoPilha;
```


Implementação de Pilhas por meio de Apontadores

- Há uma célula cabeça 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 item na antiga.



ET - Implementação

```
int main(int argc, char *argv[])
{ TipoPilha Pilha; TipoItem x;
  FPVazia(&Pilha); x.Chave = getchar();
  if (x.Chave == '\n') x.Chave = ' ';
  while (x.Chave != MARCAEOF)
  { if (x.Chave == CANCELACARATER)
    { if (!Vazia(Pilha)) Desempilha(&Pilha, &x);}
    else if (x.Chave == CANCELALINHA) FPVazia(&Pilha);
         else if (x.Chave == SALTALINHA) Imprime(&Pilha);
              else { if (Tamanho(Pilha) == MAXTAM) Imprime(&Pilha);
                     Empilha(x, &Pilha);
                   }
    x.Chave = getchar(); if (x.Chave == '\n') x.Chave = ' ';
  }
  if (!Vazia(Pilha)) Imprime(&Pilha); return 0;
}
```


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 que utiliza apontadores sem causar impacto no programa.

```
#define MAXTAM 70
#define CANCELACARATER '#'
#define CANCELALINHA '\\'
#define SALTALINHA '*'
#define MARCAEOF '~'
typedef char TipoChave;
/* Entram aqui os tipos da transparência 30 */
var Pilha: TipoPilha;
    x    : TipoItem;
/* Entram aqui os operadores da transparência 31 */
/* Entra aqui o procedimento Imprime (transp. 40) */
```


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

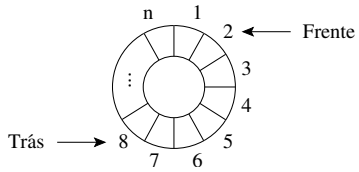
que este funciona\\\? O sinal? não### deve ficar! ~

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.

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.



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*.

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.

ET - Implementação (Procedimento Imprime)

```
void Imprime(TipoPilha *Pilha)
{ TipoPilha Pilhaux;
  TipoItem x;
  FPVazia(&Pilhaux);
  while (!Vazia(*Pilha))
    { Desempilha(Pilha, &x); Empilha(x, &Pilhaux);
    }
  while (!Vazia(Pilhaux))
    { Desempilha(&Pilhaux, &x); putchar(x.Chave);
    }
  putchar('\n');
}
```

Operações sobre Filas Usando Arranjos

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

```
void Enfileira(TipoItem x, TipoFila *Fila)
{ if (Fila->Tras % MAXTAM + 1 == Fila->Frente)
  printf(" Erro  fila est a cheia\n");
  else { Fila->Item[Fila->Tras - 1] = x;
         Fila->Tras = Fila->Tras % MAXTAM + 1;
       }
}
void Desenfileira(TipoFila *Fila, TipoItem *Item)
{ if (Vazia(*Fila))
  printf("Erro fila esta vazia\n");
  else { *Item = Fila->Item[Fila->Frente - 1];
         Fila->Frente = Fila->Frente % MAXTAM + 1;
       }
}
```


Operações sobre Filas Usando Arranjos

- 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**.
- Nesse caso, a fila está cheia quando Trás$+1$ for igual a Frente.

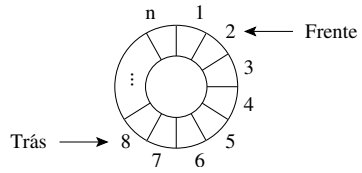
```
void FFVazia(TipoFila *Fila)
{ Fila->Frente = 1;
  Fila->Tras = Fila->Frente;
}

int Vazia(TipoFila Fila)
{ return (Fila.Frente == Fila.Tras); }
```


Estrutura da Fila Usando Arranjo

- O tamanho 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.

```
#define MAXTAM 1000
typedef int TipoApontador;
typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;
typedef struct {
  TipoItem Item[MAXTAM];
  TipoApontador Frente, Tras;
} TipoFila;
```


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.

Operações sobre Filas Usando Apontadores

```
void Desenfileira(TipoFila *Fila, TipoItem *Item)
{ TipoApontador q;
  if (Vazia(*Fila)) { printf("Erro fila esta vazia\n"); return; }
  q = Fila->Frente;
  Fila->Frente = Fila->Frente->Prox;
  *Item = Fila->Frente->Item;
  free(q);
}
```

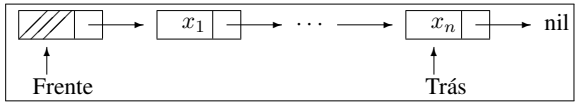

Operações sobre Filas Usando Apontadores

```
void FFVazia(TipoFila *Fila)
{ Fila->Frente = (TipoApontador) malloc(sizeof(TipoCelula));
  Fila->Tras = Fila->Frente;
  Fila->Frente->Prox = NULL;
}

int Vazia(TipoFila Fila)
{ return (Fila.Frente == Fila.Tras); }

void Enfileira(TipoItem x, TipoFila *Fila)
{ Fila->Tras->Prox = (TipoApontador) malloc(sizeof(TipoCelula));
  Fila->Tras = Fila->Tras->Prox;
  Fila->Tras->Item = x;
  Fila->Tras->Prox = NULL;
}
```


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.

```
typedef struct TipoCelula *TipoApontador;
typedef int TipoChave;
typedef struct TipoItem {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;
typedef struct TipoCelula {
  TipoItem Item;
  TipoApontador Prox;
} TipoCelula;
typedef struct TipoFila {
  TipoApontador Frente, Tras;
} TipoFila;
```


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.