

Listas Lineares

Livro “Projeto de Algoritmos” – Nívio Ziviani
Capítulo 3 – Seção 3.1

<http://www2.dcc.ufmg.br/livros/algoritmos/>

Listas Lineares

- Uma das formas mais simples de interligar os elementos de um conjunto.
- Estrutura em que as opera 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.

Listas Lineares

- 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, simulações e compiladores.

Definição de Listas Lineares

- **Seqüência de zero ou mais itens**
 - x_1, x_2, \dots, 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 $i = 1, 2, \dots, n - 1$
 - x_i sucede x_{i-1} para $i = 2, 3, \dots, n$
 - 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.**

Implementações de Listas Lineares

- **Exemplo de Conjunto de Operações:**
 - 1) `FLVazia(Lista)`. Faz a lista ficar vazia.
 - 2) `Inserex(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, $1 \leq i < \text{Último}$.
- A constante MaxTam define o tamanho máximo permitido para a lista.

Estrutura da Lista Usando Arranjo

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#define InicioArranjo 1
#define MaxTam 1000

typedef int TipoChave;

typedef int Apontador;

typedef struct {
    TipoChave Chave;
    /* outros componentes */
} TipoItem;

typedef struct {
    TipoItem Item[MaxTam];
    Apontador Primeiro, Ultimo;
} TipoLista;
```

Operações sobre Lista Usando Arranjo

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

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 */
```

Operações sobre Lista Usando Arranjo

```
void Retira(Apontador 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 */
```

Operações sobre Lista Usando Arranjo

```
void Imprime(TipoLista Lista)
{
    int Aux;

    for (Aux = Lista.Primeiro - 1;
         Aux <= (Lista.Ultimo - 2); Aux++)
        printf("%d\n", Lista.Item[Aux].Chave);
} /* Imprime */
```

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.

Estrutura da Lista Usando Apontadores

```
typedef int TipoChave;

typedef struct {
    int Chave;
    /* outros componentes */
} TipoItem;

typedef struct Celula_str *Apontador;

typedef struct Celula_str {
    TipoItem Item;
    Apontador Prox;
} Celula;

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

Operações sobre Lista Usando Apontadores

```
void FLVazia(TipoLista *Lista)
{
    Lista->Primeiro = (Apontador) malloc(sizeof(Celula));
    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 = (Apontador) malloc(sizeof(Celula));
    Lista->Ultimo = Lista->Ultimo->Prox;
    Lista->Ultimo->Item = x;
    Lista->Ultimo->Prox = NULL;
}
```

Operações sobre Lista Usando Apontadores

```
void Imprime (TipoLista Lista)
{
    Apontador Aux;
    Aux = Lista.Primeiro->Prox;
    while (Aux != NULL) {
        printf ("%d\n", Aux->Item.Chave);
        Aux = Aux->Prox;
    }
}
```

Operações sobre Lista Usando Apontadores

- **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;

Exemplo de Uso Listas - Vestibular

- **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).

Vestibular - Possível Solução

- Primeiro refinamento:

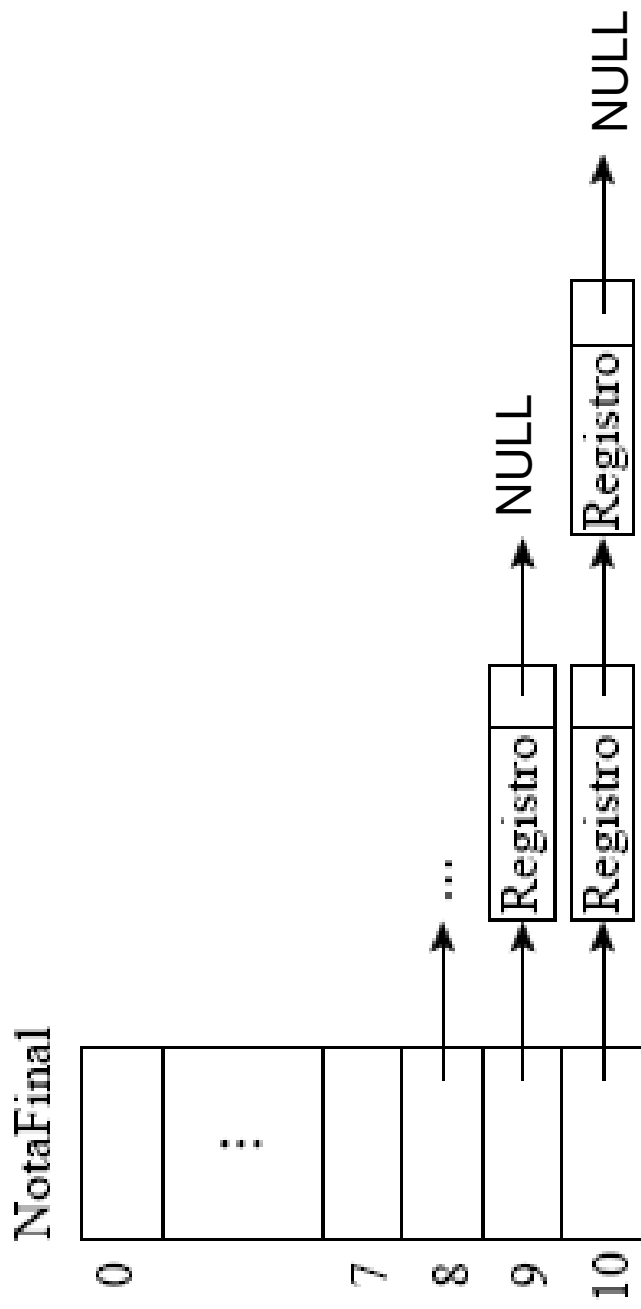
```
/* programa Vestibular */
void main ()
{
    int Nota;
    ordena os registros pelo campo NotaFinal ;
    for (Nota := 10; Nota >=0; Nota--) {
        while (houver registro com mesma nota) {
            if (existe vaga em um dos cursos de opcao do
                candidato)
                insere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

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.

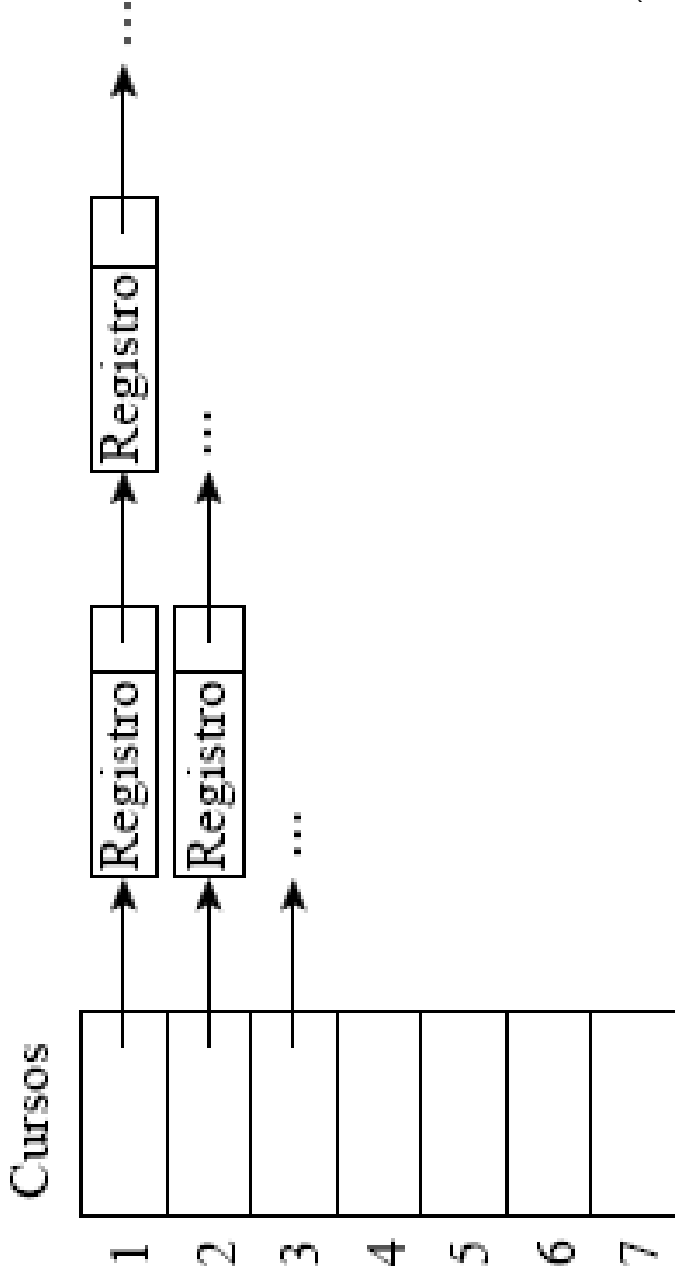
Vestibular - Classificação dos Alunos

- 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 abaixo, na primeira das três opções em que houver vaga. Após a leitura do último registro os candidatos estão automaticamente ordenados por NotaFinal.



Vestibular - Classificação dos Alunos

- 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

```
/* programa Vestibular */
Void main()
{
    int Nota;
    TipoChave Chave;

    le numero de vagas para cada curso;
    inicializa listas de classificacao, aprovados e de reprovados;
    le registro;
    while (Chave != 0) {
        insere registro nas listas de classificacao, conforme NotaFinal;
        le registro;
    }
    for (Nota = 10; Nota >= 0; Nota-- ) {
        while (houver proximo registro com mesma NotaFinal) {
            retira registro da lista;
            if (existe vaga em um dos cursos de opcao do candidato) {
                insere registro na lista de aprovados;
                decrementa o numero de vagas para aquele curso;
            }
            else insere registro na lista de reprovados;
            obtem proximo registro;
        }
    }
    imprime aprovados por curso;
    imprime reprovados;
}
}
```

Vestibular - Estrutura Final da Lista

```
#define NOPcoes          3
#define NCursos         7
#define FALSE           0
#define TRUE            1

typedef short TipoChave;

typedef struct TipoItem {
    TipoChave Chave;
    char NotaFinal;
    char Opcao[NOPcoes];
} TipoItem;

typedef struct Celula {
    TipoItem Item;
    struct Celula *Prox;
} Celula;

typedef struct TipoLista {
    Celula *Primeiro, *Ultimo;
} TipoLista;
```

Vestibular - Estrutura Final da Lista

```
#define NOPcoes 3
#define NCursos 7
#define FALSE 0
#define TRUE 1

TipoItem Registro;
TipoLista Classificacao[11];
TipoLista Aprovados[NCursos];
TipoLista Reprovados;

long Vagas[NCursos];
short Passou;
long i, Nota;
```

Vestibular - Refinamento Final

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

```
void LeRegistro(TipoItem *Registro)
{ /*---os valores lidos devem estar separados por brancos---*/
  int i;

  scanf("%d %d", &(Registro->Chave), &(Registro->NotaFinal));

  for (i = 0; i < NOPcoes; i++)
  {
    scanf("%d", &(Registro->Opcao[i]));
  }
}
```

Vestibular - Refinamento Final

```
int main(int argc, char *argv[])
{ /*---Programa principal---*/
  for (i = 1; i <= NCursos; i++) scanf("%ld ", &Vagas[i-1]);

  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

```
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;
}
```

Vestibular - Refinamento Final

- **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.**

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.**

Propriedade e Aplicações das Pilhas

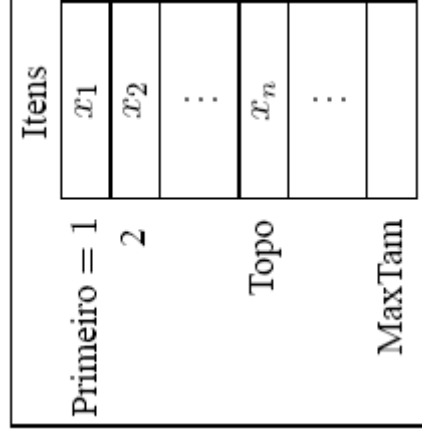
- **Aplicações em estruturas aninhadas:**
 - Quando é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente.
 - O controle de seqüências de chamadas de subprogramas.
 - A sintaxe de expressões aritméticas.
- **As pilhas ocorrem em estruturas de natureza recursiva (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.



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.

```
#define MaxTam 1000

typedef int Apontador;
typedef int TipoChave;
typedef struct {
    TipoChave Chave;
    /* --- outros componentes --- */
} TipoItem;

typedef struct {
    TipoItem Item[MaxTam];
    Apontador Topo;
} TipoPilha;
```

Operações sobre Pilhas Usando Arranjos

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

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

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;
        }
} /* Empilha */
```

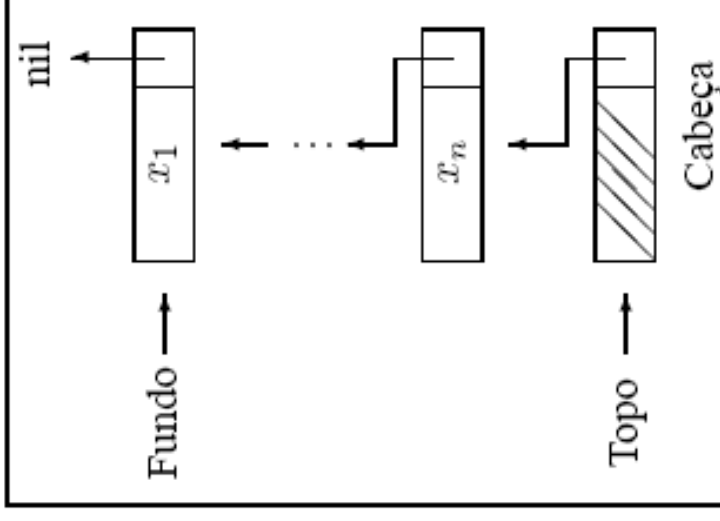
Operações sobre Pilhas Usando Arranjos

```
void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{
    if (Vazia(*Pilha))
        printf(" Erro pilha esta vazia\n");
    else { *Item = Pilha->Item[Pilha->Topo - 1];
          Pilha->Topo--;
        }
} /* Desempilha */

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

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.

Estrutura da Pilha Usando Apontadores

```
#define max 10

typedef int TipoChave;
typedef struct {
    int Chave;
    /* --- outros componentes --- */
} TipoItem;

typedef struct Celula_str *Apontador;

typedef struct Celula_str {
    TipoItem Item;
    Apontador Prox;
} Celula;

typedef struct {
    Apontador Fundo, Topo;
    int Tamanho;
} TipoPilha;
```

Operações sobre Pilhas Usando Apontadores

```
void FPVazia(TipoPilha *Pilha)
{
    Pilha->Topo = (Apontador) malloc(sizeof(Celula));
    Pilha->Fundo = Pilha->Topo;
    Pilha->Topo->Prox = NULL;
    Pilha->Tamanho = 0;
} /* FPVazia */

int Vazia(TipoPilha Pilha)
{
    return (Pilha.Topo == Pilha.Fundo);
} /* Vazia */

void Empilha(TipoItem x, TipoPilha *Pilha)
{
    Apontador Aux;

    Aux = (Apontador) malloc(sizeof(Celula));
    Pilha->Topo->Item = x;
    Aux->Prox = Pilha->Topo;
    Pilha->Topo = Aux;
    Pilha->Tamanho++;
} /* Empilha */
```

Operações sobre Pilhas Usando Apontadores

```
void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{
    Apontador 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--;
} /* Desempilha */

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

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.

Exemplo de Uso Pilhas - Editor de Textos (ET)

- **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.**

ET - Implementação

```
#include <stdio.h>

#define MaxTam      70

#define CancelaCarater '#'
#define CancelaLinha '\\\''
#define SaltaLinha  '*'
#define MarcaEof    '~'

typedef char TipoChave;

typedef int Apontador;

typedef struct {
    /* --- outros componentes --- */
    TipoChave Chave;
} TipoItem;

typedef struct {
    TipoItem Item[MaxTam];
    Apontador Topo;
} TipoPilha;
```

ET - Implementação

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

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

void Empilha(TipoItem x, TipoPilha *Pilha)
{
    if (Pilha->Topo == MaxTam)
        printf(" Erro pilha est a cheia\n");
    else { Pilha->Topo++;
          Pilha->Item[Pilha->Topo - 1] = x;
        }
} /* Empilha */
```

ET - Implementação

```
void Desempilha(TipoPilha *Pilha, TipoItem *Item)
{
    if (Vazia(*Pilha))
        printf(" Erro pilha est a vazia\n");
    else { *Item = Pilha->Item[Pilha->Topo - 1];
          Pilha->Topo--;
        }
    /* Desempilha */
}

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

ET - Implementação

```
int main(int argc, char *argv[])
{
    TipoPilha Pilha;
    TipoItem x;
    FPVazia(&Pilha);
    x.Chave = getchar();
    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 (!Vazia(Pilha)) Imprime(&Pilha);
    return 0;
} /* ET */
```

ET - Implementação

```
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');
} /* Imprime */
```

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”).**

Fila

- **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

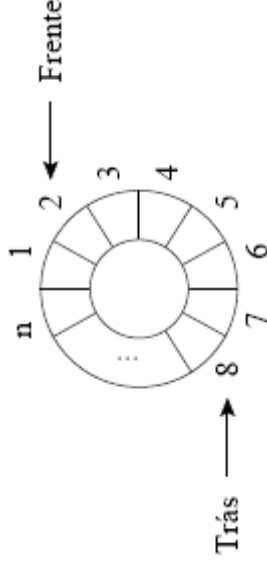
- **Conjunto de operações:**
 - 1) **FFVazia(Fila).** Faz a fila ficar vazia.
 - 2) **Enfileira(x, Fila).** Insere o item x no final da fila.
 - 3) **Desenfileira(Fila, x).** Retorna o item x no início da fila, retirando-o da fila.
 - 4) **Vazia(Fila).** Esta função retorna true se a fila está vazia;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

Implementação de Filas por meio de Arranjos

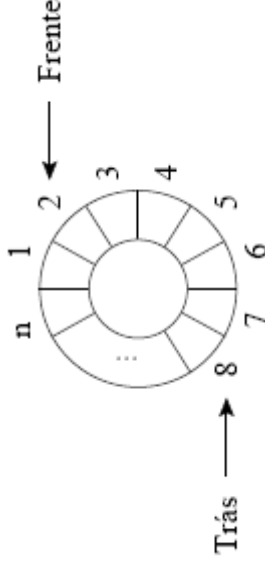
- Os itens são armazenados em posições contíguas de memória.
- A operação Enfileira faz a parte de trás da fila expandir-se.
- A operação Desenfileira faz a parte da frente da fila contrair-se.
- A fila tende a caminhar pela memória do computador, ocupando espaço na parte de trás e descartando espaço na parte da frente.

Implementação de Filas por meio de Arranjos

- 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 desinfileirar, 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` contém apontadores para a parte da frente e de trás da fila.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#define MaxTam 1000

typedef int Apontador;
typedef int TipoChave;
typedef struct {
    TipoChave Chave;
    /* outros componentes */
} TipoItem;

typedef struct {
    TipoItem Item[MaxTam];
    Apontador Frente, Tras;
} TipoFila;
```

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 $\text{Trás} + 1$ for igual a Frente.

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

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

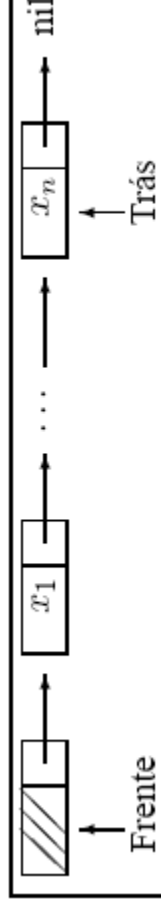
Operações sobre Filas Usando Posições Contíguas de Memória

```
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;
        }
} /* Enfileira */

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

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.
- A estrutura TipoFila contém um apontador para a frente da fila (célula cabeça) e um apontador para a parte de trás da fila.

Estrutura da Fila Usando Apontadores

```
#include <stdlib.h>
#include <sys/time.h>
#include <stdio.h>
#define max 10

typedef struct Celula_str *Apontador;

typedef int TipoChave;

typedef struct TipoItem {
    TipoChave Chave;
    /* outros componentes */
} TipoItem;

typedef struct Celula_str {
    TipoItem Item;
    Apontador Prox;
} Celula;

typedef struct TipoFila {
    Apontador Frente, Tras;
} TipoFila;
```

Operações sobre Filas Usando Apontadores

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

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

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

Operações sobre Filas Usando Apontadores

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

Exercícios

- 1) Estenda o TAD Lista, implementado por apontadores, para incluir um procedimento que remova TODOS os elementos pares da lista.
- 2) Implemente um procedimento que, utilizando os TADs Lista e Pilha, retorne os elementos da Lista em ordem invertida (em uma outra lista)
- 3) Implemente as operações Empilha e Desempilha, usando duas Filas
- 4) Um problema que pode surgir na manipulação de listas lineares simples é o de “voltar” atrás na lista, ou seja, percorre-la no sentido inverso ao dos apontadores. A solução geralmente adotada é a incorporação a célula de um apontador para o seu antecessor. Listas deste tipo são chamadas de **duplamente encadeadas**. Faça:
 - a) Declare os tipos necessários para a manipulação da lista
 - b) Escreva um procedimento para retirar da lista a célula apontada por p.
Não deixe de considerar eventuais casos especiais
- 5) Estenda o TAD lista (por apontadores) com um procedimento para trocar de posição dois elementos. Este procedimento deve receber como parâmetro, além da lista, os apontadores p e q. Ele deve tratar os casos especiais (apontadores para NULL, etc).