Estruturas de Dados Básicas*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Charles Ornelas, Leonardo Rocha, Leonardo Mata 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 estruturas auto-referenciadas.

- Exemplo de Conjunto de Operações:

  1. Lista(maxTam). Cria uma lista vazia.
  2. insere(x). Insere x após o último item da lista.
  3. retira(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(). Esta função retorna *true* se lista vazia; senão retorna *false*.
  5. imprime(). 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 $= 0$	x_1
1	x_2
	$\vdots$
último -1	x_n
	$\vdots$
maxTam -1	

Estrutura da Lista Usando Arranjo

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

```
package cap3.arranjo;
public class Lista {
  private Object item[];
  private int primeiro, ultimo, pos;
  // Operações
  public Lista (int maxTam) { // Cria uma Lista vazia
    this.item = new Object[maxTam]; this.pos = −1;
    this.primeiro = 0; this.ultimo = this.primeiro;
  }
```

Operações sobre Lista Usando Arranjo

```
public Object pesquisa (Object chave) {
  if (this.vazia () || chave == null) return null;
  for (int p = 0; p < this.ultimo; p++)
    if (this.item[p].equals (chave)) return this.item[p];
  return null;
}
public void insere (Object x) throws Exception {
  if (this.ultimo >= this.item.length)
    throw new Exception ("Erro: A lista esta cheia");
  else { this.item[this.ultimo] = x;
         this.ultimo = this.ultimo + 1; }
}
public Object retira (Object chave) throws Exception {
  if (this.vazia () || chave == null)
    throw new Exception ("Erro : A lista esta vazia");
  int p = 0;
  while(p < this.ultimo && !this.item[p].equals(chave))p++;
  if (p >= this.ultimo) return null; // Chave não en-
contrada
  Object item = this.item[p];
  this.ultimo = this.ultimo − 1;
  for (int aux = p; aux < this.ultimo; aux++)
    this.item[aux] = this.item[aux + 1];
  return item;
}
```

Operações sobre Lista Usando Arranjo

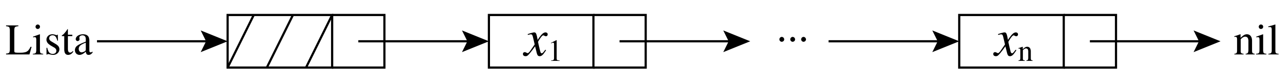
```
public Object retiraPrimeiro () throws Exception {
  if (this.vazia ()) throw new Exception
          ("Erro : A lista esta vazia");
  Object item = this.item[0];
  this.ultimo = this.ultimo − 1;
  for (int aux = 0; aux < this.ultimo; aux++)
    this.item[aux] = this.item[aux + 1];
  return item;
}
public Object primeiro () {
        this.pos = −1; return this.proximo (); }
public Object proximo () {
  this.pos++;
  if (this.pos >= this.ultimo) return null;
  else return this.item[this.pos];
}
public boolean vazia () {
        return (this.primeiro == this.ultimo); }
public void imprime () {
  for (int aux = this.primeiro; aux < this.ultimo; aux++)
    System.out.println (this.item[aux].toString ());
}
}
```

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 exigir a realocação de memória.

Implementação de Listas por meio de Estruturas Auto-Referenciadas

- Cada item da lista contém a informação que é necessária para alcançar o próximo item.
- 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.



Implementação de Listas por meio de Estruturas Auto-Referenciadas

- A lista é constituída de células.
- Cada célula contém um item da lista e uma referência para a célula seguinte.
- A classe $Lista$ contém uma referência para a célula cabeça, uma referência para a última célula da lista e uma referência para armazenar a posição corrente na lista.

```
package cap3.autoreferencia;
public class Lista {
  private static class Celula { Object item; Celula prox; }
  private Celula primeiro, ultimo, pos;
  // Operações
  public Lista () { // Cria uma Lista vazia
    this.primeiro = new Celula (); this.pos = this.primeiro;
    this.ultimo = this.primeiro; this.primeiro.prox = null;
  }
```

Lista Usando Estruturas Auto-Referenciadas

```
public Object pesquisa (Object chave) {
  if (this.vazia () || chave == null) return null;
  Celula aux = this.primeiro;
  while (aux.prox != null) {
    if (aux.prox.item.equals (chave)) return aux.prox.item;
    aux = aux.prox;
  } return null;
}
public void insere (Object x) {
  this.ultimo.prox = new Celula ();
  this.ultimo = this.ultimo.prox;
  this.ultimo.item = x; this.ultimo.prox = null;
}
public Object retira (Object chave) throws Exception {
  if (this.vazia () || (chave == null))
    throw new Exception
    ("Erro: Lista vazia ou chave invalida");
  Celula aux = this.primeiro;
  while (aux.prox!=null && !aux.prox.item.equals(chave))
          aux=aux.prox;
  if (aux.prox == null) return null; // não encontrada
  Celula q = aux.prox;
  Object item = q.item; aux.prox = q.prox;
  if (aux.prox == null) this.ultimo = aux;  return item;
}
```

Lista Usando Estruturas Auto-Referenciadas

```
public Object retiraPrimeiro () throws Exception {
   if (this.vazia ()) throw new Exception
            ("Erro: Lista vazia");
   Celula aux = this.primeiro; Celula q = aux.prox;
   Object item = q.item; aux.prox = q.prox;
   if (aux.prox == null) this.ultimo = aux;  return item;
 }
 public Object primeiro () {
          this.pos = primeiro; return proximo (); }
 public Object proximo () {
   this.pos = this.pos.prox;
   if (this.pos == null) return null;
   else return this.pos.item;
 }
 public boolean vazia () {
          return (this.primeiro == this.ultimo); }
 public void imprime () {
   Celula aux = this.primeiro.prox;
   while (aux != null) {
     System.out.println (aux.item.toString ());
     aux = aux.prox; }
 }
}
```

Lista Usando Estruturas Auto-Referenciadas - 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 as referências.

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.
  - $opcao$: vetor contendo a primeira, a segunda e a terceira opções de curso do candidato.

```
short chave;        // assume valores de 1 a 999.
byte  notaFinal;    // assume valores de 0 a 10.
byte  opcao[];      // arranjo de 3 posições;
```

- 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.

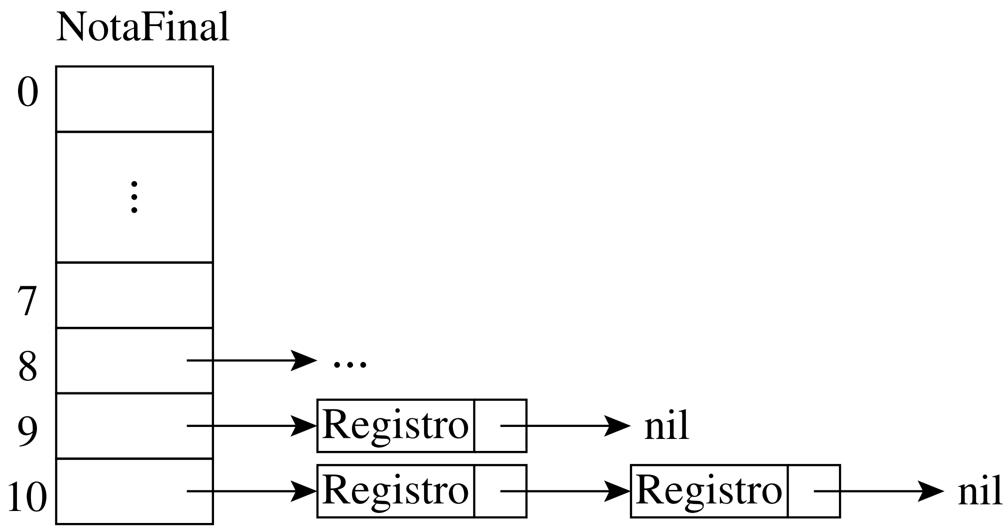
Vestibular - Possível Solução

- 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:

```
void Vestibular {
  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;
}
```

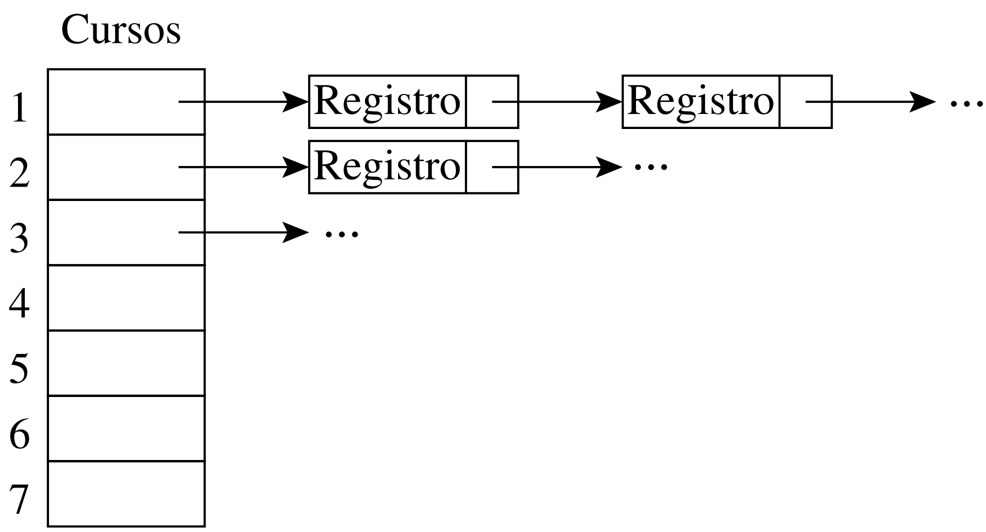
Vestibular - Classificação dos Alunos

- 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

void Vestibular {
  lê número de vagas para cada curso;
  inicializa listas de classificação, de aprovados e de reprovados;
  lê registro; *// vide formato do registro na transparência 15*
  while (chave $\neq 0$) {
    insere registro nas listas de classificação, conforme notaFinal;
    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;
}

Vestibular - Refinamento Final

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

```
package cap3;
import java.io.*;
import cap3.autoreferencia.Lista; // vide programa da trans-
parência 11
public class Vestibular {
  private class Definicoes {
    public static final int nOpcoes = 3;
    public static final int nCursos = 7;
  }
  private static class Registro {
    short chave;  byte notaFinal;
    byte opcao[] = new byte[Definicoes.nOpcoes];
    public String toString () {
            return new String ("" + this.chave); }
  }
  private static BufferedReader in =
      new BufferedReader(new InputStreamReader(System.in));
```

Vestibular - Refinamento Final

```
static Registro leRegistro () throws IOException {
  // os valores lidos devem estar separados por brancos
  Registro registro = new Registro ();
  String str = in.readLine ();
  registro.chave = Short.parseShort (str.substring (0,
                                     str.indexOf (" ")));
  registro.notaFinal = Byte.parseByte (str.substring (
                                  str.indexOf (" ") + 1));
  for (int i = 0; i < Definicoes.nOpcoes; i++)
    registro.opcao[i] = Byte.parseByte (in.readLine ());
  return registro;
}
public static void main (String[] args) {
  Registro registro = null;
  Lista classificacao[] = new Lista[11];
  Lista aprovados[] = new Lista[Definicoes.nCursos];
  Lista reprovados = new Lista ();
  long vagas[] = new long[Definicoes.nCursos];
  boolean passou;
  int i;
  try {
    for (i = 0; i < Definicoes.nCursos; i++)
      vagas[i] = Long.parseLong (in.readLine ());
    for(i = 0; i < 11; i++)classificacao[i] = new Lista();
  // Continua na próxima transparência
```

Vestibular - Refinamento Final (Cont.)

```
    for (i = 0; i < Definicoes.nCursos; i++)
      aprovados[i] = new Lista ();
    registro = leRegistro ();
    while (registro.chave != 0) {
      classificacao[registro.notaFinal].insere (registro);
      registro = leRegistro ();
    }
    for (int Nota = 10; Nota >= 0; Nota--) {
      while (!classificacao[Nota].vazia ()) {
        registro =
         (Registro) classificacao[Nota].retiraPrimeiro();
        i = 0; passou = false;
        while (i < Definicoes.nOpcoes && !passou) {
         if (vagas[registro.opcao[i]-1] > 0) {
          aprovados[registro.opcao[i]-1].insere(registro);
          vagas[registro.opcao[i]-1]--; passou = true;
         }
         i++;
        }
        if (!passou) reprovados.insere (registro);
      }
    }
  } catch (Exception e) {
          System.out.println (e.getMessage ()); }
  // Continua na próxima transparência
```

Vestibular - Refinamento Final (Cont.)

```
 for (i = 0; i < Definicoes.nCursos; i++) {
      System.out.println ("Relacao dos aprovados no Curso" +
                                                        (i + 1));
      aprovados[i].imprime ();
    }
    System.out.println ("Relacao dos reprovados");
    reprovados.imprime ();
  }
}
```

- 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. Cria uma pilha Vazia.
  2. Verifica se a lista está vazia. Retorna *true* se a pilha está vazia; caso contrário, retorna *false*.
  3. Empilhar o item x no topo da pilha.
  4. Desempilhar o item x no topo da pilha, retirando-o da pilha.
  5. Verificar o tamanho atual 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 *estruturas auto-referenciadas*.

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 $= 0$	x_1
1	x_2
	$\vdots$
topo -1	x_n
	$\vdots$
maxTam -1	

Estrutura e Operações sobre Pilhas Usando Arranjos

- Os itens são armazenados em um arranjo de tamanho suficiente para conter a pilha.
- O outro campo do mesmo registro contém uma referência para o item no topo da pilha.
- A constante $maxTam$ define o tamanho máximo permitido para a pilha.

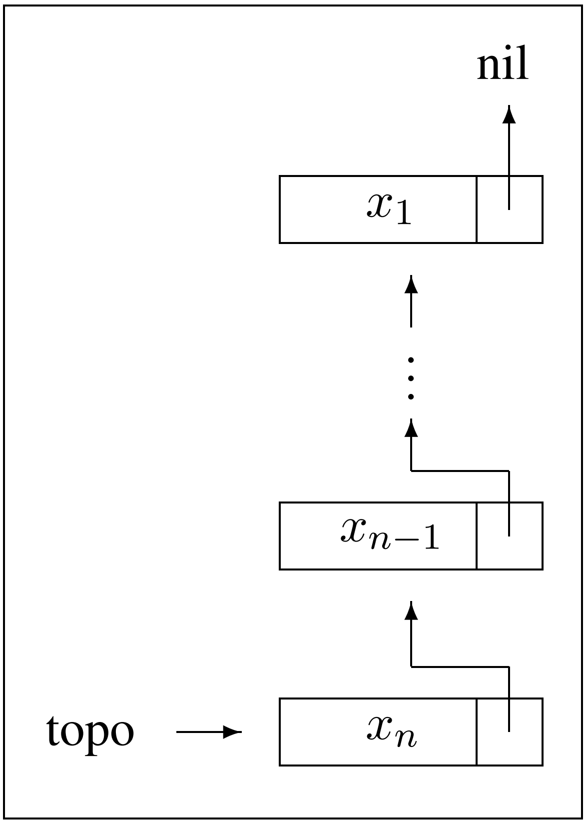
```
package cap3.arranjo;
public class Pilha {
  private Object item[];
  private int    topo;
  // Operações
  public Pilha (int maxTam) { // Cria uma Pilha vazia
    this.item = new Object[maxTam]; this.topo = 0;
  }
  public void empilha (Object x) throws Exception {
    if (this.topo == this.item.length)
      throw new Exception ("Erro: A pilha esta cheia");
    else this.item[this.topo++] = x;
  }
    // Continua na próxima transparência
```

Estrutura e Operações sobre Pilhas Usando Arranjos

```
  public Object desempilha () throws Exception {
    if (this.vazia())
      throw new Exception ("Erro: A pilha esta vazia");
    return this.item[--this.topo];
  }
  public boolean vazia () {
    return (this.topo == 0);
  }
  public int tamanho () {
    return this.topo;
  }
}
```

Implementação de Pilhas por meio de Estruturas Auto-Referenciadas

- Ao contrário da implementação de listas lineares por meio e estruturas auto-referenciadas não há necessidade de manter uma célula cabeça é no topo da pilha.
- Para desempilhar um item, basta desligar a célula que contém x_n e a célula que contém x_{n-1} passa a ser a célula de topo.
- Para empilhar um novo item, basta fazer a operação contrária, criando uma nova célula para receber o novo item.



Estrutura e operações sobre Pilhas Usando Estruturas Auto-Referenciadas

- O campo tam evita a contagem do número de itens no método $tamanho$.
- Cada célula de uma pilha contém um item da pilha e uma referência para outra célula.
- A classe $Pilha$ contém uma referência para o topo da pilha.

```
package cap3.autoreferencia;
public class Pilha {
  private static class Celula {
    Object item;
    Celula prox;
  }
  private Celula topo;
  private int    tam;
  // Operações
  public Pilha () { // Cria uma Pilha vazia
    this.topo = null; this.tam = 0;
  }
  // Continua na próxima transparência
```

Estrutura e operações sobre Pilhas Usando Estruturas Auto-Referenciadas

```
public void empilha (Object x) {
    Celula aux = this.topo;
    this.topo = new Celula ();
    this.topo.item = x;
    this.topo.prox = aux;
    this.tam++;
  }
  public Object desempilha () throws Exception {
    if (this.vazia ())
      throw new Exception ("Erro: A pilha esta vazia");
    Object item = this.topo.item;
    this.topo = this.topo.prox;
    this.tam--;
    return item;
  }
  public boolean vazia () {
    return (this.topo == null);
  }
  public int tamanho () {
    return this.tam;
  }
}
```

Exemplo de Uso Pilhas - Editor de Textos (ET)

- "#": cancelar caractere anterior na linha sendo editada. Ex.: UEM##FMB#G $\rightarrow$ 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.* $\rightarrow$
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

JAVA.*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 estruturas auto-referenciadas sem causar impacto no programa.

```
package cap3;
import cap3.arranjo.Pilha; // vide programa da transparência 11

public class ET {
  private class Definicoes {
    public static final int  maxTam          = 70;
    public static final char cancelaCarater = '#';
    public static final char cancelaLinha    = '\\';
    public static final char saltaLinha      = '*';
    public static final char marcaEof        = '~';
  }
  private static void imprime(Pilha pilha)throws Exception {
    Pilha pilhaAux = new Pilha (Definicoes.maxTam);
    Character x;
    // Continua na próxima transparência
```

ET - Implementação

```
    while (!pilha.vazia ()) {
      x = (Character) pilha.desempilha ();
      pilhaAux.empilha (x);
    }
    while (!pilhaAux.vazia ()) {
      x = (Character) pilhaAux.desempilha ();
      System.out.print (x);
    }
    System.out.print ('\n');
  }
  public static void main (String[] args) {
    Pilha pilha = new Pilha (Definicoes.maxTam);
    try {
      char c = (char) System.in.read ();
      Character x = new Character (c);
      if (x.charValue () == '\n') x =
                  new Character (' ');
      while (x.charValue () != Definicoes.marcaEof) {
        if (x.charValue () == Definicoes.cancelaCarater) {
          if (!pilha.vazia ()) x =
                  (Character) pilha.desempilha ();
        }
        else if (x.charValue () == Definicoes.cancelaLinha)
          pilha = new Pilha (Definicoes.maxTam);
            // Continua na próxima transparência
```

ET - Implementação

```
        else if (x.charValue () == Definicoes.saltaLinha)
          imprime (pilha);
        else {
          if (pilha.tamanho () == Definicoes.maxTam)
                  imprime (pilha);
          pilha.empilha (x);
        }
        c = (char) System.in.read ();
        x = new Character (c);
        if (x.charValue () == '\n') x = new Character (' ');
      }
      if (!pilha.vazia ()) imprime (pilha);
    } catch (Exception e) {
            System.out.println (e.getMessage ()); }
  }
}
```

Filas

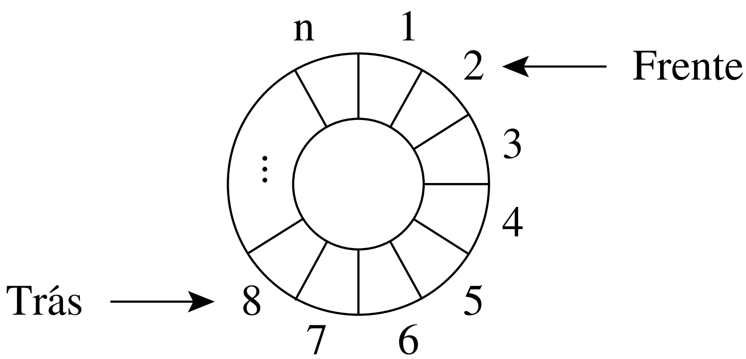
- É uma lista linear em que todas as inserções são realizadas em um extremo da lista, e todas as retiradas e, geralmente, os acessos são realizados no outro extremo da lista.
- O modelo intuitivo de uma fila é o de uma fila de espera em que as pessoas no início da fila são servidas primeiro e as pessoas 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

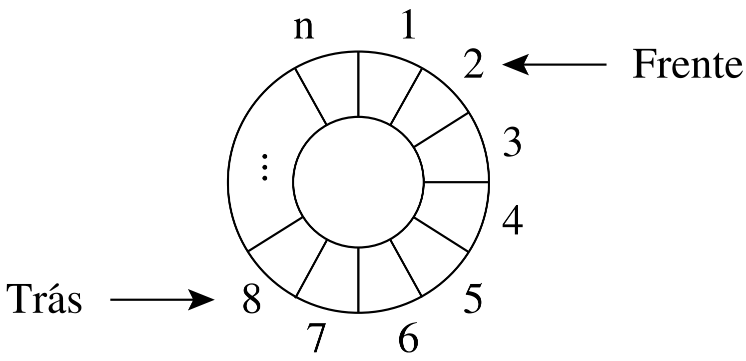
- Conjunto de operações:
  1. Criar uma fila vazia.
  2. Enfileirar o item x no final da fila.
  3. Desenfileirar. Essa função retorna o item x no início da fila e o retira da fila.
  4. Verificar se a fila está vazia. Essa função retorna *true* se a fila está vazia; do contrário, retorna *false*.

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 arranjo 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 arranjo circular é definido pela constante $maxTam$.
- Os outros campos da classe $Fila$ contêm referências para a parte da frente e de trás da fila.
- 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 arranjo.
- Neste caso, a fila está cheia quando $trás+1$ for igual a $frente$.
- A implementação utiliza aritmética modular nos procedimentos $enfileira$ e $desenfileira$ (% do Java).

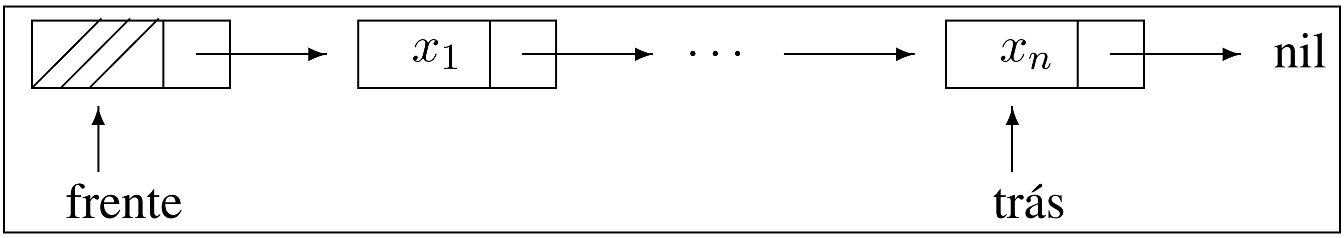
```
package cap3.arranjo;
public class Fila {
  private Object item[];
  private int    frente, tras;
    // Continua na próxima transparência
```

Estrutura e operações sobre Filas Usando arranjo

```
// Operações
  public Fila (int maxTam) { // Cria uma Fila vazia
    this.item = new Object[maxTam];
    this.frente = 0;
    this.tras = this.frente;
  }
  public void enfileira (Object x) throws Exception {
    if ((this.tras + 1) % this.item.length == this.frente)
      throw new Exception ("Erro: A fila esta cheia");
    this.item[this.tras] = x;
    this.tras = (this.tras + 1) % this.item.length;
  }
public Object desenfileira () throws Exception {
    if (this.vazia ())
      throw new Exception ("Erro: A fila esta vazia");
    Object item = this.item[this.frente];
    this.frente = (this.frente + 1) % this.item.length;
    return item;
  }
  public boolean vazia () {
    return (this.frente == this.tras);
  }
}
```

Implementação de Filas por meio de Estruturas Auto-Referenciadas

- 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 $trás$ referenciam 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 Estruturas Auto-Referenciadas

- A fila é implementada por meio de células.
- Cada célula contém um item da fila e uma referência para outra célula.
- A classe $Fila$ contém uma referência para a frente da fila (célula cabeça) e uma referência para a parte de trás da fila.

```
package cap3.autoreferencia;
public class Fila {
  private static class Celula { Object item; Celula prox; }
  private Celula frente;
  private Celula tras;
    // Operações
  public Fila () { // Cria uma Fila vazia
    this.frente = new Celula ();
    this.tras = this.frente;
    this.frente.prox = null;
  }
   // Continua na próxima transparência
```

Estrutura Operações da fila usando estruturas auto-referenciadas

```
  public void enfileira (Object x) {
    this.tras.prox = new Celula ();
    this.tras = this.tras.prox;
    this.tras.item = x;
    this.tras.prox = null;
  }
  public Object desenfileira () throws Exception {
    Object item = null;
    if (this.vazia ())
      throw new Exception ("Erro: A fila esta vazia");
    this.frente = this.frente.prox;
    item = this.frente.item;
    return item;
  }

  public boolean vazia () {
    return (this.frente == this.tras);
  }
}
```