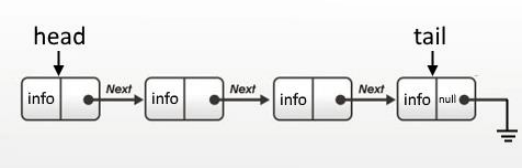
Universidade Cruzeiro do Sul

Estrutura de dados

Conteúdo

- Listas ordenadas.



Prof. Manuel Fdez. Paradela Ledón.

Revisão: ordenando os objetos de uma lista (LinkedList)

```
Comparator c =
    ((obj1, obj2) -> (((Trabalhador) obj1).compareTo(((Trabalhador) obj2))));
//O "comparador" anterior especifica como comparar trabalhadores e funciona
//porque a classe Trabalhador implementou o método compareTo.

LinkedList lista = new LinkedList();
lista.addLast(new Traballhador(.....));
lista.addLast(new Traballhador(.....));
.....
lista.sort(c);
//agora podemos mostrar os dados ordenados:
for (Object obj : lista) {
    System.out.println(obj.toString());
}
```

- Não seria muito eficiente ordenar cada vez que inserimos um objeto em uma lista, conhecendo que os melhores métodos de ordenação são, na média, de O(n log(n)).
- Lembre que **O(n)** é melhor que O(n log(n)), que é melhor que $O(n^2)$.
- Se utilizarmos uma lista ordenada, buscar sequencialmente a posição correta de inserção será, no pior caso e na média, de **O(n)**.

Listas dinâmicas encadeadas ordenadas

- Uma **lista ordenada**, como outras estruturas de dados, muda constantemente (por causa da inserção ou eliminação de elementos), mas sempre observa **a característica e necessidade de se manter ordenada após cada inserção ou remoção**.
- Devido a este comportamento altamente dinâmico, esta estrutura é melhor implementada com **alocação dinâmica encadeada**, pois a inserção e remoção de nós em posições internas da lista serão frequentes.
- Considerando um elemento qualquer dentro da lista, todos os elementos à sua esquerda são menores ou iguais e os elementos à direita são maiores ou iguais que ele.

Operações mais frequentes nas listas ordenadas

- **insert:** insere um novo elemento na lista ordenada;
- **remove:** procura e elimina um elemento específico da lista;
- **find:** procura um elemento na lista;
- **toString:** retorna todos os elementos da lista;
- **isEmpty:** retorna *true* se lista vazia, *false* em caso contrário;
- **clear:** destrói a lista, apagando todos os seus elementos;
- **size**: retorna a quantidade de objetos na lista;
- **toArray**: retorna um vetor com os objetos que estão na lista.

Veja que operações como insert(int pos), remove(int pos), addFirst() e addLast(), por exemplo, não serão implementadas (insert(pos) poderia desordenar a lista, remove(pos) não teria muito sentido, addFirst e addLast poderiam desordenar a lista). Também, a utilização de um ponteiro no final ou no centro da lista não é usual.

Exemplo

	ListaOrd: [ ] (vazia)
insert(c)	ListaOrd: [c]
insert(a) insert(h) insert(b) insert(e) remove(e)	ListaOrd: [a, c] ListaOrd: [a, c, h] ListaOrd: [a, b, c, h] ListaOrd: [a, b, c, e, h] ListaOrd: [a, b, c, h]
remove(m)	ListaOrd: [a, b, c, h]
clear()	ListaOrd: []

Universidade Cruzeiro do Sul

Implementação de uma lista ordenada em Java, como projeto NetBeans

(veja os projetos em ExemploListaOrd.zip e ExemploListaOrdAlunos.zip)

```
package exemplolistaord;

class ListaOrd <E extends Comparable<E>> {

    private Node head;   //se a lista estiver vazia → head=null
    private int size = 0;   //qtde. de objetos na lista

    //Cria uma Lista no estado vazia (construtor)
    public ListaOrd() {
        head = null;  size = 0;
    }

    //Verifica se a lista está vazia
    public boolean isEmpty() {
        return (head == null);   //if(head==null) return true; else return false;
    }
```

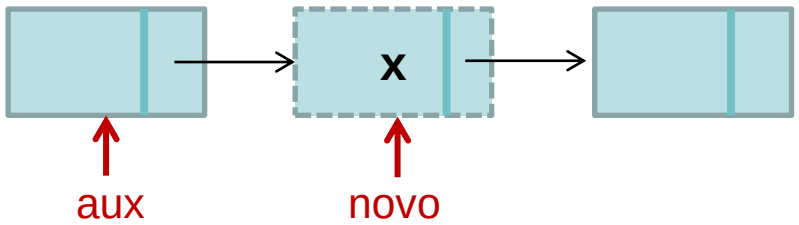
```
public int size () {
        return this.size;
}

public void clear () {  //limpar o conteúdo da lista ordenada
        Node aux = head;
        while (aux != null) {
            Node tmp = aux;
            aux = aux.getNext();
            tmp.setNext(null);  // liberar memória
        }
        head = null;  size = 0;
}
```

```
public Object[] toArray() {
   //retorna um vetor com os objetos guardados na lista ordenada, o que poderia
   //ser útil para efetuar qualquer processamento em geral
   if(isEmpty()) return null;  //operação impossível se a fila estiver vazia
   Object vet[] = new Object[size];
   Node aux = head;
   for(int i=0; i<size; i++) {
      vet[i] = aux.getValue();
      aux = aux.getNext();
   }
   return vet;
 }
```

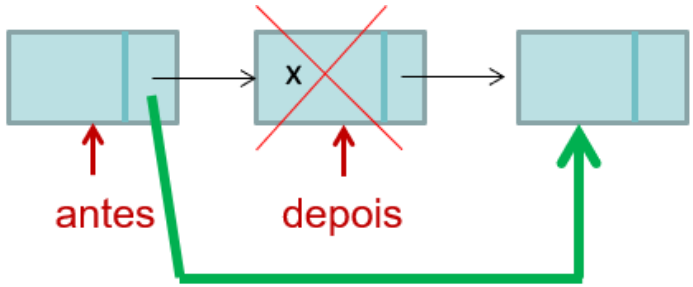
```
public <E extends Comparable<E>> E insert (E x) {
    if (x == null || Runtime.getRuntime().freeMemory() < x.toString().length() + 1024)
        return null;
    Node novo = new Node();  //solicitamos memória para o novo nodo
    novo.setValue(x);  novo.setNext(null);  size++;
    if (isEmpty() || x.compareTo((E) head.getValue()) < 0) { //lista vazia ou x menor que o 1º
       novo.setNext(head);
       head = novo;
    } else {
       Node aux = head;
       while (aux.getNext() != null && x.compareTo((E) aux.getNext().getValue()) > 0)
           aux = aux.getNext();
       novo.setNext(aux.getNext());
       aux.setNext(novo);
    }
    return x;
}
```

Inserção em uma lista que está ordenada em **ordem crescente**



```
public <E extends Comparable<E>> E remove (E x) {    //eliminar o objeto x
    if (isEmpty() || x == null || x.compareTo((E) head.getValue()) < 0) { //pré-condições
       return null;  //se lista vazia, ou o valor x é menor que o primeiro da lista, ou x é nulo
    } else {
       size--;  //um objeto está sendo eliminado
       if (x.compareTo((E)head.getValue()) == 0) {  //se x for o primeiro elemento da lista
          x = (E)head.getValue();  //para pegar todos os dados do objeto
          head = head.getNext();  //avançando head eliminamos o nodo
          return x;
       } else {  //se for maior que o primeiro, então tentar achar a posição:
          Node antes = head, depois = head.getNext();
          while (depois != null && x.compareTo((E) depois.getValue()) > 0) {
             antes = antes.getNext();   //movemos simultaneamente os dois ponteiros
             depois = depois.getNext();
          }
          if (depois != null && x.compareTo((E) depois.getValue()) == 0) {
             antes.setNext(depois.getNext());  //encontramos e eliminaremos o objeto x, caso geral
             return (E)depois.getValue();
          } else  return null;  //não encontramos o objeto x
       }
    }
}
```

Eliminar um item de uma lista que está ordenada em **ordem crescente**



```
//Retorna o conteúdo da lista ordenada, os objetos separados por vírgulas
public String toString () {
   if (!isEmpty()) {
      Node aux;
      aux = head;
      String saida = "";
      while (aux != null) {
         saida += aux.getValue().toString();
         aux = aux.getNext();  //avançar o ponteiro aux
         if (aux != null) saida += ", ";
      }
      return "ListaOrd: [" + saida + "]";
   } else {
      return "ListaOrd: []";
   }
}
```

```
//Retorna o conteúdo da lista ordenada, os objetos em linhas diferentes
public String toString2 () {
   if (!isEmpty()) {
      Node aux;
      aux = head;
      String saida = "";
      while (aux != null) {
         saida += "   " + aux.getValue().toString();
         aux = aux.getNext();  //avançar o ponteiro aux
         if (aux != null) saida += "\n";
      }
      return "ListaOrd: [\n" + saida + "\n]";
   } else {
      return "ListaOrd: []";
   }
}
```

```
//Buscar um objeto na lista e retornar a posição onde foi encontrado

public int find (E x) {
    if (x == null) return -1;
    //navegamos até encontrar o nó de interesse:
    Node aux = head;  int pos = 0;
    while (aux != null && x.compareTo((E) aux.getValue()) > 0) {
        aux = aux.getNext(); pos++;
    }
    //se achamos o elemento:
    if (aux != null && x.compareTo((E) aux.getValue()) == 0) {
        return pos;
    } else {
        return -1; //não encontramos o objeto x
    }
}
```

```
ListaOrd list = new ListaOrd();
System.out.println( list.toString() );
list.insert(10);
list.insert(9);
list.insert(2);
list.insert(6);
list.insert(5);
list.insert(10);
list.insert(25);
list.insert(98);
list.insert(55);
list.insert(1);
list.insert(14);
System.out.println( "Lista ordenada inicialmente: \n" + list.toString() );

int pos = list.find(9);
if (pos != -1) System.out.println( "Encontramos o 9 na posição " +   pos );

pos = list.find(10);
if (pos != -1) System.out.println( "Encontramos o 1º 10 na posição " +   pos );
System.out.println( list.toString() );
System.out.println( "Eliminamos o segundo " + list.remove(10) + ":");
System.out.println( list.toString() );
```

Para testar a classe ListOrd, em ExemploListaOrd.zip

```
System.out.println( "Eliminamos o " + list.remove(25) + ":");
System.out.println( list.toString() );
System.out.println( "Eliminamos o " + list.remove(2)  + ":");
System.out.println( list.toString() );
System.out.println( "Eliminamos o " + list.remove(1)  + ":");
System.out.println( list.toString() );
System.out.println( "Eliminamos o " + list.remove(98)  + ":");
System.out.println( list.toString() );
list.clear();
System.out.println( "\nEsvaziamos a lista (clear) e ficou: \n" + list.toString() );
```

Para testar a classe ListOrd, em ExemploListaOrd.zip

```
System.out.println("\nAgora vamos criar uma lista ordenada de objetos strings...");
    ListaOrd listn = new ListaOrd();
    System.out.println( "Inicialmente vazia: " + listn.toString() );
    listn.insert("Julio");
    listn.insert("Ana");
    listn.insert("Lucas");
    listn.insert("Betty");
    listn.insert("Jenildo");
    listn.insert("Kaio");
    listn.insert("Zoe");
    listn.insert("Amilton");
    System.out.println( listn.toString() );
```

A saída do programa anterior será:

ListaOrd: []
Lista ordenada inicialmente:
ListaOrd: [1, 2, 5, 6, 9, 10, 10, 14, 25, 55, 98]
Encontramos o 9 na posição 4
Encontramos o 1º 10 na posição 5
Eliminamos o primeiro 10:
ListaOrd: [1, 2, 5, 6, 9, 10, 14, 25, 55, 98]
Eliminamos o segundo 10:
ListaOrd: [1, 2, 5, 6, 9, 14, 25, 55, 98]
Eliminamos o 25:
ListaOrd: [1, 2, 5, 6, 9, 14, 55, 98]
Eliminamos o 2:
ListaOrd: [1, 5, 6, 9, 14, 55, 98]
Eliminamos o 1:
ListaOrd: [5, 6, 9, 14, 55, 98]
Eliminamos o 98:
ListaOrd: [5, 6, 9, 14, 55]
Esvaziamos a lista (clear) e ficou:
ListaOrd: []

Agora vamos criar uma lista ordenada de objetos strings...
Inicialmente vazia: ListaOrd: []
ListaOrd: [Amilton, Ana, Betty, Jenildo, Julio, Kaio, Lucas, Zoe]

Universidade Cruzeiro do Sul

Exemplo com objetos da classe **Aluno**, em ExemploListaOrdAlunos.zip

```
package exemplolistaord;

public class ExemploListaOrd {

    public static void main(String[] args) {
        ListaOrd listaAlunos = new ListaOrd();
        listaAlunos.insert(new Aluno("8888", "Caio Silva", 54, 'M'));
        listaAlunos.insert(new Aluno("7777", "Roberto Ledón", 92, 'M'));
        listaAlunos.insert(new Aluno("6666", "Zoe Valdes", 47, 'M'));
        listaAlunos.insert(new Aluno("2222", "Amilton Alves", 29, 'M'));
        listaAlunos.insert(new Aluno("4444", "Ana Lopes", 26, 'F'));
        listaAlunos.insert(new Aluno("9999", "Juan Felipe", 87, 'M'));
        listaAlunos.insert(new Aluno("1111", "Renata Souza", 24, 'F'));
        listaAlunos.insert(new Aluno("3333", "Pedro Lima", 25, 'M'));
        System.out.println("\n\nLista de objetos (Aluno) em ordem crescente pelos nomes: \n"
                + listaAlunos.toString2() );
    }

}
```

Lista de objetos da classe Aluno, em ordem crescente pelos RGMs:

```
ListaOrd: [
  Aluno{rgm=1111, nome=Renata Souza, idade=24, sexo=F}
  Aluno{rgm=2222, nome=Amilton Alves, idade=29, sexo=M}
  Aluno{rgm=3333, nome=Pedro Lima, idade=25, sexo=M}
  Aluno{rgm=4444, nome=Ana Lopes, idade=26, sexo=F}
  Aluno{rgm=6666, nome=Zoe Valdes, idade=47, sexo=M}
  Aluno{rgm=7777, nome=Roberto Ledón, idade=92, sexo=M}
  Aluno{rgm=8888, nome=Caio Silva, idade=54, sexo=M}
  Aluno{rgm=9999, nome=Juan Felipe, idade=87, sexo=M}
]
```

ExemploListaOrdAlunos.zip

(o método **compareTo** da classe **Aluno** decide o critério de ordenação da lista)

Lista de objetos da classe Aluno, em ordem crescente pelos nomes:

```
ListaOrd: [
  Aluno{rgm=2222, nome=Amilton Alves, idade=29, sexo=M}
  Aluno{rgm=4444, nome=Ana Lopes, idade=26, sexo=F}
  Aluno{rgm=8888, nome=Caio Silva, idade=54, sexo=M}
  Aluno{rgm=9999, nome=Juan Felipe, idade=87, sexo=M}
  Aluno{rgm=3333, nome=Pedro Lima, idade=25, sexo=M}
  Aluno{rgm=1111, nome=Renata Souza, idade=24, sexo=F}
  Aluno{rgm=7777, nome=Roberto Ledón, idade=92, sexo=M}
  Aluno{rgm=6666, nome=Zoe Valdes, idade=47, sexo=M}
]
```

Lista de objetos da classe Aluno, em ordem crescente pelas idades:

```
ListaOrd: [
  Aluno{rgm=1111, nome=Renata Souza, idade=24, sexo=F}
  Aluno{rgm=3333, nome=Pedro Lima, idade=25, sexo=M}
  Aluno{rgm=4444, nome=Ana Lopes, idade=26, sexo=F}
  Aluno{rgm=2222, nome=Amilton Alves, idade=29, sexo=M}
  Aluno{rgm=6666, nome=Zoe Valdes, idade=47, sexo=M}
  Aluno{rgm=8888, nome=Caio Silva, idade=54, sexo=M}
  Aluno{rgm=9999, nome=Juan Felipe, idade=87, sexo=M}
  Aluno{rgm=7777, nome=Roberto Ledón, idade=92, sexo=M}
]
```

Universidade Cruzeiro do Sul

Na classe **Aluno**, em ExemploListaOrdAlunos.zip

```
/*
   public int compareTo(Object outro) {
      //Para ter uma lista ordenada pelos nomes dos alunos:
      Aluno al = (Aluno)outro;
      if(getNome().compareToIgnoreCase(al.getNome()) < 0)return -1;
      else if (getNome().compareToIgnoreCase(al.getNome()) == 0)return 0;
      else return 1;
   }
```

```
public int compareTo(Object outro) {
      //Para uma lista ordenada pelas idades:
      Aluno al = (Aluno)outro;
      if(getIdade() < al.getIdade())return -1;
      else if (getIdade() == al.getIdade())return 0;
      else return 1;
}
*/
```

```
public int compareTo(Object outro) {
   //Para ter a lista ordenada pelos RGMs dos alunos:
   Aluno al = (Aluno)outro;
   if(getRgm().compareToIgnoreCase(al.getRgm()) < 0)return -1;
   else if (getRgm().compareToIgnoreCase(al.getRgm()) == 0)return 0;
   else return 1;
}
```

```
//exemplo: um processamento geral: contar homens e mulheres:
int qm=0, qf=0;
Object vet[] = listaAlunos.toArray();
for(int i=0; i < vet.length; i++) {
    Aluno alu = (Aluno)vet[i];  //conversão de tipo
    if(alu.getSexo() == 'M') qm++;  else qf++;
}
System.out.println("Quantidade de sexo M: " + qm);
System.out.println("Quantidade de sexo F: " + qf);
```

ExemploListaOrdAlunos.zip

Lista de objetos da classe Aluno, em ordem crescente pelos nomes:

ListaOrd: [

Aluno{rgm=2222, nome=Amilton Alves, idade=29, sexo=M}

Aluno{rgm=4444, nome=Ana Lopes, idade=26, sexo=F}

Aluno{rgm=8888, nome=Caio Silva, idade=54, sexo=M}

Aluno{rgm=9999, nome=Juan Felipe, idade=87, sexo=M}

Aluno{rgm=3333, nome=Pedro Lima, idade=25, sexo=M}

Aluno{rgm=1111, nome=Renata Souza, idade=24, sexo=F}

Aluno{rgm=7777, nome=Roberto Ledón, idade=92, sexo=M}

Aluno{rgm=6666, nome=Zoe Valdes, idade=47, sexo=M}

]

Quantidade de sexo M: 6

Quantidade de sexo F: 2

ExemploListaOrdAlunos.zip

Bibliografia sugerida

BIBLIOGRAFIA BÁSICA	BIBLIOGRAFIA COMPLEMENTAR
CORMEN, T. H.; et al. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012. GOODRICH, M. T.; TAMASSIA, R. Estrutura de dados e algoritmos em java. 5. ed. Porto Alegre: Bookman, 2013. (livro físico e e-book) CURY, T. E., BARRETO, J. S., SARAIVA, M. O., et al. Estrutura de Dados (1. ed.) ISBN 9788595024328, Porto Alegre: SAGAH, 2018 (e-book)	ASCENCIO, A. F. G.; ARAÚJO, G. S. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. São Paulo: Pearson Prentice Hall, 2010. (eBook) PUGA, S.; RISSETTI, G. Lógica de programação e estruturas de dados, com aplicações em Java. 3. ed. São Paulo: Pearson Education do Brasil, 2016. (eBook) DEITEL, P.;DEITEL, H. Java como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017. (eBook) BARNES, D. J.; KOLLING, M. Programação Orientada a Objetos com Java: uma introdução prática usando o Blue J. São Paulo: Pearson Prentice Hall, 2004. (eBook) BORIN, V. POZZOBON. Estrutura de Dados. ISBN: 9786557451595, Edição: 1ª . Curitiba: Contentus, 2020 (e-book)