Universidade Cruzeiro do Sul

Estrutura de dados

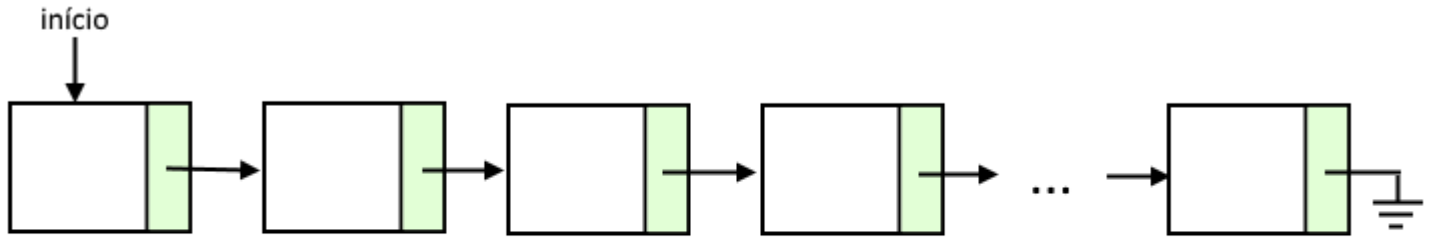
Conteúdo

- Listas dinâmicas lineares duplamente encadeadas.
- Listas circulares.

Elaboração

Prof. Manuel Fernández Paradela Ledón

Revisão: lista dinâmica linear simplesmente encadeada



Lista dinâmica linear simplesmente encadeada

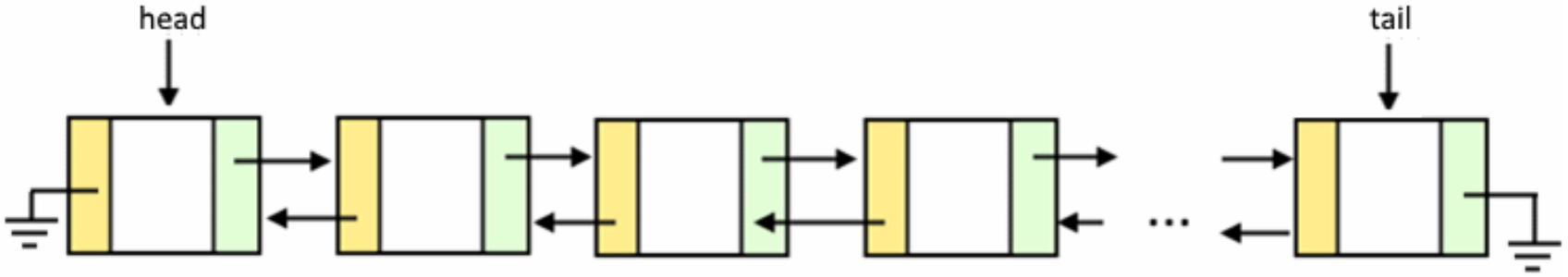
- Será necessário conhecer o ponteiro (endereço) do primeiro elemento na lista para poder acessar os elementos na mesma. Normalmente utilizamos dois ponteiros: head e tail.
- O percurso ou varredura será feita em um único sentido.
- O último elemento da lista possui enlace nulo.
- Poderíamos ter outros ponteiros como referências a estas listas.

Revisão: estruturas de dados simplesmente encadeadas

```
public class Node {
    Object value;
    Node next;
   ...
}

public class LinkedList {
    Node head, tail; //ou só head
   ...
}
```

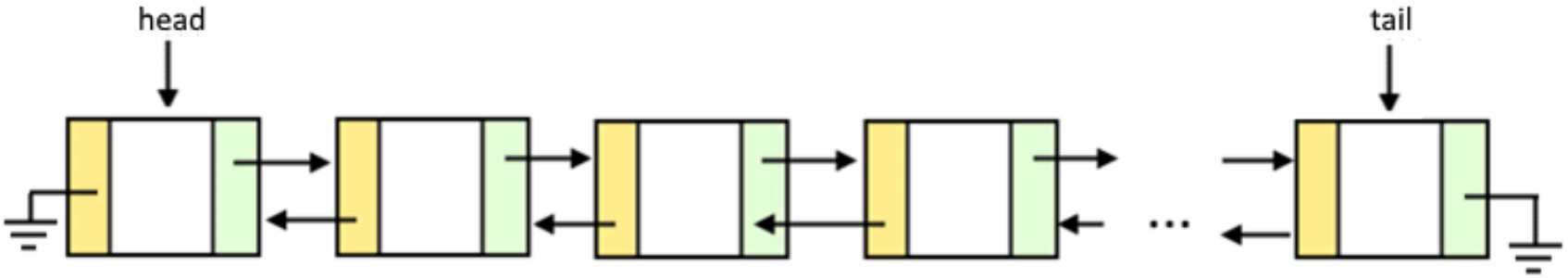
```
public class Queue {   // fila dinâmica
     Node head, tail;  // dois ponteiros
   ...
}

public class Stack {   // stack ou pilha dinâmica
     Node top;  // ponteiro ao topo da pilha
   ...
}

public class Queue extends LinkedList {
    // ou implementar a fila derivando de LinkedList
}
```

Listas dinâmicas lineares duplamente encadeadas



- É suficiente conhecer um ponteiro ou endereço de qualquer elemento na lista para poder acessar os restantes elementos na mesma. Mas poderíamos ter outros.
- O percurso ou varredura poderá ser feita nos dois sentidos.
- Conhecido o endereço de um item e percorrendo em um único sentido podemos visitar uma parte dos elementos, mas nem sempre todos.
- Dependo do critério de armazenamento (ex. elementos ordenados) algumas operações, como procurar ou inserir um item, poderiam ser mais eficientes.
- Cada nodo ou nó da lista possui dois endereços, um deles apontando para o elemento anterior e outro para o próximo => mais memória.
- As operações em uma lista duplamente ligada serão, em geral, mais complexas.
- O nodos terminais da lista possuem enlaces nulos.
- Poderíamos utilizar um nodo cabeça (fictício, sentinela).

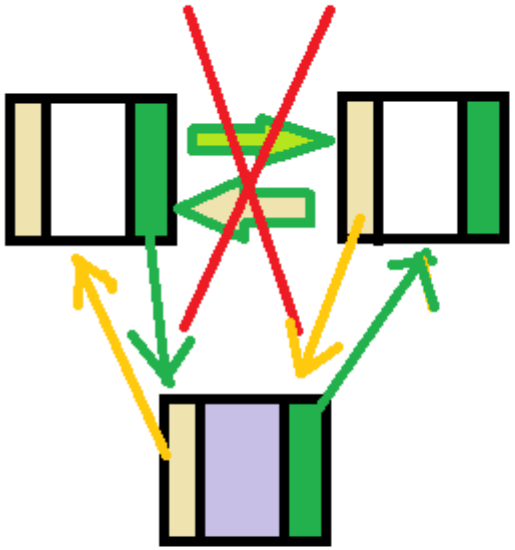
Listas dinâmicas lineares duplamente encadeadas




```
public class Node {
   Object value;
   Node prev, next;
   ...
}
```

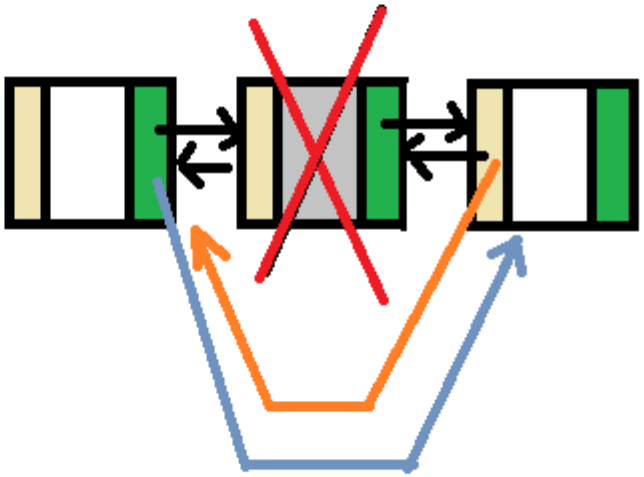
```
public class ListaDuplamenteLigada {
    Node head, tail;  // um único ponteiro é suficiente,
                      // mas poderíamos usar outros ponteiros,
                      // como mostrado
    ...
}
```

Listas dinâmicas lineares duplamente encadeadas



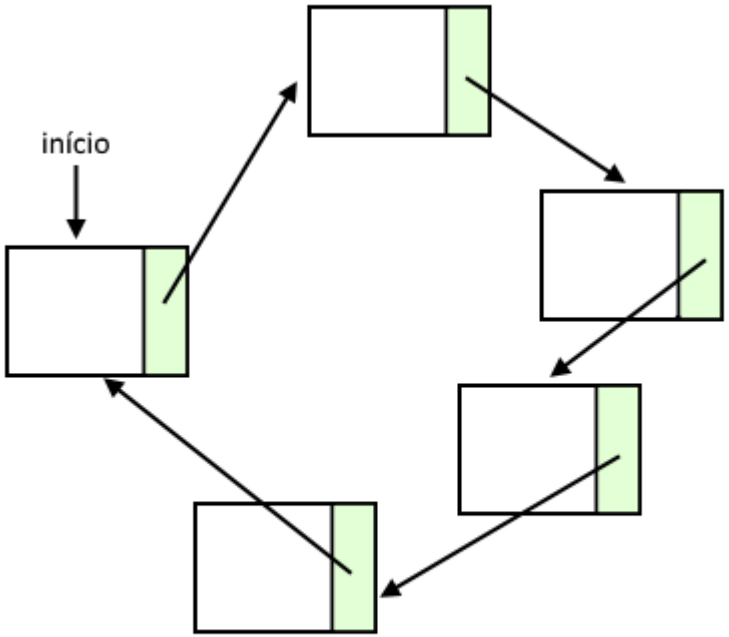
Inserir um novo nó

Eliminar um nó intermediário



Observe a **complexidade** de atualização dos enlaces em operações de inserção ou eliminação.

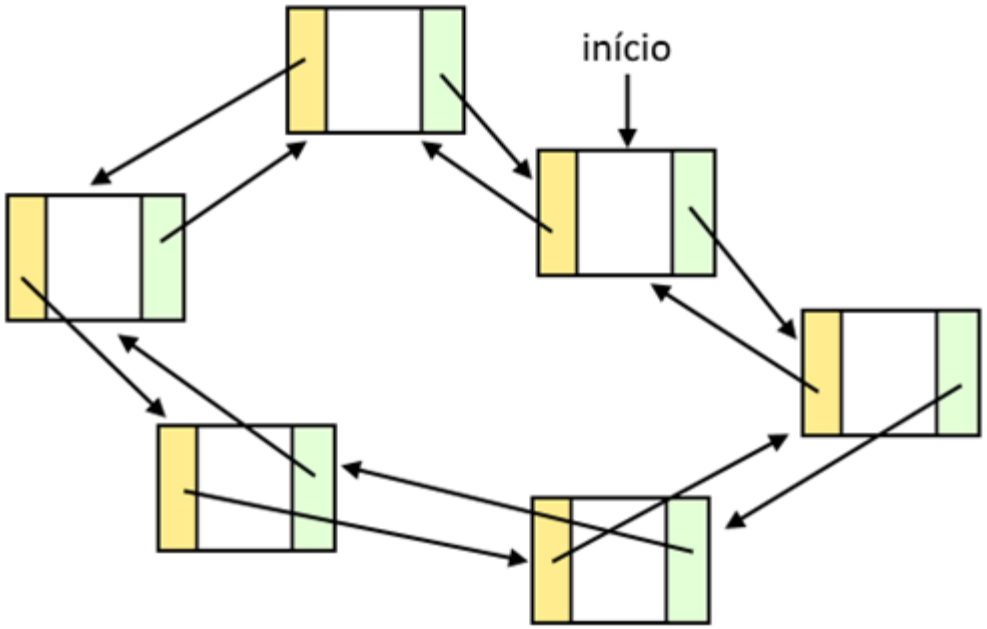
Listas circulares dinâmicas simplesmente encadeadas



Lista circular simplesmente encadeada

- É suficiente conhecer um ponteiro ou endereço de qualquer elemento para poder acessar todos os elementos na lista, ou seja, conhecido o endereço de um item qualquer e percorrendo em um único sentido podemos visitar todos os elementos da lista.
- O percurso ou varredura será feita em um único sentido, por exemplo, em "sentido horário".
- Cada elemento aponta para o próximo na lista.
- Não existem itens com o endereço do próximo elemento com valor nulo (somente no caso de uma lista circular com um único item, mas o enlace poderia apontar para ele mesmo).

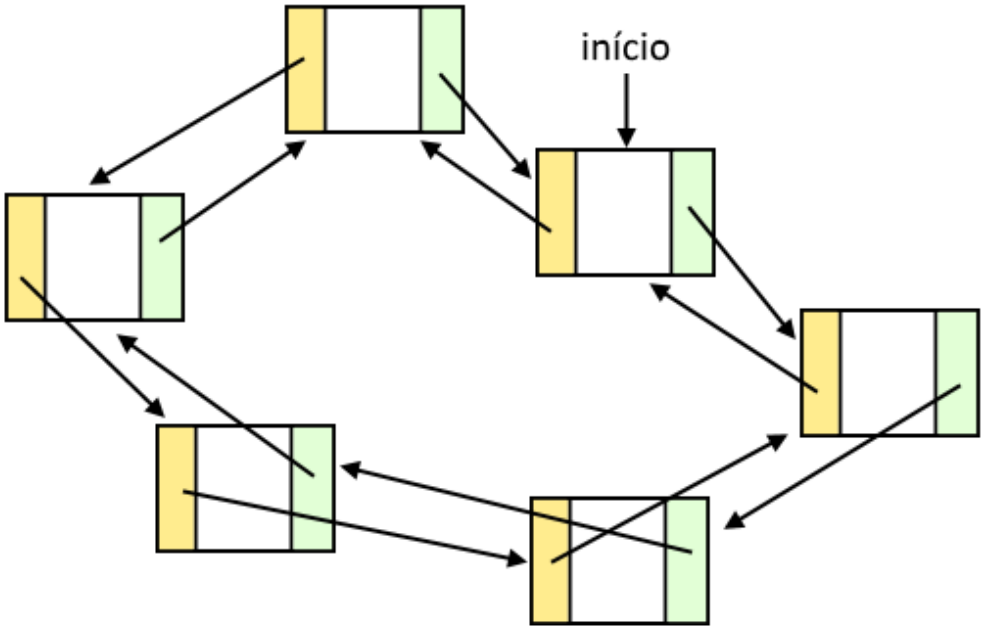
Listas circulares dinâmicas duplamente encadeadas



Lista circular duplamente encadeada

- É suficiente conhecer um ponteiro (endereço) de qualquer elemento na lista para poder acessar todos os elementos na mesma.
- O percurso ou varredura poderá ser feita em ambos sentidos (sentido horário e anti-horário).
- Algumas operações, como procura de um item, poderiam ser mais eficientes com este encadeamento duplo.
- Cada nodo ou nó possui dois endereços, um deles apontando para o elemento anterior e outro para o próximo.
- Não existem itens com os endereços dos enlaces com valores nulos (somente no caso de uma lista circular com um único item, mas os enlaces poderiam apontar para ele mesmo).

Listas circulares dinâmicas duplamente encadeadas

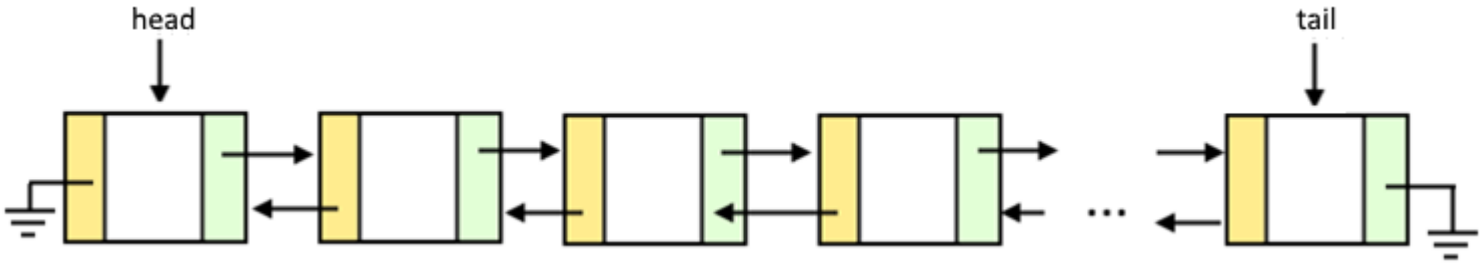


Lista circular duplamente encadeada

```
public class Node {
   Object value;
   Node prev, next;
   ...
}

public class Lista {
     Node inicio;
   ...
}
```

DoubleSortedLinkedList: implementação completa em Java

- Classe **Node**.
- Projeto com a classe **DoubleSortedLinkedList** que implementa a lista duplamente ligada, encadeada, ordenada.
  - Métodos: isEmpty, add, remove, find, clear, toString, toStringReverse.



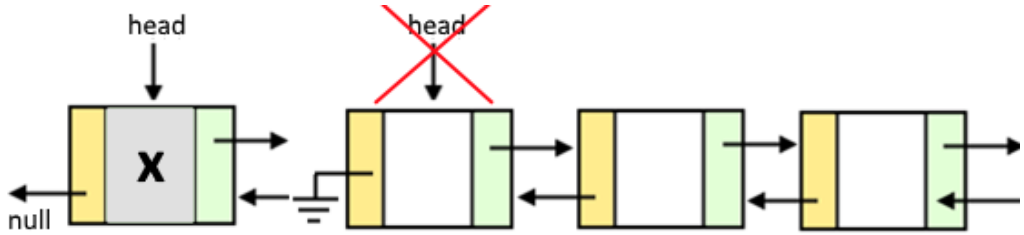
- Classe principal **ExemploListaDuplamOrd** e a classe **Aluno**, para efetuar diferentes testes.
- Veja tudo no **projeto NetBeans ListaDuplamenteEncadeada**.

```
public class Node {

    private Object value;          // dado do nó
    private Node prev, next;       // referências  anterior e próximo

        public String toString () {
             return value.toString();
        }

        //Devolve o conteúdo do nó
        public Object getValue() {
            return value;
        }

        //Atribui um valor ao conteúdo do nó
        public void setValue(Object novovalor) {
            value = novovalor;
        }

        //Devolve a referência do próximo nó
        public Node getNext() {
            return next;
        }

        //Atribui uma referência para o próximo nó
        public void setNext(Node prox) {
              next = prox;
         }

        //Devolve a referência do nó anterior
        public Node getPrev() {
            return prev;
        }

        //Atribui uma referência para o nó anterior
        public void setPrev(Node prev) {
            this.prev = prev;
        }

}
```

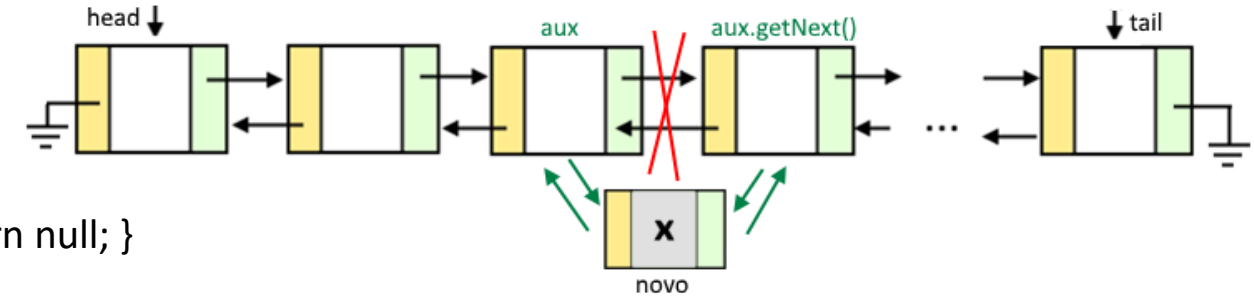
```
class DoubleSortedLinkedList <E> {

   private Node head, tail;   // nós referências do começo e final da lista

   //Inicializa uma lista no estado vazia (construtor)
   public DoubleSortedLinkedList () {   // O(1)
      head = null;
      tail = null;
   }

   //Verifica se a lista está vazia
   public boolean isEmpty () {    // O(1)
      return (head == null);
   }
```

```
   public Node getHead () {  // O(1)
      return head;
    }

    public Node getTail () {  // O(1)
      return tail;
    }
```

```
//Insere um elemento na lista (dinâmica, ordenada-crescente, duplamente encadeada)
public Object add  (Comparable x)  {    // O(n)
   if (x == null) return null;
   try {
      Node novo = new Node();
      Node aux;
      novo.setValue(x);
      //Se a lista estiver vazia ou o elemento for menor que o primeiro:
      if (isEmpty() || x.compareTo(head.getValue()) < 0) {
         if(!isEmpty())head.setPrev(novo); else tail = novo;
         novo.setNext(head);
         novo.setPrev(null);
         head = novo;
      } else {  //casos restantes - inserir no meio ou no final:
         aux = head;
         while (aux.getNext() != null && x.compareTo(aux.getNext().getValue()) > 0)  aux = aux.getNext();
         novo.setNext(aux.getNext());
         if(aux.getNext() == null)tail = novo; //caso seja inserido no final
            else aux.getNext().setPrev(novo);
         aux.setNext(novo);
         novo.setPrev(aux);
      }
      return x;
   } catch (Exception exMem) { return null; }
}
```

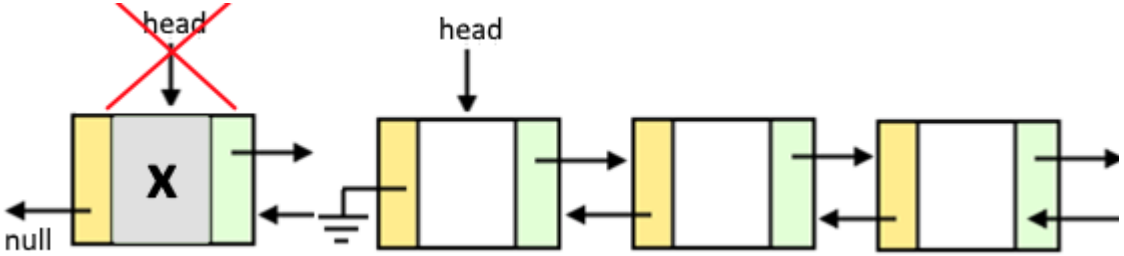


Complexo!
Em provas: só as lógicas das ligações.



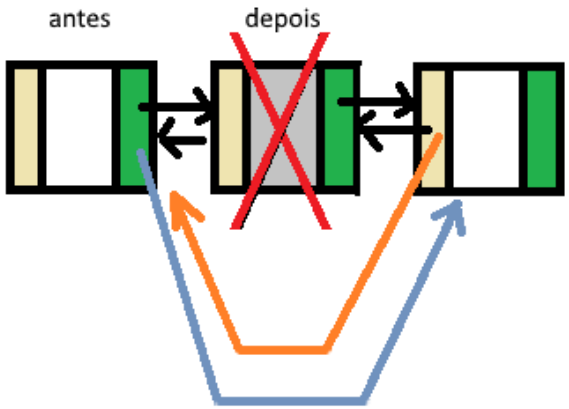
```
//Remove um elemento da lista
public Object remove (Comparable x) {    // O(n)
   Node antes = null, depois = null;
   //Se elemento nulo, lista vazia ou o elemento for menor que o primeiro (não está na lista):
   if (x == null || isEmpty() || x.compareTo(head.getValue()) < 0) return null;
   else {
          if (x.compareTo(head.getValue()) == 0) {   //se for o primeiro elemento
              head = head.getNext();
              if(head == null)tail = null;    //porque foi eliminado o único
                  else head.setPrev(null);
             return x;
          }
```

Complexo!
Em provas: só as lógicas das ligações.



```
    else {  // se for maior que o primeiro, tentamos achar a posição do elemento a eliminar:
            antes = head;
            depois = head.getNext();
            //navega até a posição onde se encontra o valor buscado:
            while (depois != null  &&  x.compareTo(depois.getValue()) > 0) {
               antes = antes.getNext();        //avançamos ponteiro
               depois = depois.getNext();    //avançamos ponteiro
            }
            //verificamos se encontrou ou se chegou no final da lista:
            if (depois != null  &&  x.compareTo(depois.getValue()) == 0) {
                  antes.setNext(depois.getNext());
                  if(antes.getNext() == null) tail = antes;    //porque foi eliminado o último
                  if(depois.getNext() != null) depois.getNext().setPrev(antes);
                  return x;
            }
            else return null;   //não encontrou o elemento
       }
    }
}
```

Complexo!
Em provas: só as lógicas das ligações.



```
//Buscar um objeto na lista e retornar a posição onde o encontrou
public int find (Comparable x) {
   if (x == null) return -1;
   //navegamos para o nodo de interesse:
   Node aux = head;
   int pos = 0;
   while (aux != null && x.compareTo(aux.getValue()) > 0) {
      aux = aux.getNext();
      pos++;
   }
   //se achamos o elemento:
   if (aux != null && x.compareTo(aux.getValue()) == 0) {
      return pos;
   } else {
      return -1;  //não encontramos o objeto x
   }
}
```

```
public void clear () {  //O(n)
      Node aux = head;
      while (aux != null) {
         Node temp = aux;
         aux = aux.getNext();
         temp.setNext(null);
      }
      head = null;  tail = null;
}
```

```
//Retorna o conteúdo da Lista
public String toString () {    // O(n)
   if (!isEmpty()) {
      Node aux;
      aux = head;
      String saida = "";
      while (aux != null) {
         saida += aux.getValue().toString();
         aux = aux.getNext();
         if (aux != null) {
            saida += ", ";
         }
      }
      return "Lo: [" + saida + "]";
   } else {
      return "Lo: []";
   }
}
```

```
//Retorna o conteúdo da Lista
   public String toString2 () {    // O(n)
      if (!isEmpty()) {
         Node aux;
         aux = head;
         String saida = "";
         while (aux != null) {
            saida += aux.getValue().toString();
            aux = aux.getNext();
            if (aux != null) {
               saida += "\n";
            }
         }
         return "Lo: [\n" + saida + "\n]";
      } else {
         return "Lo: []";
      }
   }
```

```
//Retorna o conteúdo da Lista
public String toStringReverse () {   // O(n)
   if (!isEmpty()) {
      Node aux;
      aux = tail;
      String saida = "";
      while (aux != null) {
         saida += aux.getValue().toString();
         aux = aux.getPrev();
         if (aux != null) {
            saida += ", ";
         }
      }
      return "LoInv: [" + saida + "]";
   } else {
      return "LoInv: []";
   }
}
```

```
//Retorna o conteúdo da Lista
public String toStringReverse2 () {    // O(n)
   if (!isEmpty()) {
      Node aux;
      aux = tail;
      String saida = "";
      while (aux != null) {
         saida += aux.getValue().toString();
         aux = aux.getPrev();
         if (aux != null) {
            saida += "\n";
         }
      }
      return "LoInv: [\n" + saida + "\n]";
   } else {
      return "LoInv: []";
   }
}
```

Universidade Cruzeiro do Sul

Testando DoubleSortedLinkedList com exemplos

```
DoubleSortedLinkedList<Integer> list = new DoubleSortedLinkedList<Integer>();
System.out.println("Adicionamos " + list.add(10) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(9) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(7) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(2) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(5) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(1) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(14) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(7) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(15) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(8) + " e a lista fica: " + list.toString());
System.out.println("Adicionamos " + list.add(3) + " e a lista fica: " + list.toString());
System.out.println(list.toString());
System.out.println("Percorrendo em ordem inversa:");
System.out.println(list.toStringReverse());
list.clear();
System.out.println("Esvaziamos a lista (clear) e ficou: \n" + list.toString());
```

Testando DoubleSortedLinkedList com exemplos

```
System.out.println("Uma lista ordenada, duplamente ligada, com strings");
DoubleSortedLinkedList listnomes = new DoubleSortedLinkedList();
listnomes.add("Julio");
listnomes.add("Ana");
listnomes.add("Lucas");
listnomes.add("Betty");
listnomes.add("Jenildo");
listnomes.add("Amilton");
System.out.println(listnomes.toString());
System.out.println("Que em ordem reversa será:\n" +
                                listnomes.toStringReverse());
listnomes.clear();
System.out.println(listnomes.toString());
```

Testando DoubleSortedLinkedList com exemplos

```
System.out.println("Vamos criar uma lista ordenada, duplamente ligada,  " +
                   "ordenada pelos nomes, com objetos da classe Aluno");
DoubleSortedLinkedList listaAlunos = new DoubleSortedLinkedList();
listaAlunos.add(new Aluno("999-9", "Mary", 'F', 9.5f));
listaAlunos.add(new Aluno("111-1", "Luiz", 'M', 4.5f));
listaAlunos.add(new Aluno("666-6", "Ana", 'F', 9.2f));
listaAlunos.add(new Aluno("444-4", "Betty", 'F', 9.0f));
listaAlunos.add(new Aluno("888-8", "Caio", 'M', 5.5f));
listaAlunos.add(new Aluno("333-3", "Lara", 'F', 7.8f));
System.out.println(listaAlunos.toString2());
System.out.println("Que em ordem reversa será: \n"
                   + listaAlunos.toStringReverse2());
listaAlunos.clear();
System.out.println(listaAlunos.toString2());
```

Exercício final

- Crie um projeto com os dados dos alunos de duas turmas, guardados em duas listas duplamente ligadas ordenadas.
- Crie vários objetos da classe Aluno e adicione os mesmos na lista de cada turma, para facilitar os testes.
- Crie uma terceira lista duplamente ligada ordenada, com os dados dos alunos das turmas, mas considerando apenas os alunos reprovados.
- Mostre o conteúdo da terceira lista em ordem alfabética.
- Mostre o conteúdo da terceira lista em ordem alfabética reversa.

Bibliografia

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)