Aula 07

Estruturas de dados recursivas

Listas ligadas

Programação II, 2015-2016

v0.5, 30-03-2016

DETI, Universidade de Aveiro

07.1

Objectivos:
- Estrutura de dados recursivas: lista ligadas;
- Funções recursivas (cont.)

Conteúdo

07.2

1 Lista Ligada

Como guardar colecções de dados?

- Temos utilizado vectores (ou *arrays*)
- São muito úteis para guardar coisas numa determinada ordem
- Permitem acesso directo a cada elemento
- No entanto, **os vectores têm limitações**:
  - A sua capacidade tem de ser definida/fixada quando são criados
  - Isto obriga a sobre-dimensionar um vector quanto o número de elementos não é conhecido à partida
  - Ou então, re-dimensionar o vector à chegada de novos elementos, com custos em tempo de processamento
  - Inserções (`insert`) e remoções (`delete`) numa posição intermédia podem demorar bastante tempo se for necessário deslocar muitos elementos

07.3

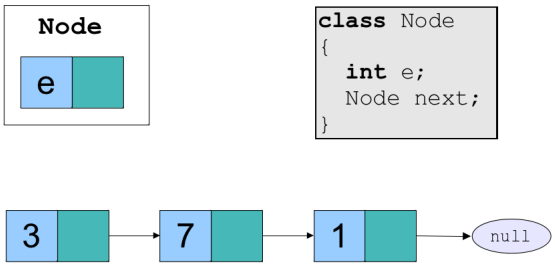
Lista Ligada

- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento.
  - Essa referência terá o valor `null` caso esse elemento não exista.

- É uma estrutura de dados **recursiva** (dado que contém uma referência para si própria).
- Ao contrário do vector, é **completamente dinâmica**.
  - No entanto, obriga a um acesso sequencial.
- Requer a criação de uma estrutura (um *nó*) para armazenar cada elemento.

07.4

Lista ligada simples: exemplo



07.5

Nós para uma lista de inteiros

```
class NodeInt {

   final int elem;
   NodeInt next;

   NodeInt(int e, NodeInt n) {
      elem = e;
      next = n;
   }

   NodeInt(int e) {
      elem = e;
      next = null;
   }
}
```

07.6

Lista ligada: tipo de dados abstracto

- Nome do módulo:
  - `LinkedList`
- Serviços:
  - `addFirst`: insere um elemento no início da lista
  - `addLast`: insere um elemento no fim da lista
  - `first`: retorna o elemento no início da lista
  - `removeFirst`: retira o elemento no início da lista
  - `isEmpty`: verifica se a lista está vazia
  - `size`: retorna a dimensão actual da lista
  - `clear`: limpa a lista (remove todos os elementos)

07.7

Lista ligada: semântica

- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty()`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

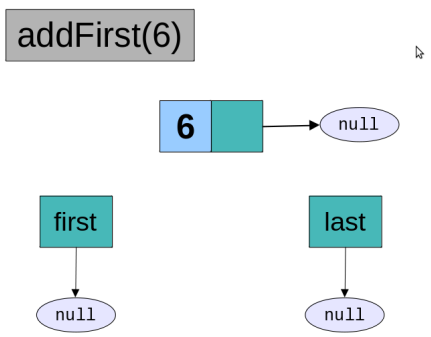
07.8

Lista de inteiros: esqueleto da implementação

```
public class LinkedListInt {
   public LinkedListInt() { }
   public void addFirst(int e) {
      ...
      assert !isEmpty();
   }
   public void addLast(int e) {
      ...
      assert !isEmpty();
   }
   public int first() {
      assert !isEmpty();
      ...
   }
   public void removeFirst() {
      assert !isEmpty();
      ...
   }
   public boolean isEmpty() { ... }
   public int size() { ... }
   public void clear() {
      ...
      assert isEmpty();
   }
   private NodeInt first=null;
   private NodeInt last=null;
   private int size;
}
```

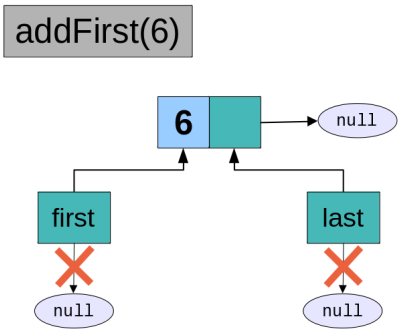
07.9

- `addFirst` - inserção do primeiro elemento



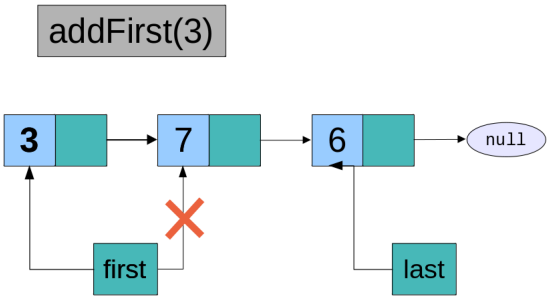
07.10

- `addFirst` - inserção do primeiro elemento



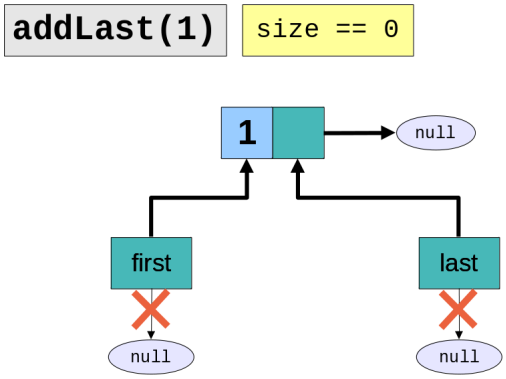
07.11

- `addFirst` - inserção de elementos adicionais no início



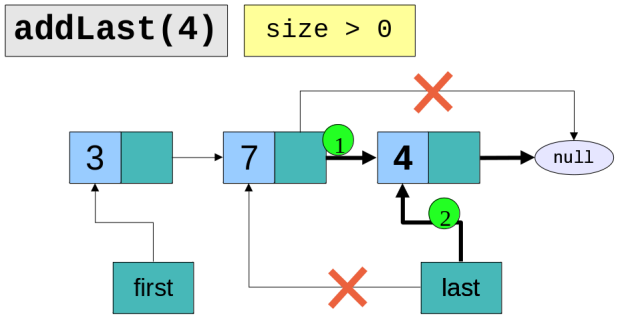
07.12

- Novo elemento no fim: `addLast`
- Caso de lista vazia: similar a `addFirst`



07.13

- Novo elemento no fim: `addLast`



07.14

```
public class LinkedListInt {

   public void addFirst(int e) {
      first = new NodeInt(e, first);
      if (isEmpty())
         last = first;
      size++;

      assert !isEmpty();
   }

   public void addLast(int e) {
      NodeInt n = new NodeInt(e);
      if (first == null)
         first = n;
      else
         last.next = n;
      last = n;
      size++;

      assert !isEmpty();
   }

   public int first() {
      assert !isEmpty();

      return first.elem;
   }
```

```
   public void removeFirst() {
      assert !isEmpty();

      size--;
      first = first.next;
      if (first == null)
         last = null;
   }

   public int size() {
      return size;
   }

   public boolean isEmpty() {
      return size() == 0;
   }

   public void clear() {
      first = last = null;
      size = 0;
   }

   private NodeInt first = null;
   private NodeInt last = null;
   private int size = 0;
}
```

07.15

2 Polimorfismo Paramétrico

Polimorfisto paramétrico

- Problema da `LinkedListInt`:
  - Desenvolvida específicamente para elementos inteiros
  - Se quisermos ter listas de elementos de outros tipos, podemos duplicar o código e fazer pequenas alterações para adaptar ao tipo pretendido
  - O código assim obtido é praticamente igual, mas *não é pratico* fazer esta "clonagem"de código para cada nova necessidade
- **Solução**: Construir módulos aplicáveis a quaisquer tipos
  - Diz-se que são parametrizados por tipo, ou seja, o tipo é também um parâmetro
  - Em Java, estes tipos são chamados **genéricos**
  - As estruturas e funções passam a ser polimórficas
  - Este mecanismo é conhecido como **polimorfismo paramétrico**

07.16

Tipos genéricos em Java

- Java permite a implementação e utilização de classes (módulos) com parâmetros que são tipos genéricos
- Os tipos genéricos de uma classe são indicados entre < ... > a seguir ao nome da classe na definição desta

```
public class LinkedList<E> {
   ...
   public void addFirst(E e) {
      ...
   }
   ...
}
...
public static void main(String args[]) {
   ...
   LinkedList<Double>  p1 = new LinkedList<Double>();
   LinkedList<Integer> p2 = new LinkedList<Integer>();
   ...
}
```

07.17

Convenção sobre nomes de tipos genéricos

- Por convenção, os nomes dos tipos genéricos são letras maiúsculas
  - `E` - *element*
  - `K` - *key*
  - `N` - *number*
  - `T` - *type*
  - `V` - *value*
- Assim, mais facilmente se distingue uma variável que representa um tipo genérico de uma variável normal, que começa (também por convenção) com letra minúscula (exemplo: `numberOfElements`)

07.18

Tipos genéricos em Java: limitações

- *Problema*: Não é possível aplicar módulos genéricos directamente a tipos primitivos! (`int`, `short`, `long`, `byte`, `boolean`, `char`, `float`, `double`);
- *Solução*:
  - utilizar os tipos referência correspondentes (`Integer`, `Double`, etc.);
  - A linguagem faz a conversão automática entre os tipos primitivos e os tipos referência respectivos (*boxing* e *unboxing*).
- *Problema*: Não é possível instanciar arrays de genéricos!
- *Solução*:
  - criar arrays de elementos do tipo Object e fazer a coerção de tipo para o *array* de genéricos:

```
T[] a = (T[]) new Object[maxSize];
```

- Para evitar o aviso gerado pelo compilador como resultado desta coerção pode-se associar ao método onde a coerção é feita a seguinte anotação:

```
@SuppressWarnings(value = "unchecked")
```

07.19

```
public class LinkedList<E> {

   public void addFirst(E e) {
      first = new Node<>(e, first);
      if (isEmpty())
         last = first;
      size++;

      assert !isEmpty();
   }

   public void addLast(E e) {
      Node<E> n = new Node<>(e);
      if (first == null)
         first = n;
      else
         last.next = n;
      last = n;
      size++;

      assert !isEmpty();
   }

   public E first() {
      assert !isEmpty();

      return first.elem;
   }
```

```
   public void removeFirst() {
      assert !isEmpty();

      size--;
      first = first.next;
      if (first == null)
         last = null;
   }

   public int size() {
      return size;
   }

   public boolean isEmpty() {
      return size() == 0;
   }

   public void clear() {
      first = last = null;
      size = 0;
   }

   private Node<E> first = null;
   private Node<E> last = null;
   private int size = 0;
}
```

07.20

3 Processamento recursivo de listas

Processamento recursivo de listas

- Quando a acção a realizar implica aceder ao meio da lista, é preciso percorrer a lista até ao nó que vai ter alteração
- Sendo uma estrutura recursiva, as listas prestam-se naturalmente à utilização de algoritmos recursivos
- **Exemplo**: saber se um elemento `e` existe na lista
  - Condições de terminação da recursividade:
    - Chegou ao fim da lista (retorna `false`), ou
    - Encontrou o elemento `e` (retorna `true`)
  - Variabilidade: passar do nó actual (`n`) ao seguinte (`n.next`)
  - Convergência: está garantida!

07.21

Exemplo: lista contém elemento

```
public boolean contains(E e) {
   return contains(first,e);
}
private boolean contains(Node<E> n, E e) {
   if (n == null) return false; // condicao de terminacao
   if (n.elem.equals(e)) return true; // condicao de terminacao
   return contains(n.next,e); // chamada recursiva (continuacao)
}
```

07.22