Aula 07

Estruturas de dados recursivas

Listas ligadas

Programação II, 2016-2017

v1.1, 26-03-2017

DETI, Universidade de Aveiro

07.1

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

Conteúdo

1 Lista Ligada **1**

2 Polimorfismo Paramétrico **6**

3 Processamento recursivo de listas **8**

07.2

As estruturas de dados servem não só para registar e aceder a informação, como também para disciplinar (estruturar) essas utilizações. Em linguagens de programação com um sistema de tipos estático, como é o caso da linguagem Java, a correcção formal nessas utilizações é garantida em tempo de compilação, evitando as dificuldades envolvidas na depuração do programa em tempo de execução.

O sistema de tipos dá, grosso modo, duas garantias a um programa:

1. compatibilidade de tipos na atribuição de valores;
2. correcção na utilização (formal) de um membro da classe.

A primeira aplica-se tanto à instrução de atribuição propriamente dita, como também à passagem de argumentos a uma função, que pode ser vista como a atribuição de valores aos parâmetros formais correspondentes. A segunda garante que quando se utiliza um membro de uma classe (método ou campo), ele tem de existir e ser compatível no número e tipos dos eventuais argumentos (no caso de métodos).

Vamos seguir uma abordagem modular na apresentação e implementação de algumas estruturas de dados de propósito geral. Assim, começaremos por definir o seu tipo de dados abstracto (a sua interface e os respectivos contratos), partindo depois para algumas possíveis concretizações.

Nesta aula, apresentamos uma dessas estrutura de dados de propósito geral, a *lista ligada*. Em aulas seguintes, veremos as pilhas e filas, bem como diferentes tipos de dicionários.

1 Lista Ligada

Como guardar colecções de dados?

- Temos utilizado vectores (ou *arrays*).
- São muito úteis para guardar elementos 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 sobredimensionar um vector quando o número de elementos não é conhecido à partida.
  - Ou então, redimensionar o vector quando chegam novos elementos, com custos em tempo de processamento.
  - Inserir (`insert`) ou remover (`delete`) elementos numa posição intermédia podem demorar bastante tempo se for necessário deslocar muitos elementos.

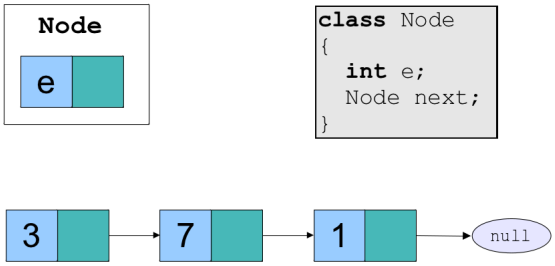
07.3

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 a sua definição 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

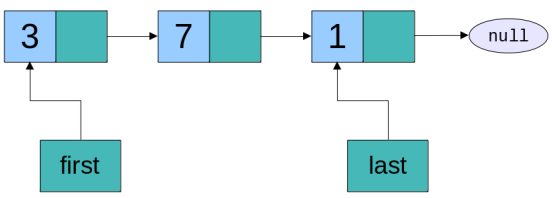
Lista ligada simples: exemplo



07.5

Lista ligada com dupla entrada

- A lista possui acesso directo ao primeiro e último elementos.
- É possível acrescentar elementos no início e no fim da lista.
- É possível remover elementos do início da lista.
- Exemplo - lista com os elementos 3, 7 e 1:



07.6

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;
   }
}
```


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`: devolve o primeiro elemento da lista.
  - `last`: devolve o último elemento da lista.
  - `removeFirst`: retira o elemento no início da lista.
  - `size`: devolve a dimensão actual da lista.
  - `isEmpty`: verifica se a lista está vazia.
  - `clear`: limpa a lista (remove todos os elementos).

Lista ligada: semântica

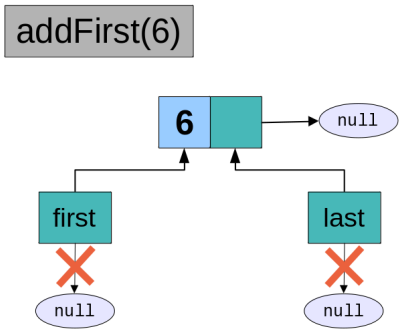
- **addFirst(v)**
  - Pós-condição: `!isEmpty() && (first() == v)`
- **addLast(v)**
  - Pós-condição: `!isEmpty() && (last() == v)`
- **removeFirst()**
  - Pré-condição: `!isEmpty()`
- **first()**
  - Pré-condição: `!isEmpty()`

Lista de inteiros: esqueleto da implementação

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

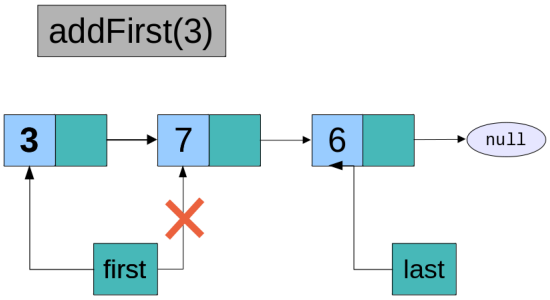
07.10

- `addFirst` - inserção do primeiro elemento.



07.11

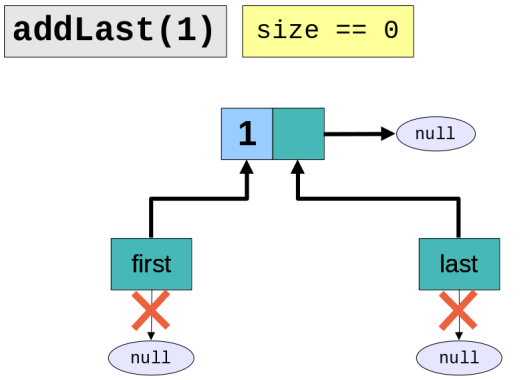
- `addFirst` - inserção de elementos adicionais no início.



07.12

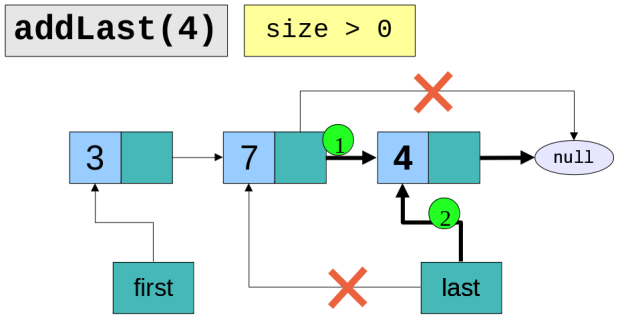
- Novo elemento no fim: `addLast`.

- Caso de lista vazia: similar a `addFirst`.



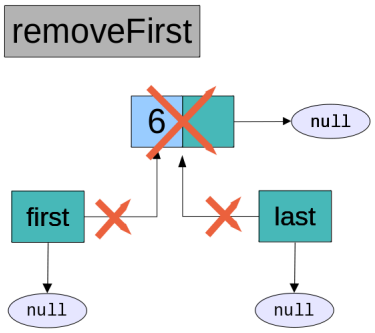
07.13

- Novo elemento no fim: `addLast`.



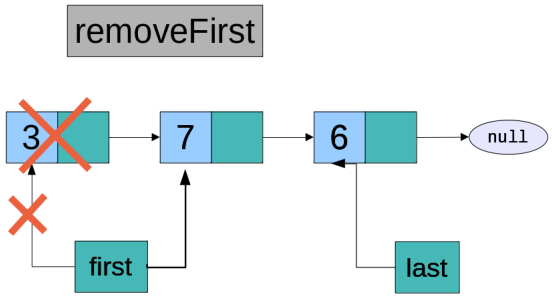
07.14

- Remoção do primeiro elemento: `removeFirst`.
- `size==1`



07.15

- Remoção do primeiro elemento: `removeFirst`.
- `size>1`



07.16

```
public class LinkedListInt {

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

      assert !isEmpty() && first()==e;
   }

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

      assert !isEmpty() && last()==e;
   }

   public int size() {
      return size;
   }

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

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

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

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

      return first.elem;
   }

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

      return last.elem;
   }

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

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

07.17

2 Polimorfismo Paramétrico

Polimorfismo paramétrico

- **Problema**: A classe `LinkedListInt`:
  - Foi desenvolvida especificamente 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 prático* 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.
  - As estruturas e funções passam a ser polimórficas.
  - Este mecanismo é conhecido como **polimorfismo paramétrico**.

07.18

Tipos genéricos em Java

- Em Java, as classes e funções que têm parâmetros que representam tipos são chamadas **genéricas**.
- Os parâmetros de tipo 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>();
   ...
}
```

07.19

Convenção sobre nomes de variáveis de tipo

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

Tipos genéricos em Java: limitações

- *Problema*: Não é possível invocar módulos genéricos com argumentos de 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")
```



```
public class LinkedList<E> {

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

      assert !isEmpty() && first().equals(e);
   }

   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() && last().equals(e);
   }

   public int size() {
      return size;
   }

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

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

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

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

      return first.elem;
   }

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

      return last.elem;
   }

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

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

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 ser alterado.
- 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 (devolve `false`), ou
    - Encontrou o elemento `e` (devolve `true`).
  - Variabilidade: passar do nó actual (`n`) ao seguinte (`n.next`).
  - Convergência: está garantida!

07.23

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)
}
```

07.24

Um padrão que se repete ...

- Muitas funções sobre listas fazem um percurso da lista
- Esse percurso segue um padrão que convém desde já assimilar

Implementação Iterativa

```
public class LinkedList<E> {
  ....
  public ... xpto(...) {
     Node<E> n = first;
     ...
     while (n!=null && ...) {
        ...
        n = n.next;
     }
     return ...;
  }
  ....
}
```

Implementação Recursiva

```
public class LinkedList<E> {
  ....
  public ... xpto(...) {
     return xpto(first,e);
  }
  private ... xpto(Node<E> n, ...) {
     if (n == null) return ...;
     ...
     ... xpto(n.next,...);
     return ...
  }
  ....
}
```

07.25