IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Programação Avançada

2a

Árvores | Estruturas de dados ~ Conceitos

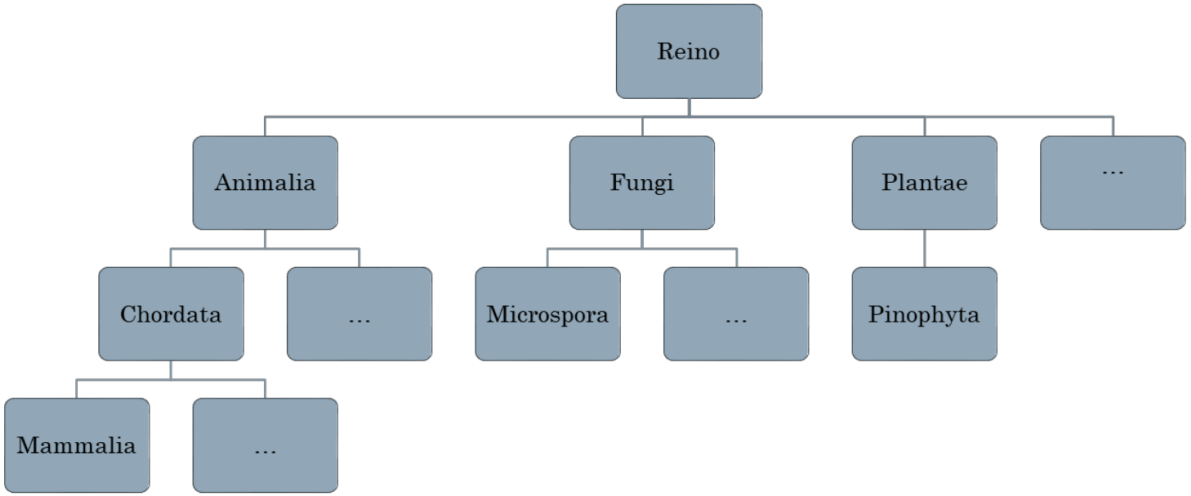
Bruno Silva, Patrícia Macedo

Sumário 📝

- Árvores como estruturas de dados hierárquicas
- Conceitos
  - Grau, Ordem, Níveis, Altura e Sub-árvores
  - Travessia (inglês: *traversal*) de árvores
    - *breadth-first* e *depth-first*
  - Árvores binárias
- Exercícios

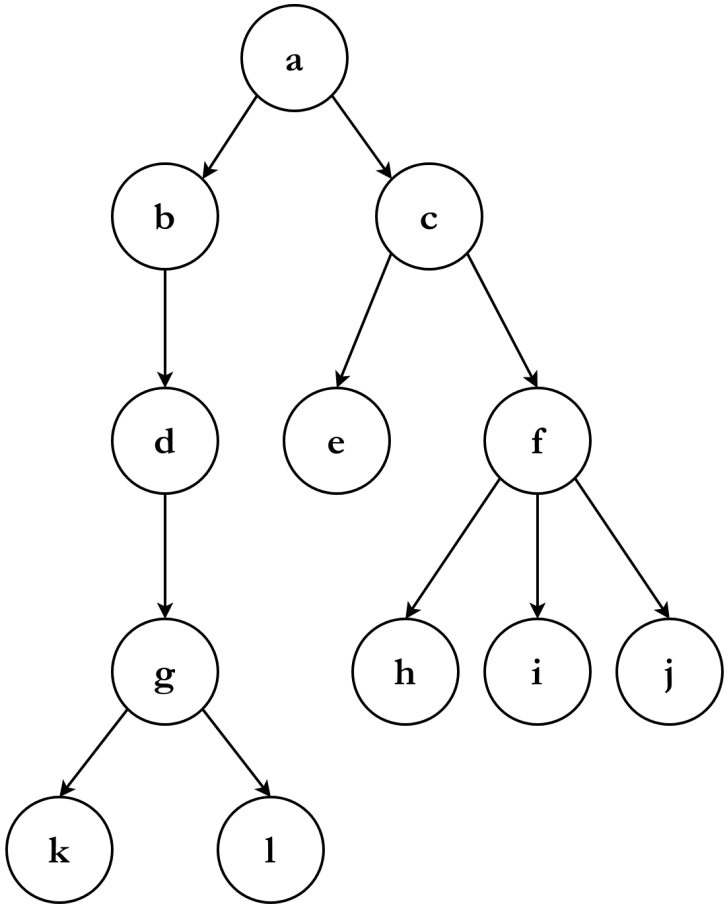
Árvores

- As **árvores**, no contexto das ciências da computação, são estruturas de dados **não-lineares** e **hierárquicas**.
- Permitem representar (informação de) elementos com relações hierárquicas, i.e., relações de *pai*, *filho*, *ascendente* e *descendente*. Exemplo:



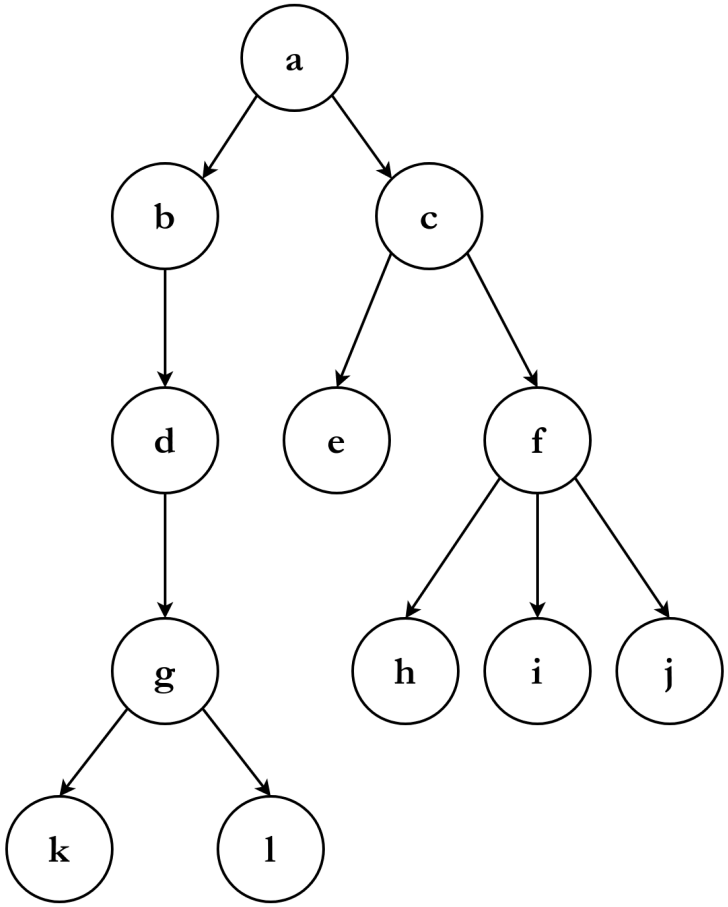
Árvores | Conceitos

- Uma árvore é composta por **nós**;
- No topo da árvore existe um nó especial, a **raiz**; não possui *ascendentes* - nó **a**.
- Dos restantes, um nó pode ter vários *filhos* (descendentes diretos), mas apenas um *pai* (ascendente direto).
    - Em relação ao nó **c**:
        - filhos: {**e**, **f**} - e *irmãos* entre si, e;
        - descendentes: {**e**, **f**, **h**, **i**, **j**}.



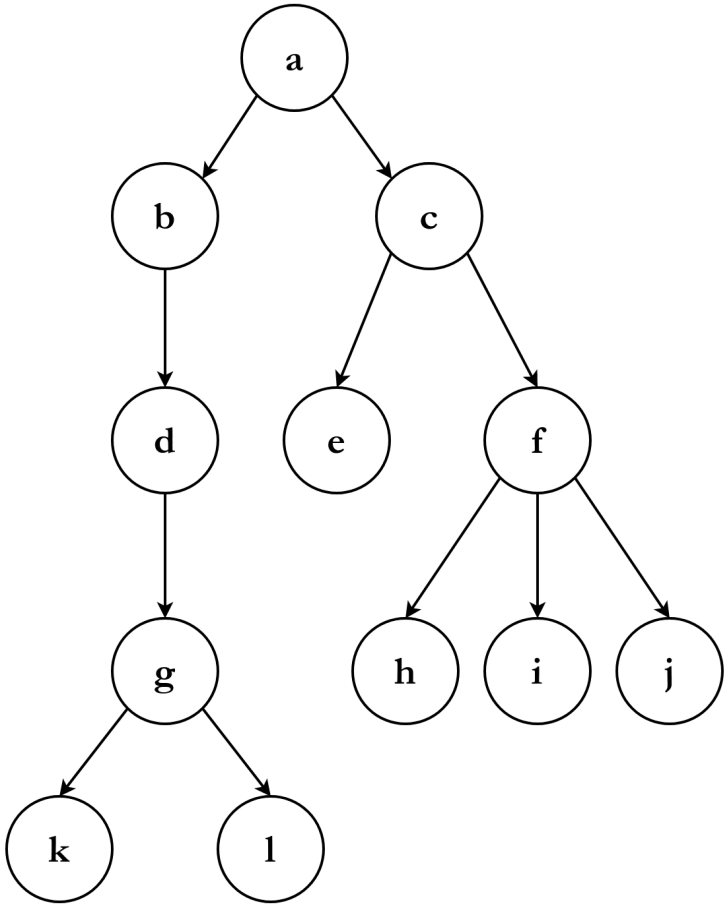
Árvores | Conceitos

- Nós que não têm descendentes são chamados de **nós externos** ou **folhas**.
  - {e, k, l, h, i, j}
- Nós que **não são** a raiz e **não são** folhas são chamados de **nós internos**.
  - {b, c, d, f, g}



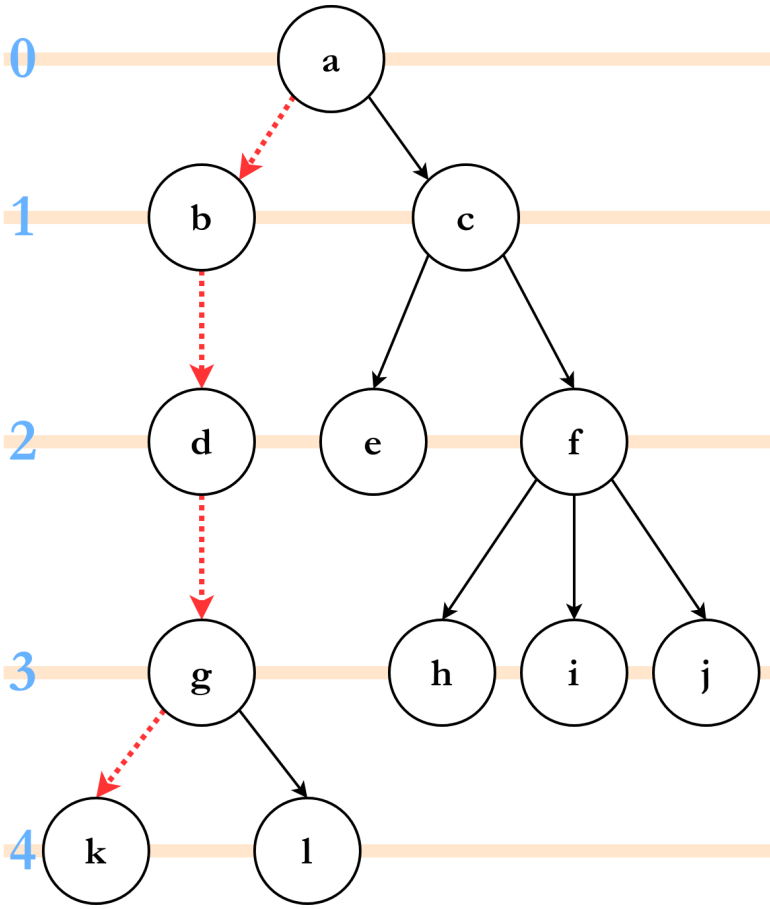
Árvores | Grau e Ordem

- O **grau de um nó** corresponde ao seu número de filhos
  - e.g., a : 2, b : 1, f : 3
- A **ordem de uma árvore** consiste no grau máximo permitido para os seus nós.
  - e.g., **árvores binárias** são de ordem 2.



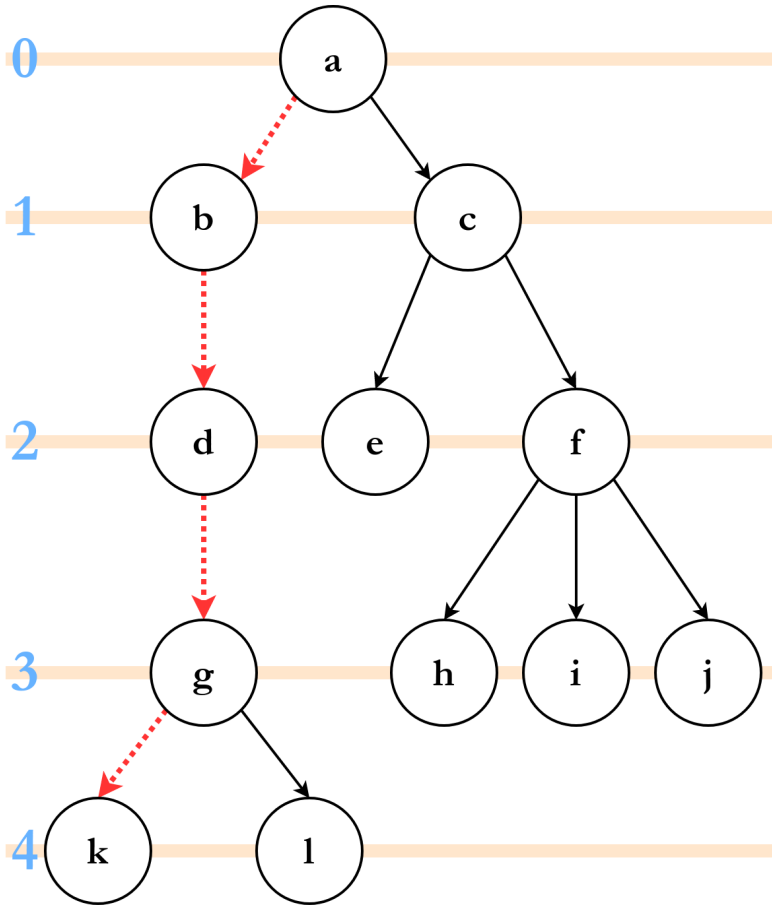
Árvores | Níveis e Altura

- As árvores podem ser organizadas em **níveis**
  - Raiz está no nível 0;
  - O nível 2 contém todos os filhos do nível 1, etc.
- A **altura da árvore** corresponde ao maior nível da árvore
  - (ou, equivalente) ao maior caminho presente na árvore;
  - Na figura ➡️, a altura é 4;



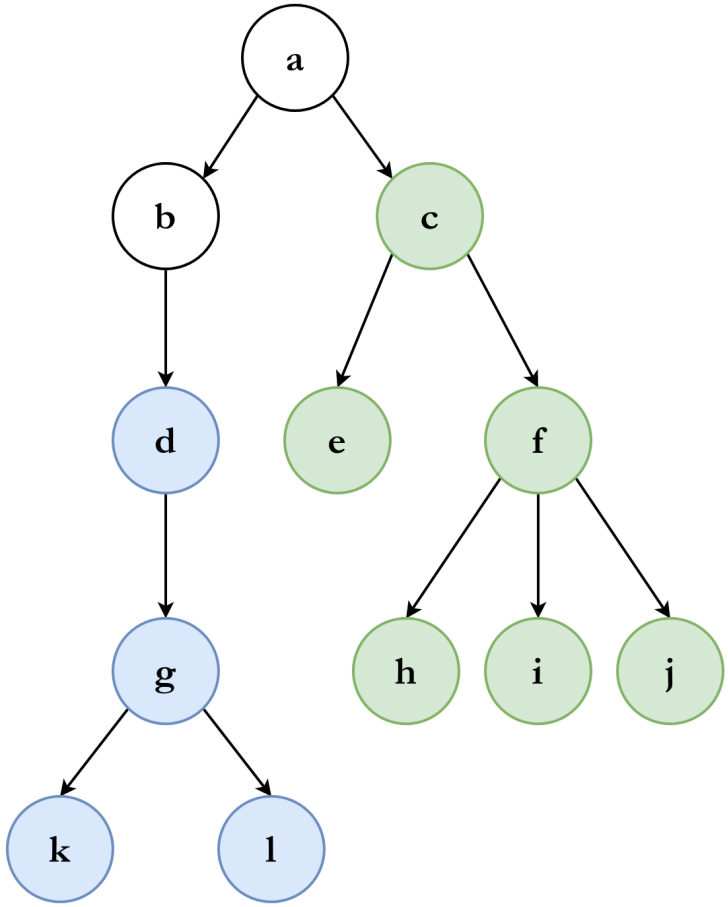
Árvores | Níveis e Altura

- A **altura da árvore** corresponde ao maior nível da árvore
  - (...)
  - Uma árvore "vazia" tem altura -1;
  - Uma árvore contendo apenas a raiz tem altura 0 (zero).



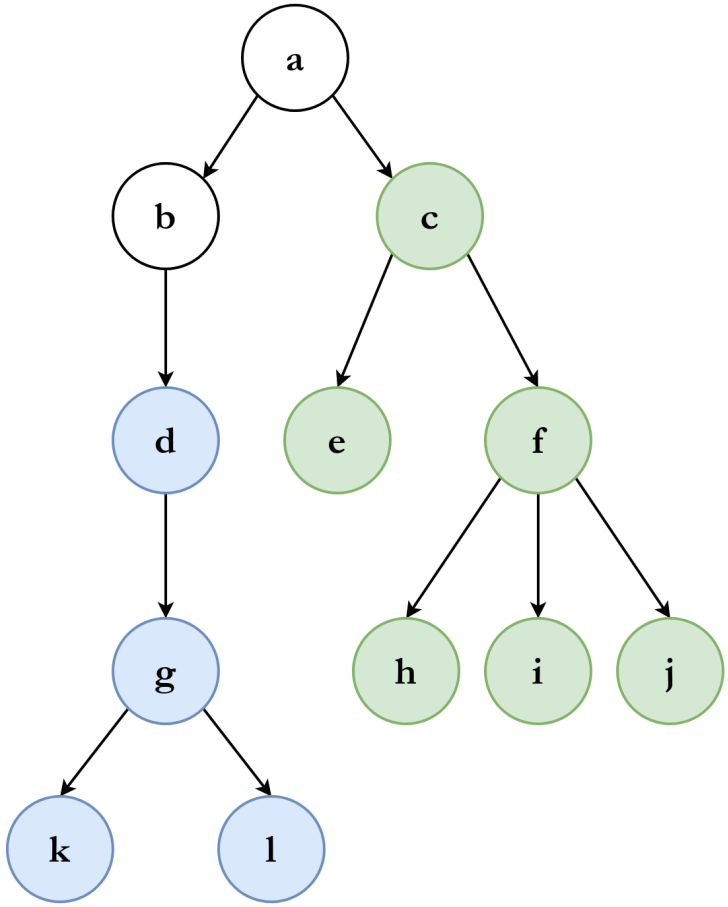
Árvores | Sub-árvores

- Dado uma àrvore e um nó n, o conjunto de todos os nós que possuem n como ascendente é chamada a **sub-árvore com raiz em n.**
- Exemplos:
  - sub-árvore com raiz em d
  - sub-árvore com raiz em c



Árvores | Sub-árvores

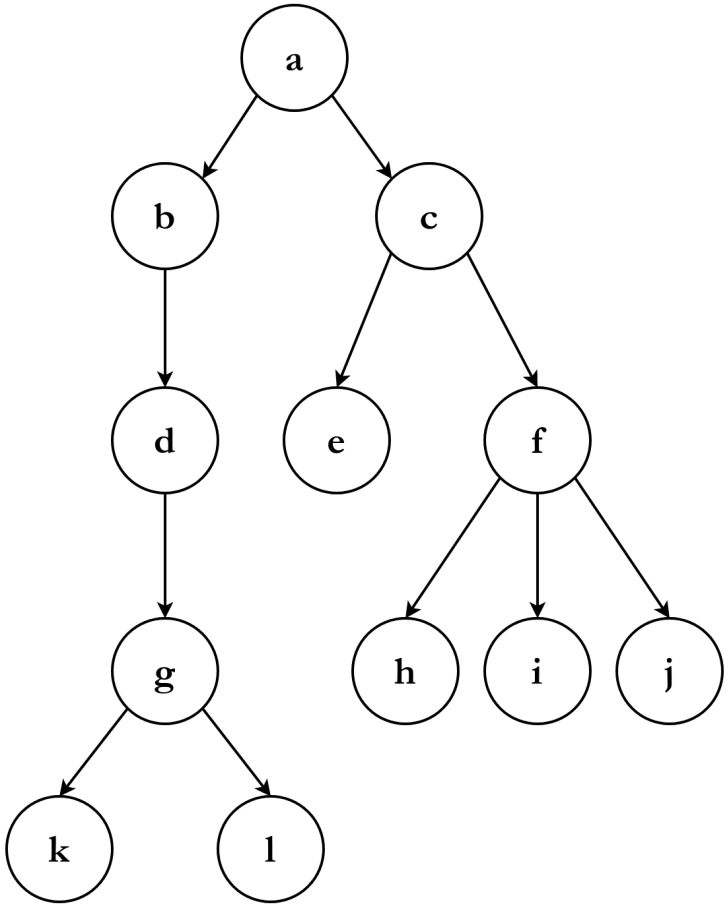
- Isto permite a abstração **recursiva** de uma árvore:
  - Uma árvore é composta por um nó (raiz) que possui um determinado número de filhos, que por sua vez representam árvores menores.



Árvores | Travessia

Em **largura** (*breadth-first traversal*):

- [esq/dir] **a b c d e f g h i j k l**
- [dir/esq] **a c b f e d j i h g l k**

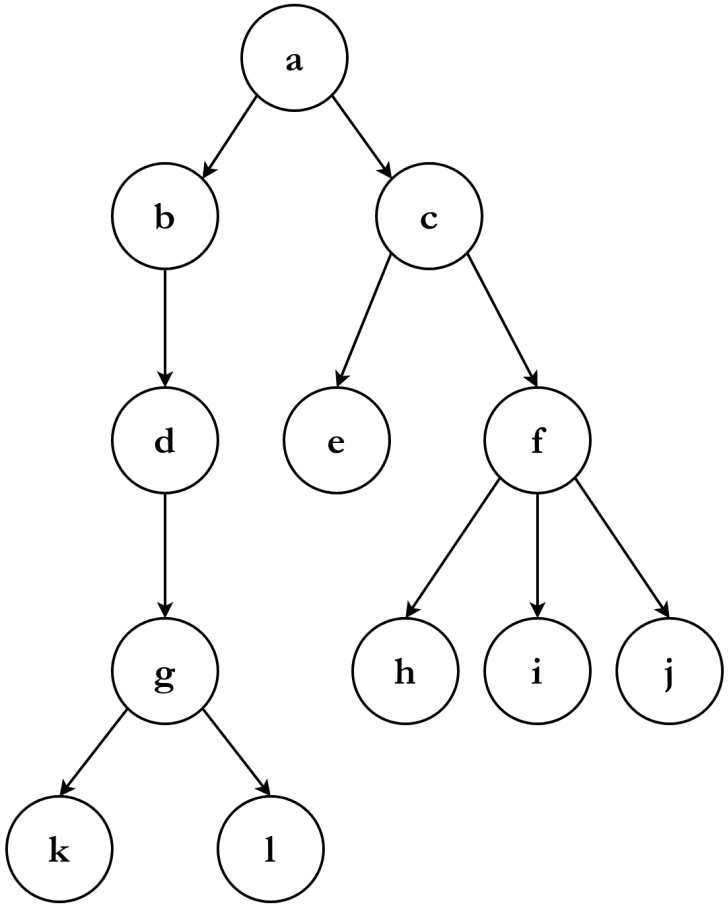


Árvores | Travessia

Em **profundidade** (*depth-first traversal*):

- [pré-ordem] **a b d g k l c e f h i j**
  - Os nós são visitados antes dos seus descendentes.
- [pós-ordem] **k l g d b e h i j f c a**
  - Os nós são visitados depois dos seus descendentes.

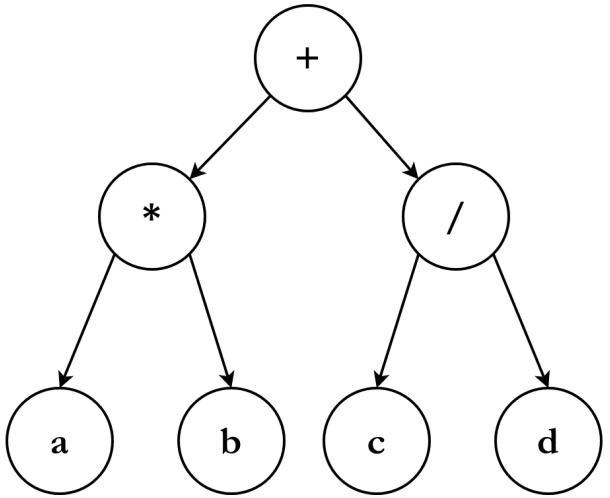
Nota: Nos exemplos os descendentes são visitados da esquerda para a direita.



Árvores | Árvores Binárias

Consistem em árvores de **ordem 2**, i.e., cada nó pode ser no máximo de *grau 2* (máx. dois filhos).

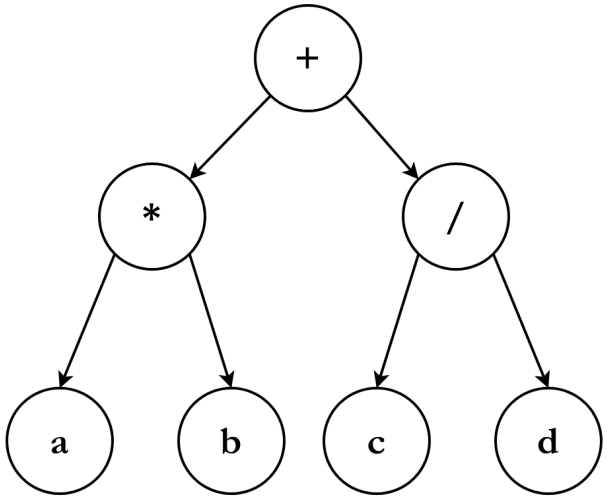
Na figura ➡️ um exemplo de uma árvore binária para representar uma expressão matemática.



Árvores | Árvores Binárias

A **travessia** de árvores binárias contempla um modo adicional **em-ordem** (antes de um nó ser visitado é visitado o seu filho esquerdo; e no final o filho direito) :

- [em-ordem]: `a * b + c / d`
  - *Notação Convencional*
- [pré-ordem]: `+ * a b / c d`
  - *Notação Polaca*
- [pós-ordem]: `a b * c d / +`
  - *Notação Polaca Invertida*



Exercícios ✏️

Elabore a ficha de atividades disponível no Moodle:

`2a_FichaAtividades.pdf`

IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Programação Avançada

2b

Árvores Binárias de Pesquisa | Algoritmos

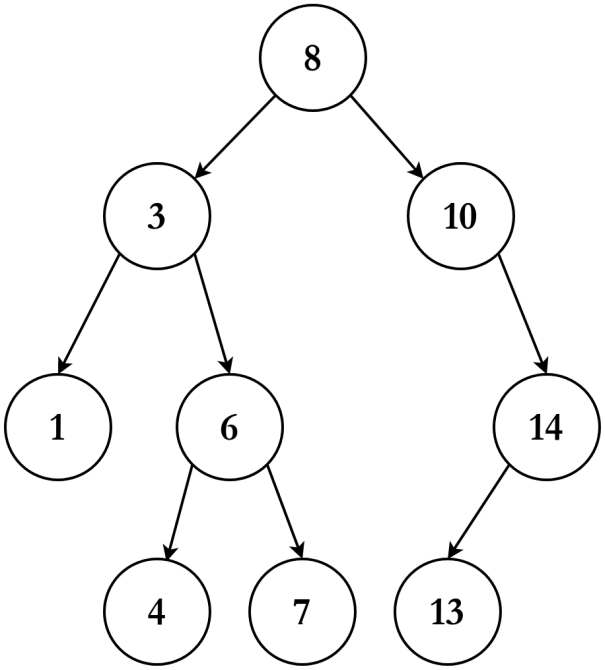
Bruno Silva, Patrícia Macedo

Sumário 📝

- Caracterização
- Motivação de uso
- Algoritmos
  - Pesquisa
  - Inserção
  - Remoção
- Exercícios

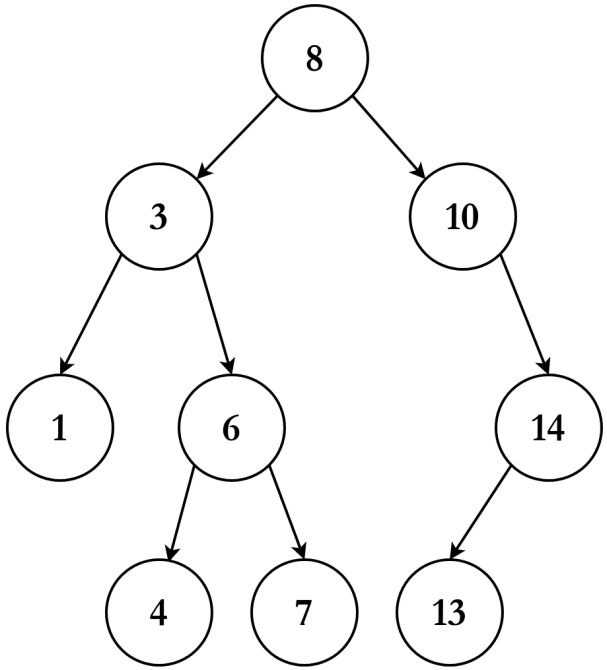
Características

- Uma árvore binária de pesquisa (em inglês, *binary search tree*) é uma estrutura de dados hierárquica especializada;
  - Consiste numa árvore binária "ordenada".
- É **caracterizada** pelo seguinte:
  - um **nó** contém um elemento/chave maior que os da **sub-árvore esquerda**;
  - um **nó** contém um elemento/chave menor que os da **sub-árvore direita**;



Características

- Deverá ser óbvio que é necessário um **critério de comparação** para os elementos/chaves numa árvore binária de pesquisa;
- **Não são permitidos** elementos/chaves **repetidos**;
- Vamos ilustrar com números (comparação natural), mas são válidas para quaisquer outros dados "comparáveis".



Motivação

Como o nome sugere, permitem acelerar a pesquisa de elementos.

- Pesquisa sequencial num array - complexidade $O(n)$
  - 10.000 elementos ➡️ 10.000 comparações no pior caso 😒
- Pesquisa numa árvore binária de pesquisa - complexidade $O(log\ n)$
  - 10.000 elementos ➡️ $\approx$ 13 comparações no pior caso 😄

Algoritmos

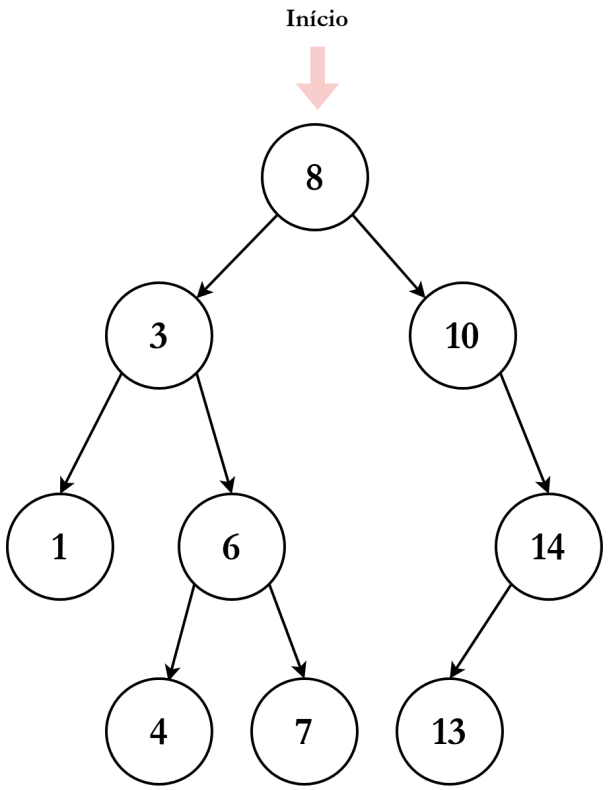
- Os principais algoritmos são:
  - 🔍 Pesquisa (elemento/chave, mínimo e máximo);
  - ➕ Inserção de um novo nó (⚠️)
  - ➖ Remoção de um nó (⚠️)
- Os algoritmos que alteram uma árvore binária de pesquisa (⚠️) têm de garantir/manter as suas características.
- Os algoritmos serão apresentados em *linguagem natural*.

🔍 Pesquisa

Para pesquisar um elemento/chave *target*:

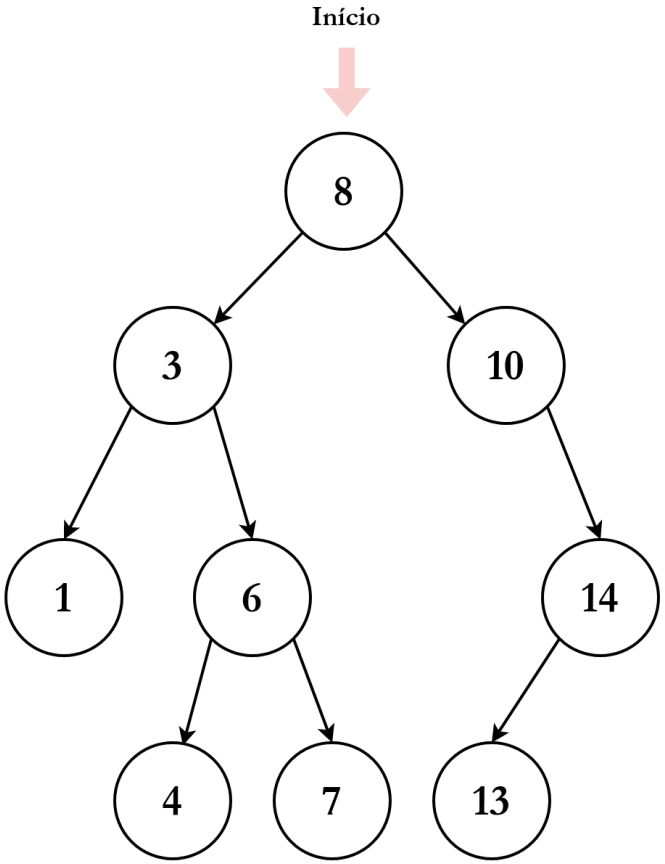
1. Se a árvore estiver vazia, não existe;
2. Se *target* é igual ao elemento na raiz da árvore, sucesso!;
3. Senão:

- Se target é menor que o elemento na raiz, retornar o resultado da pesquisa na sub-àrvore esquerda;
- Se target é maior que o elemento na raiz, retornar o resultado da pesquisa na subárvore direita;



🔍 Pesquisa

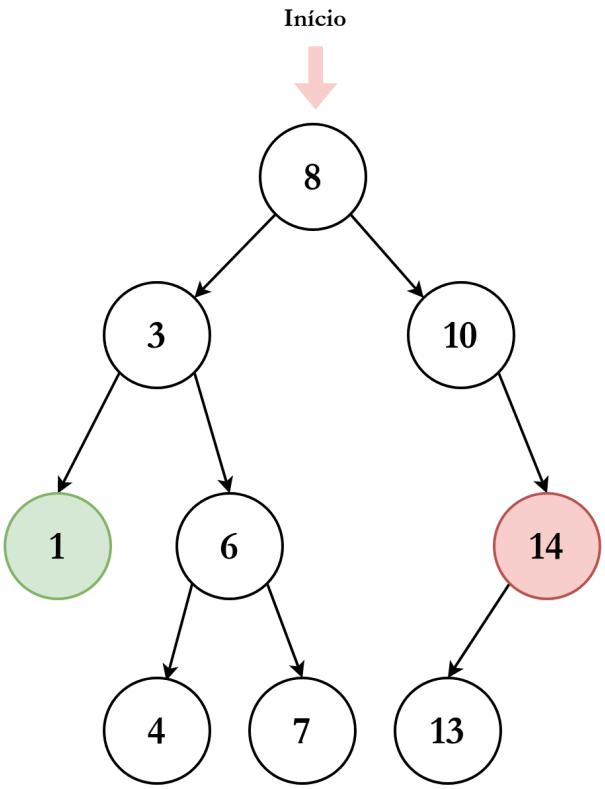
- Quais os passos para *target* =
  - 5?
  - 8?
  - 13?
  - 15?



🔍 Mínimo & Máximo

- Elemento/chave mínimo esta contido no nó mais à esquerda da árvore;
- Elemento/chave máximo esta contido no nó mais à direita da árvore;

❓ **Exemplifique com um algoritmo.**

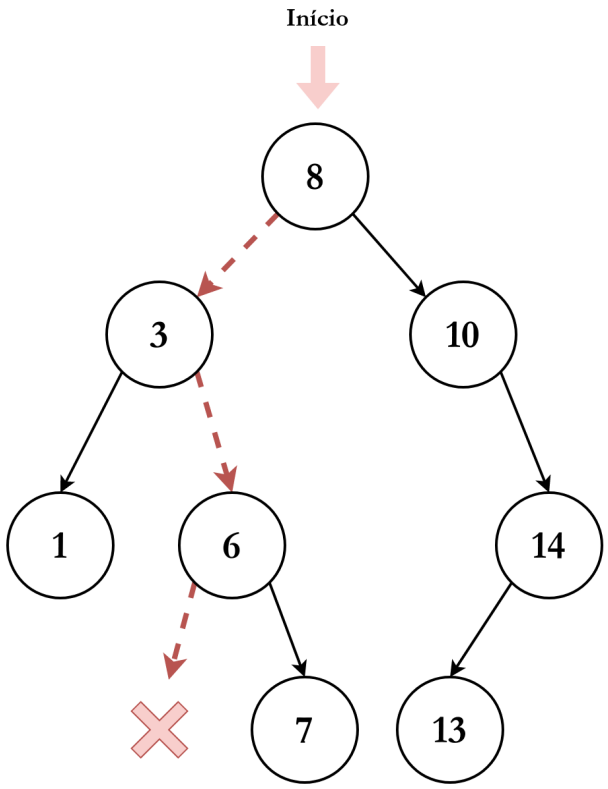


➕ Inserção (⚠️)

- A inserção de um elemento deverá garantir que são mantidas as características de uma árvore binária de pesquisa:
  - Sem repetição de elementos/chaves;
  - Sub-árvores esquerdas contêm elementos menores que a raiz;
  - Sub-árvores direitas contêm elementos maiores que a raiz;

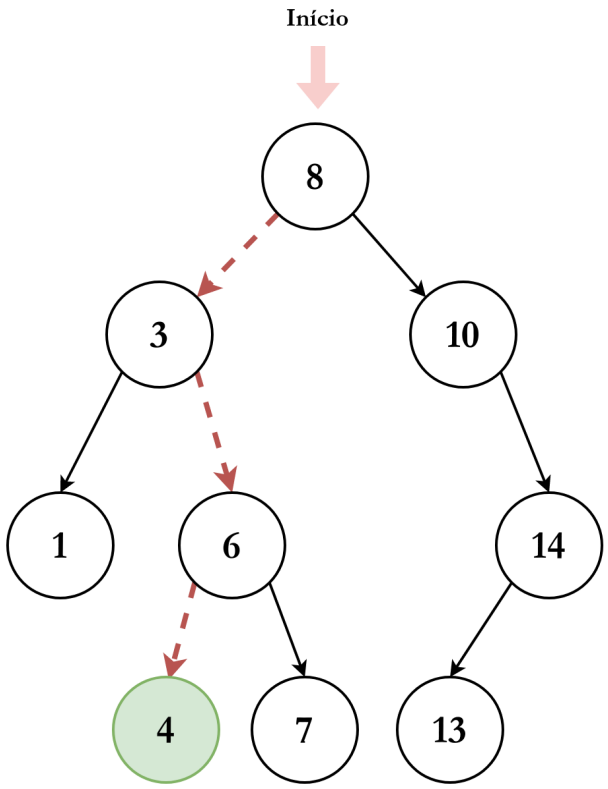
+ Inserção

- O algoritmo de inserção procede de forma análoga ao de pesquisa até que a sub-árvore que teria de conter o elemento esteja vazia.
- A figura ➡️ ilustra esta situação para o elemento 4.



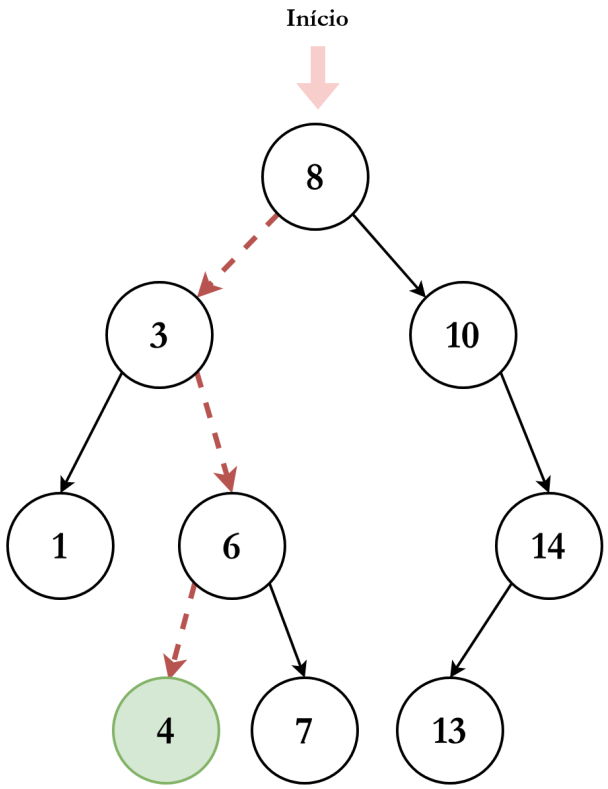
+ Inserção

- Nesta situação é adicionado um novo nó contendo o elemento (correspondendo a uma nova sub-árvore com o elemento como raiz).



+ Inserção

- Se o elemento a inserir (e.g., o elemento 4) já existir então o algoritmo nada faz.

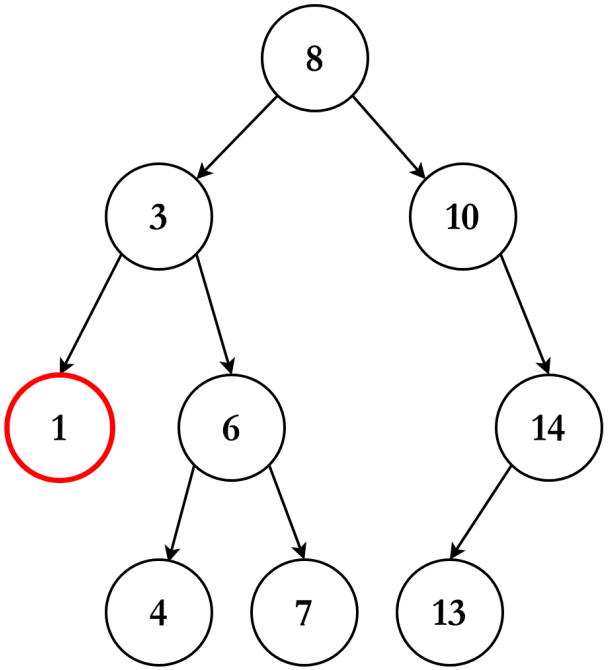


— Remoção (⚠️)

- Existem três situações possíveis a contemplar para o nó que contém o elemento a remover:
  - $[1]$ Não possui sub-árvores 😄
  - $[2]$ Apenas possui uma sub-árvore 😕
  - $[3]$ Possui duas sub-árvores 😖
- Os algoritmos de remoção dos elementos também têm de manter as características de uma àrvore binária de pesquisa.

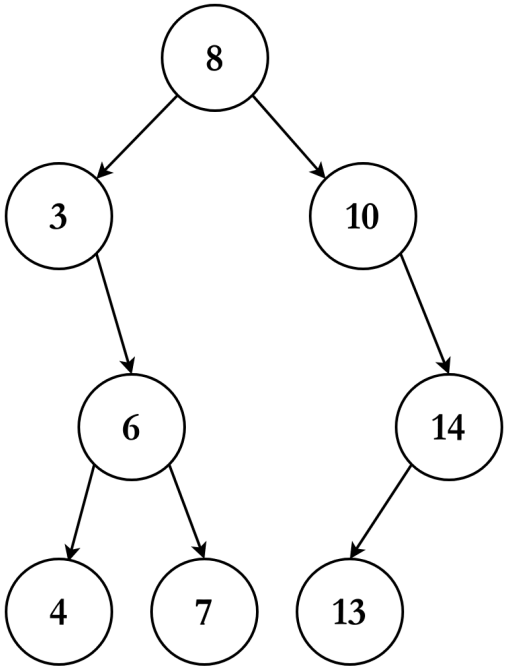
— Remoção

- [1] Não possui sub-árvores:
  - Figura ➡️ com exemplo para o elemento 1.



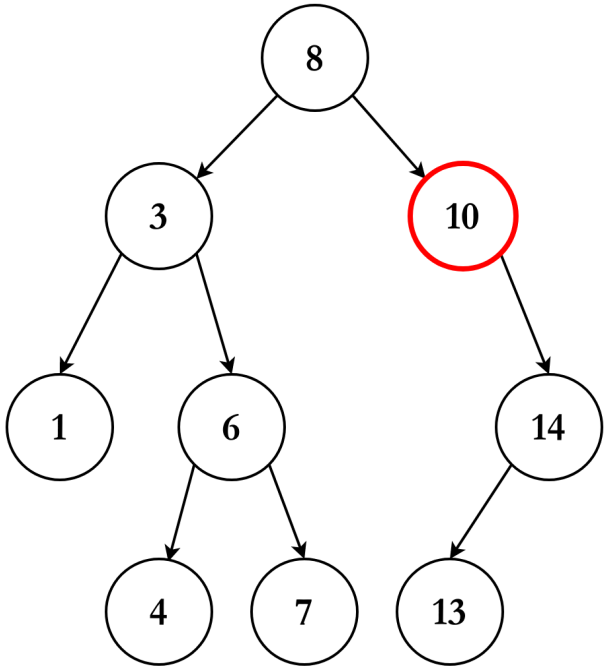
— Remoção

- [1] Não possui sub-árvores:
  - Figura ➡️ com exemplo para o elemento 1.
  - **Remove-se simplesmente o nó**.



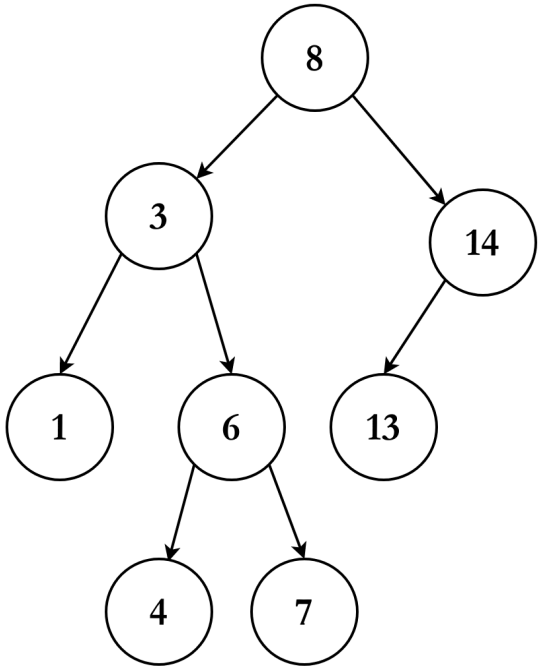
— Remoção

- [2] Apenas possui uma sub-árvore:
  - Figura ➡️ com exemplo para o elemento 10.



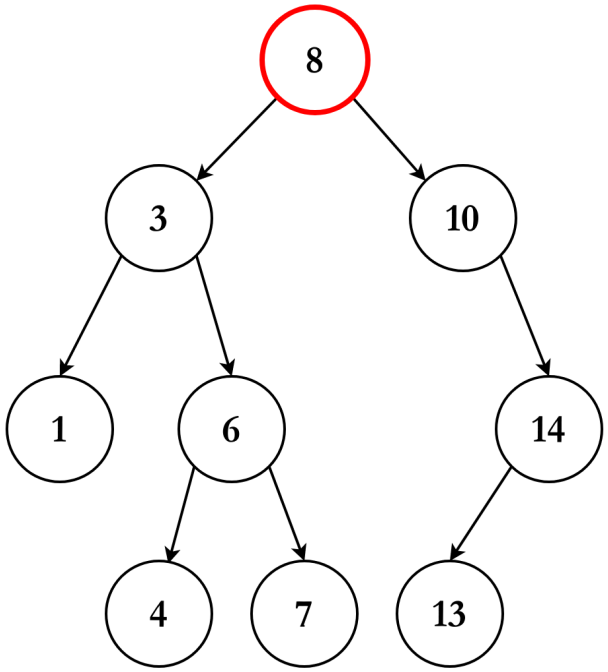
— Remoção

- [2] Apenas possui uma sub-árvore:
  - Figura ➡️ com exemplo para o elemento 10.
  - **Substitui-se o nó pela sua sub-árvore**.



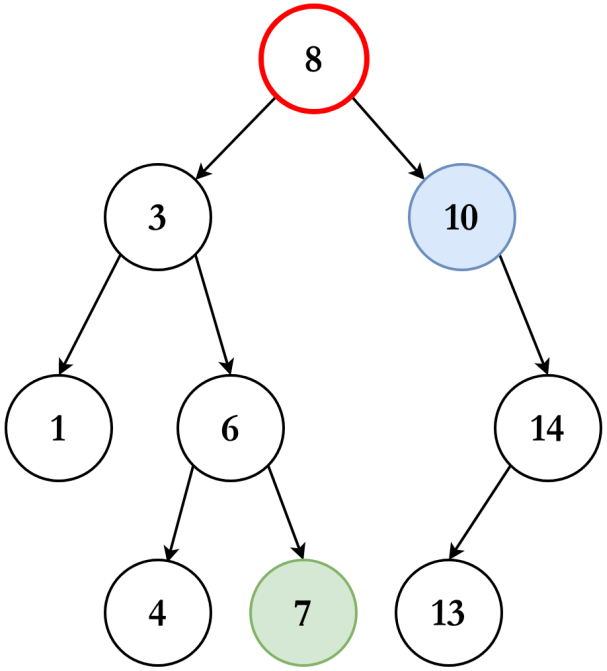
— Remoção

- [3] Possui duas sub-árvores:
  - Figura ➡️ com exemplo para o elemento 8.



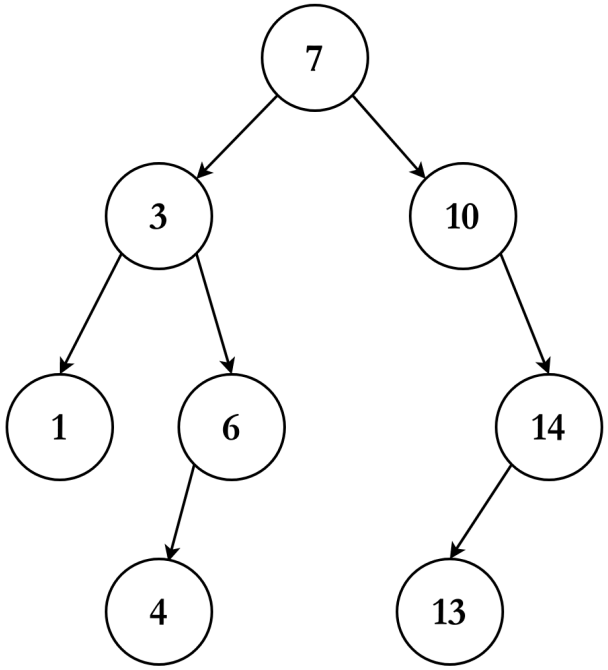
— Remoção

- [3] Possui duas sub-árvores:
  - Figura ➡️ com exemplo para o elemento 8.
  - Duas hipóteses (subsituição e algoritmo de remoção):
    - Elemento máximo da **sub-árvore esquerda**.
    - Elemento mínimo da **sub-árvore direita**.



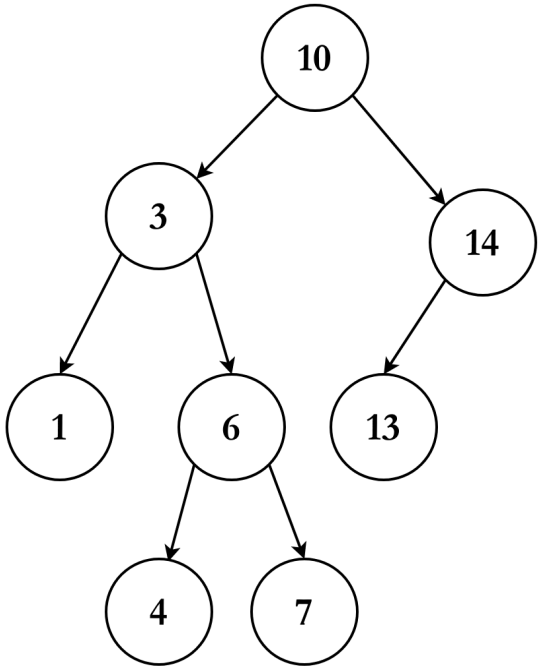
— Remoção

- [3] Possui duas sub-árvores:
  - Figura ➡️ com exemplo para o elemento 8.
  - 1ª Hipótese (subsituição e algoritmo de remoção):
    - Elemento máximo da **sub-árvore esquerda**.



— Remoção

- [3] Possui duas sub-árvores:
  - Figura ➡️ com exemplo para o elemento 8.
  - Hipótese (subsituição e algoritmo de remoção):
    - Elemento mínimo da **sub-árvore direita**.



Exercícios ✏️

1. Elabore a ficha de atividades disponível no Moodle:
    - `2b_FichaAtividades.pdf`

Exercícios 🖋️

2. Aceda ao seguinte link que permite visualizar a manipulação de uma árvore binária de pesquisa:

   https://www.cs.usfca.edu/~galles/visualization/BST.html

3. Tente uma ordem de inserção de elementos por forma a obter a àrvore ilustrada no slide #22.

4. Qual a àrvore resultante da inserção sequencial de `{1,4,7,9,10,18}`?

   - Vê algum problema na àrvore resultante?
   - https://en.wikipedia.org/wiki/AVL_tree

IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Programação Avançada

2c

ADT Map | Impl. com Árvore Binária de Pesquisa

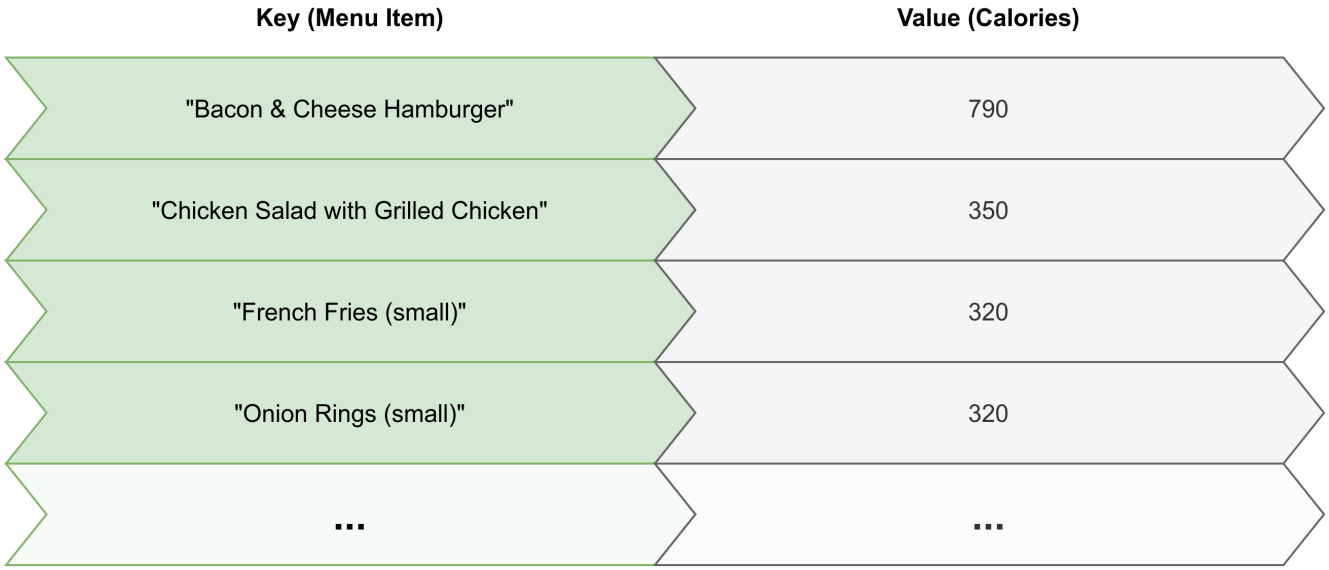
Bruno Silva, Patrícia Macedo

Sumário 📝

- ADT Map
- Interface `Map<K,V>`
  - Implementação (fornecida) com `List` e exemplo
  - Implementação (a finalizar) com BST
    - Motivação
    - Algoritmos recursivos
- Exercícios

ADT Map

- O ADT Map consiste num contentor de mapeamentos *chave:valor*, vulgarmente também chamado de *dicionário*.
  - Não permite chaves duplicadas;
  - O mesmo valor pode estar associado a múltiplas chaves.



Interface `Map<K,V>`

A especificação do ADT Map na linguagem Java é descrito numa *interface*:

```java
package pt.pa.adts;


/**
 * An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
 * @param <K> the type of keys maintained by this map
 * @param <V> the type of mapped values
 */
public interface Map<K, V> {
    V put(K key, V value) throws NullPointerException;
    V get(K key) throws NullPointerException;
    V remove(K key) throws NullPointerException;
    boolean containsKey(K key) throws NullPointerException;
    Collection<K> keys();
    Collection<V> values();
    int size();
    boolean isEmpty();
    void clear();
}
```

Versão comentada da interface disponível no projeto base:

https://github.com/pa-estsetubal-ips-pt/ADTMap_Template

Implementação (fornecida) com `List` e exemplo

- No **projeto base** (faça *git clone*) é fornecida uma implementação completa na classe `MapList` e um exemplo de utilização que mapeia números para o seu número de ocorrências.

```
int[] numbers = {1,4,3,7,4,8,9,1,4,6,4,7,6,9,5,3,6,8,4,6,9};

Map<Integer, Integer> uniqueCount = new MapList<>();

for(int num : numbers) {
    if(uniqueCount.containsKey(num)) {
        int curCount = uniqueCount.get(num);
        uniqueCount.put(num, curCount + 1);
    } else {
        uniqueCount.put(num, 1);
    }
}
//Do not use .toString() for this!
//TODO: 1. show only unique numbers
//TODO: 2. show unique numbers and how many times they occur
```

- ❓ Complete o código em falta, utilizando as operações de `Map`.

Implementação (fornecida) com `List` e exemplo

Implementação fornecida utilizando uma estrutura de dados linear:

```
public class MapList<K,V> implements Map<K,V> {

    private final List<KeyValue> mappings;

    public MapList() {
        mappings = new ArrayList<>();
    }

    @Override
    ...

    private class KeyValue {
        private K key;
        private V value;

        public KeyValue(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

- Guarda os mapeamentos `KeyValue` numa instância de `ArrayList` (já contém funcionalidades de manipulação de um array).

ADT Map | Impl. com BST

- Se notar, todos os métodos principais de `Map` envolvem a pesquisa de uma chave na estrutura de dados subjacente; aliás, basta olhar para o *javadoc* da interface.
- Como visto anteriormente, as **árvores binárias de pesquisa** permitem acelerar significativamente a pesquisa de elementos.
  - Estrutura de dados linear ➡️ $O(n)$
  - Árvore binária de pesquisa ➡️ $O(log\ n)$
- ❗ Temos então o objetivo de obter uma implementação de `Map` baseada nesta última estrutura de dados.
  - A classe `MapBST` no projeto base contém uma implementação praticamente total.

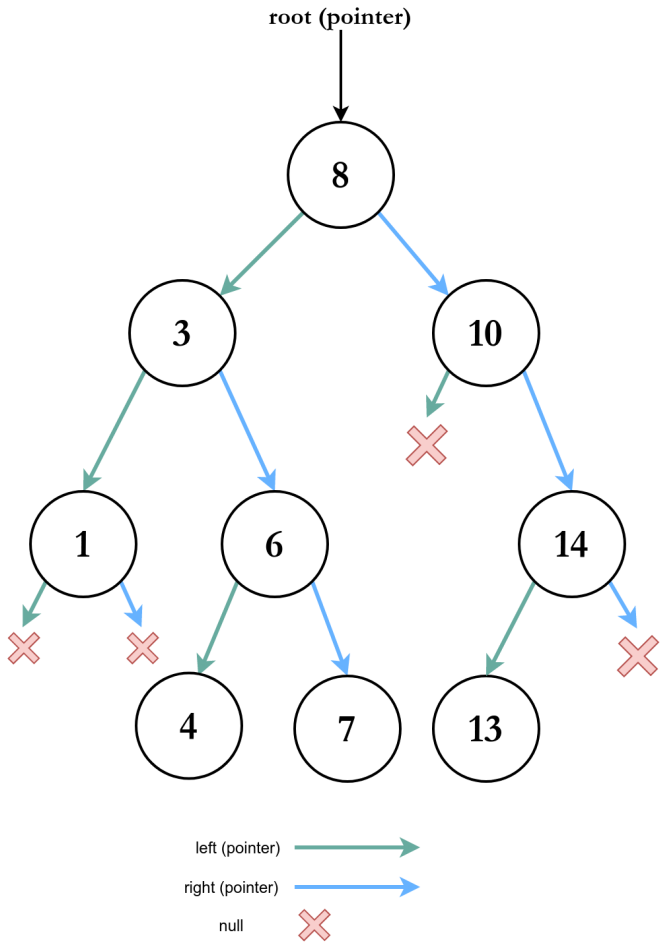
ADT Map | Impl. com BST

- Particularidade desta implementação: **só poderá aceitar chaves que sejam comparáveis** (por motivos que deverão ser óbvios).

```
public class MapBST<K extends Comparable<K>, V> implements Map<K,V> {

    private BSTNode root;

    public MapBST() {
        this.root = null;
    }

    @Override
    ...
    private class BSTNode {
        private K key;
        private V value;

        private BSTNode parent; //útil para operação de remoção
        private BSTNode left;
        private BSTNode right;

        public BSTNode(K key, V value, BSTNode parent, BSTNode left, BSTNode right) {
            ...
        }
    }
```

ADT Map | Impl. com BST

- Exemplo da **estrutura de dados** contendo 9 números (**keys**) ➡️
  - Ponteiros `parent` estão omitidos.
  - **values** estão omitidos.
- ⚠️ A **abstração recursiva** de árvores permitirá a utilização de algoritmos *recursivos* em algumas situações.



ADT Map | Impl. com BST

- Implementações "óbvias":

```
public class MapBST<K extends Comparable<K>, V> implements Map<K,V> {

    private BSTNode root;

    public MapBST() {
        this.root = null;
    }

    @Override
    public boolean isEmpty() {
        return (this.root == null);
    }

    @Override
    public void clear() {
        tis.root = null;
    }

    ...
```

ADT Map | Impl. com BST

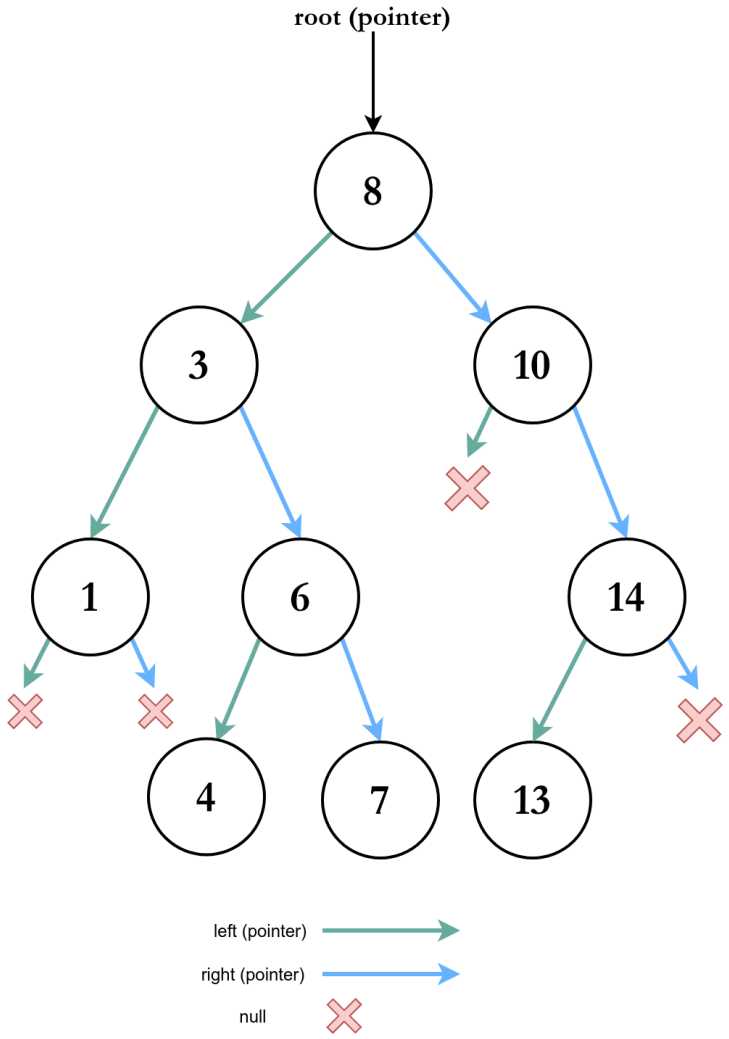
- Implementação *recursiva* de `size()`:

```
public class MapBST<K extends Comparable<K>, V> implements Map<K,V> {

    @Override
    public int size() {
        return size(this.root);
    }

    private int size(BSTNode treeRoot) {
        if(treeRoot == null) return 0;
        else return 1 + size(treeRoot.left) + size(treeRoot.right);
    }

    ...
```

- ❗ Note que a utilização de um atributo `int size` é mais eficiente. O propósito aqui é o de **introduzir/utilizar a abstração recursiva**.

ADT Map | Impl. com BST

- Implementação *recursiva* de `size()`:
  - Simule o algoritmo anterior no diagrama.



ADT Map | Impl. com BST

- ❓ Por forma a poder testar a implementação da classe `MapBST` forneça a implementação dos seguintes dois métodos auxiliares:
  - `private BSTNode searchNodeWithKey(K key, BSTNode treeRoot)`
    - Dada a raiz de uma (sub-)árvore, pesquisa o nó que contém essa chave; `null` se não existir. **Forneça uma implementação recursiva**.
  - `private BSTNode getLeftmostNode(BSTNode treeRoot)`
    - Dada a raiz de uma (sub-)árvore, pesquisa o seu nó mais à esquerda (*contém a chave "mínima"*); `null` se não existir. **Forneça uma implementação recursiva ou iterativa**.

ADT Map | Impl. com BST

- ❓ Execute o método `main()` utilizando a implementação completa de `MapBST`;
- ❓ Utilize o método `MapBST.toString()` que irá mostrar uma representação textual da árvore subjacente:

```
MapBST of size = 8:
│            ┌── {key=9, value=3
│        ┌── {key=8, value=2
│    ┌── {key=7, value=2
│    │   └── {key=6, value=4
└── {key=5, value=2
     │   ┌── {key=4, value=5
     │   │   └── {key=3, value=2
     └── {key=1, value=2
```

- ❓ Teste a remoção de mapeamentos, verificando as árvores resultantes.

Exercícios | Implementação ✒️

1. Altere a implementação por forma a que os métodos `keys()` e `values()` utilizem uma *travessia* **em-ordem** da árvore.
    - No caso de `keys()`, dado que são as chaves da árvore, a coleção irá conter esses elementos ordenados.

Exercícios | Implementação ✒️

2. Adicione ao *output* do método `toString()` informação sobre a **altura da árvore**, e.g.:

```
MapBST of size = 8 and height = 3:
│            ┌── {key=9, value=3
│        ┌── {key=8, value=2
│    ┌── {key=7, value=2
│    │   └── {key=6, value=4
└── {key=5, value=2
     │   ┌── {key=4, value=5
     │   │   └── {key=3, value=2
     └── {key=1, value=2
```

- Implemente/utilize um método auxiliar recursivo:
    - `private int height(BSTNode treeRoot)`

Exercícios | Utilização 🖋️

3. Crie uma classe `MainMenu` que, no método `main()` replique numa instância de ADT Map o menu de calorias apresentado no início da aula.
    - Adicionalmente, e utilizando as operações de `Map`, mostre apenas os *items* do menu com calorias superiores a um *threshold* t.

IPS Instituto Politécnico de Setúbal
Escola Superior de Tecnologia de Setúbal

Programação Avançada

Implementação do TAD Tree
Programação Avançada 2020-21

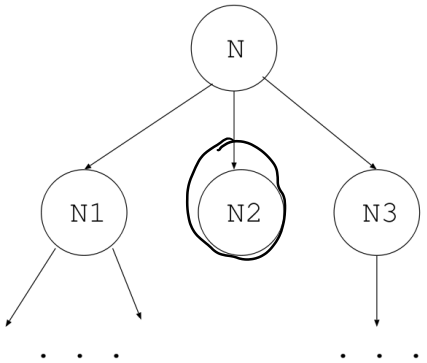
Bruno Silva, Patrícia Macedo

Sumário 

- Especificação do TAD Tree
- Implementação em Java
  - Tipo de Dados Position
  - Interface Tree
  - TreeNode
  - TreeLinked

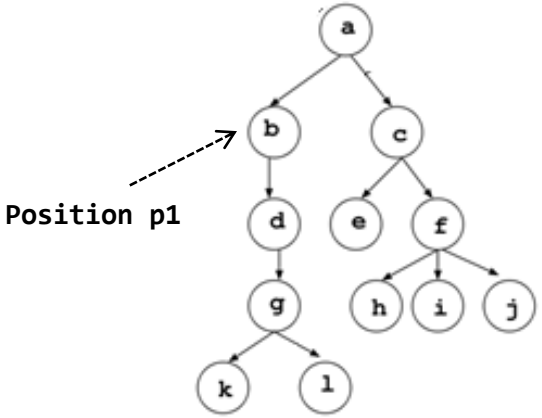
Árvores – Conceitos (revisão)

- Uma árvore é composta por ***nós***;
- No topo da árvore existe um nó especial, chamado ***raiz***;
  - Não possui ascendentes.
- Todos os outros nós têm exatamente um ascendente direto;
- Na Figura, dizemos que N_1, N_2 e N_3 são ***filhos*** de N. Consequentemente, N é ***pai*** de N_1, N_2 e N_3.
- Nós que não têm descendentes são chamados nós ***externos*** ou ***folhas***.
  - Ex.: N_2.
- Nós que não são a raiz nem folhas, são chamados ***internos***.
  - Ex.: N_1 e N_3.

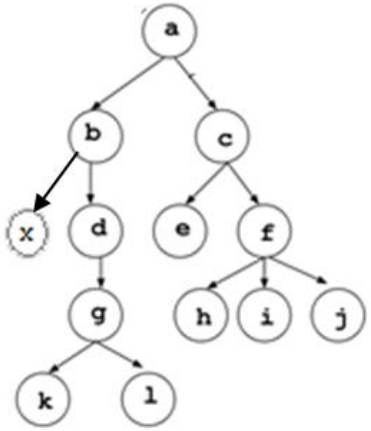


Posição - Conceito

- De forma a podermos facilmente referenciar os nós da árvore sem expormos a usa implementação, introduzimos a noção de Posição (Position).
- A **Posição** modela a noção de **"lugar"** dentro de uma estrutura de dados, onde um único objeto é armazenado.
- É através da referencia para o lugar na árvore que se insere e remove elementos numa árvore.



`insert(x,p1)`



TAD Tree - especificação

- Operações Modificadoras
  - **replace (x, pos):** substitui o elemento que se encontra na posição pos pelo valor **x**, devolve erro se for uma posição que não pertence à árvore.
  - **insert(x,pos):** insere um elemento *x* como sendo filho do nó da árvore que está na posição pos, devolve erro se for uma posição que não pertence à árvore.
  - **insert(x,n,pos):** insere um elemento *x* como sendo o n-ésimo filho do nó da árvore que está na posição pos, devolve erro se for uma posição que não pertence à árvore, ou se n for um numero superior ao numero de filhos daquele nó.
  - **remove(pos):** remove o nó de uma determinada posição pos, devolve erro se for uma posição que não pertence à árvore.

TAD Tree - especificação

- Operações de Verificação
  - **isInternal (pos) :** verifica se a posição **pos** se refere a um nó interno, devolve erro se for uma posição que não pertence à árvore.
  - **isExternal (pos):** verifica se verifica se a posição **pos** se refere a um nó externo, devolve erro se for uma posição que não pertence à árvore.
  - **isRoot(pos)**: verifica se a posição **pos** se refere a um nó do tipo raiz, devolve erro se for uma posição que não pertence à árvore.
  - **isEmpty:** verifica se a árvore está vazia.

TAD Tree - especificação

- Operações Selectoras:
  - **size:** devolve o tamanho da árvore.
  - **elements:** devolve uma coleção iterável com os elementos da árvore.
  - **positions:** devolve uma coleção iterável de posições da árvore.
  - **root**: devolve a raiz da árvore
  - **parent (pos):** devolve a posição do nó pai do nó que se encontra na posição pos, devolve erro se for uma posição que não pertence à árvore.
  - **children (pos)** : devolve a coleção dos filhos de um nó da árvore que se encontra na posição pos, devolve erro se for uma posição que não pertence à árvore.
  - **remove(pos):** remove o nó de uma determinada posição pos, devolve erro se for uma posição que não pertence à árvore.

TAD Position (Posição)

- Modela a noção de **"lugar"** dentro de uma estrutura de dados, onde um único objeto é armazenado;
- Possui apenas um método:
  - E element(): retorna o elemento armazenado nessa posição.

```
public interface Position<E>{
    public E element() throws InvalidPositionException;
}
```

TAD Tree - Interface

```
public interface Tree<E> {
    public int size();

    public boolean isEmpty();

    public Iterable<Position<E>> positions();

    public Iterable<E> elements();

    public E replace(Position<E> position, E elem) throws InvalidPositionException;

    public Position<E> root() throws EmptyTreeException;

    public Position<E> parent(Position<E> position)    throws InvalidPositionException, BoundaryViolationException;

    public Iterable<Position<E>> children(Position<E> position) throws InvalidPositionException;

    public boolean isInternal(Position<E> position) throws InvalidPositionException;

    public boolean isExternal(Position<E> position)  throws InvalidPositionException;

    public boolean isRoot(Position<E> position)   throws InvalidPositionException;

    public Position<E> insert(Position<E> parent, E elem, int order)throws InvalidPositionException,BoundaryViolationException;

    public Position<E> insert(Position<E> parent, E elem) throws InvalidPositionException;

    public E remove(Position<E> position)throws InvalidPositionException;

    public int height();
}
```

TAD Tree: Aplicação

```
TREE Animal
  -Mamifero
    -Cao
    -Gato
    -Vaca
  -Ave
    -Papagaio
    -Aguia
      -Aguia Real
```

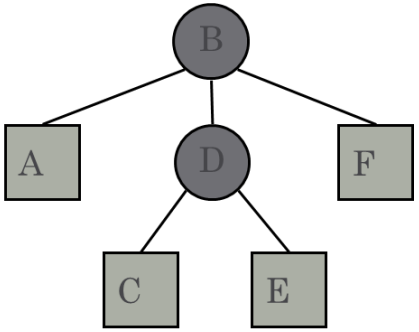
Taxonomia de Animais:

- Construir uma árvore, usando a noção de Posição (Position)
- Após a inserção de um elemento na árvore a posição onde ficou o elemento é devolvida.
- Quando se insere um elemento indica-se qual a posição do pai.

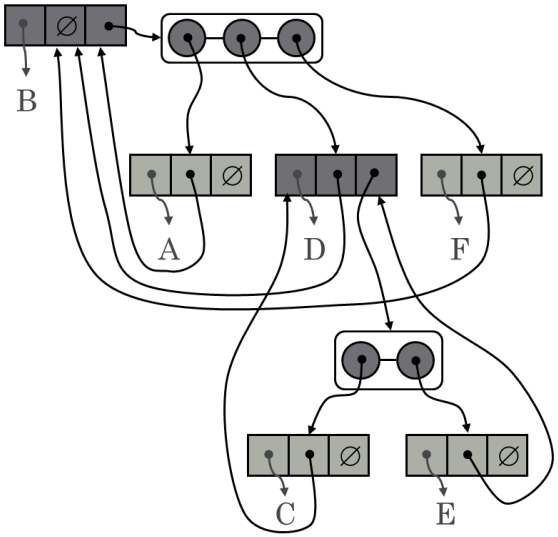
```
public class TADTreeMain {

    public static void main(String[] args) {
        TreeLinked<String> myTree = new TreeLinked("Animal");
        Position<String> root = myTree.root();
        Position<String> posMamifero = myTree.insert(root, "Mamifero");
        Position<String> posAve = myTree.insert(root, "Ave");
        myTree.insert(posMamifero, "Cao");
        Position<String> posGato = myTree.insert(posMamifero, "Gato");
        myTree.insert(posMamifero, "Vaca");
        myTree.insert(posAve, "Papagaio");
        Position<String> posAguia = myTree.insert(posAve, "Aguia");
        myTree.insert(posAguia, "Aguia Real");
        System.out.println("TREE " + myTree);
        System.out.println(myTree);
    }
}
```

TAD Tree Implementação: Usando uma estrutura de nós ligados

- Uma árvore é composta por um conjunto de nós interligados entre si.
- Um nó de uma árvore guarda:
  - element - o elemento
  - parent - referência para o nó pai.
  - children - Uma referencia para a lista de nós filhos



árvore



estrutura de dados da árvore

TAD Tree Implementação: Usando uma estrutura de nós ligados

```
private class TreeNode implements Position<E> {

    private E element;  // element stored at this node
    private TreeNode parent;  // adjacent node
    private List<TreeNode> children;  // children nodes

    TreeNode(E element) {
        this.element = element;
        parent = null;
        children = new ArrayList<>();
    }

    TreeNode(E element, TreeNode parent) {
        this.element = element;
        this.parent = parent;
        this.children = new ArrayList<>();
    }

    public E element() {
        if (element == null) {
            throw new InvalidPositionException();
        }
        return element;
    }
```

Lista de filhos inicializada a vazia. Evita verificações desnecessárias, quando se pretende adicionar um filho

TAD Tree Implementação: Usando uma estrutura de nós ligados

```
private class TreeNode implements Position<E> {

    private E element;  // element stored at this node
    private TreeNode parent;  // adjacent node
    private List<TreeNode> children;  // children nodes

    TreeNode(E element) {
        this.element = element;
        parent = null;
        children = new ArrayList<>();
    }

    TreeNode(E element, TreeNode parent) {
        this.element = element;
        this.parent = parent;
        this.children = new ArrayList<>();
    }

    public E element() {
        if (element == null) {
            throw new InvalidPositionException();
        }
        return element;
    }
```

Lista de filhos inicializada a vazia. Evita verificações desnecessárias, quando se pretende adicionar um filho

TreeLinked : Atributos e Construtor

```
public class TreeLinked<E> implements Tree<E> {


    private TreeNode root;

    public TreeLinked() {
     this.root=null;
    }

    public TreeLinked(E root) {
        this.root = new TreeNode(root);
    }
```

Conversão Position ->TreeNode

- Se observarmos a interface, verifica-se que a maior parte dos métodos tem um parâmetro do tipo Position, parâmetro esse que se refere indiretamente a um nó da árvore.
- A *inner* classe TreeNode é do tipo Position, logo é possível fazer um "cast" para o tipo Position.
- O método *checkPosition* é um método auxiliar que recebe uma posição valida-a e "converte-a" num TreeNode.

```
private TreeNode checkPosition(Position<E> position)
        throws InvalidPositionException {
    if (position == null) {
        throw new InvalidPositionException();
    }

    try {
        TreeNode treeNode = (TreeNode) position;
        if (treeNode.children == null) {   // a lista de filhos nunca pode ser null
            throw new InvalidPositionException("The position is invalid");
        }
        return treeNode;
    } catch (ClassCastException e) {
        throw new InvalidPositionException();
    }
}
```

TreeLinked : insert

- Para inserir um elemento na árvore genérica, é necessário indicar a posição do nó pai.
- Adiciona-se **um nó** à lista de nós filhos, na ordem especificada no parâmetro order.

```
public Position<E> insert(Position<E> parent, E elem, int order)
        throws BoundaryViolationException, InvalidPositionException {
    if(isEmpty()){
        if( parent!= null) throw new InvalidPositionException("Pai não é nulo");
        if (order != 0 ) throw new BoundaryViolationException("Fora de limites");
        this.root = new TreeNode(elem);
        return root;
    }
    TreeNode parentNode = checkPosition(parent);
    if (order < 0 || order > parentNode.children.size()) {
        throw new BoundaryViolationException("Fora de limites");
    }
    TreeNode node = new TreeNode(elem, parentNode);
    parentNode.children.add(order, node);
    return node;

}
```

TreeLinked: métodos recursivos

- A implementação de métodos recursivos seguem a mesma estratégia usada com a implementação de métodos recursivos na BST.
- Uso de um método auxiliar para implementar o mecanismo de recursão a partir de um nó.
- **Exemplo**: Devolver todas os elementos da árvore numa coleção iterável

```
@Override
public Iterable<E> elements() {
    ArrayList<E> lista = new ArrayList<>();
    if (!isEmpty()) {
        elements(root, lista);
    }
    return lista;
}

private void elements(Position<E> position, ArrayList<E> lista) {

    lista.add(position.element()); // visit (position) primeiro, pre-order
    for (Position<E> w : children(position)) {
        elements(w, lista);
    }

}
```

ADT Tree | Exercícios de implementação

1. Faça *clone* do projeto base **ADTTree_Template** (projeto **IntelliJ**) do *GitHub*:

   https://github.com/pa-estsetubal-ips-pt/ADTTree_Template

2. Forneça o código dos métodos por implementar, i.e., os que estão a lançar `NotImplementedException` ;

   Nota: Em relação ao método size, sugere-se que reveja a implementação realizada para a BST

3. Compile e teste o programa fornecido.

ADT Tree | Exercícios Adicionais

1. Considere o algoritmo Breath-first para percorrer a árvore em largura, por níveis.

```
BFS(arvore)
   Coloque a raiz da árvore na fila
   Enquanto a fila não está vazia faça:
           seja n o primeiro nó da fila
           processe n
           para todo o f nó filho de n
                       coloque f na fila
```

Implemente na classe TreeLinked o método Breath-first que devolve uma Coleção Iteravel com os elementos da árvore ordenados segundo o algoritmo Breath-first.

ADT Tree | Exercícios Adicionais

2. O método `checkPosition` fornecido não está a validar se à posição fornecida pertence à árvore.
   2.1 Implemente um método auxiliar (denominado `belongs` que dado um nó verifica se este pertence à arvore T.
   Sugere-se a utilização do seguinte algoritmo:

```
belongs(tree,node)
    Enquanto  parent(node) <> NULL faça
            node<-parent(node)
    Se node=root(tree)
            retorna verdade
     senão
            retorna falso
```

   2.2 Altere o método checkPosition, para incluir esta validação.

Estudar e Rever

Tree

- Páginas 280-296

Ficha 2b

Ficha de atividades - Conceitos sobre árvores binárias

1 – Arvores Binárias

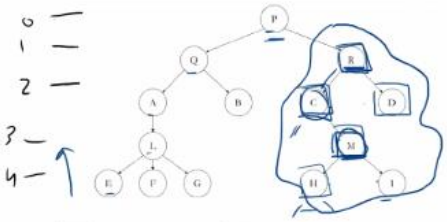
Considerando a árvore acima indique:

Ordem	2
Altura	3
Travessia em-ordem	A B C D E F G
Travessia pré-ordem	C B A F E D G
Travessia pós-ordem	A B D E G F C

Ficha 2a

Programação Avançada
Ficha de Atividades | Conceitos de árvores

Considere a árvore apresentada na figura e preencha a tabela à luz dos conceitos aprendidos.



Raiz	P
Nós nível 2	A B C D
Altura da árvore	4
Nós internos	-
Nós externos	-
Grau do nó B	0
Grau do nó L	3
Caminho de G até raiz	G - L - A - Q - P
Travessia pré-ordem	
Travessia pós-ordem	

P Q A L E F G B R C M H I D

E F G L A B Q H I M C D R P