# Aula 13

## Estruturas de Dados

### Árvores Binárias

*Programação II, 2015-2016*

*v1.01, 18-05-2016*

DETI, Universidade de Aveiro

13.1

**Objectivos:**
- Árvores binárias;
- Árvores binárias de procura.

## Conteúdo

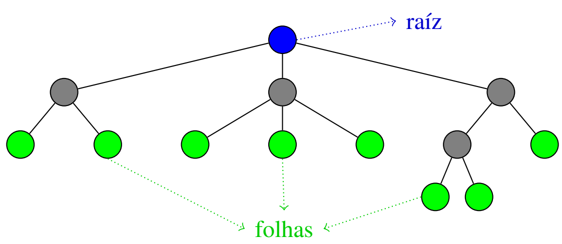


13.2

## 1 Árvore

### Árvores: Introdução

- O que são estruturas de dados em Árvore?




- A árvore consiste de nós ligados por *ramos* orientados (é um caso particular de grafo).
- Cada nó (*pai*) pode ter ligações para outros nós (*filhos*).
- Um dos nós não tem pai e é chamado *raiz*.
- Todos os outros nós têm um pai (e apenas um).
- Nós sem filhos são chamados *folhas*.
- A raiz representa-se no topo e as folhas na base.
- Uma árvore não pode incluir ciclos.
- Cada nó pode ser considerado como a raiz de uma *subárvore*.

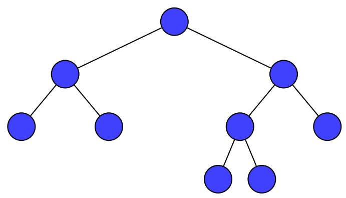
13.3

- Cada nó é atingível a partir da raiz através de uma sequência única de ramos, chamada de *caminho*.
- O número de ramos de um caminho é chamado de *comprimento* do caminho.
- O *nível* de um nó é:
  - comprimento do caminho $+1$
  - o nó raiz tem nível 1
- A *altura* de uma árvore é o nível máximo de um nó na árvore.

13.4

## 2 Árvore Binária

- Estrutura de dados recursiva em que cada nó se pode ligar, no máximo, a dois nós filhos.
- Cada nó pode ser encarado ele próprio como uma árvore binária



```
class Node<T>
{
   T elem;
   Node<T> leftChild;
   Node<T> rightChild;
}
```

13.5

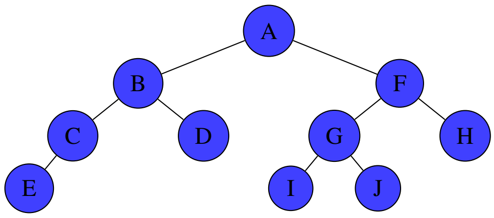
### Árvores Binárias: Percursos

- Percurso ou travessia da árvore:
  - Quando pretendemos percorrer todos os nós de uma árvore de forma sistemática temos necessidade de ter um algoritmo de travessia
  - Baseia-se na ordem em que a raiz é visitada em relação a seus descendentes.
- Os diferentes percursos têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
  - Para cada aplicação, pode haver um percurso mais adequado.

13.6

- *Prefixo* [*Pre-order*] (`RED`: Raiz, Esquerda, Direita)
  - Processar o nó raiz.
  - Percurso prefixo da sub-árvore esquerda
  - Percurso prefixo da sub-árvore direita
- *Infixo* [*In-order*] (`ERD`: Esquerda, Raiz, Direita)
  - Percurso infixo da sub-árvore esquerda
  - Processar o nó raiz
  - Percurso infixo da sub-árvore direita
- *Posfixo* [*Post-order*] (`EDR`: Esquerda, Direita, Raiz)
  - Percurso posfixo da sub-árvore esquerda
  - Percurso posfixo da sub-árvore direita

13.7 – Processar o nó raiz




```
Prefixo (RED):  A, B, C, E, D, F, G, I, J, H
Infixo (ERD):   E, C, B, D, A, I, G, J, F, H
Posfixo (EDR): E, C, D, B, I, J, G, H, F, A
```

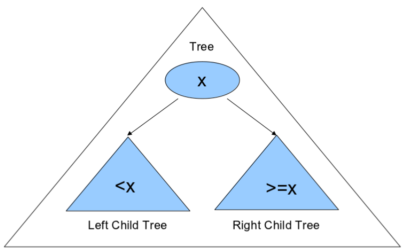
13.8

## 3 Árvore Binária de Procura

- São outra forma de implementar **dicionários**
- Como já tinhamos analisado nas tabelas de dispersão:
  - A complexidade de uma estrutura de dados tem duas componentes: Espaço e Tempo.
  - As listas ligadas têm bom desempenho no Espaço pois permitem uma alocação dinâmica;
  - Os vectores (*arrays*) têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
  - Num vector ordenado podemos utilizar "pesquisa binária";
  - Numa estrutura dinâmica com listas ligadas temos o problema do acesso sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação dinâmica com desempenho temporal (na pesquisa) similar ao de um vector ordenado. 13.9

- Árvore binária em que todos os nós estão estruturalmente ordenados por uma chave.
- Todos os nós de uma eventual sub-árvore filha à esquerda terão uma chave inferior à do nó raiz.
- Todos os nós de uma eventual sub-árvore filha à direita terão uma chave igual ou superior à da raiz.
  - Esta regra é aplicável a qualquer nó de uma árvore binária de procura 13.10

### Árvore Binária de Procura




- Sendo as árvores binárias um exemplo de uma estrutura de dados recursiva, os algoritmos mais simples para as manipular tendem também a ser recursivos;
- Algoritmos recursivos em estruturas de dados recursivas replicam a recursividade existente na estrutura de dados para os próprios algoritmos;
- Neste caso, temos uma árvore construída por um nó e duas subárvores, pelo que o algoritmo recursivo repetirá, na ordem desejada, esta estrutura: processamento do nó, e invocação recursiva para as duas subárvores (se existirem).

13.11

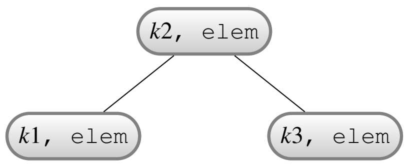
## 3.1 Dicionário implementado como árvore binária de procura

- Nome do módulo:
  - `BinarySearchTree`
- Serviços:
  - `BinarySearchTree()`: construtor;
  - `set(key,elem)`: criar/actualizar uma associação;
  - `get(key)`: devolve elemento associado a uma chave;
  - `remove(key)`: apaga uma chave com o elemento associado;
  - `contains(key)`: existe uma chave;
  - `isEmpty()`: árvore vazia;
  - `size() -> int`: número de entradas;
  - `clear()`: esvazia a estrutura;
  - `keys()`: devolve um vector com todas as chaves existentes.

13.12

### Árvore Binária de Procura

- Os elementos `(key,elem)` estão armazenados na árvore binária da seguinte forma:
  - Todos os elementos na sub-árvore esquerda de cada nó $X$ têm uma `key` menor ao valor da `key` do nó $X$.
  - Todos os elementos na sub-árvore direita de cada nó $X$ têm uma `key` maior do que o valor da `key` do nó $X$.




$k1 < k2 < k3$

13.13

### Árvores Binárias de Procura: pesquisa

- Algoritmo (tirando proveito da ABP):

```
search n in Tree.root
  if n.key < Tree.root.key then
    search n in LeftChildTree.root
  else if n.key > Tree.root.key then
    search n in RightChildTree.root
  else // n.key == Tree.root.key
    result = Tree.root // FOUND!
```

13.14

### Árvores binárias de procura: inserir um elemento

- Algoritmo (inserir como "folha")

```
insert n in Tree.root
  if Tree.root == null then
    Tree.root = n
  else if n.key < Tree.key then
    insert n in LeftChildTree.root
  else // n.key >= Tree.key
    insert n in RightChildTree.root
```

13.15

## Árvores binárias de procura: remover um elemento

- Várias Hipóteses:
  - Nó folha (sem filhos):
    - Colocar, no nó pai, a referência para este nó a `null`;
  - Nó só com uma subárvore:
    - Suprimir o nó a remover fazendo o ligação do seu pai ao nó da subárvore
  - Nó tem as duas subárvores:
    - Inserir uma dos filhos como folha do outro, e substituir o no pela raiz resultante;
    - Substituir o nó a eliminar pelo menor elemento na subárvore da direita (ou vice-versa).
- Conclusão: Necessitamos sempre de uma referência para o nó pai do elemento a remover (caso exista. . . )

13.16

## Árvores binárias de procura: remoção por inserção como folha

- Algoritmo

```
delete n from Tree.root
  if n == Tree.root then
    if LeftChildTree.root == null then
      Tree.root = RightChildTree.root
    else if RightChildTree.root == null then
      Tree.root = LeftChildTree.root
    else
      Tree.root = insert LeftChildTree.root in RightChildTree.root
  else if n.key < Tree.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.key
    delete n from RightChildTree.root
```

13.17

## Árvores binárias de procura: remoção por procura de mínimo

- Algoritmo

```
delete n from Tree.root
  if n == Tree.root then
    if LeftChildTree.root == null then
      Tree.root = RightChildTree.root
    else if RightChildTree.root == null then
      Tree.root = LeftChildTree.root
    else
      min = searchMinimum from RightChildTree.root
      delete min from RightChildTree.root
      min.LeftChildTree = LeftChildTree
      min.RightChildTree = RightChildTree
      Tree.root = min
  else if n.key < Tree.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.key
    delete n from RightChildTree.root
```

13.18

## Árvores binárias: balanceamento

- Uma árvore está equilibrada se:
  - a diferença das alturas das suas sub-árvores não é superior a 1;
  - todas as sub-árvores estão equilibradas;
- Para mantermos a árvore equilibrada temos de implementar operações de `insert` e *delete* que mantenham a árvore equilibrada
- A manutenção do equilíbrio de uma árvore faz com que mantenhamos a complexidade $O(\log(n))$

13.19