Aula 13

Estruturas de Dados

Árvores Binárias

Programação II, 2017-2018

v1.12, 22-05-2018

DETI, Universidade de Aveiro

13.1

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

Conteúdo

13.2

Colecções de dados: o que vimos até agora

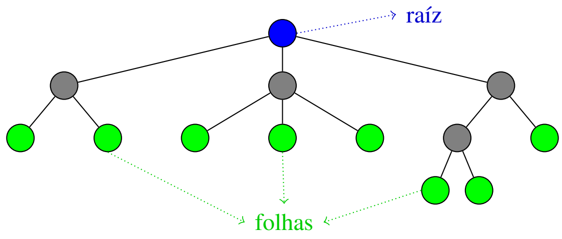
- `LinkedList`
  - `addFirst()`, `addLast()`, `removeFirst()`, `first()`, ...
- `SortedList`
  - `insert()`, `remove()`, `first()`, ...
- `Stack`
  - `push()`, `pop()`, `top()`, ...
- `Queue`
  - `in()`, `out()`, `peek()`, ...
- `KeyValueList` e `HashTable` (implementam o conceito de **dicionário**)
  - `set()`, `get()`, `remove()`, ...

13.3

1 Árvore

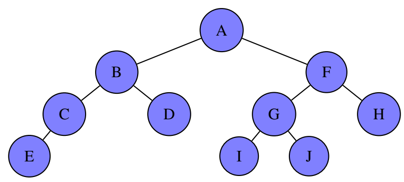
Árvores: Introdução

- 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 ramos 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*.

13.4



- Cada nó é atingível a partir da raiz através de uma sequência única de ramos, chamada de *caminho* do nó.
  - O caminho do nó J é: A-F-G-J.
- O número de ramos de um caminho é chamado de *comprimento* do caminho.
  - O comprimento do caminho A-F-G-J é: 3.
- O *nível* de um nó é o comprimento do caminho $+ 1$.
  - O nível do nó J é: 4.
  - O nó raiz (A) tem nível 1.
- A *altura* de uma árvore é o nível do nó mais profundo.
  - A altura desta árvore é: 4.
  - Uma árvore vazia tem altura 0.

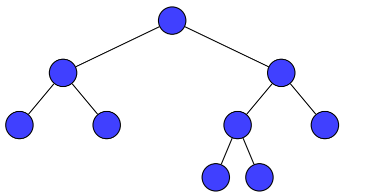
13.5

- *Atenção*: há outras definições de árvore!
- A definição acima é a mais usual em Informática.
- Na Matemática (teoria de grafos), uma *árvore* é definida de forma mais geral, como um *grafo* (não-orientado) *conexo* e *acíclico*.

13.6

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;
}
```

13.7

Árvores Binárias: Percursos

- *Percurso* ou *travessia* de uma árvore:
  - É um algoritmo que permite percorrer todos os nós da árvore de forma sistemática, sem repetições.
- Há muitas travessias possíveis e podem classificar-se em
  - *Travessias em largura*: percorrem nós irmãos antes de avançar para os filhos, por exemplo da esquerda para a direita, de cima para baixo.
  - *Travessias em profundidade*: percorrem nós filhos antes dos nós irmãos.
- Os diferentes percursos têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
  - Para cada aplicação, pode haver um percurso mais adequado.

13.8

- As *travessias em profundidade* podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- *Prefixo* (*Pré-ordem*) (RED: Raiz, Esquerda, Direita)

  R: Processar o nó raiz.

  E: Percurso prefixo da sub-árvore esquerda.

  D: Percurso prefixo da sub-árvore direita.
- *Infixo* (*Em-ordem*) (ERD: Esquerda, Raiz, Direita)

  E: Percurso infixo da sub-árvore esquerda.

  R: Processar o nó raiz.

  D: Percurso infixo da sub-árvore direita.
- *Posfixo* (*Pós-ordem*) (EDR: Esquerda, Direita, Raiz)

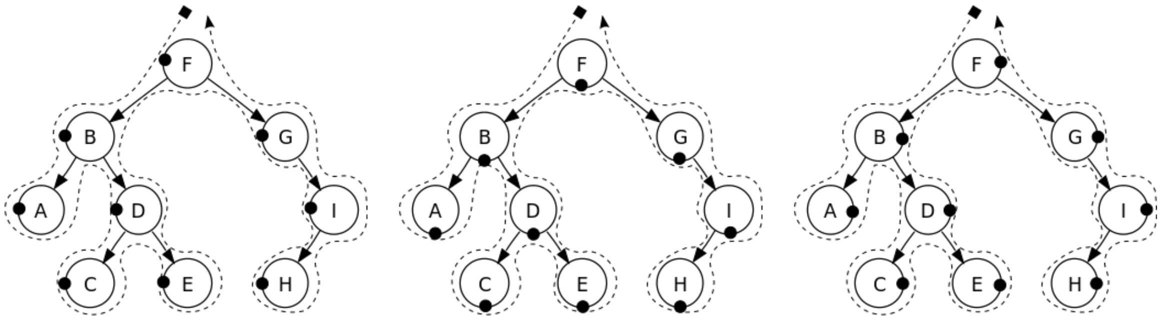
  E: Percurso posfixo da sub-árvore esquerda.

  D: Percurso posfixo da sub-árvore direita.

  R: Processar o nó raiz.

13.9

Pré-ordem **Em-ordem** **Pós-ordem**

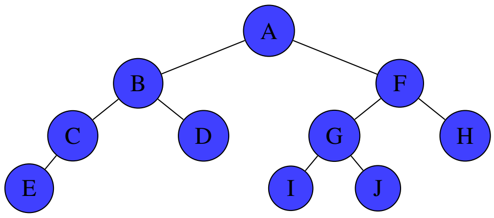


F, B, A, D, C, E, G, I, H

A, B, C, D, E, F, G, H, I

A, C, E, D, B, H, I, G, F

13.10




```
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
```

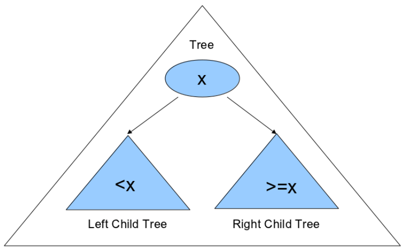
13.11

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.

13.12

- Uma *árvore binária de procura* é uma árvore binária em que a *chave* armazenada em cada nó:
  - é maior que todas as chaves na sua subárvore esquerda
  - é menor* que todas as chaves na sua subárvore direita.
  - (* Chaves iguais podem ser colocadas à direita, por exemplo.)



13.13

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 constituída por um nó raiz e duas subárvores, pelo que o algoritmo recursivo repetirá, na ordem desejada, esta estrutura: processamento do nó raiz, invocação recursiva para cada subárvore.

13.14

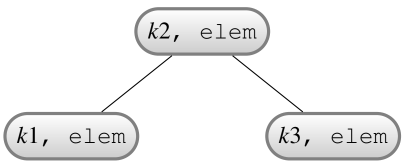
3.1 Dicionário implementado como árvore binária de procura

- 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()`: número de entradas;
  - `clear()`: esvazia a estrutura;
  - `keys()`: devolve um vector com todas as chaves existentes.

13.15

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.



$k1 < k2 < k3$

13.16

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!
```

13.17

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
```

13.18

Árvores binárias de procura: remover um elemento

- Se é um nó folha (zero filhos):
  - Colocar, no nó pai, a referência para este nó a `null`.
- Se é um nó só com uma subárvore (1 filho):
  - Suprimir o nó a remover fazendo o ligação do seu pai ao nó da subárvore.
- Se é um nó com duas subárvores (2 filhos):
  - Substituir o nó a eliminar pelo menor elemento na subárvore da direita (ou pelo maior da esquerda).
  - (Uma alternativa seria inserir um dos filhos como folha do outro e substituir o nó pela raiz resultante. Mas cria árvores menos eficientes.)

13.19

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
```

13.20

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
```

- *Cuidado*: pode aumentar a altura da árvore!

13.21

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 `remove` que mantenham a árvore equilibrada.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.

13.22