

Aula 13

Estruturas de Dados

Árvores Binárias

Programação II, 2018-2019

v1.13 2019-06-02

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

1 **Árvore**

2 **Árvore Binária**

3 **Árvore Binária de Procura**

4 **Dicionário implementado com árvore binária de procura**

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

1 **Árvore**

2 **Árvore Binária**

3 **Árvore Binária de Procura**

4 **Dicionário implementado com árvore binária de procura**

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

Coleções de dados: o que vimos até agora

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

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

Coleções de dados: o que vimos até agora

Árvore

Árvore Binária

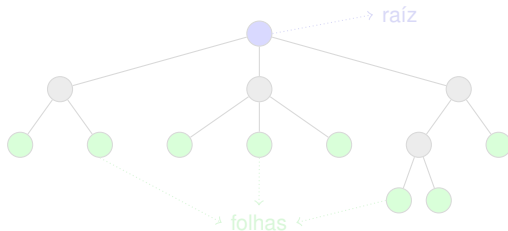
Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

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

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**.

Árvore

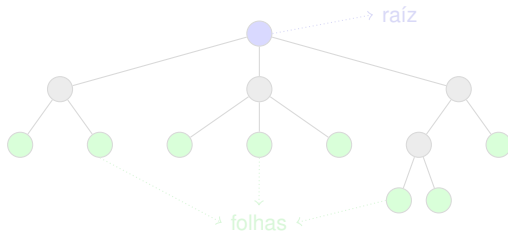
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

Árvores: Introdução

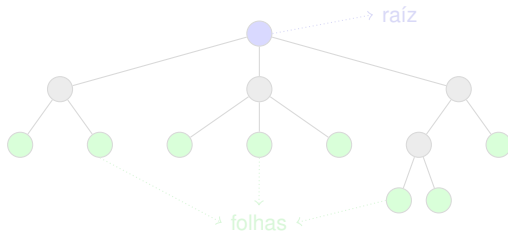
- O que são estruturas de dados em Árvore?



- 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**.

Árvores: Introdução

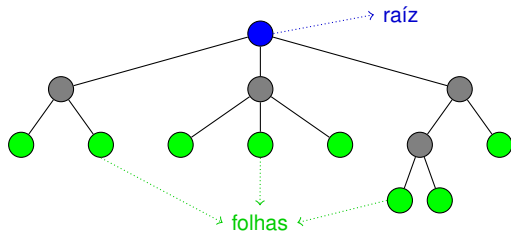
- O que são estruturas de dados em Árvore?



- 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**.

Árvores: Introdução

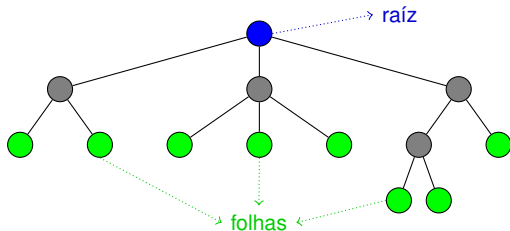
- O que são estruturas de dados em Árvore?



- 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**.

Árvores: Introdução

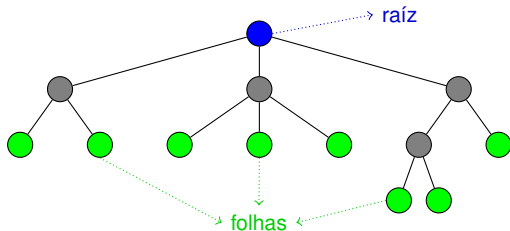
- O que são estruturas de dados em Árvore?



- 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**.

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**.

Árvore

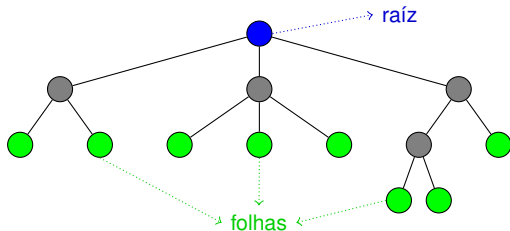
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

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**.

Árvore

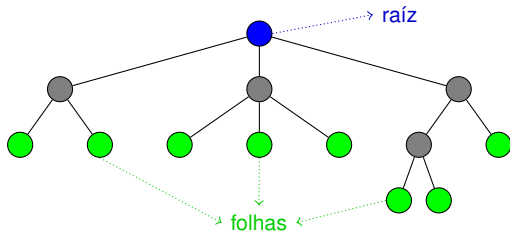
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

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**.

Árvore

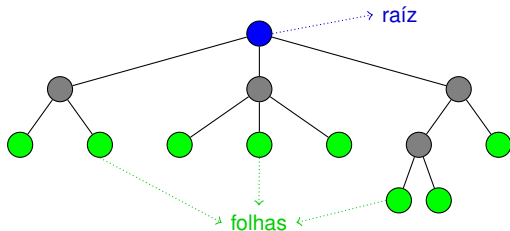
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

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**.

Árvore

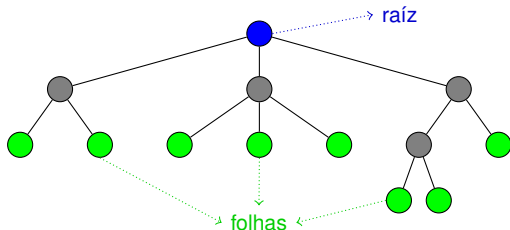
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

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**.

Árvore

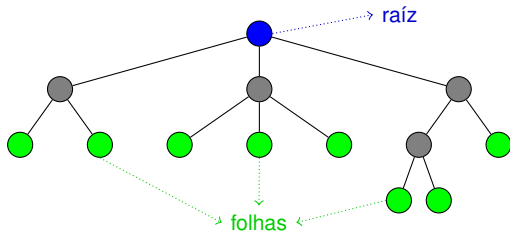
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

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**.

Árvore

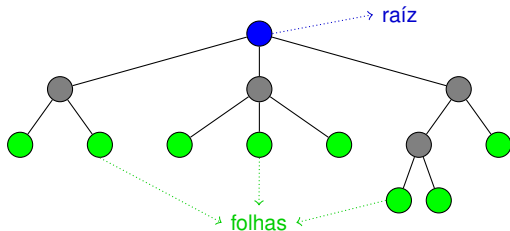
Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

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**.

Árvore

Árvore Binária

Árvore Binária de Procura

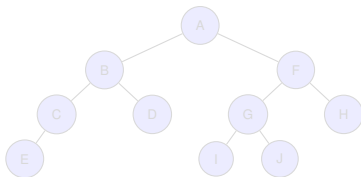
Dicionário implementado com árvore binária de procura

Árvore

Árvore Binária

Árvore Binária de Procura

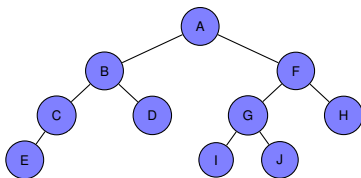
Dicionário implementado com árvore binária de procura



- 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.

Árvore

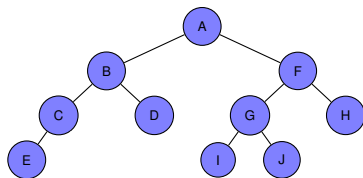
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

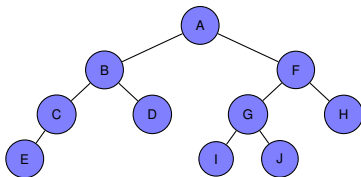
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

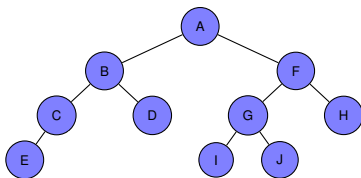
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

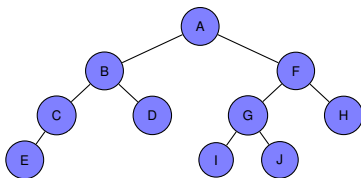
- Cada nó é atingível a partir da raiz através de uma sequência única de ramos, chamada de **caminho** do nó.
 - O caminho do nó J é: A-F-G-J.
- O número de ramos de um caminho é chamado de **comprimento** do caminho.
 - O comprimento do caminho A-F-G-J é: 3.
- O **nível** de um nó é o comprimento do 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.

Árvore

Árvore Binária

Árvore Binária de Procura

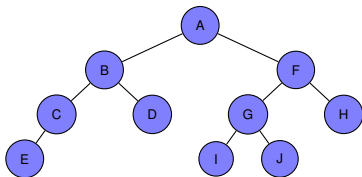
Dicionário implementado com árvore binária de procura



- 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.

Árvore

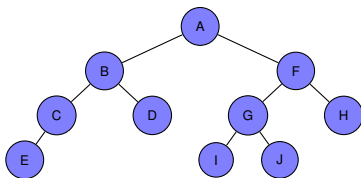
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

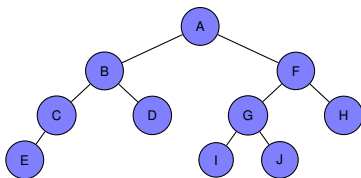
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

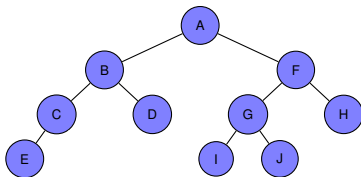
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

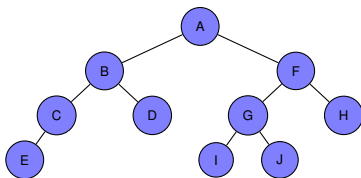
Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

Árvore

Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- 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.

- **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*.

- **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*.

- **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*.

Árvore

Árvore Binária

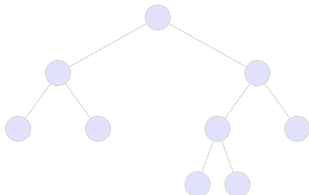
Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

- **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*.

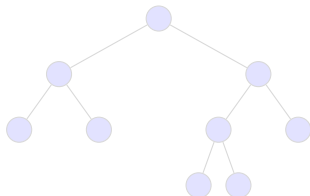
Árvore binária

- 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.



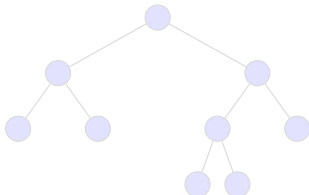
```
class Node<T>
{
    T elem;
    Node<T> leftChild;
    Node<T> rightChild;
}
```

- 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.



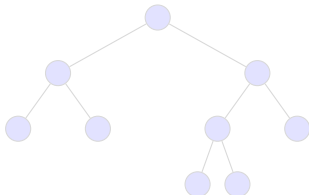
```
class Node<T>
{
    T elem;
    Node<T> leftChild;
    Node<T> rightChild;
}
```

- 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.



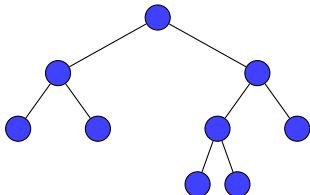
```
class Node<T>
{
    T elem;
    Node<T> leftChild;
    Node<T> rightChild;
}
```

- 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.



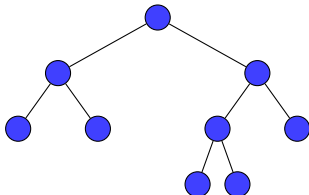
```
class Node<T>
{
    T elem;
    Node<T> leftChild;
    Node<T> rightChild;
}
```

- 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;
}
```

- 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;
}
```

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

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- **Travessia** ou **percurso** 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.
- As diferentes travessias têm normalmente o mesmo custo.
- A diferença está no efeito produzido.
 - Para cada aplicação, pode haver uma travessia mais adequada.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - 1. Visitar a raiz
 - 2. Percurso em pré-ordem a sub-árvore esquerda
 - 3. Percurso em pré-ordem a sub-árvore direita
- **Em ordem** (ERD: Esquerda, Raiz, Direita)
 - 1. Percurso em ordem a sub-árvore esquerda
 - 2. Visitar a raiz
 - 3. Percurso em ordem a sub-árvore direita
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - 1. Percurso em pós-ordem a sub-árvore esquerda
 - 2. Percurso em pós-ordem a sub-árvore direita
 - 3. Visitar a raiz

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

- As **travessias em profundidade** podem subclassificar-se em função da ordem em que a raiz é visitada em relação a seus descendentes.
- **Em pré-ordem** (RED: Raiz, Esquerda, Direita)
 - R: Processar o nó raiz.
 - E: Percorrer em pré-ordem a sub-árvore esquerda.
 - D: Percorrer em pré-ordem a sub-árvore direita.
- **Em-ordem** (ERD: Esquerda, Raiz, Direita)
 - E: Percorrer em-ordem a sub-árvore esquerda.
 - R: Processar o nó raiz.
 - D: Percorrer em-ordem a sub-árvore direita.
- **Em pós-ordem** (EDR: Esquerda, Direita, Raiz)
 - E: Percorrer em pós-ordem a sub-árvore esquerda.
 - D: Percorrer em pós-ordem a sub-árvore direita.
 - R: Processar o nó raiz.

Árvores Binárias: Travessias em Profundidade

Estruturas de Dados

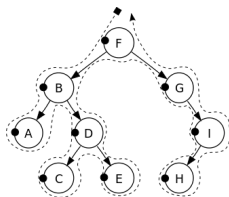
Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

Pré-ordem

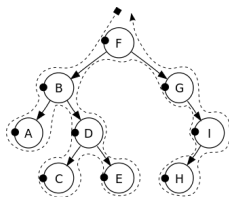


F, B, A, D, C, E, G, I, H

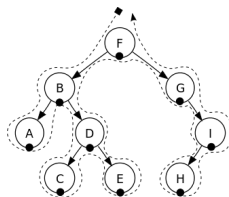
Árvores Binárias: Travessias em Profundidade

Pré-ordem

Em-ordem



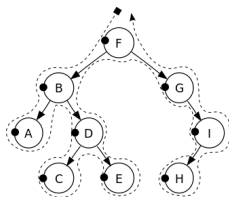
F, B, A, D, C, E, G, I, H



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

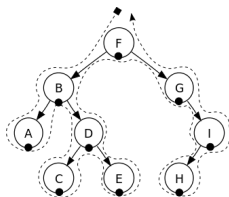
Árvores Binárias: Travessias em Profundidade

Pré-ordem



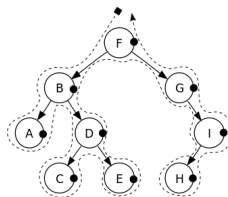
F, B, A, D, C, E, G, I, H

Em-ordem



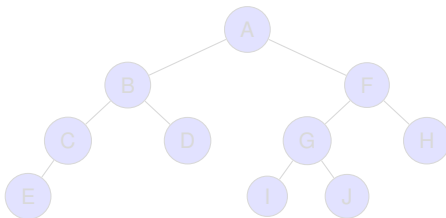
A, B, C, D, E, F, G, H, I

Pós-ordem



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

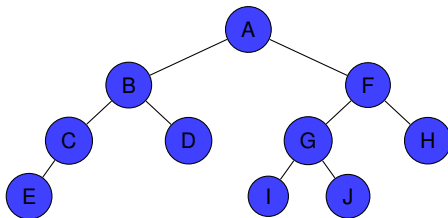
Árvores Binárias: Travessias



Prefixo (RED): A, B, C, E, D, F, G, I, J, H

Infixo (ERD): E, C, B, D, A, I, G, J, F, H

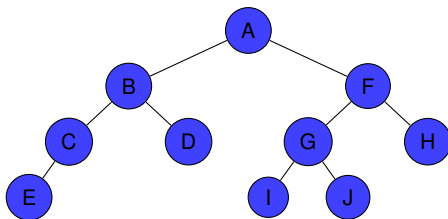
Posfixo (EDR): E, C, D, B, I, J, G, H, F, A



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



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

Árvores Binárias de Procura: Motivação

- São outra forma de implementar **dicionários**
- Como já tínhamos analisado nas tabelas de dispersão:
 - A complexidade de uma estrutura de dados tem duas componentes: Espaço e Tempo.
 - As tabelas de dispersão têm bom desempenho no Espaço, pois permitem uma alocação dinâmica.
 - Os vetores (arrays) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vetor ordenado podemos usar a pesquisa binária;
 - Num lista ligada estamos limitados à pesquisa sequencial (precisamos percorrer todos os elementos até encontrar o pretendido).
- **Árvore Binária de Procura**: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vetor ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

- São outra forma de implementar **dicionários**
- Como já tínhamos 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*) ordenados têm bom desempenho no Tempo.
- Se quisermos pesquisar um elemento:
 - Num vector ordenado podemos utilizar pesquisa binária;
 - Numa lista ligada estamos limitados à pesquisa sequencial (percorrer todos os elementos até encontrar o pretendido).
- Árvore Binária de Procura: uma implementação com alocação dinâmica de espaço e desempenho temporal similar ao de um vector ordenado.

Árvore Binária de Procura: Definição

- 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.
- (Se houver chaves iguais, podem ser colocadas à direita, por exemplo.)

Árvore Binária de Procura: Definição

- 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.
 - (Se houver chaves iguais, podem ser colocadas à direita, por exemplo.)

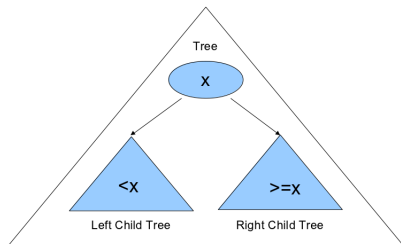
Árvore Binária de Procura: Definição

- 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.
 - (Se houver chaves iguais, podem ser colocadas à direita, por exemplo.)

- 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.
 - (Se houver chaves iguais, podem ser colocadas à direita, por exemplo.)

Árvore Binária de Procura: Definição

- 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.
 - (Se houver chaves iguais, podem ser colocadas à direita, por exemplo.)



- 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.

- 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.

- 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.

- 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.

Dicionário implementado com árvore binária de procura

Estruturas de Dados

Árvore

Árvore Binária

Árvore Binária de Procura

Dicionário implementado com árvore binária de procura

- Nome do módulo:

```
dict.BinarySearchTree
```

- Serviços:

```
BinarySearchTree(): construtor  
get(key, elem): retorna elem se existir;  
put(key): insere elemento chave=key e valor elem  
remove(key): remove elem chave=key e valor elem  
isEmpty(): retorna True se não há elementos  
contains(key): retorna True se contém  
isEmpty(): retorna False  
size(): retorna o número de elementos  
clear(): remove todos os elementos  
keys(): retorna um vetor com todas as chaves  
values():
```


- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

- 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.

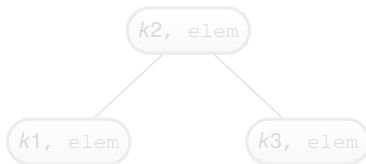
- Nome do módulo:
 - `BinarySearchTree`
- Serviços:
 - `BinarySearchTree()`: construtor;
 - `set(key, elem)`: criar/actualizar uma associação;
 - `get(key)`: devolve elemento associado a uma chave;
 - `remove(key)`: apaga uma chave com o elemento associado;
 - `contains(key)`: existe uma chave;
 - `isEmpty()`: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

- 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.

- 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.

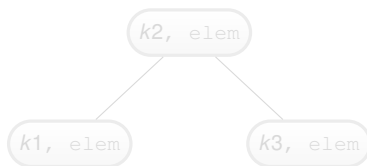
Árvore Binária de Procura

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



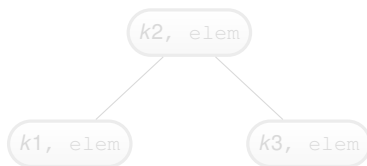
$$k1 < k2 < k3$$

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



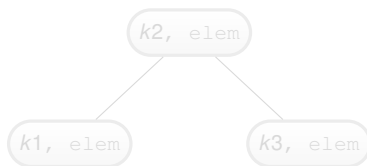
$$k1 < k2 < k3$$

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



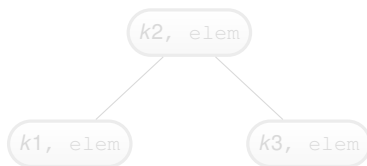
$$k1 < k2 < k3$$

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



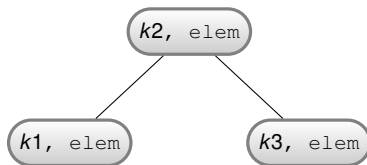
$$k1 < k2 < k3$$

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



$$k1 < k2 < k3$$

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



$$k1 < k2 < k3$$

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

- Algoritmo (tirando proveito da ABP):

```
search n in Tree.root
  if Tree.root == null then
    result = null // NOT FOUND!
  else 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!
```

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

- Algoritmo (tirando proveito da ABP):

```
search n in Tree.root
  if Tree.root == null then
    result = null // NOT FOUND!
  else 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!
```

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

- Algoritmo (tirando proveito da ABP):

```
search n in Tree.root
  if Tree.root == null then
    result = null // NOT FOUND!
  else 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!
```

Árvore

Árvore Binária

Árvore Binária de
ProcuraDicionário
implementado com
árvore binária de
procura

- Algoritmo (tirando proveito da ABP):

```
search n in Tree.root
  if Tree.root == null then
    result = null // NOT FOUND!
  else 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!
```

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

- Algoritmo (inserir como “folha”)

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


- Algoritmo (inserir como “folha”)

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

- Algoritmo (inserir como “folha”)

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

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):
 - Substituir o nó fazendo a ligação do seu pai ao nó da subárvore.
- Se é um nó com duas subárvores (2 filhos):
 - Substituir o nó e eliminar pelo menor elemento da subárvore da direita (ou pelo maior da esquerda).
 - (Uma alternativa seria fazer uma fusão entre as duas subárvores e substituir o nó pelo resultado. Mas esta árvore poderia ficar desequilibrada.)

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 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.)

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 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.)

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 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.)

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 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.)

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 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.)

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 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.)

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 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.)

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

• Cuidado: pode aumentar a altura da árvore!

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

- Cuidado: pode aumentar a altura da árvore!

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

- Cuidado: pode aumentar a altura da árvore!

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.root.key then
    delete n from LeftChildTree.root
else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

- Cuidado: pode aumentar a altura da árvore!

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.root.key then
    delete n from LeftChildTree.root
  else // n.key >= Tree.root.key
    delete n from RightChildTree.root
```

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

Árvore

Árvore Binária

Árvore Binária de
Procura

Dicionário
implementado com
árvore binária de
procura

- Uma árvore está **equilibrada** se:
 - a diferença das alturas das suas sub-árvores não é superior a 1;
 - ambas as sub-árvores estão equilibradas.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.
- É possível manter a árvore sempre equilibrada com implementações mais complexas das operações de `insert` e `remove`. (Mas sai fora do âmbito desta disciplina.)

- Uma árvore está **equilibrada** se:
 - a diferença das alturas das suas sub-árvores não é superior a 1;
 - todas as sub-árvores estão equilibradas.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.
- É possível manter a árvore sempre equilibrada com implementações mais complexas das operações de `insert` e `remove`. (Mas sai fora do âmbito desta disciplina.)

- Uma árvore está **equilibrada** se:
 - a diferença das alturas das suas sub-árvores não é superior a 1;
 - todas as sub-árvores estão equilibradas.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.
- É possível manter a árvore sempre equilibrada com implementações mais complexas das operações de `insert` e `remove`. (Mas sai fora do âmbito desta disciplina.)

- Uma árvore está **equilibrada** se:
 - a diferença das alturas das suas sub-árvores não é superior a 1;
 - todas as sub-árvores estão equilibradas.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.
- É possível manter a árvore sempre equilibrada com implementações mais complexas das operações de `insert` e `remove`. (Mas sai fora do âmbito desta disciplina.)

- Uma árvore está **equilibrada** se:
 - a diferença das alturas das suas sub-árvores não é superior a 1;
 - todas as sub-árvores estão equilibradas.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.
- É possível manter a árvore sempre equilibrada com implementações mais complexas das operações de `insert` e `remove`. (Mas sai fora do âmbito desta disciplina.)

- Uma árvore está **equilibrada** se:
 - a diferença das alturas das suas sub-árvores não é superior a 1;
 - todas as sub-árvores estão equilibradas.
- Manter uma árvore equilibrada permite garantir complexidade $O(\log n)$ para as operações de pesquisa, inserção e remoção.
- É possível manter a árvore sempre equilibrada com implementações mais complexas das operações de `insert` e `remove`. (Mas sai fora do âmbito desta disciplina.)