Sistemas de Informação | AMF
Classificação e Pesquisa de Dados

Aula 10 - Pesquisa de dados digital. Árvores de busca

Cristiano Santos

Baseado no material de proposto por Rhauani Fazul

Conteúdo Programático

1. Métodos de Classificação de Dados

2. Pesquisa de Dados

1. Famílias de métodos de pesquisa de dados
2. Pesquisa sequencial (pesquisa linear)
3. Pesquisa binária
4. Pesquisa digital
5. Árvores de busca
    1. Árvores binárias de pesquisa sem balanceamento
    2. Árvores binárias de pesquisa com balanceamento
6. Tabelas de dispersão (*hash tables*)
    1. Funções de transformação de chave (*hashing*)
    2. Cálculo de endereços e tratamento de colisões
    3. Endereçamento aberto
    4. Listas encadeadas
    5. Hashing perfeito
7. Pesquisa de dados em memória secundária
    1. Acesso sequencial indexado
    2. Árvores de pesquisa
    3. Árvores-B
    4. Árvores-B*

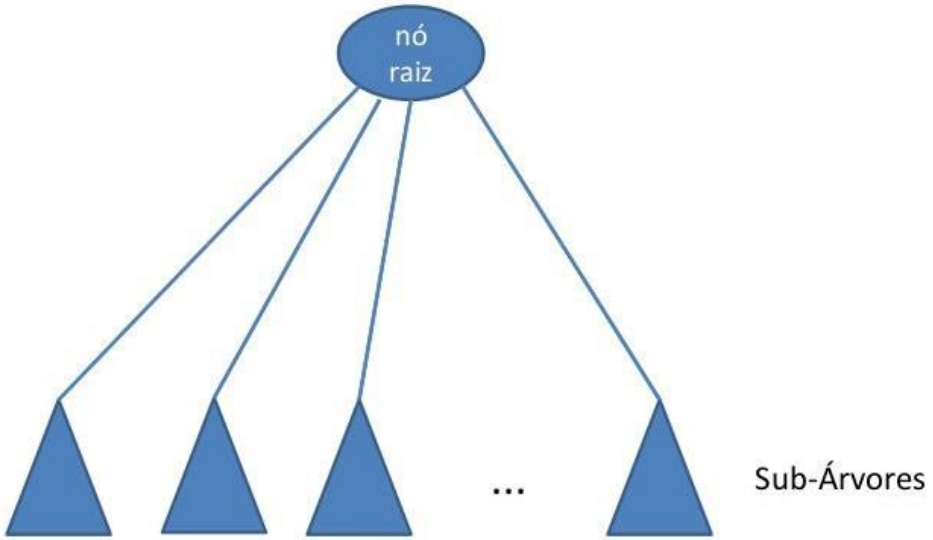
3. Introdução à Análise da Complexidade de Algoritmos

Agenda

- Relembrando árvores
- Pesquisa digital
- Árvores Binárias de Busca
- *Hands-on*

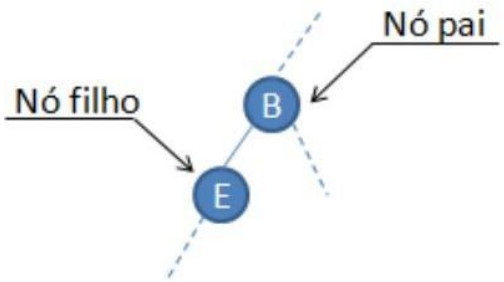
Árvores

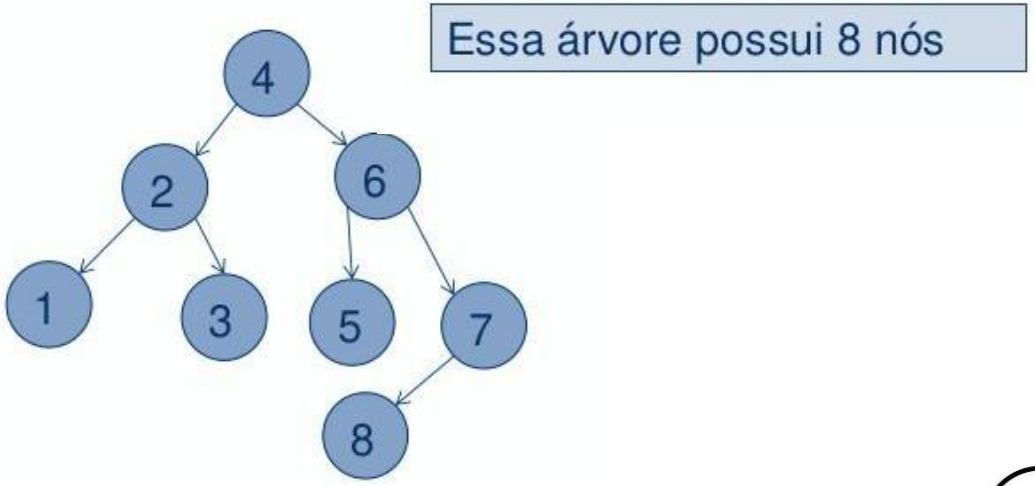
- Árvores são EDs muito utilizadas na computação:
  - Estrutura de um diretório de arquivos;
  - Árvores de parsing em LPs;
  - Documentos HTML, XML, ...
- Representação gráfica:



Árvores

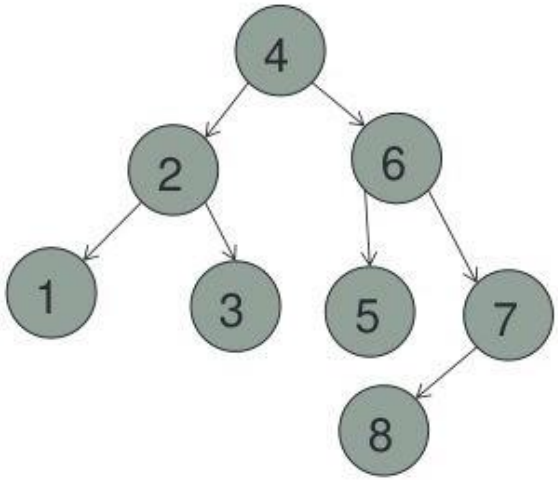
- São compostas por um conjunto de **nós** (ou **nodos**):
  - Existe um nó *r* chamado de nó **raiz** que contém zero ou mais **subárvores**, cujas raízes são ligadas a *r*;
  - Os nós raízes destas sub-árvores são ditos nós **filhos** de *r*;
  - Nós com filhos são chamados de **nós internos**;
  - Nós sem filhos são chamados de **nós folhas** (ou **nós externos**).



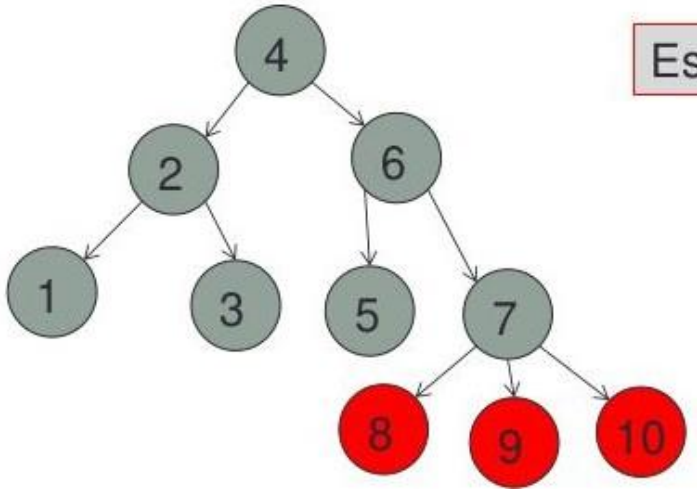


Árvores

- Binárias (um nó pode ter no máximo 2 filhos);
- *n*-árias.



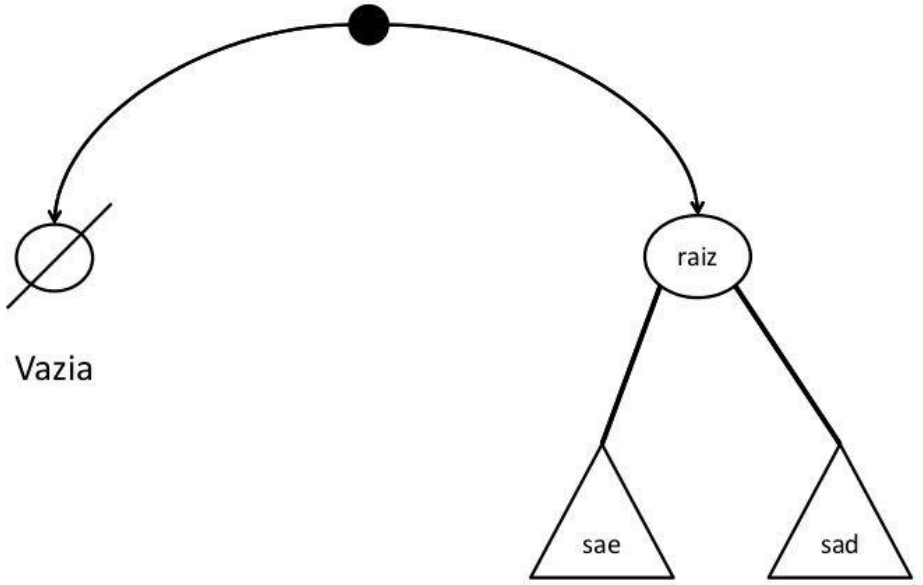
Essa árvore é binária

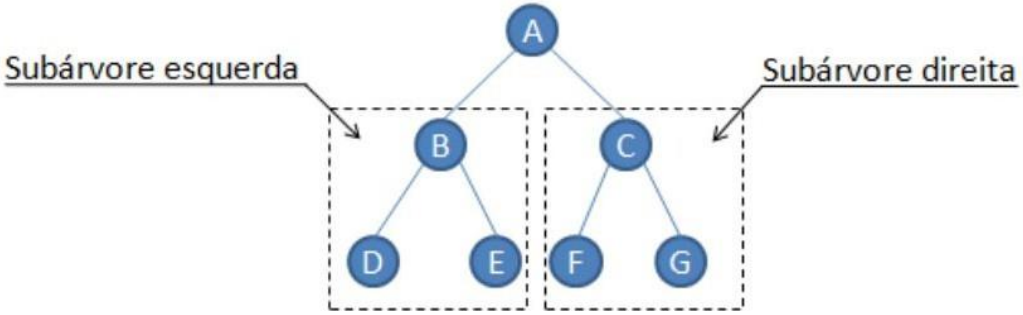


Essa árvore não é binária

Árvores

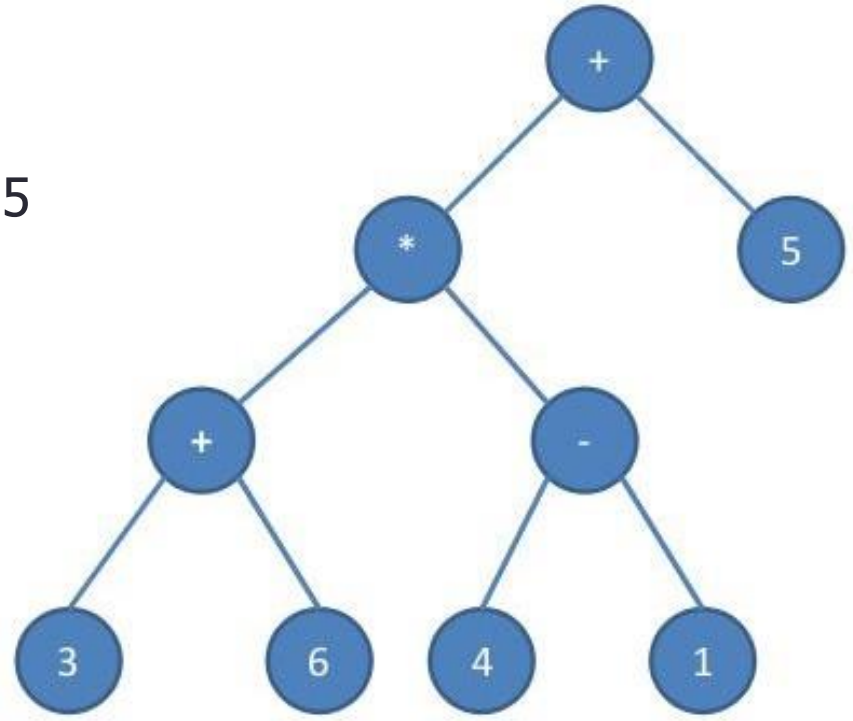
- Em uma árvore binária, cada nó tem 0, 1 ou 2 filhos;
- Uma árvore binária é:
  - uma árvore vazia; ou
  - um nó raiz com duas sub-árvores:
    - a sub-árvore da direita (sad)
    - a sub-árvore da esquerda (sae).





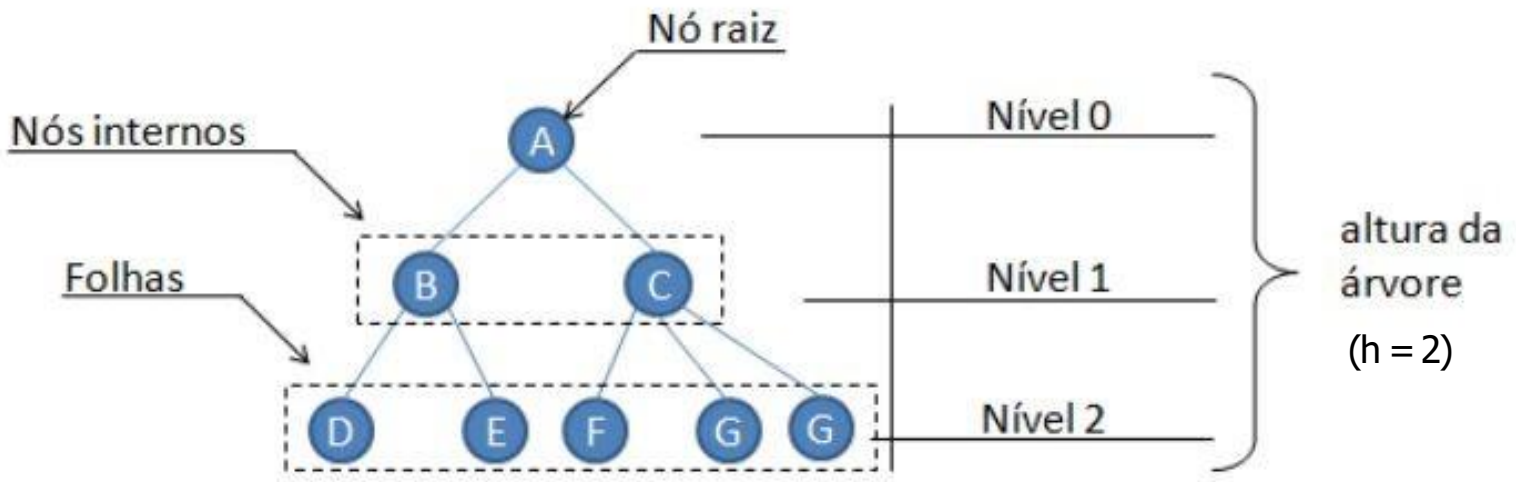
Árvores

- Exemplo:
  - Árvores binárias representando expressões aritméticas:
    - nós folhas representam operandos;
    - nós internos operadores.
  - Expressão: (3+6) *(4-1) + 5

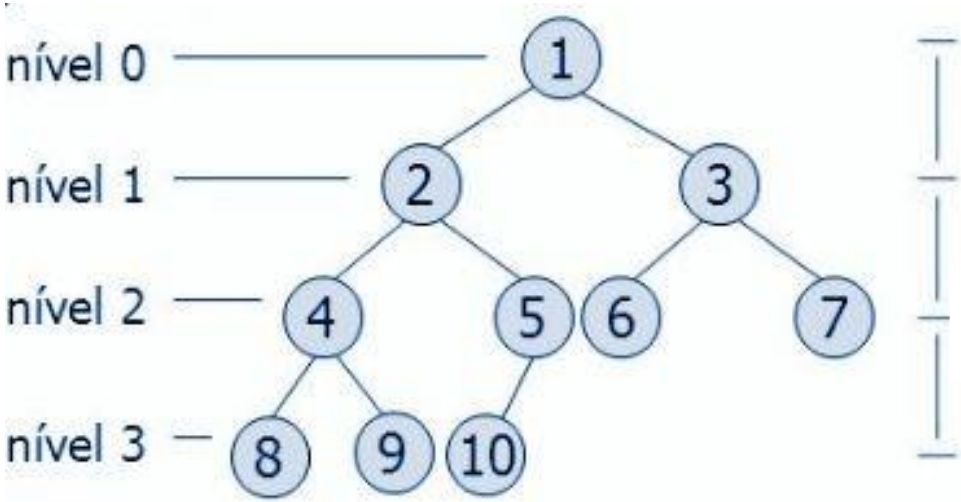


Árvores

- Uma das propriedades do nó de uma árvore é seu **nível** (profundidade):
  - O nível do nó raiz é 0.
- Outra das propriedades do nó é a sua **altura**:
  - O comprimento do caminho mais longo deste nó até um nó folha.
  - A altura de uma árvore (*h*) é a altura do nó raiz.

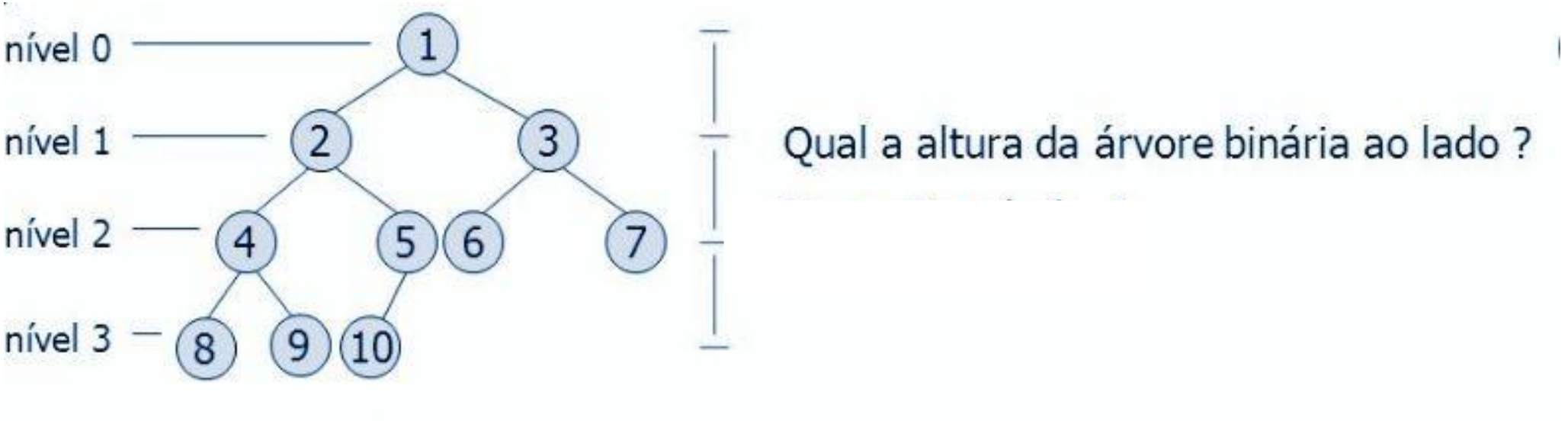


Árvores



Qual a altura da árvore binária ao lado ?

Árvores



Uma árvore binária de altura **h** tem no máximo $\mathbf{2^{h+1} - 1}$ nós

- No exemplo: $2^{3+1} - 1 = 16 - 1 = 15$

Árvores

- Existem diferentes ordens possíveis para percorrer uma árvore (**ordens de percurso**), tais como:

pré-ordem:

trata *raiz*, percorre *sae*, percorre *sad*

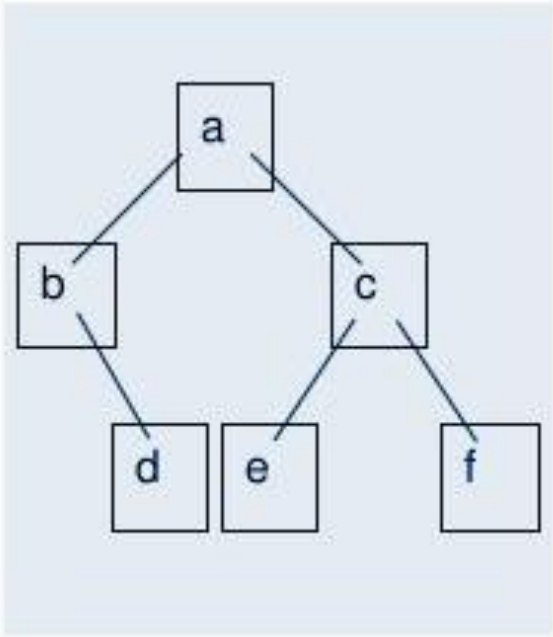
exemplo: a b d c e f

ordem simétrica:

- percorre *sae*, trata *raiz*, percorre *sad*
- exemplo: b d a e c f

pós-ordem:

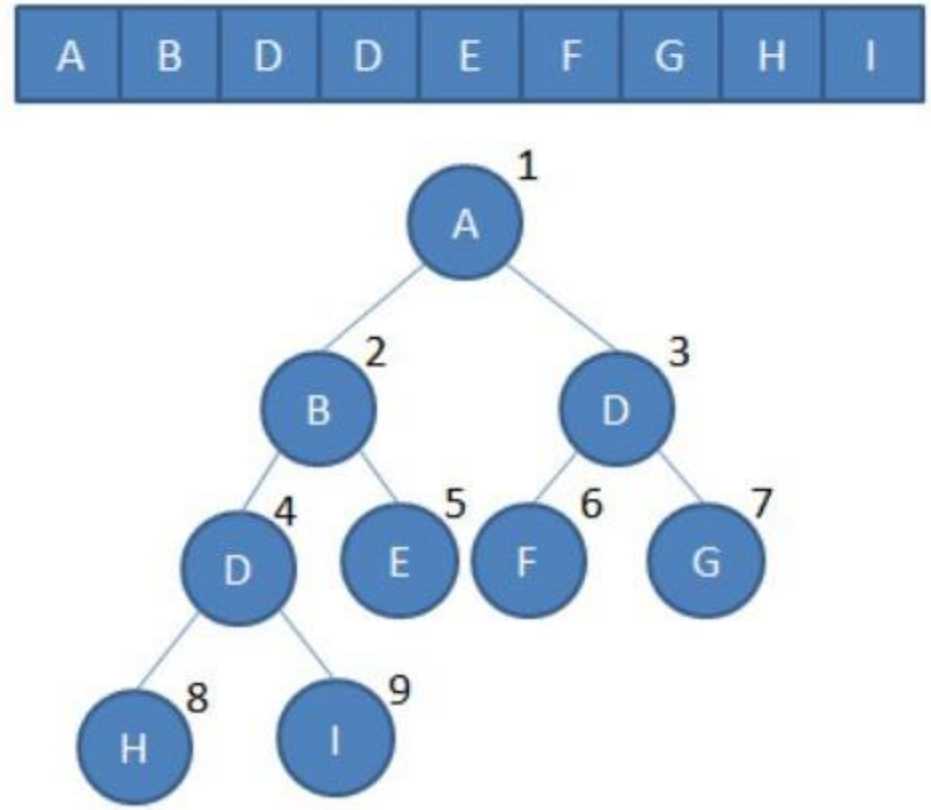
- percorre *sae*, percorre *sad*, trata *raiz*
- exemplo: d b e f c a



Pre-ordem	Em-ordem	Pos-ordem
1 - Processa o nó 2 - Caminha p/ esquerda 3 - Caminha p/ direita	1 - Caminha p/ esquerda 2 - Processa o nó 3 - Caminha p/ direita	1 - Caminha p/ esquerda 2 - Caminha p/ direita 3 - Processa o nó

Árvores

- Conforme veremos, uma árvore binária é uma ED comumente utilizada por **algoritmos de busca**;
- Podemos representar como um vetor (problemas com árvores não completas ou dinâmicas) ou com uma estrutura de nós (nodos) ligados.



Árvores em Python (exemplo)

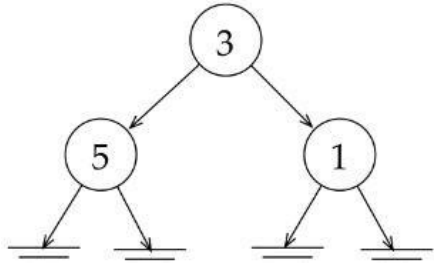
- Os nodos de uma árvore binária possuem um valor (chamado de chave) e dois apontadores/ponteiros, um para o filho da esquerda e outro para o filho da direita.

```
1   class Nodo:
2       def__init__(self, chave=None, esquerda=None, direita=None):
3           self.chave = chave
4           self.esquerda = esquerda
5           self.direita = direita

6       def__repr__(self):
7           return '%s <- %s -> %s' % (self.esquerda and self.esquerda.chave,
                                       self.chave,
                                        self.direita and self.direita.chave)
```

```
raiz = Nodo(3)
raiz.esquerda = Nodo(5)
raiz.direita  = Nodo(1)
print("Árvore: ", raiz)
```

```
< Árvore:  5 <- 3 -> 1
```



Árvores em Python (exemplo)

- Com base nos nodos podemos criar uma estrutura de árvore:

```
1   class ArvoreBinaria:
2       def__init__(self):
3           self.raiz = Nodo(None, None, None)
4           self.raiz = None
...
5    def _imprime_pre_ordem(self, no_atual):
6        if no_atual is not None:
7            print(str(no_atual.chave))
8            self._imprime_pre_ordem(no_atual.esquerda)
9            self._imprime_pre_ordem(no_atual.direita)
```

Veremos como implementar métodos para **inserir** e **buscar** elementos na sequência

Agenda

- Relembrando árvores
- Pesquisa digital
- Árvores Binárias de Busca
- *Hands-on*

Pesquisa digital

- É baseada na **representação das chaves** como uma sequência de **caracteres ou de dígitos**:
  - Representação usada para estruturar os dados na memória;
  - Por exemplo, a representação de um número em binário.
- Os métodos de pesquisa digital são particularmente vantajosos quando as chaves são grandes e de tamanho variável;
- **Pesquisa não é baseada** em **comparações** de **chaves**, **mas sim em processamento feito sob a chave**.

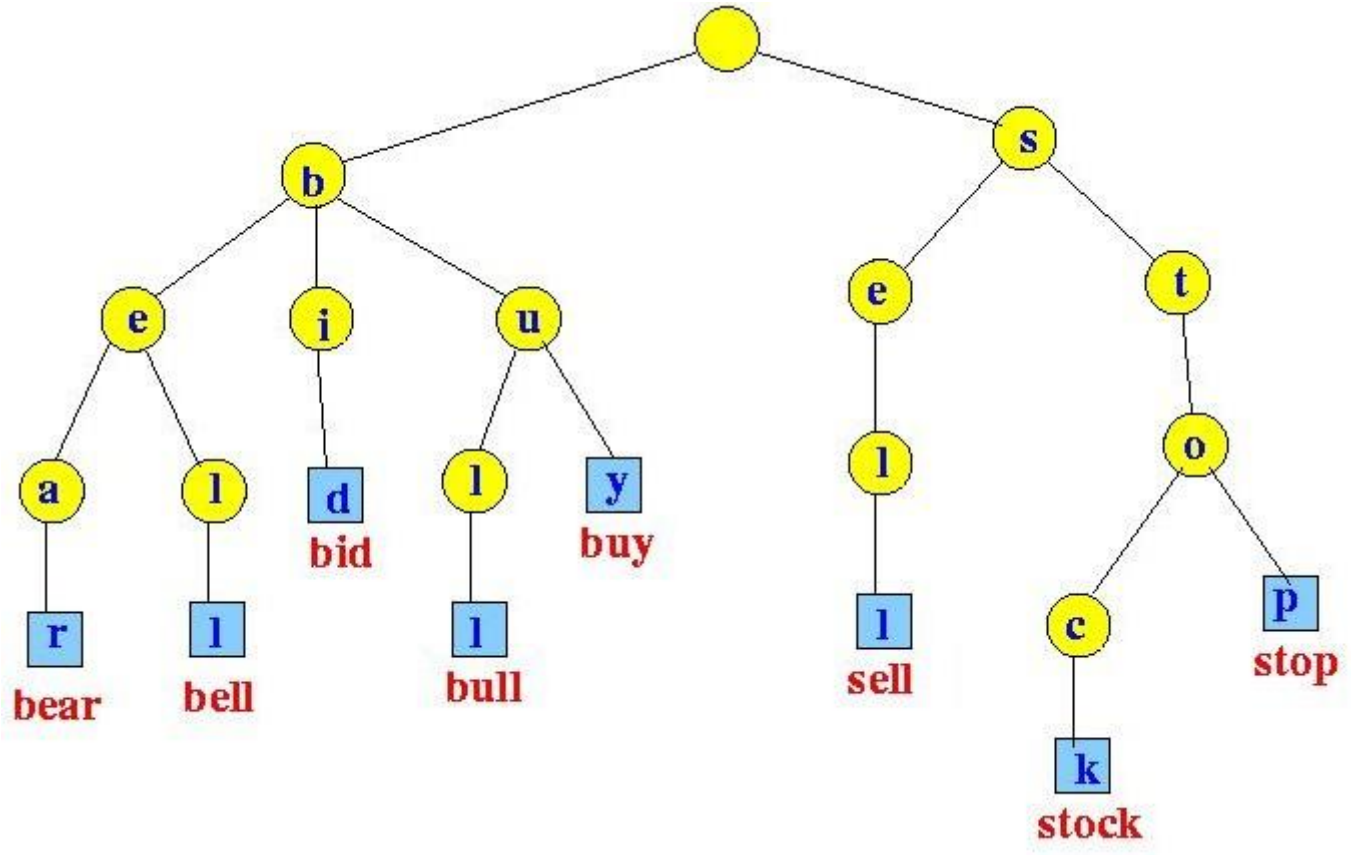
Pesquisa digital

- Estruturas geralmente utilizadas para a pesquisa digital são as árvores digitais, como por exemplo:
  - **Trie;**
  - **Patricia.**
- Veremos cada uma delas na sequência.

Trie

- Árvore M-ária **cujos nós são vetores de M componentes com campos correspondentes aos dígitos ou caracteres que formam as chaves:**
  - **Também conhecida como *digital tree* ou *prefix tree*.**
- Cada nó no nível *i* representa o conjunto de todas as chaves que começam com a mesma sequência de *i* dígitos ou caracteres:
  - Cada filho corresponde a um possível valor do i-ésimo "bit";
  - Por exemplo, cada nó numa trie M-ária para armazenar strings pode ter 256 filhos (um para cada valor possível de um caractere).

Trie


b
s
e
i
u
e
t
a
l
d
l
y
l
o
r
l
l
l
c
p
bear
bell
bid
bull
buy
sell
stop
k
stock

Trie (binária)

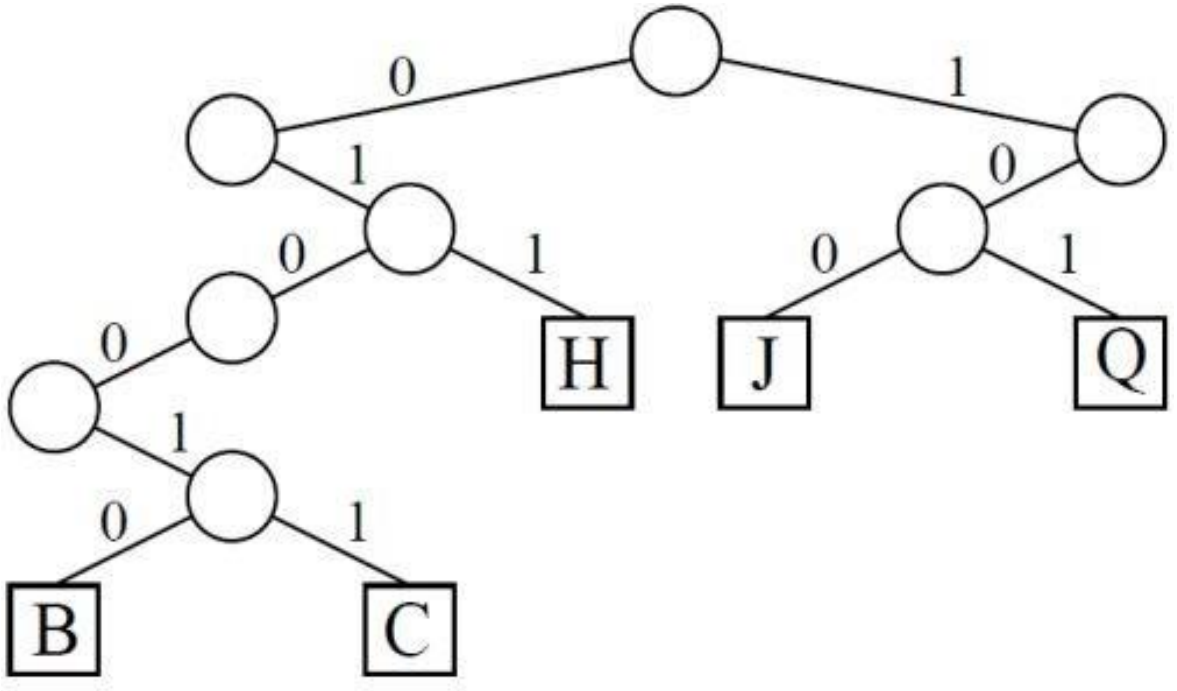
- Considerando as chaves como sequência de bits (M = 2), o algoritmo de pesquisa digital é semelhante ao de pesquisa em árvore binária:
  - Porém não percorremos a árvore de acordo com o resultado de comparação entre chaves;
  - Percorremos a *trie* de acordo com os bits da chave.

Trie

- Dada as chaves de seis bits, como podemos representá-las em uma trie?
  - B = 010010
  - C = 010011
  - H = 011000
  - J = 100000
  - Q = 101000

Trie

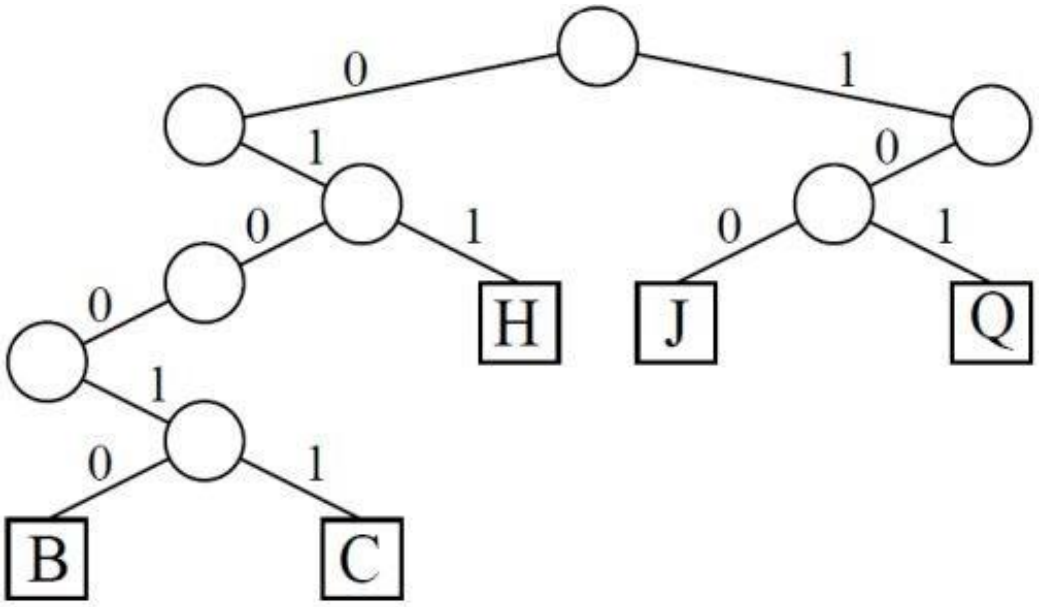
- Dada as chaves de seis bits, como podemos representá-las em uma trie?
  - B = 010010
  - C = 010011
  - H = 011000
  - J = 100000
  - Q = 101000



Trie – Inserção

- Fazemos uma pesquisa na árvore para descobrir onde a chave deverá ser inserida;
  - **Caso 1:** se o nó externo onde a pesquisa terminar tiver uma chave, criamos nós internos até encontrar o bit onde a nova chave difere da chave já existente.

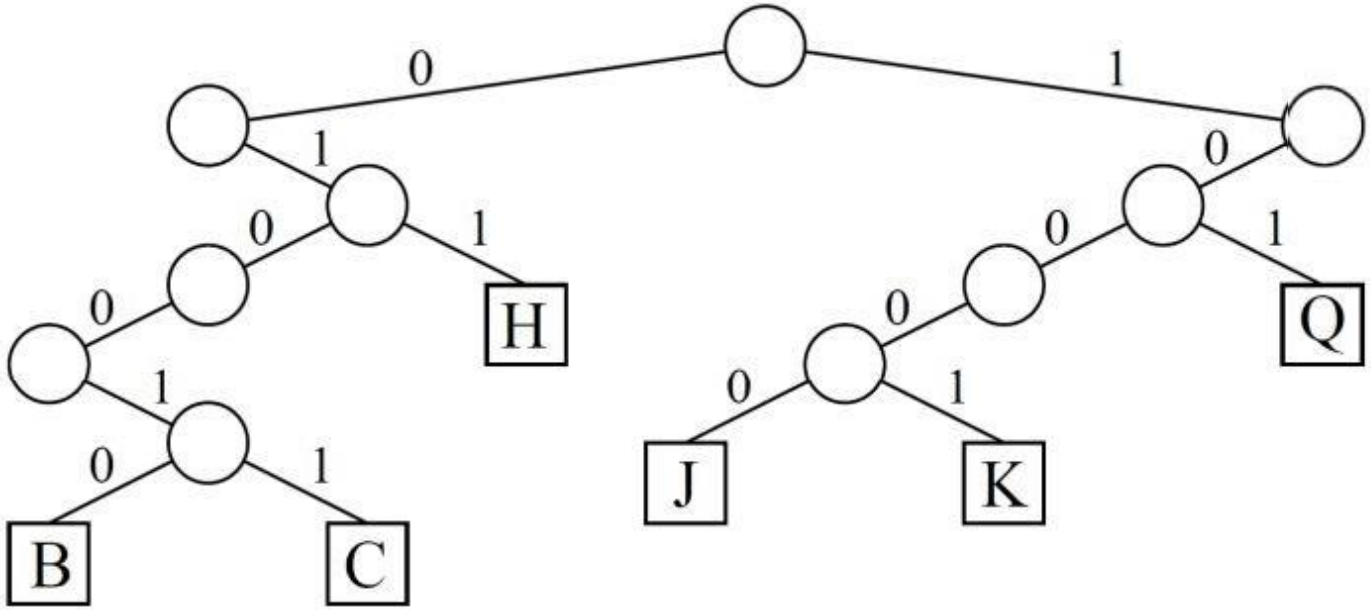
    Inserindo K = 100010



Trie – Inserção

- Fazemos uma pesquisa na árvore para descobrir onde a chave deverá ser inserida;
  - **Caso 1:** se o nó externo onde a pesquisa terminar tiver uma chave, criamos nós internos até encontrar o bit onde a nova chave difere da chave já existente.

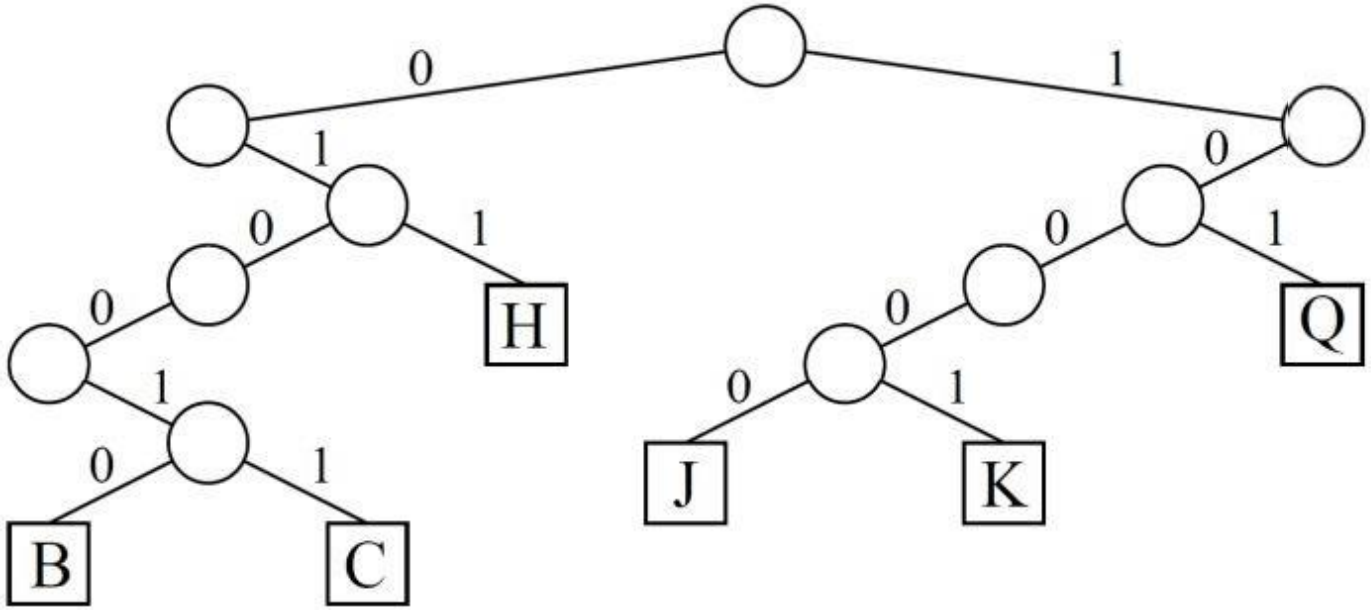
    Inserindo K = 100010



Trie – Inserção

- Fazemos uma pesquisa na árvore para descobrir onde a chave deverá ser inserida;
  - **Caso 2:** se o nó externo onde a pesquisa terminar for vazio, basta criar um novo nó para conter a nova chave.

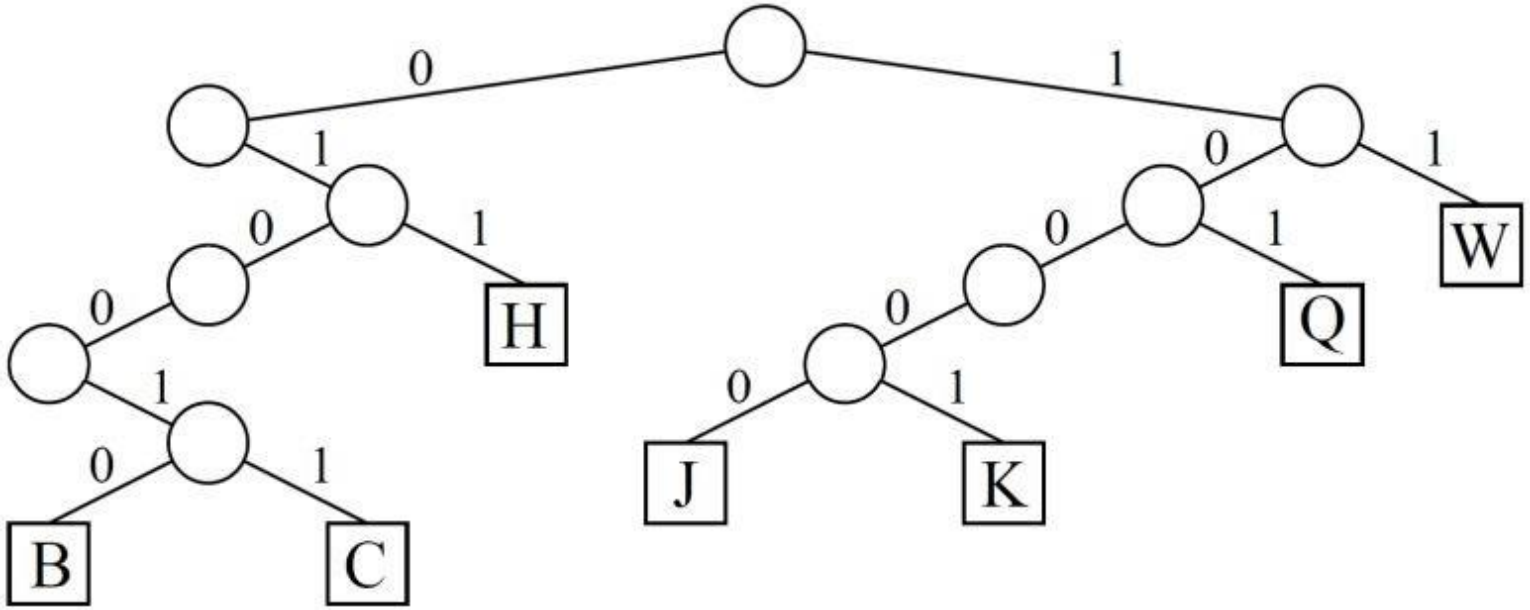
    Inserindo W = 110000



Trie – Inserção

- Fazemos uma pesquisa na árvore para descobrir onde a chave deverá ser inserida;
  - **Caso 2:** se o nó externo onde a pesquisa terminar for vazio, basta criar um novo nó para conter a nova chave.

    Inserindo W = 110000



Trie

Vantagens:

- O formato das *tries*, diferentemente das árvores binárias comuns, **não depende da ordem** em que as chaves são inseridas:
  - Depende da **estrutura das chaves** (distribuição de seus bits);
  - "Balanceamento natural".
- Inserção e busca numa *trie* com *n* chaves aleatórias requer aproximadamente log(n) comparações de bits no caso médio:
  - Não depende do tamanho da chave;
  - O pior caso é limitado pelo número de bits das chaves.

Trie

Desvantagens:

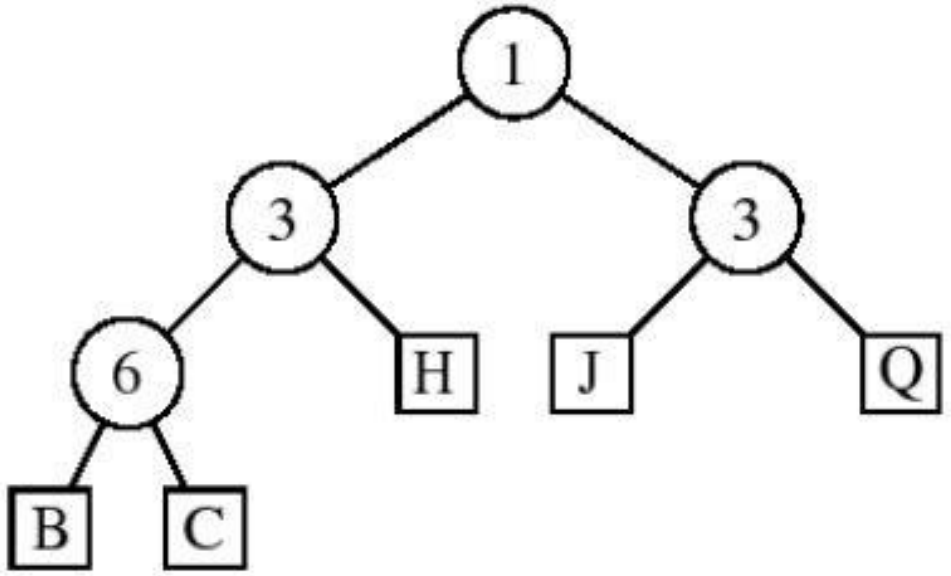
- **Caminhos de uma única direção** acontecem quando chaves compartilham vários bits em comum:
  - Por exemplo, as chaves B (00010) e C (00011) são idênticas exceto no último bit;
  - Requer inspeção de todos os bits da chave independente do número de registros na *trie*.
- Os registros são armazenados **apenas nas folhas**, o que **desperdiça memória** em nós intermediários

Patricia

- *Practical Algorithm To Retrieve Information Coded in Alphanumeric:*
  - Criada para recuperação de informação em arquivos de texto.
- Diferentes variações do algoritmo foram propostas ao longo do tempo;
- Usa o conceito de pesquisa digital, mas estrutura os dados de forma a **evitar as desvantagens** inerentes das *tries:*
  - Os problemas dos caminhos de única direção e desperdício de memória em nós internos é eliminado de uma forma elegante.

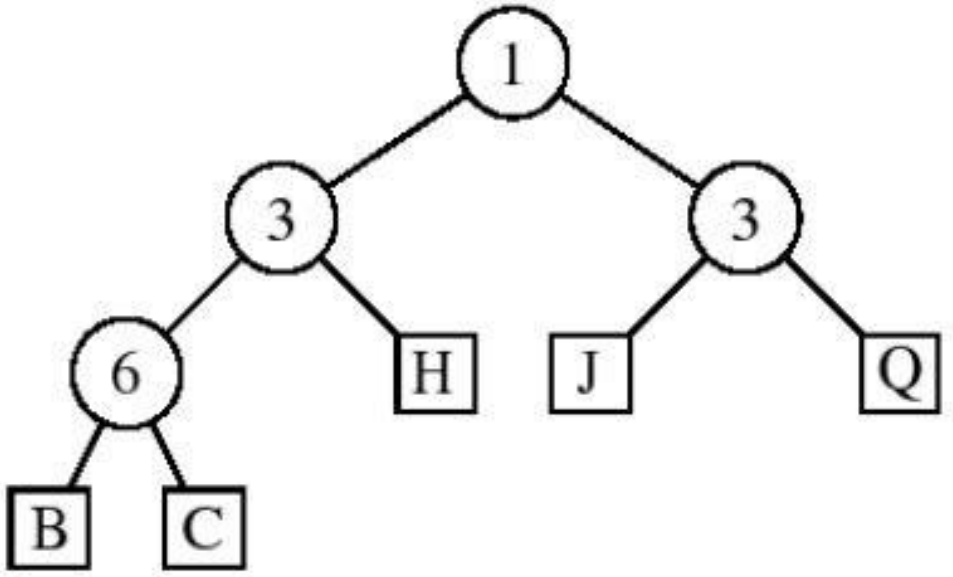
Patricia

- Cada nó da árvore contém uma chave e um índice indicando qual bit deve ser testado para decidir qual ramo seguir;
- Por exemplo, dadas as chaves de 6 bits:
  - B = 010010
  - C = 010011
  - H = 011000
  - J = 100001
  - Q = 101000



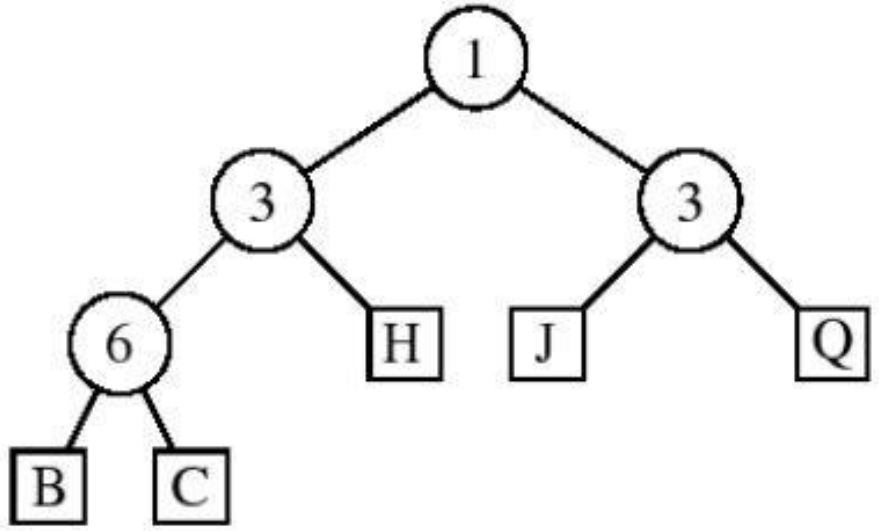
Patricia – inserção

- **Caso 1 –** O novo nó substitui um nó externo (folha):
  - Acontece quando o bit que diferencia a nova chave da chave encontrada não foi utilizado na busca.
- **Caso 2 –** O novo nó substitui um nó interno:
  - Acontece quando o bit que diferencia a nova chave da chave encontrada foi pulado durante a busca.



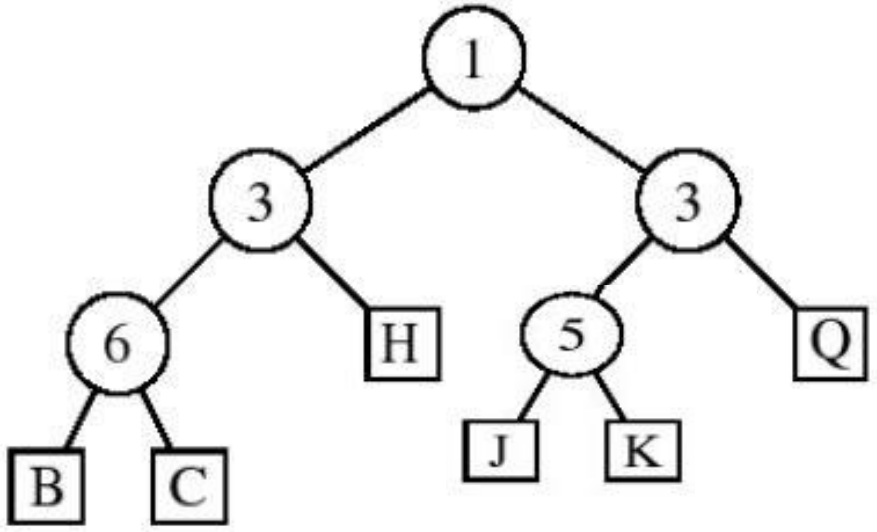
Patricia – Inserção

- **Caso 1:** Inserção de K = 100010;
- Os índices dos bits nas chaves estão ordenados esq → dir (início pela raiz):
  - Bit de índice 1 de K é 1 → subárvore direita;
  - Bit de índice 3 é 0→ subárvore esquerda que neste caso é um nó externo.
- Chaves J (100001) e K mantêm o padrão de bits 1x0xxx, assim como qualquer outra chave que seguir este caminho de pesquisa:
  - A diferença está no índice 5.



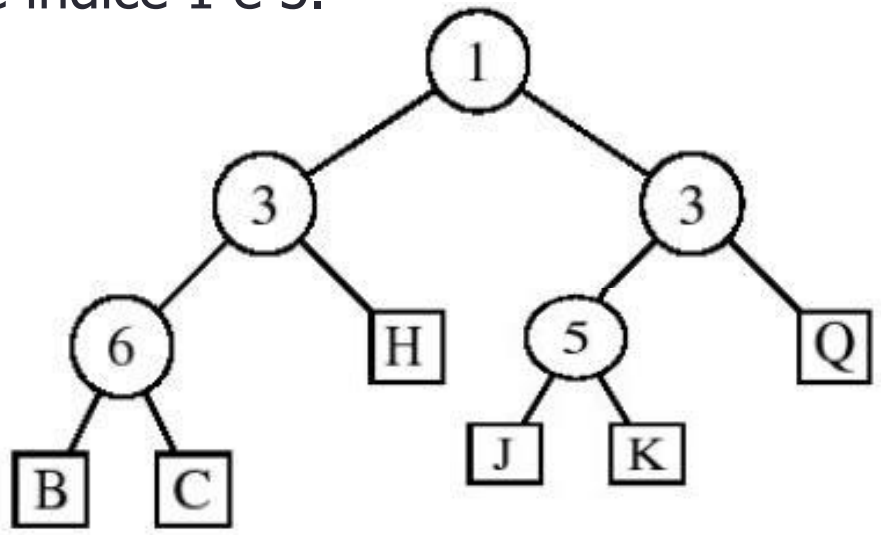
Patricia – Inserção

- **Caso 1:** Inserção de K = 100010;
- Os índices dos bits nas chaves estão ordenados esq → dir (início pela raiz):
  - Bit de índice 1 de K é 1 → subárvore direita;
  - Bit de índice 3 é 0→ subárvore esquerda que neste caso é um nó externo.
- Chaves J (100001) e K mantêm o padrão de bits 1x0xxx, assim como qualquer outra chave que seguir este caminho de pesquisa:
  - A diferença está no índice 5.



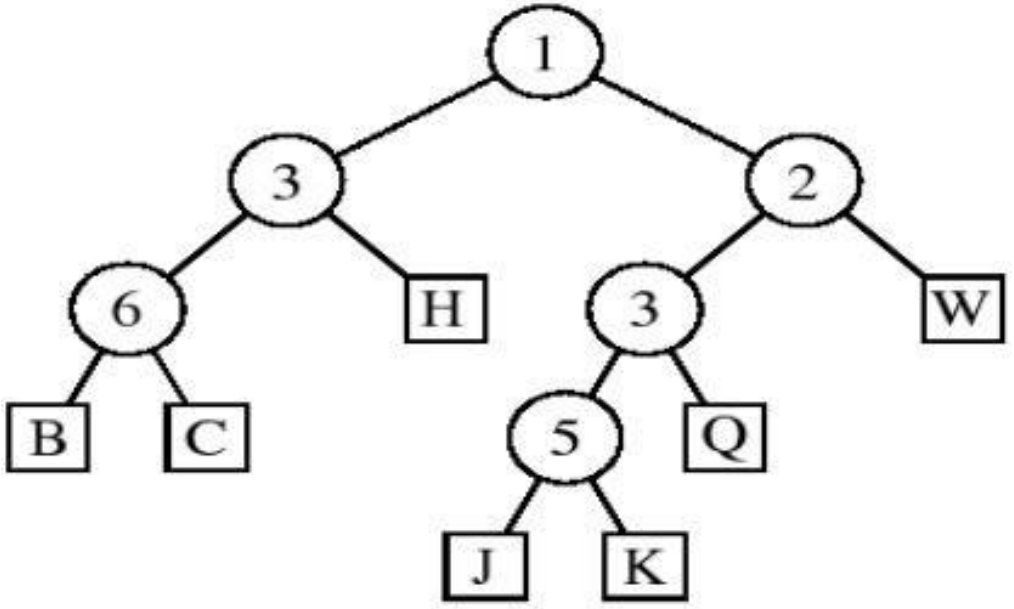
Patricia – Inserção

- **Caso 2:** inserção de W = 110110:
  - Bit de índice 1 de W é 1 → a subárvore direita;
  - Bit de índice 3 é 0→ subárvore esquerda;
  - Bit de índice 5 é 1 → subárvore direita → nó folha (K = 100010).
- Os bits das chaves K e W são comparados a partir do 1º bit para determinar em qual índice eles diferem, sendo, neste caso, o de índice 2;
- Portanto, o ponto de inserção agora será no caminho de pesquisa entre os nós internos de índice 1 e 3.



Patricia – Inserção

- **Caso 2:** inserção de W = 110110;
- O ponto de inserção será no caminho entre os nós internos de índice 1 e 3:
  - Cria-se um novo nó interno de índice 2, cujo descendente direito é um nó externo contendo W e cujo descendente esquerdo é a subárvore de raiz de índice 3.



Patricia

Considerações:

- Inserção em uma árvore patricia com **n** chaves aleatórias requer aproximadamente log(n) comparações de bits no caso médio:
  - 2 *log(n) comparações no pior caso.
- O número de comparações de bits é limitado pelo tamanho das chaves;
- Não existe desperdício de memória nos nós internos;

Agenda

- Relembrando árvores
- Pesquisa digital
- Árvores Binárias de Busca
- *Hands-on*

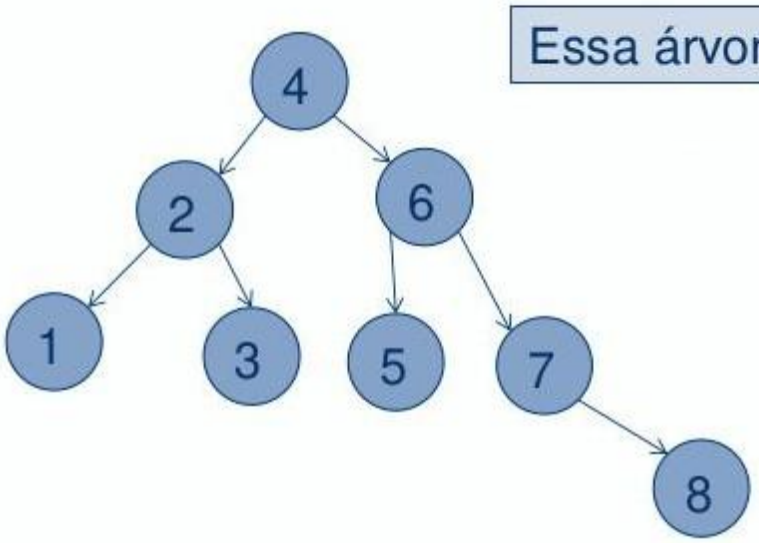
Árvores Binárias de Busca

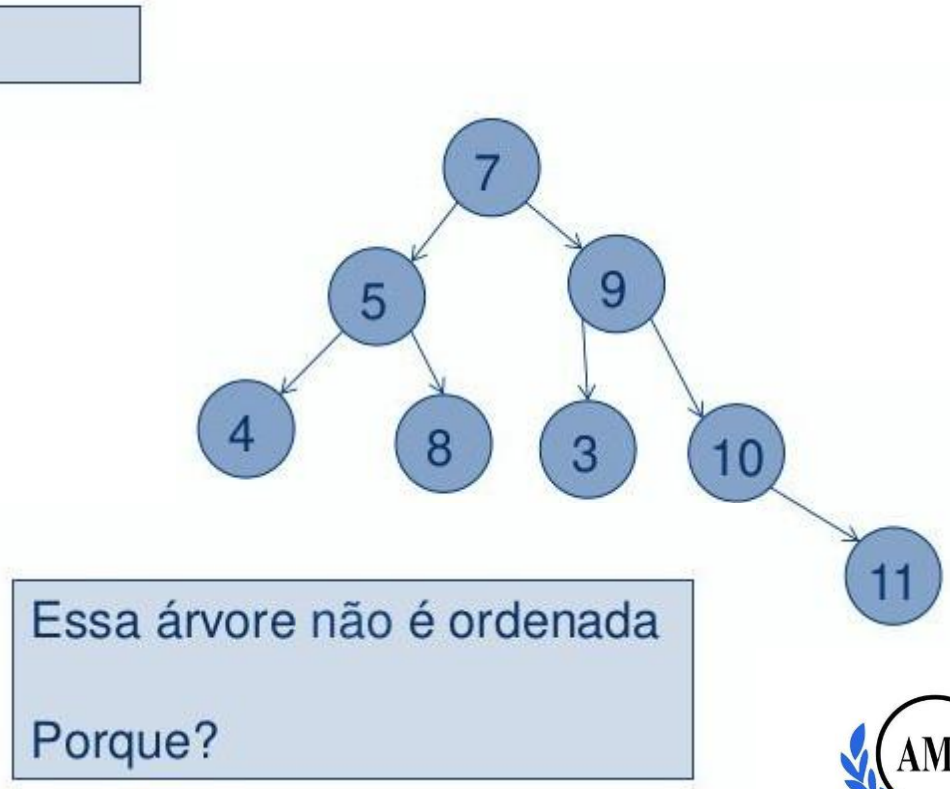
- **Busca binária em vetor**:
  - dados armazenados em vetor, de forma ordenada;
  - bom desempenho computacional para pesquisa;
  - **inadequado quando inserções e remoções são frequentes**:
    - exige re-arrumar o vetor para abrir espaço uma inserção;
    - exige re-arrumar o vetor após uma remoção.

- **Árvore binária de busca – ABB** (ou árvore binária ordenada) são utilizadas para otimizar buscas em coleções de dados:
  - ***Binary search tree*** **– BST**;
  - Podem ser utilizadas, por exemplo, no contexto de índices em bancos de dados.

Árvores Binárias de Busca

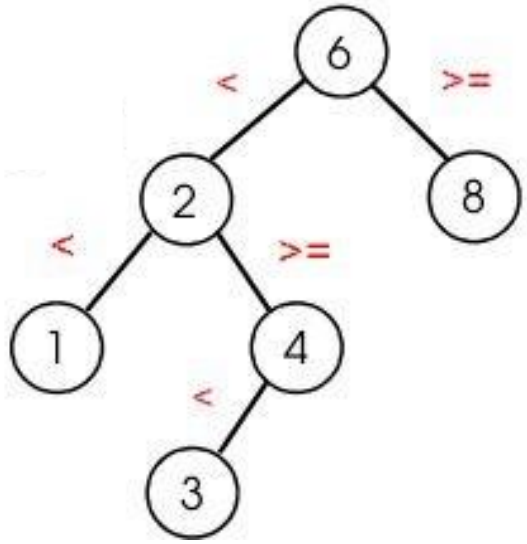
- Em uma árvore binária ordenada, todos os nós satisfazem as seguintes regras:
  - Todos nós à sua esquerda tem valor menor ao nó pai;
  - Todos nós a sua direita tem valor maior ou igual ao nó pai.





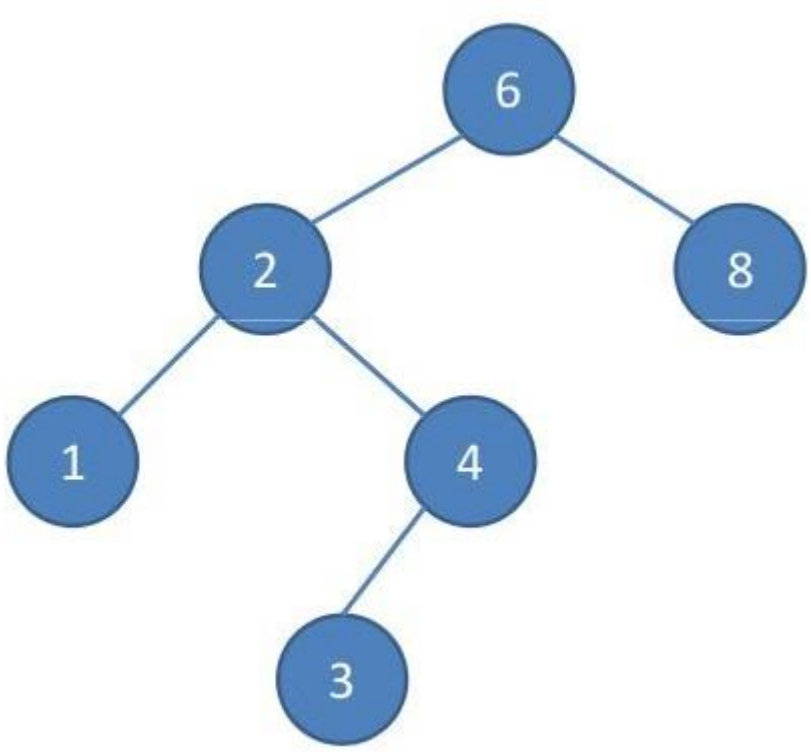
Árvores Binárias de Busca

- Sendo assim, ABBs possuem as seguintes propriedades:
  - O valor associado à raiz é sempre **maior** que o valor associado a qualquer nó da sub-árvore à **esquerda** (**sae**); e
  - O valor associado à raiz é sempre **menor ou igual** (para permitir valores repetidos) que o valor associado a qualquer nó da sub-árvore à **direita** (**sad**).



Árvores Binárias de Busca

- Tais propriedades garantem que, quando a árvore é percorrida em **ordem simétrica** (sae - raiz - sad), os valores são encontrados em ordem crescente.



Ordem simétrica: 1 2 3 4 6 8

Árvores Binárias de Busca

- **Vantagem ao pesquisar** em ABB:
  - Na média, a **busca por uma informação** precisa **percorrer por menos nós**;
  - Isso ocorre porque em **uma árvore ordenada apenas um dos ramos da árvore precisa ser pesquisado**:
    - Já em árvores sem ordenação a pesquisa pode acabar sendo propagada para os dois ramos.

Árvores Binárias de Busca

- **As operações nas ABBs são equivalentes** às operações **das outras árvores binárias**, **exceto** pelo **fato** de que **precisam manter as propriedades da ABB**:
  - Para isto, **árvore deve ser mantida ordenada**, acarretando mudanças tanto na inserção quanto na remoção de nós.
- Veremos as seguintes operações:
  - Inserção;
  - Remoção;
  - Busca.

Inserção em ABBs

- A operação adiciona um nó no local correto para preservar a ordem dos valores na ABB:
  - **Se a árvore é vazia**, **deve ser substituída por uma árvore cujo único nó** tem o **valor a ser inserido**;
  - **Se não for vazia**, comparamos o valor a ser inserido com o valor na raiz, inserindo-o na SAE ou SAD, de acordo com a comparação.
  - A função retorna o (eventual novo) nó raiz da árvore.

Inserção em ABBs

- A operação adiciona um nó no local correto para preservar a ordem dos valores na ABB:
  - Se a árvore é vazia, deve ser substituída por uma árvore cujo único nó tem o valor a ser inserido;
  - Se não for vazia, comparamos o valor a ser inserido com o valor na raiz, inserindo-o na SAE ou SAD, de acordo com a comparação.
  - A função retorna o (eventual novo) nó raiz da árvore.

```
1   def insert(root, key):
2       if root is None:
3           return Node(key)
4       else:
5           if key < root.val:
6               root.left = insert(root.left, key)
7           else:
8               root.right = insert(root.right, key)
9        return root
```

Remoção em ABBs

- Esta operação é mais complexa:
  - Se a árvore for vazia → nada tem de ser feito;
  - **Se não for,** comparamos o valor da raiz com o valor a ser removido:
    - Se o valor na raiz for maior que o valor a ser retirado, o valor a ser excluído está na SAE;
    - Se o valor na raiz for menor que o valor a ser retirado, o valor a ser excluído está na SAD;
    - Se o valor for igual, remove-se o nó raiz.
  - Existem três casos possíveis para o nó a ser removido:
    - **O nó a ser retirado é folha;**
    - **O nó a ser retirado possui um único filho;**
    - O nó a ser retirado possui dois filhos (caso mais complexo).

Remoção em ABBs

- Esboço da implementação:

```
def remove(root, key):
    if root is None:
        return root

    if key < root.val:    # Procura o valor na árvore da esquerda (sae)
        root.left = remove(root.left, key)
    elif key > root.val:  # Procura o valor na árvore da direita (sad)
        root.right = remove(root.right, key)
    else:
        # Tratar caso 1: nó sem filhos
        # Tratar caso 2: nó com apenas um filho (esquerda ou direita)
        # Tratar caso 3: nó com os dois filhos

    return root
```

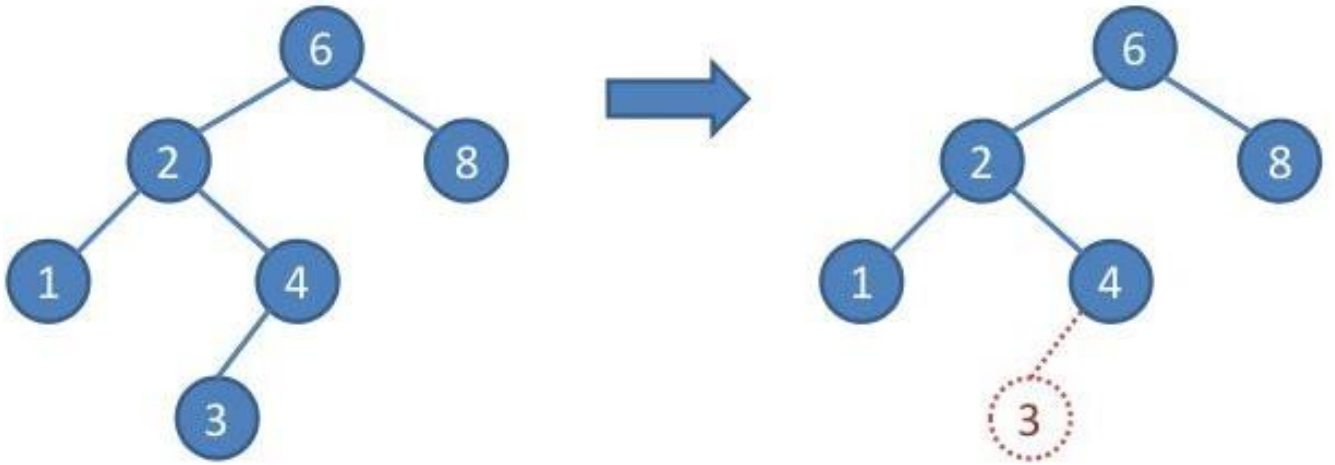
Achou o valor.. agora precisa removê-lo

Remoção em ABBs

- **Caso 1:** Nó sem filhos (folha)
  - Neste caso, basta "liberar" a memória alocada (em Python será feito automaticamente pelo *garbage collector*) pelo nó e retorna a raiz atualizada, que passa a ser *null*.

Exemplo: remover(a, 3);

```
if r.left is None and r.right is None:
    r = None
```

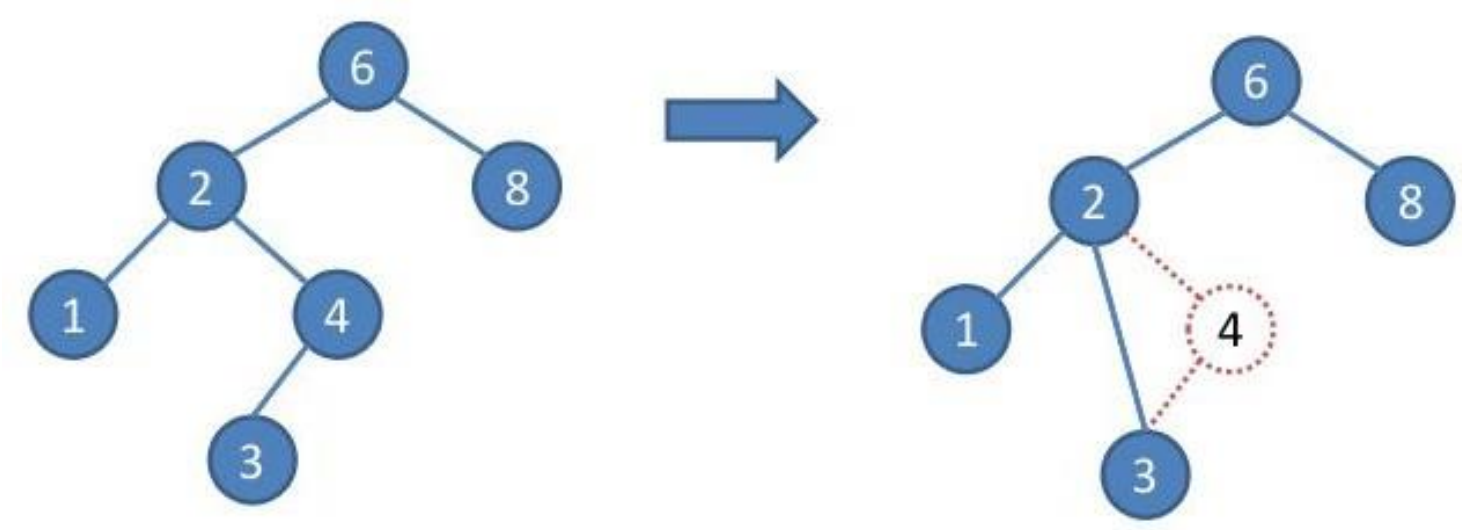


Remoção em ABBs

- **Caso 2:** Somente um filho
  - Ao retirarmos o nó raiz de uma subárvore, que possui um filho apenas, tal filho passa a ser a raiz da subárvore.

Exemplo: remover(a, 4);

```
elif r.left is None: # somente filho dir.
    t = r
    r = r.right
    t = None
elif r.right is None: # somente filho esq.
    t = r
    r = r.left
    t = None
```

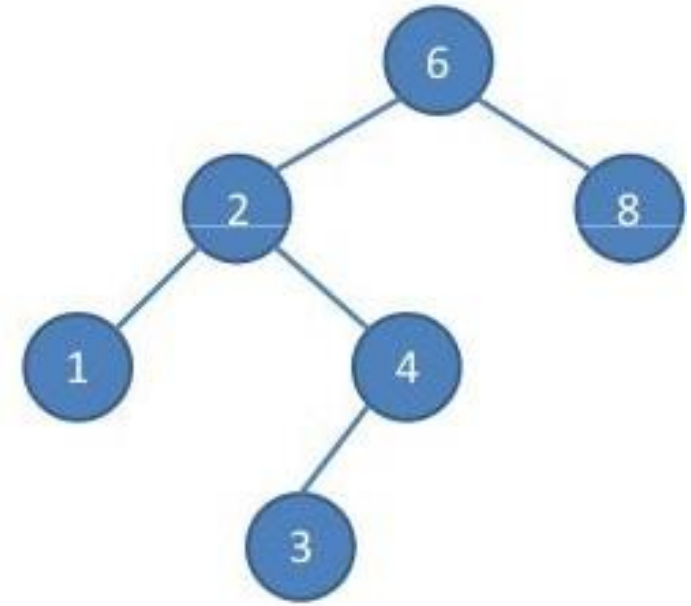


Remoção em ABBs

- **Caso 3:** Tem os dois filhos
  - Procede-se da seguinte forma:
    - Encontrar o elemento a ser excluído;
    - Encontramos na SAE o elemento que precede o elemento a ser removido na ordenação;
    - Trocamos a informação entre os dois nós (o elemento a ser removido e seu precedente);
    - Retiramos da SAE o nó a ser removido.

Exemplo: remover(a, 6)

```
else:
    f = r.left  # vai para a SAE
    # busca o filho mais à direita
    while f.right is not None:
        f = f.right
    r.val = f.val  # troca os valores
    f.val = v
    r.left = remove(r.left, v)  # remove v
```

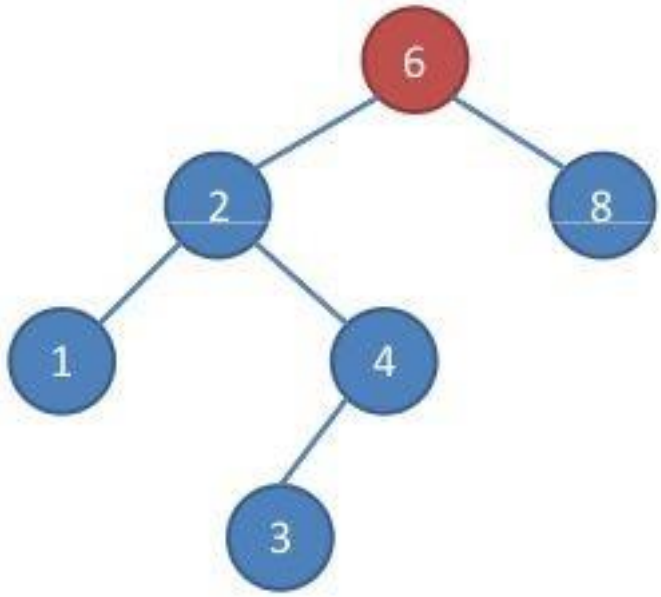


Remoção em ABBs

- **Caso 3:** Tem os dois filhos
  - Procede-se da seguinte forma:
    - Encontrar o elemento a ser excluído → **encontrar o 6**
    - Encontramos na SAE o elemento que precede o elemento a ser removido na ordenação;
    - Trocamos a informação entre os dois nós (o elemento a ser removido e seu precedente);
    - Retiramos da SAE o nó a ser removido.

Exemplo: remover(a, 6)

```
else:
    f = r.left  # vai para a SAE
    # busca o filho mais à direita
    while f.right is not None:
        f = f.right
    r.val = f.val  # troca os valores
    f.val = v
    r.left = remove(r.left, v)  # remove v
```

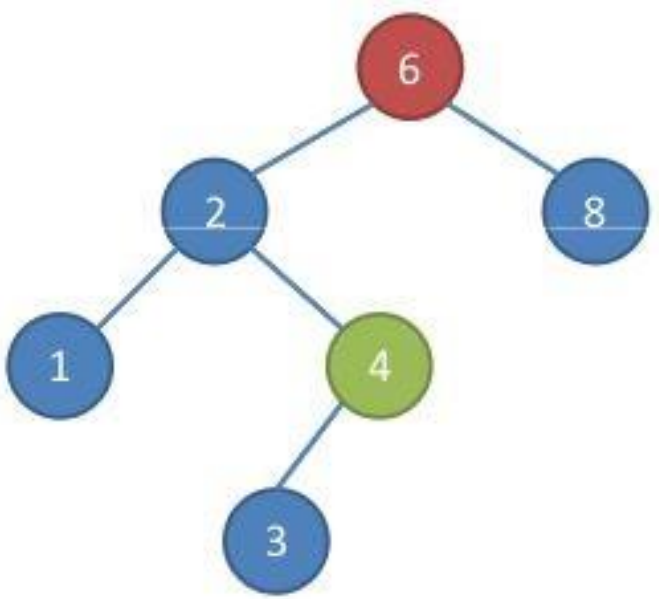


Remoção em ABBs

- **Caso 3:** Tem os dois filhos
  - Procede-se da seguinte forma:
    - Encontrar o elemento a ser excluído → encontrar o 6
    - Encontramos na SAE o elemento que precede o elemento a ser removido na ordenação→ **buscar na SAE o predecessor**
    - Trocamos a informação entre os dois nós (o elemento a ser removido e seu precedente);
    - Retiramos da SAE o nó a ser removido.

Exemplo: remover(a, 6)

```
else:
    f = r.left  # vai para a SAE
    # busca o filho mais à direita
    while f.right is not None:
        f = f.right
    r.val = f.val  # troca os valores
    f.val = v
    r.left = remove(r.left, v)  # remove v
```

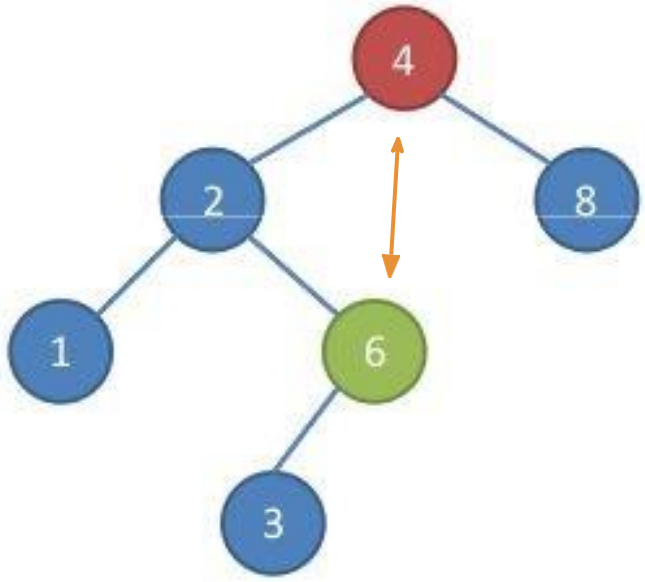


Remoção em ABBs

- **Caso 3:** Tem os dois filhos
  - Procede-se da seguinte forma:
    - Encontrar o elemento a ser excluído → encontrar o 6
    - Encontramos na SAE o elemento que precede o elemento a ser removido na ordenação→ buscar na SAE o predecessor
    - Trocamos a informação entre os dois nós (o elemento a ser removido e seu precedente) → **trocar a informação dos dois nós**
    - Retiramos da SAE o nó a ser removido

Exemplo: remover(a, 6)

```
else:
    f = r.left  # vai para a SAE
    # busca o filho mais à direita
    while f.right is not None:
        f = f.right
    r.val = f.val  # troca os valores
    f.val = v
    r.left = remove(r.left, v)  # remove v
```

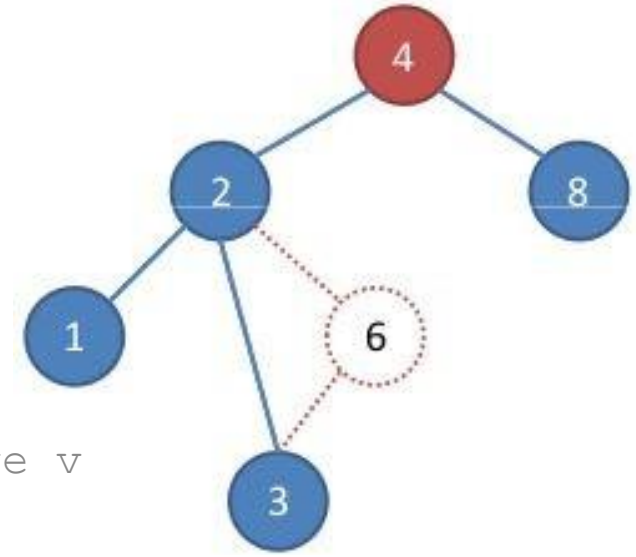


Remoção em ABBs

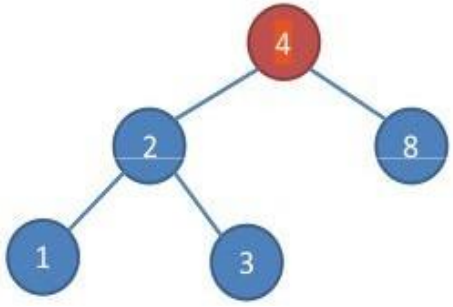
- **Caso 3:** Tem os dois filhos
  - Procede-se da seguinte forma:
    - Encontrar o elemento a ser excluído → encontrar o 6
    - Encontramos na SAE o elemento que precede o elemento a ser removido na ordenação→ buscar na SAE o predecessor
    - Trocamos a informação entre os dois nós (o elemento a ser removido e seu precedente) → trocar a informação dos dois nós
    - Retiramos da SAE o nó a ser removido → **remover o 6**

Exemplo: remover(a, 6)

```
else:
    f = r.left  # vai para a SAE
    # busca o filho mais à direita
    while f.right is not None:
        f = f.right
    r.val = f.val  # troca os valores
    f.val = v
    r.left = remove(r.left, v)  # remove v
```



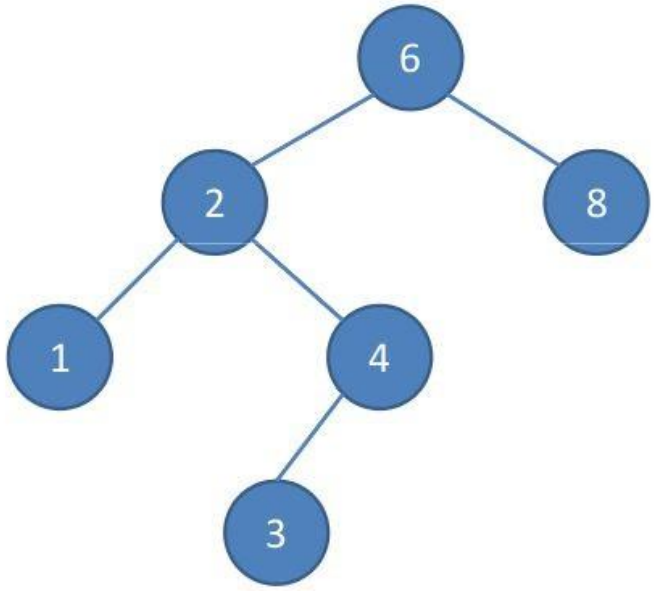
Árvore resultante:



Busca em ABBs

- O algoritmo funciona da seguinte forma:
  - Se a árvore for vazia, retorna-se *null*;
  - Se o valor da raiz da árvore for **<** que o valor procurado, busca-se na SAD;
  - Se o valor da raiz da árvore for **>** que o valor procurado, busca-se na SAE;
  - Se o valor na raiz da árvore for **==** ao valor procurado, retorna-se o nó raiz.

```
def search(root, value):
    if root is None:
        return None
    if value < root.val:
        return search(root.left, value)
    if value > root.val:
        return search(root.right, value)
     return root
```



Exemplo:

search(a, 6)

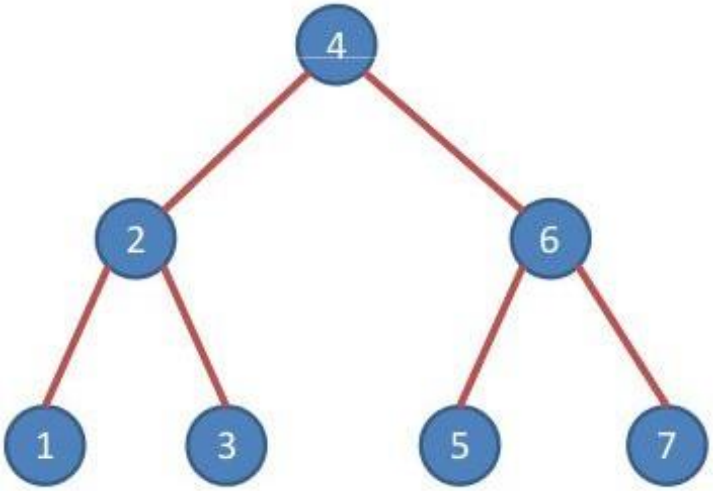
search(a, 10)

Problemas com ABBs

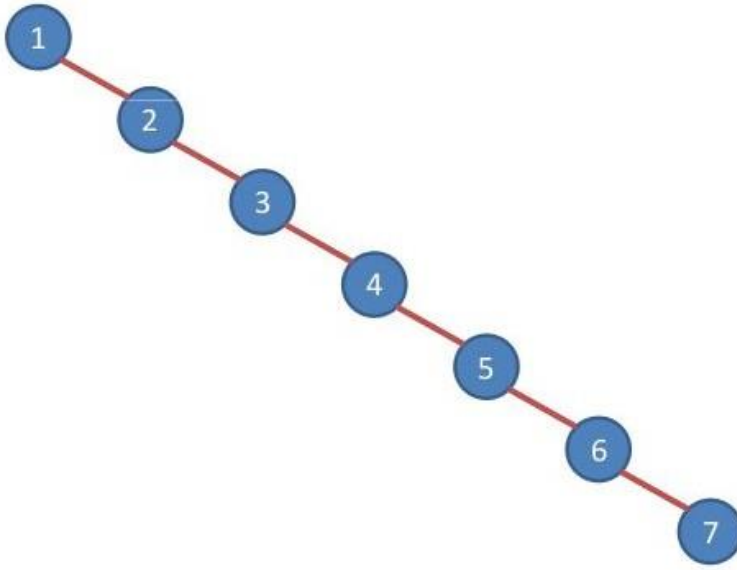
- Pesquisas utilizando **árvores binárias de busca aumentam o desempenho** em **relação** a busca em **listas encadeadas ou vetores**;
- No entanto, à medida em que a árvore vai sendo modificada (através de inserções e remoções), ela pode ficar degenerada.

Inserção, na ordem indicada, em A:

4, 2, 6, 1, 3, 5 e 7 → pior caso é O(log n)

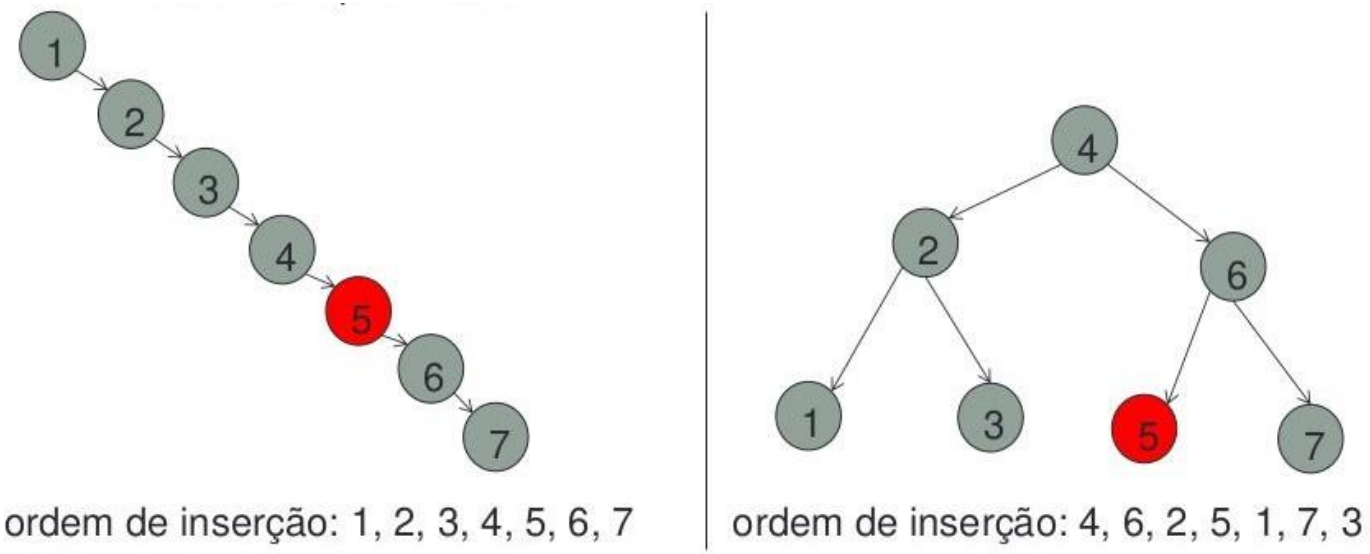


1, 2, 3, 4, 5, 6, 7→ pior caso é O(n)



Problemas com ABBs

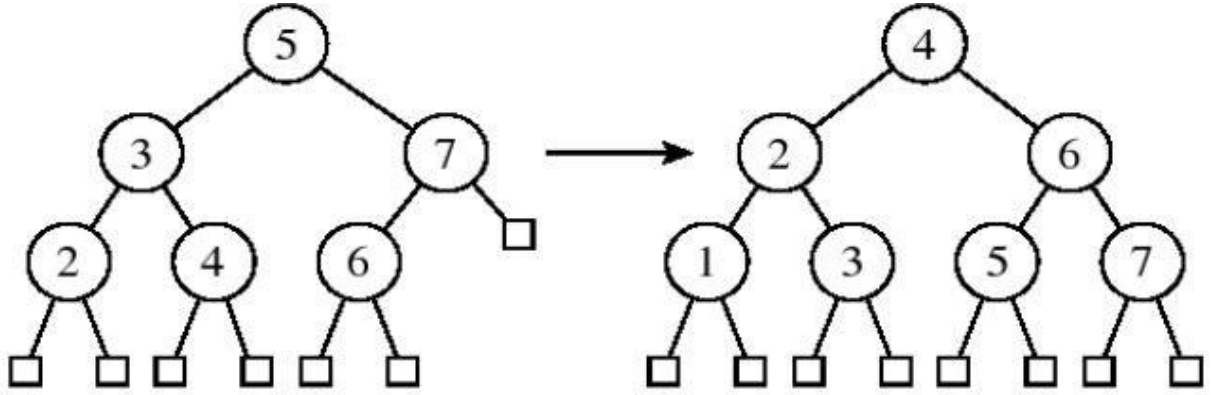
- As ABBs não são livres de **limitações**:
  - À medida em que nós são inseridos e removidos, a árvore pode ficar **desbalanceada**;
  - O desempenho, portanto, depende da ordem que os elementos são inseridos. Considere a busca pelo elemento 5 nas árvores abaixo:



  - Para resolver este problema, foram desenvolvidos algoritmos para árvores de busca (binárias ou não binárias) balanceadas.

Problemas com ABBs

- Idealmente, deseja-se que a árvore esteja **completamente balanceada**, para qualquer nó da árvore:
  - A distância média para qualquer nó da árvore é mínima;
  - Isso diminui o tempo de busca.
- **Manter uma árvore completamente balanceada tem um custo alto**:
  - Considere inserir a chave 1 na árvore à esquerda e obter a árvore a direita → é necessário movimentar todos os nós da árvore original.



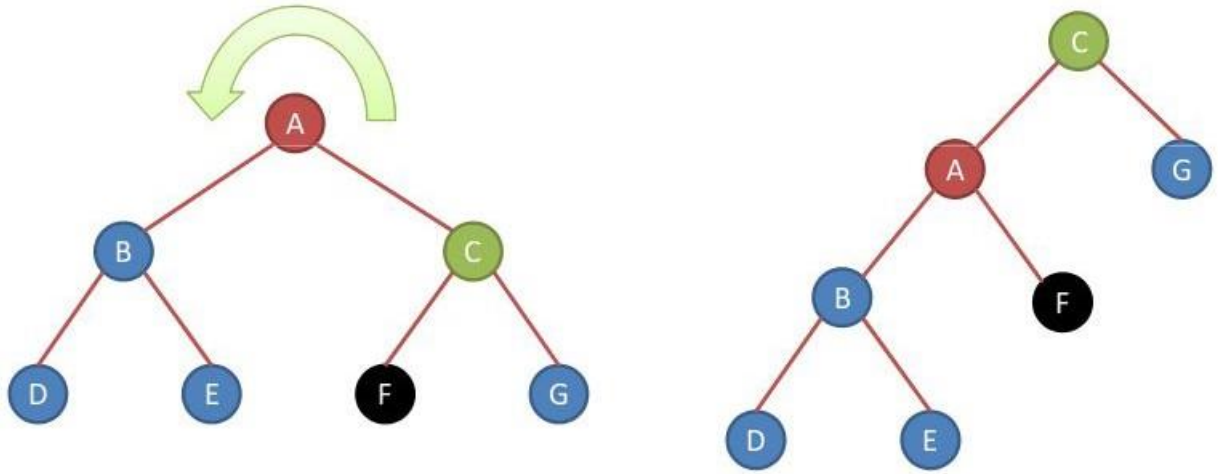
Balanceamento de árvores

- **Uma boa condição de balanceamento possui as seguintes características**:
  - Assegurar que a altura da árvore com n nós é O(log n).
  - Deve poder ser mantida balanceada com um custo baixo;
  - Para fazer o balanceamento de árvores são usadas as chamadas rotações.

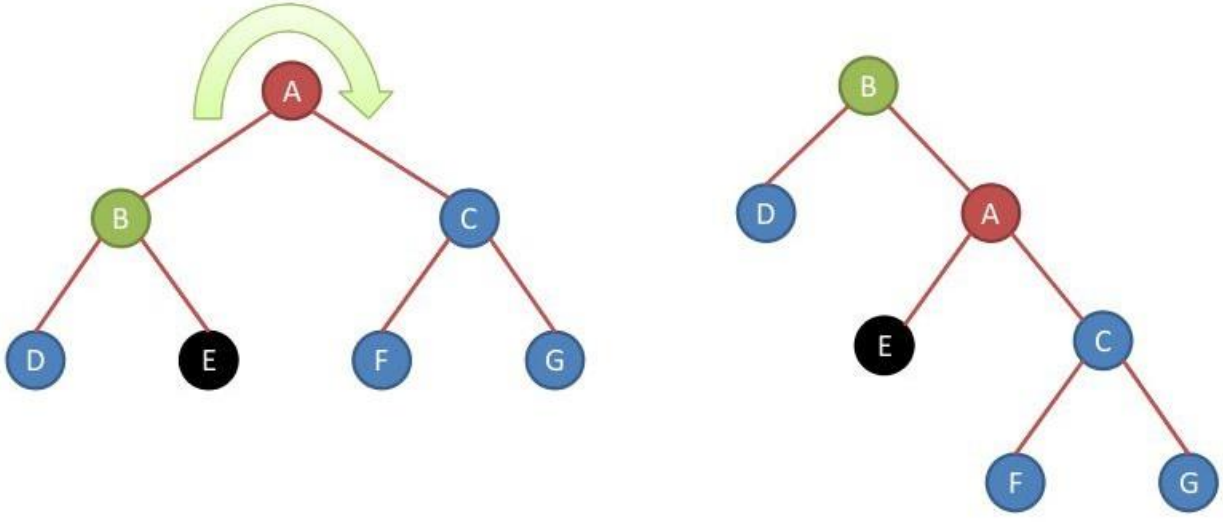
- **Rotações** em ABBs podem ser feitas rotações em dois sentidos:
  - Rotações à direita;
  - Rotações à esquerda.

Rotações de árvores

- Rotação à esquerda:



- Rotação à direita:



Balanceamento de árvores

- Ao invés de tentar manter a árvore **completamente** balanceada, **procura-se uma solução intermediária que possa manter, de forma controlada**, a árvore balanceada:
  - Árvores balanceadas visam maximizar o desempenho de buscas em árvores binárias.
- **Objetivo:** obter bons tempos de pesquisa:
  - Próximo do tempo ótimo da árvore completamente balanceada;
  - Sem pagar muito para inserir e retirar da árvore.
- Exemplos de soluções possíveis:
  - **Árvores AVL;**
  - Árvores SBB (*Symmetric Binary B-Trees*)

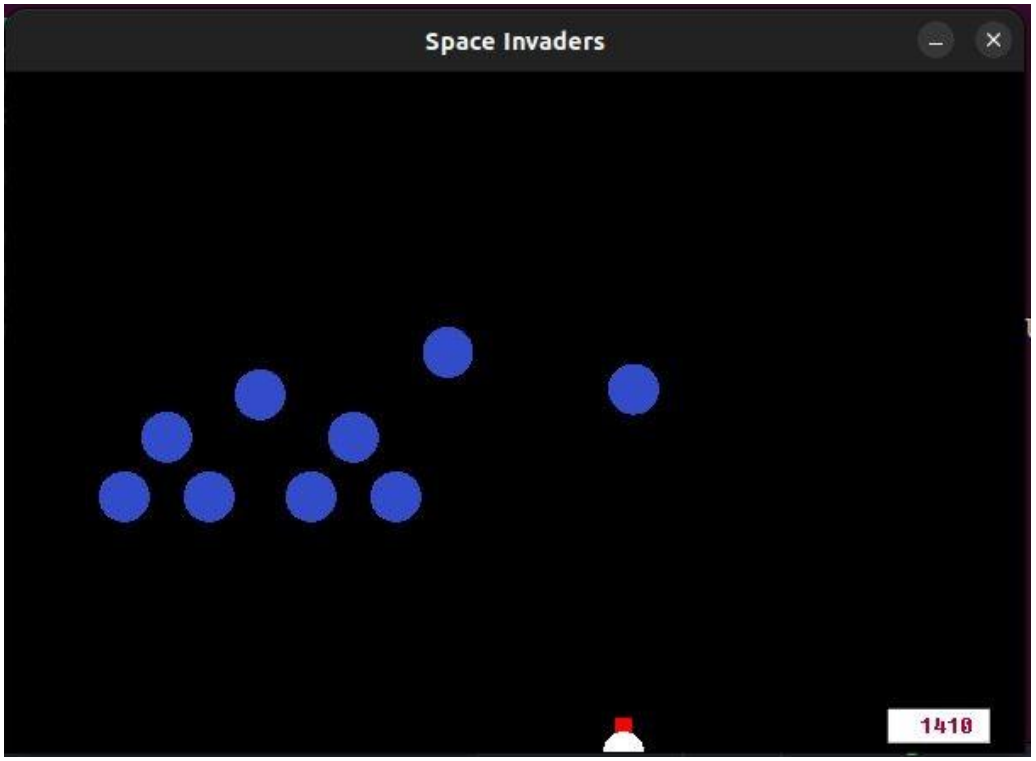
Árvores Balanceadas (AVL)

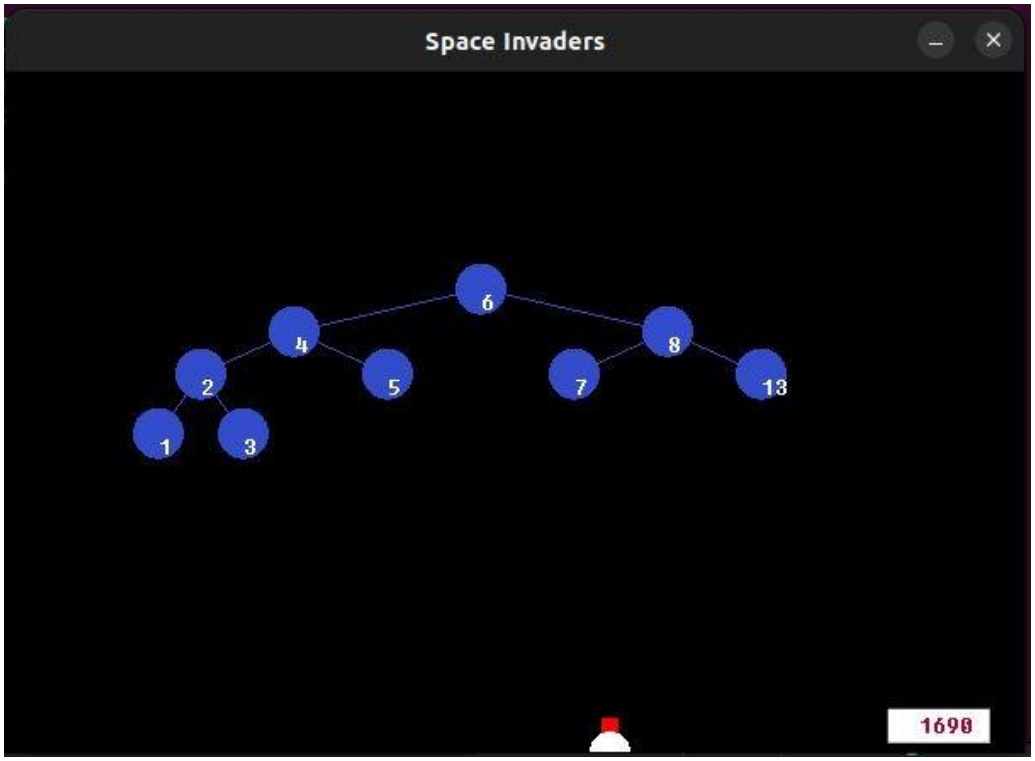
- Nome vem de seus criados (Adelson Velsky e Landis);
- Uma árvore binária é denominada AVL se:
  - Para todos os nós, as **alturas** de suas duas sub-árvores diferem **no máximo de uma unidade** → logo, é balanceada.
- Operações de consulta, inserção e remoção de nós **tem custo O(log n),** sendo n o número de nós da árvore, em específico:
  - Busca é O(log n) → altura da árvore é O(log n), não necessita de reestruturação;
  - Inserção é O(log n) → **busca inicial é O(log n), reestruturação exige uma rotação**;
  - Remoção é O(log n) → **busca inicial é O(log n), reestruturação pode exigir mais de uma rotação);**
  - Uma única rotação é O(1) → tempo constante.

Árvores Balanceadas (AVL)

- Exemplo rotações AVL:
  - Space Invaders (com Allegro)

Comparação jogo ABB x AVL após eliminar 6 inimigos





Agenda

- Relembrando árvores
- Pesquisa digital
- Árvores Binárias de Busca
- *Hands-on*

Atividade Individual

1. Crie um **TAD** (Tipo Abstrato de Dados) de ABB, com, no mínimo, funções para:
   - criação;
   - inserção;
   - impressão (percurso em pré-ordem, pós-ordem, ordem simétrica);
   - deleção;
   - busca.
2. Pesquisar sobre outros tipos de árvore:
   - *Red–black tree* (Árvore rubro-negra)
   - *B tree, B+ tree*

Próxima aula...

- Tabelas de dispersão (*hash tables*):
  - Funções de transformação de chave (*hashing*);
  - Cálculo de endereços e tratamento de colisões;
  - Endereçamento aberto;
  - Listas encadeadas;
  - Hashing perfeito.