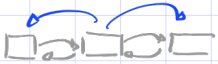
Árvores

↳ Terminologia
↳ Árvores binárias [de busca]

Lista linear:

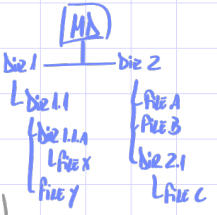


Árvore:

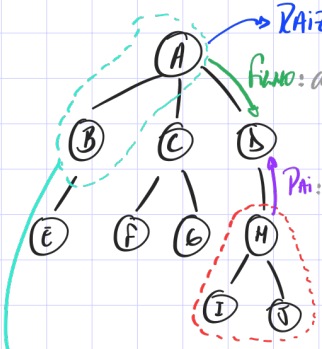
Árvore: Estrutura de dados não linear que modela relações entre seus elementos.

↳ Representar hierarquias.

Ex: Sistema de arquivos
Estrutura de uma empresa
Mapa mental (composição, derivação)



Elementos → Nodos (nós) → Círculos (ou qualquer forma geométrica)
Relacionamentos → Arestas → Linhas (ou setas)



Raiz: Primeiro nodo; acesso inicia por ele.

Filho: descendente direto (sucessor(es))

Irmãos: nodos com o mesmo pai / predecessor

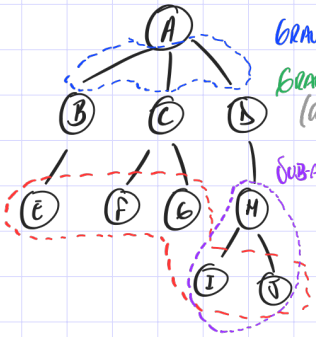
Pai: ascendente direto (predecessor)

Descendentes (de D): estão "abaixo" de determinado nodo (relação de dependência)

Ascendentes (de E): estão "acima" de determinado nodo

Exemplos:

- E é descendente de A, tal como C, H, J, ...
- F é filho de C. A é pai de B, C, D.
- D é ascendente de I, bem como de H e J.
- F e G são irmãos.



GRAU DO NODO: número de filhos

GRAU DA ÁRVORE: o maior valor entre os graus dos seus nodos (aridade)

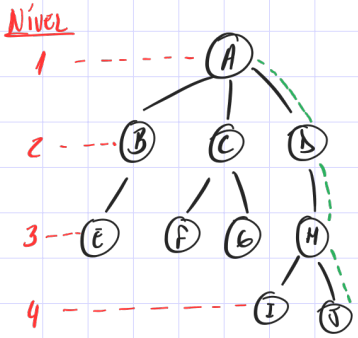
SUB-ÁRVORE: um subconjunto de nodos conectados da árvore

FOLHAS: nodos que não possuem filhos (grau 0) (também chamados nodos terminais ou externos)

Analogamente, um nodo **não-folha** é aquele que possui filhos. Também chamado **interno** ou de **derivação**

EXEMPLOS:

- D tem grau 1, e C tem grau 2.
- A árvore tem grau 3.
- C, F e G formam uma sub-árvore.
- A sub-árvore <u>da esquerda</u> de A é {B, E}.
- A sub-árvore de D é {H, I, J}.



CAMINHO: sequência de arestas (ou nodos) entre dois nodos

ALTURA (DA ÁRVORE): nível/profundidade máximo entre seus nodos

NÍVEL (DO NODO): número de nodos do caminho da raiz até o nodo (profundidade)

ALTURA (DO NODO): número de nodos do maior caminho entre o nodo e seus descendentes-folha

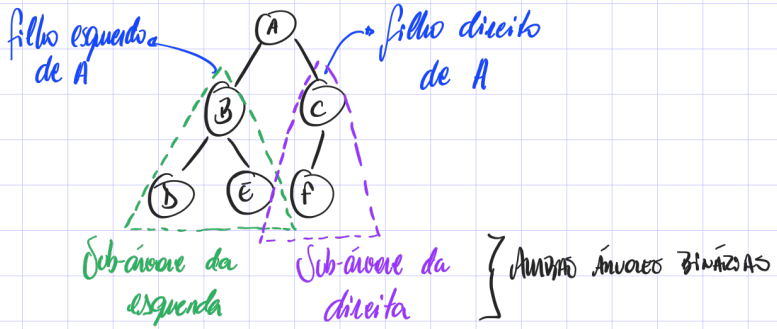
EXEMPLOS:

- G tem maior nível/profundidade que D.
- B tem menor altura que D (2 vs. 3).
- A-C-G é um caminho de A até G.
- Nenhum outro nodo tem altura ≥ que a altura da raiz.

Árvore n-ária: árvore de grau no máximo n.

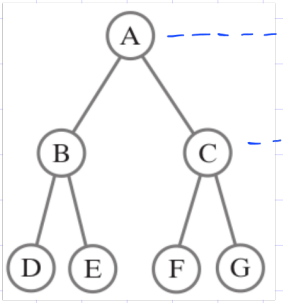
↳ Os nodos têm no máximo n filhos.

EXEMPLO: Árvore binária (0, 1 ou 2 filhos)



ÁRVORE BINÁRIA CHEIA: Seja h a altura da árvore, ela é CHEIA quando todas as suas folhas estão no nível h (último) e todo nodo não-folha possui exatamente dois filhos.

↳ Ou seja, possui todos os nodos possíveis.



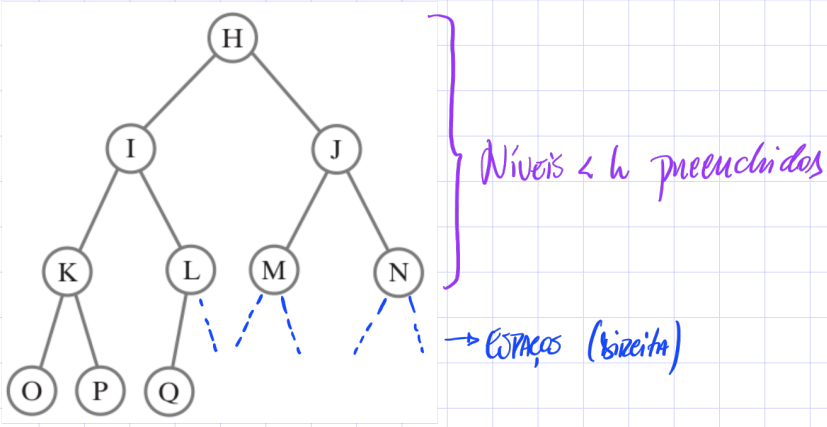
Todos os nodos não-folha têm exatamente dois filhos (e estão em níveis < h)

Folhas no último nível = altura (h)

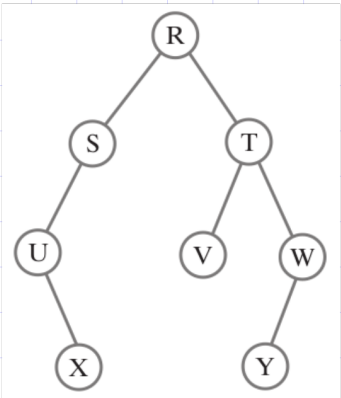
↳ Árvore cheia até o último nível!

Árvore Binária Completa: Seja h a altura da árvore, ela é completa quando todos os níveis < h possuem todos os nodos possíveis e o último nível (h) está preenchido da esquerda para a direita.

↳ Ou seja, só existe espaço no último nível, e eles se concentram à direita.



Exemplo de árvore que não é cheia, nem completa.



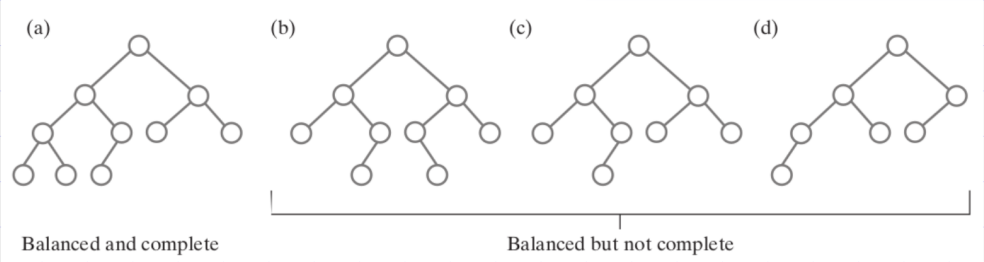
Balanceamento

↳ Uma árvore binária é **completamente balanceada** quando cada nodo possui duas sub-árvores (esquerda, direita) com exatamente a mesma altura.

↳ Somente árvores cheias são completamente balanceadas.

↳ Uma árvore binária é **balanceada** quando a diferença na altura das sub-árvores de cada nodo é de no máximo 1.

↳ Toda árvore completa é balanceada. Mas existem árvores não-completas que são balanceadas.



Algumas propriedades (árvores cheias)

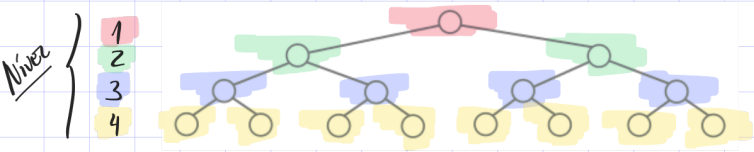
- Altura: $h = \log_2(x+1)$, para uma árvore com x nodos
- Número de nodos:
  - ↳ De um nível n: 2^{n-1}
  - ↳ Até um nível n: $2^n - 1$
  - ↳ De toda a árvore: $2^h - 1$

→ Nodos $n = 2^4 - 1 = 15$

Altura $h = \log_2 15+1 = 4$

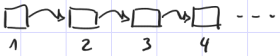
Nodos nível 3: $n' = 2^{3-1} = 4$

Nodos até nível 3: $2^3 - 1 = 7$



Percurso (Caminhamento, travessia, busca)

↳ Percorrer uma estrutura linear é simples:



↳ Como percorremos uma árvore?

Um percurso consiste em visitar (processar) todos os nodos da árvore.

Principais métodos:

- Pré-ordem ⟶ Profundidade
- Pós-ordem
- In-ordem
- Level-ordem ⟶ Largura

Busca em Profundidade (pré-ordem)

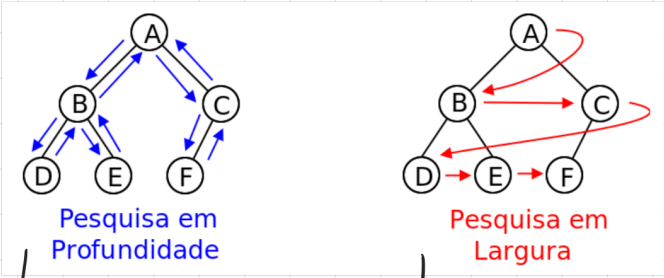
⟶ DFS: Depth-first search

↳ Visita os nodos da árvore, priorizando os nodos mais profundos, sempre que possível.

Busca em Largura (level-ordem)

⟶ BFS: Breadth-first search

↳ Visita todos os nodos de um nível antes de visitar o próximo nível.



↳ Ordem: A B D E C F

↳ Ordem: A B C D E F

Algoritmos de Percurso

Percurso:

N ← {Raiz da árvore}
Enquanto N não é vazio faça
- n ← extrai nodo de N
- Visita nodo n
- Adiciona os filhos de n a N → Para árvores n-árias, cada filho é adicionado seguindo alguma ordem desejada (ou aleatória).

Complexidade? (n é o número de nodos)
- Laço de repetição: $O(n)$
- Operações internas: $O(1)$

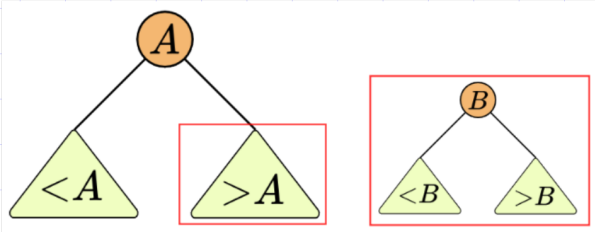
} Logo, BFS e $O(n)$
DFS e $O(n)$

Largura: N é uma fila (filho da esquerda é inserido primeiro).
Profundidade: N é uma pilha (filho da direita é inserido primeiro).

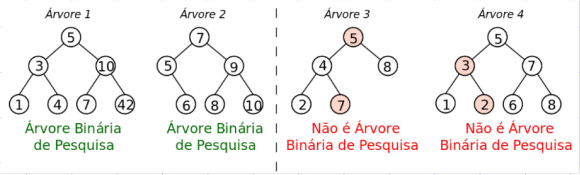
Árvores Binárias de Busca

- Busca em uma árvore qualquer: $O(n)$
- Árvore de Busca:
  - Organiza (ordena) os dados (nodos) para que a busca seja eficiente.
  - Exemplo: Árvore Binária de Busca

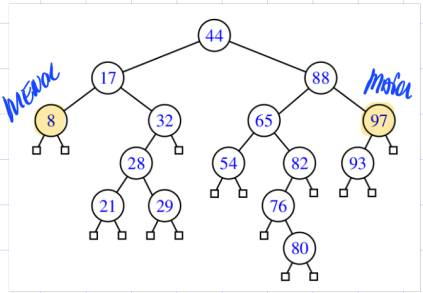
Propriedade: Se um nodo n armazena um elemento e(n),
1. Elementos da sub-árvore esquerda de n são menores que e(n);
2. Elementos da sub-árvore direita de n são maiores que e(n).



Exemplos:



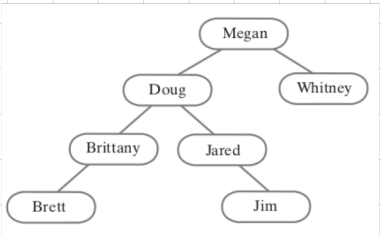
Consequências:



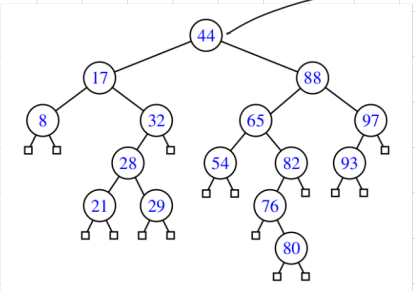
→ Menor nodo é o que está mais à esquerda, enquanto o maior é o que está mais à direita!

Podemos usar ABB para elementos que suportem tipo {Números, Strings, Pessoa, Veículo ... } Comparável!

↳ Exemplo (Nomes):



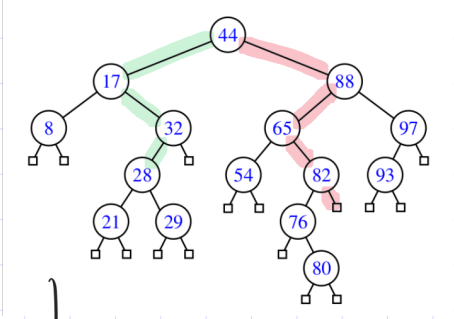
Busca:



Começa pela raiz.
Elemento buscado? Sucesso!
Maior que o buscado?
↳ Repete na sub-árvore esquerda.
Menor que o buscado?
↳ Repete na sub-árvore direita.
É null? Falha!

Busca (e, Raiz):

- Se Raiz = NULL então
  - Retorna Falha
- Se Raiz.Value = e então
  - Retorna Sucesso
- Se Raiz.Value > e então
  - Retorna Busca(e, Raiz.left)
- Se Raiz.Value < e então
  - Retorna Busca(e, Raiz.right)



Buscando o 28

Buscando o 85

Em resumo, percorre somente um ramo da árvore para completar a busca.

↳ Pior caso? Quando percorre o ramo mais longo.

Logo, complexidade proporcional à altura da árvore: $O(h)$.

↳ Árvore balanceada? $O(\log n)$

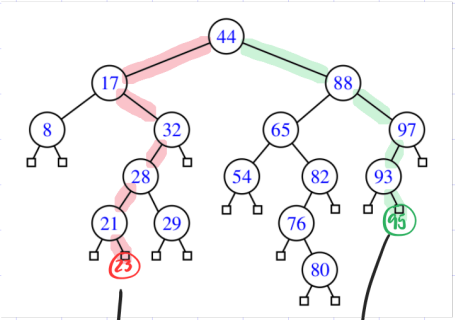
Árvore desbalanceada? $O(n)$

Inserção: 1. Busca a posição correta; → Sempre como filho de uma folha

2. Insere novo elemento.

Insere (e, Raiz):

- Se Raiz = NULL então
  - Novo Nodo (c/ elemento e) Aqui
- Se Raiz.Value > e então
  - Insere(e, Raiz.left)
- Se Raiz.Value < e então
  - Insere(e, Raiz.right)



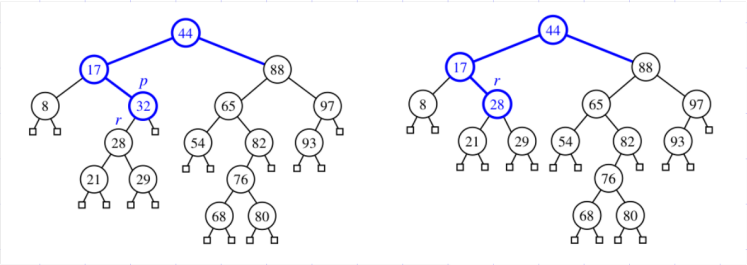
Inserção do 23

Inserção do 95

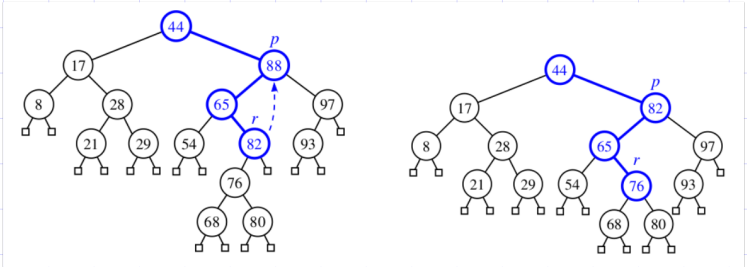
Remoção:

1. Remove o nodo.
2. Se o nodo é uma folha, termina.
3. Se o nodo possui um único filho, (Promover filho)
   o filho assume sua posição.
4. Se o nodo possui dois filhos, rearranja. (Rearranjar)

Promover filho:



Rearranjar:



- p: elemento a remover.
- r: maior elemento menor que p (i.e. da sua sub-árvore esquerda).
- r assume a posição de p.
- O filho esquerdo de r assume sua posição.
  (r não terá filho direito)

Complexidade (inserção/remoção)? $O(h)$

↳ Ou seja, Desbalanceada ⟷ Balanceada

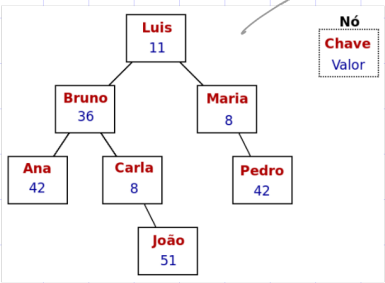
$O(n)$ ⟷ $O(\log n)$

Comparação de complexidade (sem admitir repetição)

	Lista ord.	Lista não ord.	ABB
contains	$O(\log n)$	$O(n)$	$O(\log n)$
insert	$O(n)$	$O(n)$	$O(\log n)$
remove	$O(n)$	$O(n)$	$O(\log n)$

Revisitando Mapas:

Árvore de busca binária



TreeMap

	Array ordenado	Tree Map
get	$O(\log n)$	$O(\log n)$
put	$O(n)$	$O(\log n)$
remove	$O(n)$	$O(\log n)$

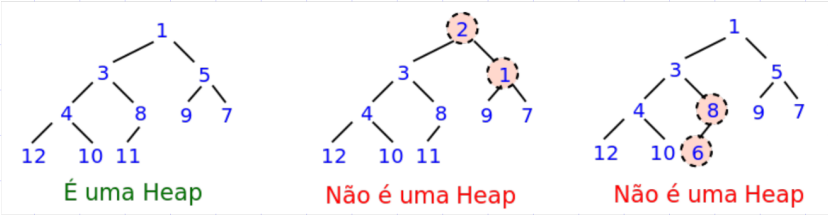
Heaps

Uma heap é uma árvore binária ordenada e completa, onde:

MIN-HEAP: O pai tem sempre um valor menor que seus filhos.

MAX-HEAP: O pai tem sempre um valor maior que seus filhos.

MIN-HEAP

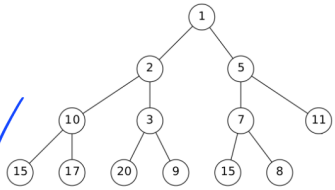


MIN-HEAP é ideal para implementar filas de prioridade.

↳ Ordem definida pela chave das entradas → min é sempre a raiz!

↳ Por ser completa, $h = O(\log n)$. Logo, há garantia de eficiência ($\neq$ ABB).

Implementação (usando um arranjo)



Acesso aos vizinhos na árvore

Seja um elemento i da árvore:

- $\texttt{leftChild}(i) = 2i + 1$
- $\texttt{rightChild}(i) = 2i + 2$
- $\texttt{parent}(i) = \lfloor (i-1)/2 \rfloor$

Percurso em nível

1	2	5	10	3	7	11	15	17	20	9	15	8			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

→ H

↳ Acesso à raiz, ao último nodo e à primeira posição livre em tempo constante – $O(1)$.

Fila de Prioridade: min, removeMin, insert

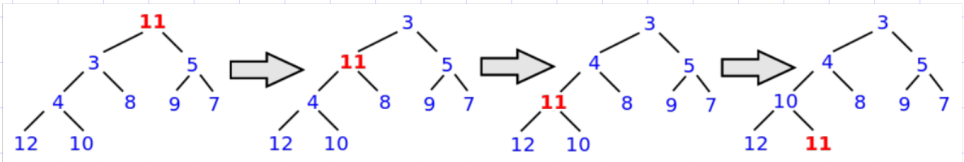
↳ Devolvem raiz da árvore (H[0]): $O(1)$.

RemoveMin

1. Remove o nodo da raiz → "Buraco"
2. Move o último nodo para a raiz
3. Recursivamente: troca o nodo movido com o menor dos seus filhos
   ↳ Até que a ordenação da heap esteja correta.

Heapify-down / Down-heap

Pior caso: desce até uma folha → $O(\log n)$

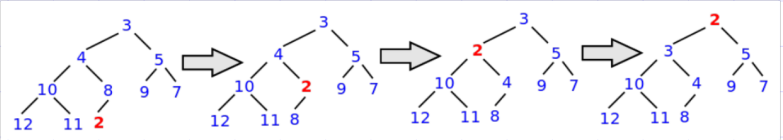


insert

1. Insere o nodo na última posição.
2. Repetidamente: troca com o pai.
   ↳ Até que a ordenação da heap esteja correta.

→ Heapify-up / Up-heap

Pior caso: sobe até a raiz → $O(\log n)$



Complexidade

	Array não-ord	Array ord.	Heap
min	$O(n)$*	$O(1)$	$O(1)$
RemoveMin	$O(n)$	$O(1)$	$O(\log n)$
insert	$O(1)$	$O(n)$	$O(\log n)$

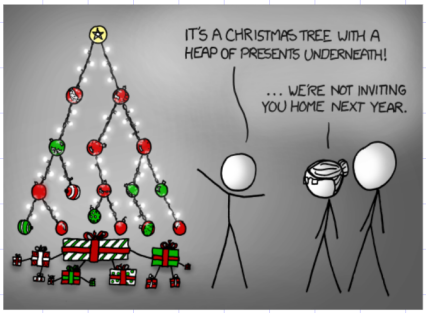
*pode ser reduzido para $O(1)$

Usando Heaps para ordenação:

Dado um conjunto de n valores:

1. Cria uma heap e insere os n valores.
2. Remove um a um (RemoveMin), obtendo os valores de forma ordenada.

$$\begin{array}{r} O(n \log n) + \\ O(n \log n) \\ \hline O(n \log n) \end{array}$$

→ O passo 1 pode ser feito em $O(n)$ usando uma estratégia de criação alternativa!

↳ HeapSort!