Árvores de Busca Binária

Rafael Alves da Costa

ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
FATEC Carapicuíba

Aula 12 - Estrutura de Dados

11/2025

Sumário

Árvores Binárias de Busca

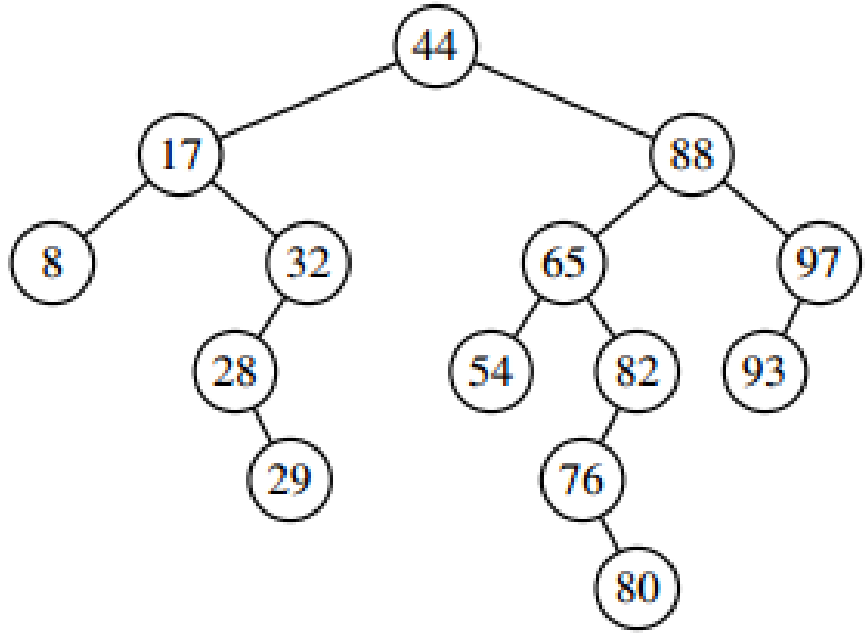
Árvores Binárias de Busca (BST) – Conceito

As Árvores Binárias de Busca (BSTs) são uma estrutura de dados fundamentalmente usadas para armazenar e recuperar dados de forma eficiente.

- Mantêm uma ordenação onde o valor de cada nó é:
  - maior que todos os valores em sua subárvore esquerda;
  - menor que todos os valores em sua subárvore direita.
- Essa propriedade permite que operações como busca, inserção e remoção sejam realizadas, em média, em tempo $O(\log n)$.
- São mais eficientes que arrays ou listas encadeadas para certas tarefas.

Estrutura para BST

- Uma árvore binária de busca com chaves inteiras, sem a exibição dos valores associados, pois eles não são relevantes para a ordenação dos itens dentro de uma árvore de busca.



BSTs – Aplicações e Eficiência

- BSTs são úteis quando é necessário recuperar dados ordenados.
- Permitem encontrar rapidamente valores mínimos ou máximos.
- Variantes balanceadas, como as árvores AVL ou Red-Black[1]:
  - Garantem desempenho ideal mesmo nos piores cenários.
  - Mantêm complexidade $O(\log n)$ para as operações.
- Essa eficiência torna as BSTs fundamentais no design de algoritmos e em sistemas de gerenciamento de dados.

[1] veja Capítulo 11 de "Goodrich, M. T., Tamassia, R., Goldwasser, M. H. (2013). Data structures and algorithms in Python"

Comparação de Estruturas de Fila de Prioridade

Estrutura	**Inserção**	**Remoção Mínimo**	**Busca Mínimo**
Lista não ordenada	$O(1)$	$O(n)$	$O(n)$
Lista ordenada	$O(n)$	$O(1)$	$O(1)$
Heap (min-heap)	$O(\log n)$	$O(\log n)$	$O(1)$
Heap (max-heap)	$O(\log n)$	$O(n)$	$O(n)$
Árv. de busca binária (BST)	$O(\log n)$	$O(\log n)$	$O(\log n)$

Comparação de complexidade entre diferentes implementações de filas de prioridade.

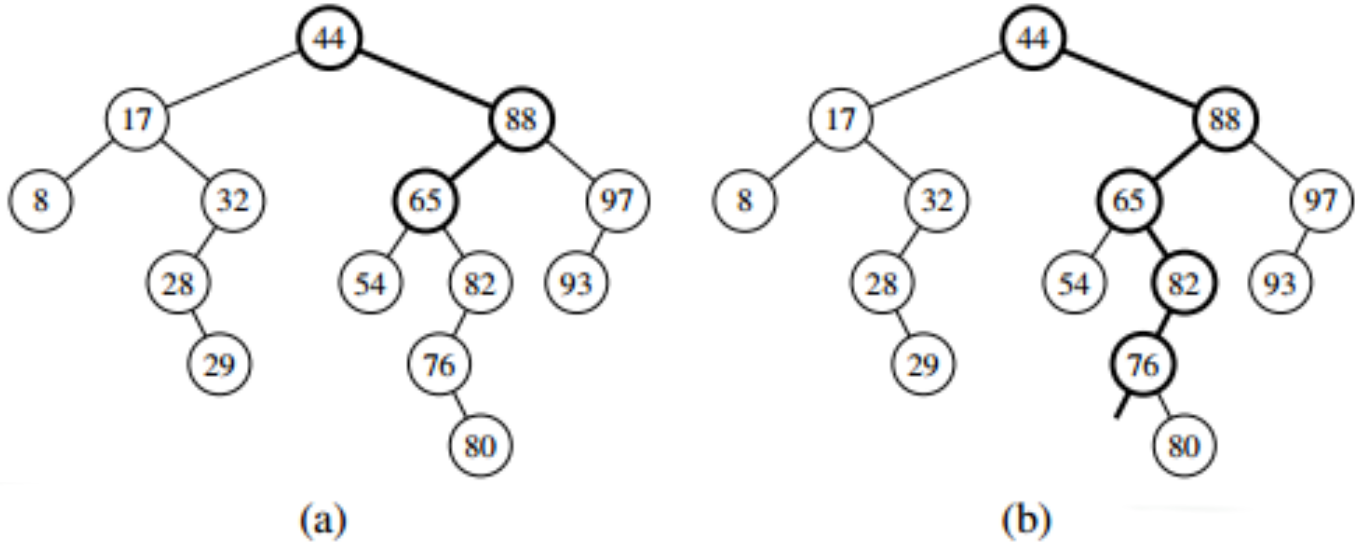
- Nota-se que a árvore de busca binária (BST) na tabela de comparação de estruturas de fila de prioridade, possui complexidade média, pois no pior caso (BST desbalanceada) pode ser **Linear**.

Busca em Árvores Binárias de Busca (BST)

A principal consequência da estrutura de uma BST é o seu algoritmo de busca:

- Em cada posição p, comparamos a chave desejada k com p.`key()`:
  - Se $k < p$.`key()`: continua na subárvore à esquerda;
  - Se $k = p$.`key()`: a busca é bem-sucedida;
  - Se $k > p$.`key()`: continua na subárvore à direita.
- Se alcançarmos uma subárvore vazia, a busca é mal-sucedida.

Figura: Exemplo de busca bem-sucedida (chave 65) e malsucedida (chave 68).



Algoritmo TreeSearch (Busca Recursiva)

Algoritmo: Busca de uma chave k em uma BST enraizada em p

```
Algorithm TreeSearch(T, p, k):
    if k == p.key():
        return p  # busca bem-sucedida
    elif k < p.key() and T.left(p) is not None:
        return TreeSearch(T, T.left(p), k)  # recur. esquerda
    elif k > p.key() and T.right(p) is not None:
        return TreeSearch(T, T.right(p), k)  # recur. direita
    return p  # busca malsucedida
```

Comportamento:

- Executa-se em tempo proporcional à altura da árvore: $O(h)$, com $h = \lceil \log(n+1) \rceil - 1$
- Em caso de falha, retorna a posição final explorada — útil para inserção.

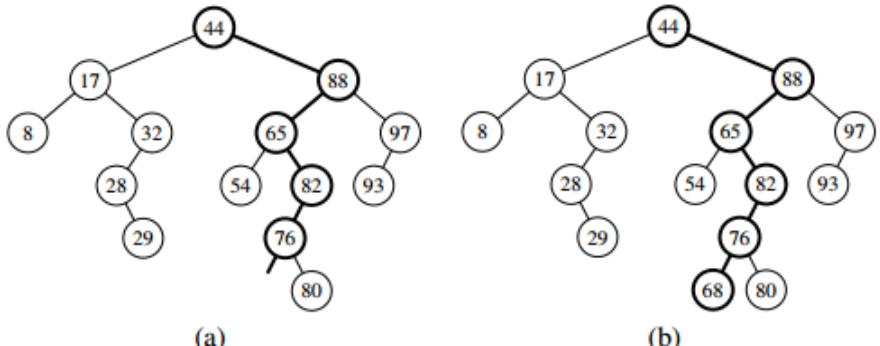
Inserção em Árvores Binárias de Busca (BST)

- Uma busca pela chave k.
- Se encontrada, o valor é atualizado.
- Caso contrário, um novo nó com o par (k, v) é inserido onde a busca falhou.
- A propriedade da BST é mantida, pois a inserção ocorre onde a chave seria encontrada.

Algoritmo TreeInsert(T, k, v):

```
p = TreeSearch(T, T.root(), k)
if k == p.key():
    set p's value to v
elif k < p.key():
    add (k,v) como filho à esquerda de p
else:
    add (k,v) como filho à direita de p
```

Figura: Exemplo de inserção de chave 68.

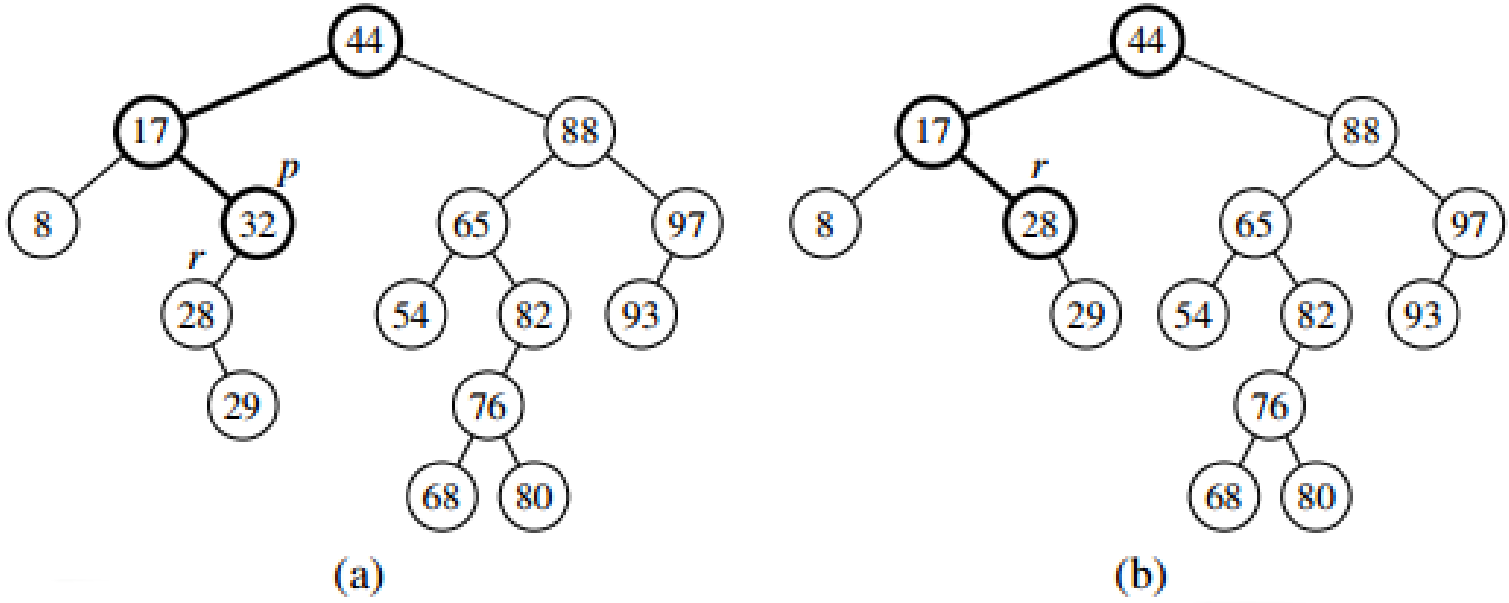


Remoção em BST – Caso com um Filho

Remoção de um nó com no máximo um filho:

- Busca-se a chave k com `TreeSearch`.
- Se o nó p tiver até um filho:
  - O nó é removido e substituído pelo seu filho (se houver).
  - A propriedade da BST é preservada.

Figura: Remoção da chave 32 com um filho.



Remoção em BST – Caso com Dois Filhos

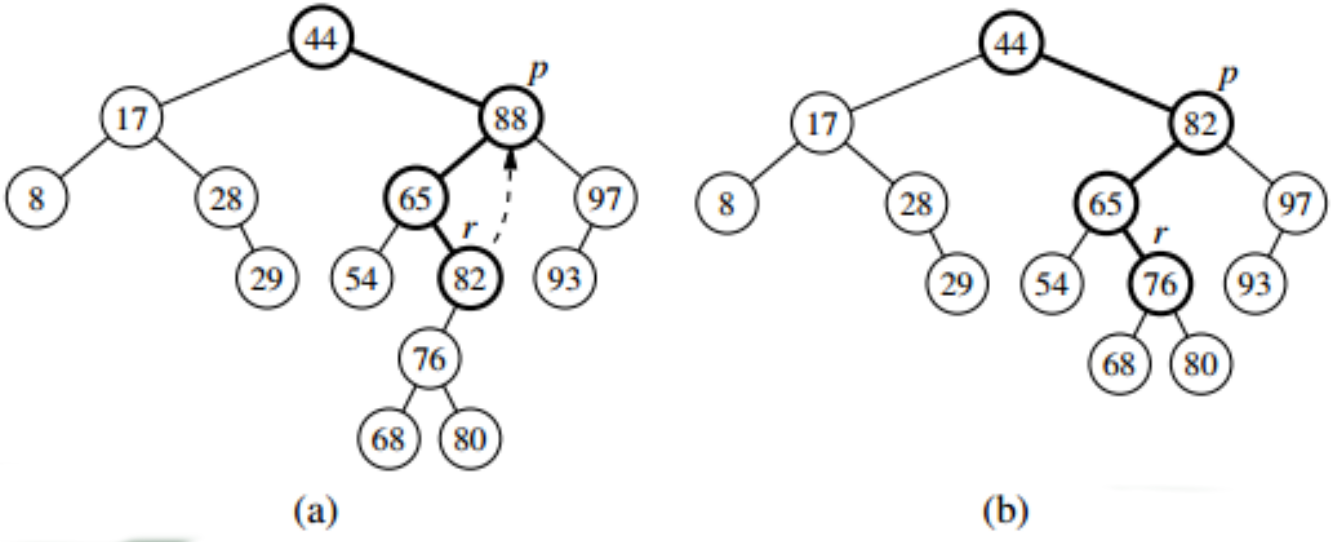
Passos quando o nó a ser removido tem dois filhos:

- Localiza-se o **predecessor** r de p: o maior valor menor que p.
- Substitui-se o conteúdo de p pelo de r.
- Remove-se r, que possui no máximo um filho (caso mais simples).

Justificativa:

- O predecessor está na subárvore esquerda de p.
- A substituição preserva a ordem da BST.

Figura: Remoção da chave 88 com dois filhos.



Performance da BST - Visão Geral

- A análise de desempenho de operações na estrutura `TreeMap` depende da altura h da árvore.
- Muitas operações percorrem da raiz até uma folha, executando trabalho $O(1)$ por nó.
- A eficiência geral está ligada à forma da árvore: mais balanceada, melhor.

Performance da BST - Principais Operações e Tempos

- `getitem`, `setitem`, `delitem` $\rightarrow O(h)$
- Iteração com `after` $\rightarrow O(n)$ total, $O(1)$ amortizado
- `find_range(start, stop)` $\rightarrow O(s + h)$

  Onde s é o número de elementos retornados no intervalo especificado.

Performance da BST - Complexidade no Pior Caso

Operação	Tempo
$k \in T$	$O(h)$
$T[k]$, $T[k] = v$	$O(h)$
`delete`(p), `del` $T[k]$	$O(h)$
`find_position`(k)	$O(h)$
`first()`, `last()`	$O(h)$
`before`(p), `after`(p)	$O(h)$
`find_range`(start, stop)	$O(s + h)$

Performance da BST - Altura Ideal da Árvore

- Melhor caso: altura mínima

$$h = \lceil \log(n+1) \rceil - 1$$

- Pior caso: altura máxima

$$h = n$$

- Vamos ver uma implantação no notebook!

Considerações

Considerações

Considerações

- **Recordando!!!**
  - Árvore de Busca Binária.
  - Exercícios.

Considerações

- **Recordando!!!**
  - Árvore de Busca Binária.
  - Exercícios.
- **Referências utilizadas!!!**

  - Cormen, T. H. et al. Algoritmos: Teoria e prática, 3a edição. Elsevier, 2012.
  - Levitin, A. Introduction to the Design and Analysis of Algorithms, 2007.
  - Goodrich, M. T., Tamassia, R., Goldwasser, M. H. (2013). Data structures and algorithms in Python.
  - Goodrich, M. T., Tamassia, R., Goldwasser, M. H. (2014). Data Structures and Algorithms in Java.
  - Deitel & Deitel (2016). Java: Como Programar.
  - Referências utilizadas também para figuras!