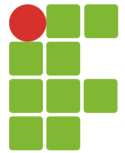
Árvores Binárias de Pesquisa

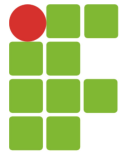
Estruturas de Dados e Algoritmos – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

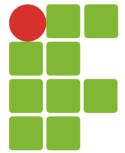
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

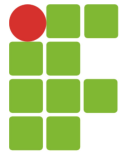


Sumário



Árvores Binárias de Pesquisa

- Uma árvore binária de pesquisa (BST ou Binary Search Tree) organiza os nós pela sua chave.
- Facilita a busca de uma determinada nó através da chave.



Árvores Binárias de Pesquisa

Definição (Árvore Binária de Pesquisa)

Uma árvore binária de pesquisa (BST) possui:

- Uma potencial raiz com valor de chave y.
- Caso a raiz exista, uma subárvore da esquerda com nós que possuem chaves menores que y.
- Caso a raiz exista, uma subárvore da direita com nós que possuem chaves maiores que y.

Árvores Binárias de Pesquisa

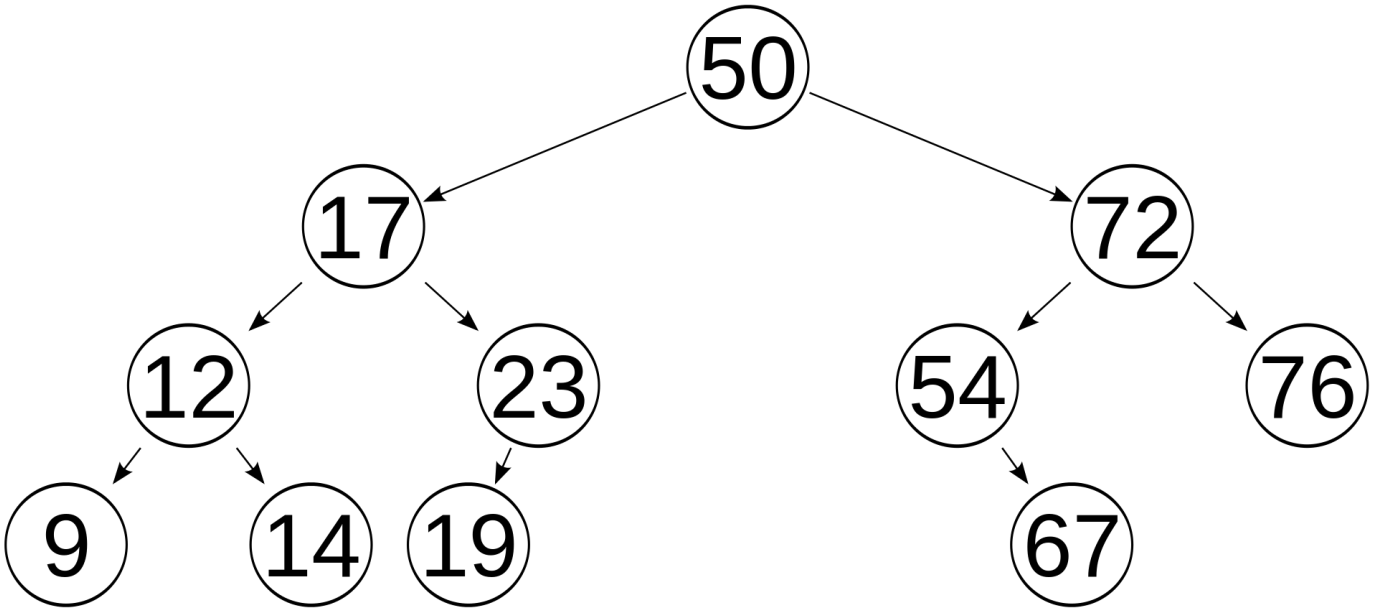
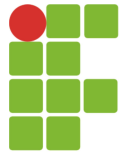
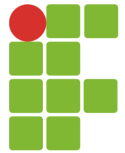


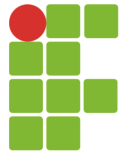
Figura: Árvores Binárias de Pesquisas



Sumário

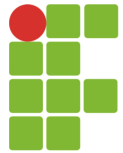


Sumário



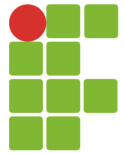
Definição

```
6  typedef void* (*bst_element_constructor_fn) (void*);
7  typedef void (*bst_element_destructor_fn)(void *);
8  typedef int (*bst_tree_element_compare_fn)(const void*,const void*);
```



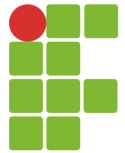
Definição

```
11  typedef struct bst_node_t{
12      void* data;
13      struct bst_node_t* left;
14      struct bst_node_t* right;
15  }bst_node_t;
```

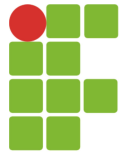


Definição

```
18 typedef struct bst_t{
19     bst_node_t* root;
20     bst_element_constructor_fn constructor;
21     bst_element_destructor_fn destructor;
22     bst_tree_element_compare_fn comparator;
23     size_t size;
24 }bst_t;
```

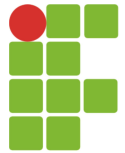


Sumário

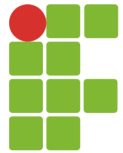


Inicialização

```
35 void bst_initialize(bst_t **t, bst_element_constructor_fn constructor,
36                     bst_element_destructor_fn destructor,
37                     bst_tree_element_compare_fn comparator) {
38     (*t) = mallocx(sizeof(bst_t));
39     (*t)->root = NULL;
40     (*t)->size = 0;
41     (*t)->comparator = comparator;
42     (*t)->constructor = constructor;
43     (*t)->destructor = destructor;
44 }
```

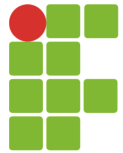


Sumário



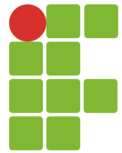
Funções Auxiliares

```
138 size_t bst_size(bst_t *t) {
139     return t->size;
140 }
```



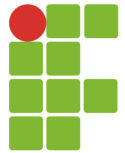
Funções Auxiliares

```
26  static bst_node_t *bst_new_node(void *data,
27                                  bst_element_constructor_fn constructor) {
28      bst_node_t *ptr = mallocx(sizeof(bst_node_t));
29      ptr->left = NULL;
30      ptr->right = NULL;
31      ptr->data = constructor(data);
32      return ptr;
33  }
```

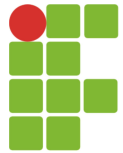


Funções Auxiliares

```
20 static void bst_delete_node(bst_node_t *node,
21                              bst_element_destructor_fn destructor) {
22     destructor(node->data);
23     free(node);
24 }
```



Sumário



Busca

- Durante uma busca por um nó com chave x em uma BST, temos os seguintes casos:
  - A árvore é vazia: o nó não encontra-se na árvore.
  - A raiz possui a chave buscada: o nó buscado foi encontrado.
  - A chave buscada é menor que a chave da raiz: procede-se recursivamente para a subárvore da esquerda.
  - Caso contrário, procede-se recursivamente para a subárvore da direita.

Busca

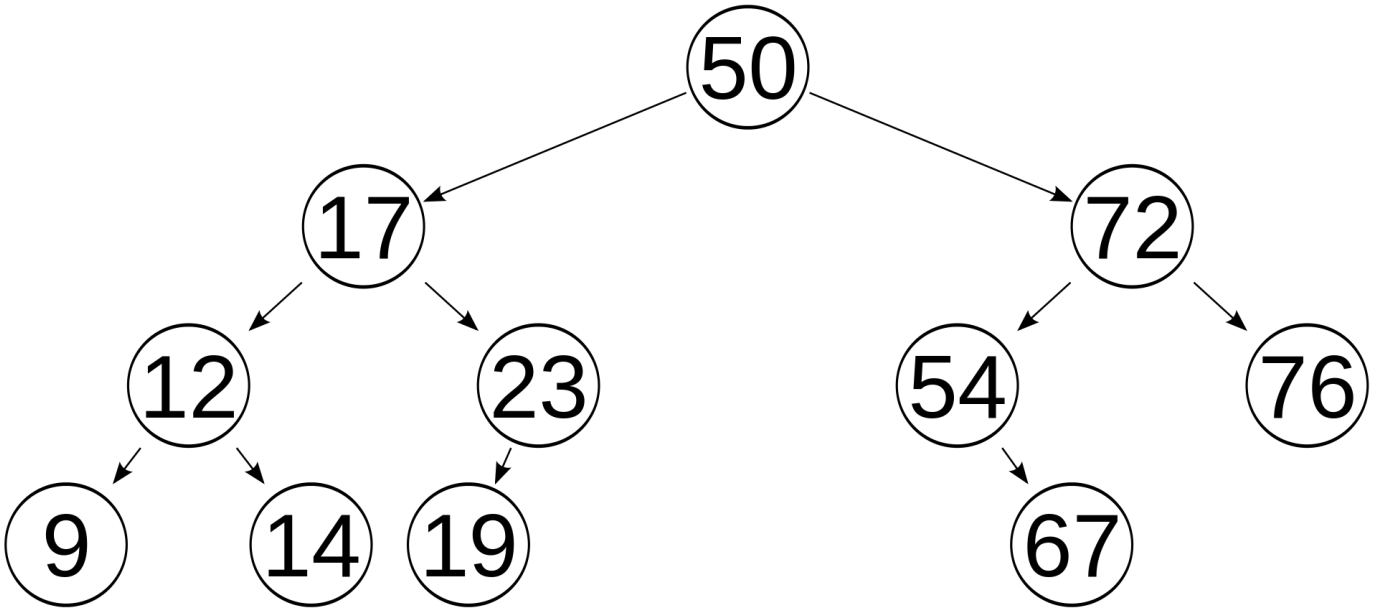


Figura: Busca Pelo 19.

Busca

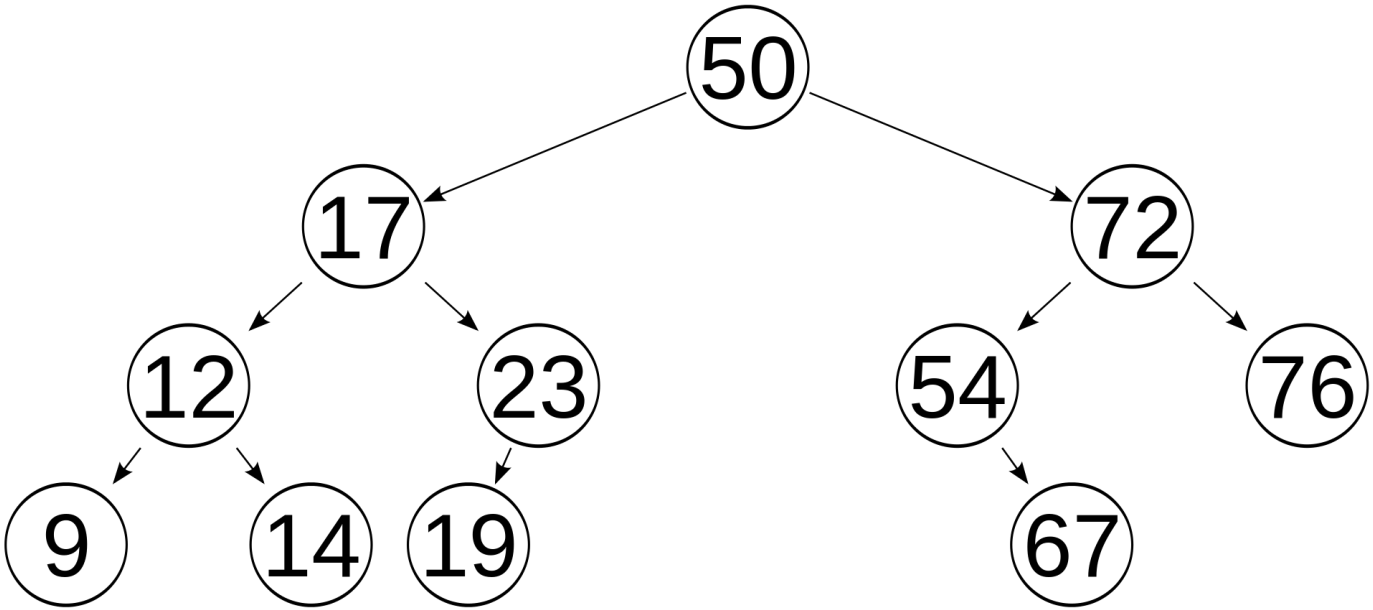


Figura: Busca Pelo 54.

Busca

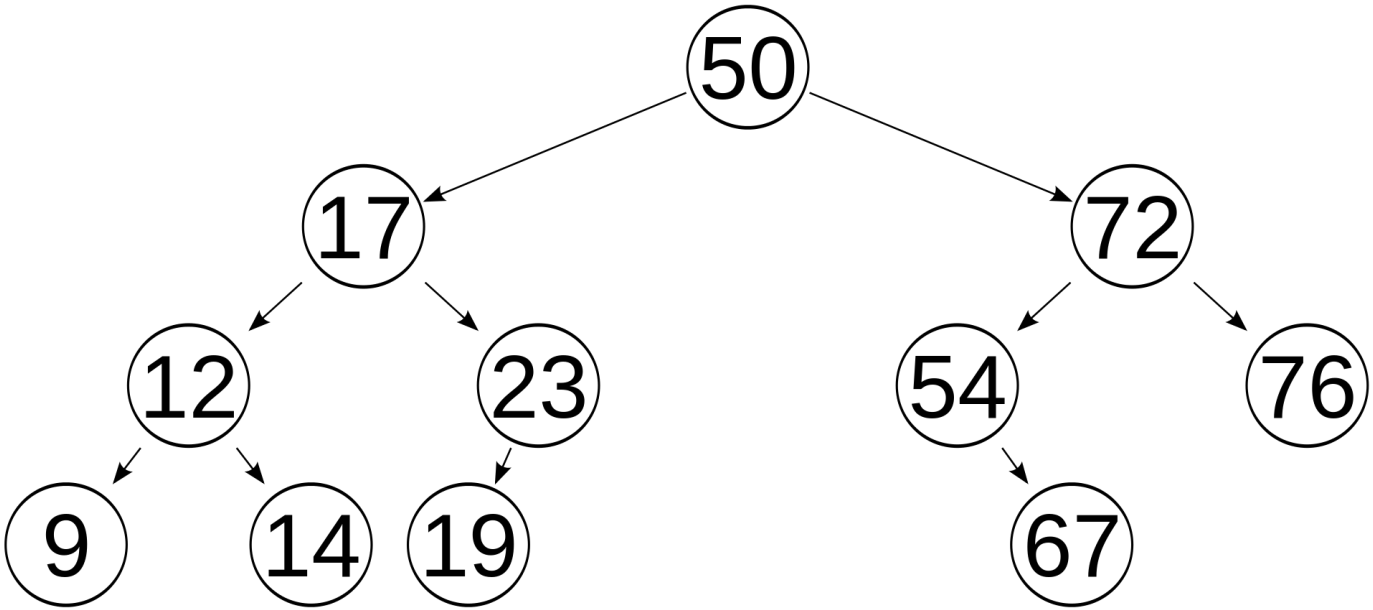
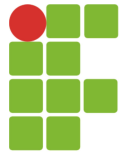
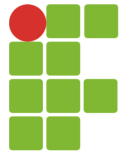


Figura: Busca Pelo 18.



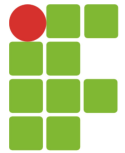
Busca

▸ BST Applet



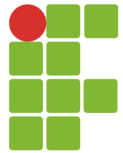
Busca

```
122 int bst_find(bst_t *t, void *data) {
123     return bst_find_helper(t, t->root, data);
124 }
```



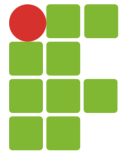
Busca

```
126 static int bst_find_helper(bst_t *t, bst_node_t *v, void *data) {
127     if (v == NULL) {
128         return 0;
129     }
130     if (t->comparator(data, v->data) < 0) {
131         return bst_find_helper(t, v->left, data);
132     } else if (t->comparator(data, v->data) > 0) {
133         return bst_find_helper(t, v->right, data);
134     }
135     return 1;
136 }
```



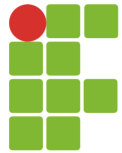
Árvores Binárias de Pesquisa

- A forma como uma BST é organizada lembra alguma técnica?



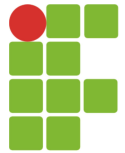
Árvores Binárias de Pesquisa

- A forma como uma BST é organizada lembra alguma técnica?
- **Busca binária**.

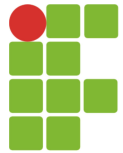


Árvores Binárias de Pesquisa

- A forma como uma BST é organizada lembra alguma técnica?
- **Busca binária**.
- A diferença agora é que a BST permite inserir e remover elementos sem precisar ordenar todo o vetor, no caso da busca binária.

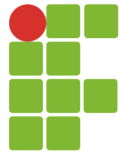


Sumário



Inserção em BSTs

- BSTs são estruturas de dados dinâmicas, permitem inserções e remoções.
- Todas as inserções são feitas nas folhas.

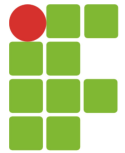


Árvores Binárias de Pesquisa

Inserção em uma BST

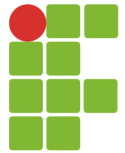
Durante a inserção de um nó com chave x em uma BST, temos os seguintes casos:

- A árvore é vazia: o nó é inserido e passa a ser a raiz daquela árvore.
- O nó a ser inserido possui a chave menor que a da raiz: procede-se recursivamente para a subárvore da esquerda.
- Caso contrário, procede-se recursivamente para a subárvore da direita.



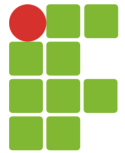
Árvores Binárias de Pesquisa

▸ BST Applet



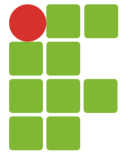
Inserção

```
68 void bst_insert(bst_t *t, void *data) {
69     t->root = bst_insert_helper(t, t->root, data);
70 }
```

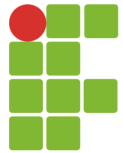


Inserção

```
72  static bst_node_t *bst_insert_helper(bst_t *t, bst_node_t *v, void *data) {
73      if (v == NULL) {
74          t->size++;
75          v = bst_new_node(data, t->constructor);
76      } else if (t->comparator(data, v->data) < 0) {
77          v->left = bst_insert_helper(t, v->left, data);
78      } else {
79          v->right = bst_insert_helper(t, v->right, data);
80      }
81      return v;
82  }
```



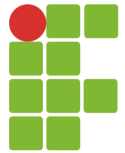
Sumário



Remoção

Remoção em uma BST

- Como visto, as inserções são simples, pois são sempre feitas nas folhas.
- No entanto, a remoção de um nó pode ocorrer em um nó interno.
- Como proceder?

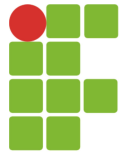


Remoção

Remoção em uma BST

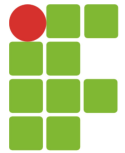
Caso 1: o nó a ser removido é uma folha.

- Este é o caso mais simples.
- O nó é removido sem complicações.



Remoção

▸ BST Applet

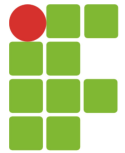


Remoção

Remoção em uma BST

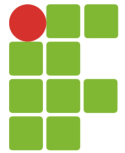
Caso 2: o nó a ser removido possui apenas um filho.

- Também pode ser lidado facilmente.
- O filho é transplantado no lugar do pai.
  - O avô do filho passa a apontar diretamente para o filho.



Remoção

▸ BST Applet

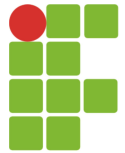


Remoção

Remoção em uma BST

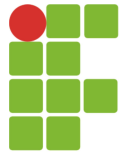
Caso 3: o nó a ser removido possui dois filhos.

- Este é o caso mais difícil.
- Solução: transformar em um caso fácil.
- Obrigatoriamente o nó possui subárvores não vazias.
- Trocamos o nó pelo seu antecessor.
  - O antecessor encontra-se no nó mais à direita da subárvore da esquerda.
- Procede-se recursivamente na subárvore da esquerda para remover o nó desejado.



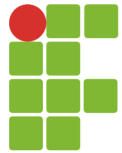
Remoção

▸ BST Applet



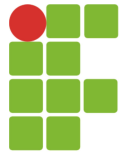
Remoção

```
84 void bst_remove(bst_t *t, void *data) {
85     t->root = bst_remove_helper(t, t->root, data);
86 }
```



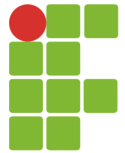
Remoção

```
88  static bst_node_t *bst_remove_helper(bst_t *t, bst_node_t *v, void *data) {
89      if (v == NULL) {
90          return v;
91      } else if (t->comparator(data, v->data) < 0) {
92          v->left = bst_remove_helper(t, v->left, data);
93      } else if (t->comparator(data, v->data) > 0) {
94          v->right = bst_remove_helper(t, v->right, data);
95      } else { /*remoção do nó*/
96          /*caso 1 e caso 2, o nó é uma folha ou só tem um filho. Solução:
97           * transplantar*/
98          if (v->left == NULL) {
99              bst_node_t *tmp = v->right;
100             bst_delete_node(v, t->destructor);
101             t->size--;
102             return tmp;
```



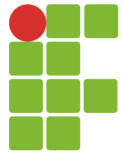
Remoção

```
103         } else if (v->right == NULL) {
104             bst_node_t *tmp = v->left;
105             bst_delete_node(v, t->destructor);
106             t->size--;
107             return tmp;
```



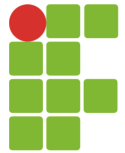
Remoção

```
108         } else {
109             /*caso 3, o nó tem dois filhos: devemos o nó antecessor do
110              que queremos deletar e realizar o swap.
111              Obrigatoriamente este nó cai no caso 1 ou caso 2.*/
112             bst_node_t *tmp = bst_find_rightmost(v->left);
113             void *swap = v->data;
114             v->data = tmp->data;
115             tmp->data = swap;
116             v->left = bst_remove_helper(t, v->left, tmp->data);
117         }
118     }
119     return v;
120 }
```

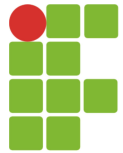


Remoção

```
60  static bst_node_t *bst_find_rightmost(bst_node_t *v) {
61      if (v == NULL || v->right == NULL) {
62          return v;
63      } else {
64          return bst_find_rightmost(v->right);
65      }
66  }
```

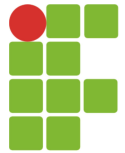


Sumário



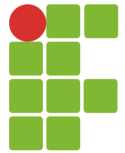
Limpeza

- Para remover a estrutura de memória é simples.
- Primeiro, recursivamente, removemos a subárvore da esquerda.
- Depois, recursivamente, removemos a subárvore da direita.
- Por fim, removemos a raiz.



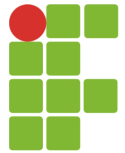
Limpeza

```
46 void bst_delete(bst_t **t) {
47     bst_delete_helper((*t), (*t)->root);
48     free(*t);
49     (*t) = NULL;
50 }
```

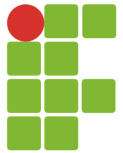


Limpeza

```
52 static void bst_delete_helper(bst_t *t, bst_node_t *v) {
53     if (v != NULL) {
54         bst_delete_helper(t, v->left);
55         bst_delete_helper(t, v->right);
56         bst_delete_node(v, t->destructor);
57     }
58 }
```



Sumário



Análise

- Todas as operações básicas de inserção, remoção e busca, fazem um esforço proporcional à altura da árvore binária de pesquisa.
- Tudo vai depender da maior altura da árvore.
- Quanto mais equilibrada, melhor.
- Idealmente, temos: $\Theta(\lg n)$ por operação básica.
- No pior caso, a BST pode estar degenerada, causando um custo de pior caso de $\Theta(n)$.

Análise

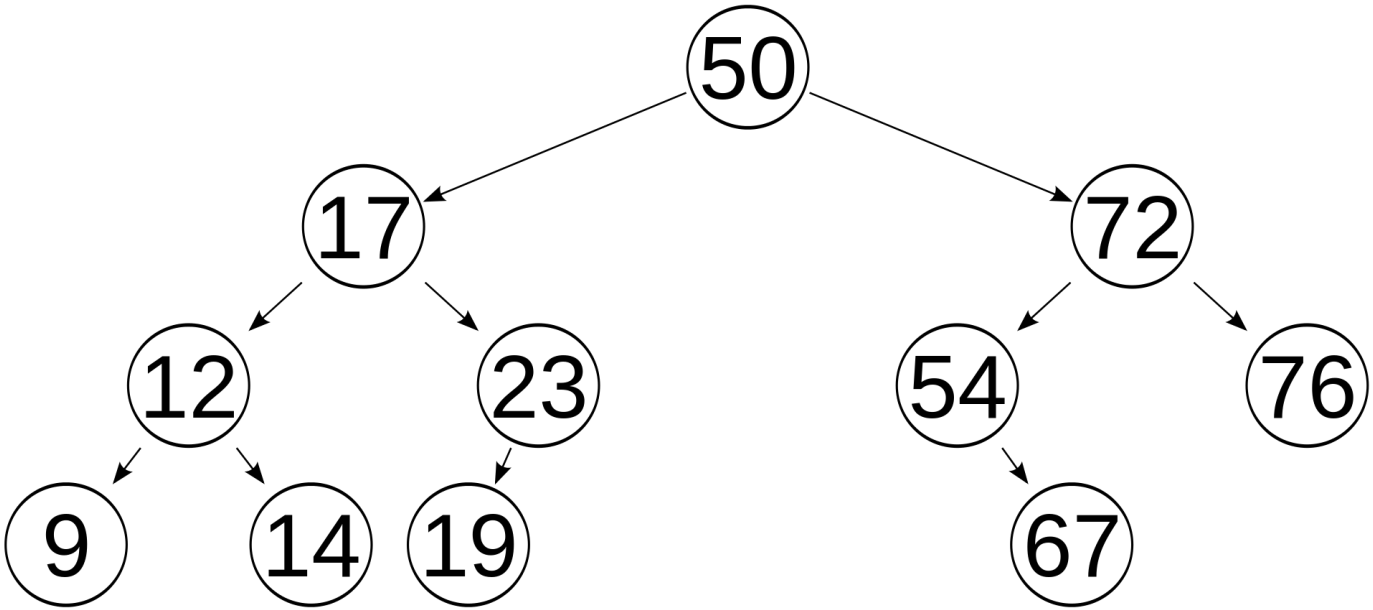


Figura: BST equilibrada.

Análise

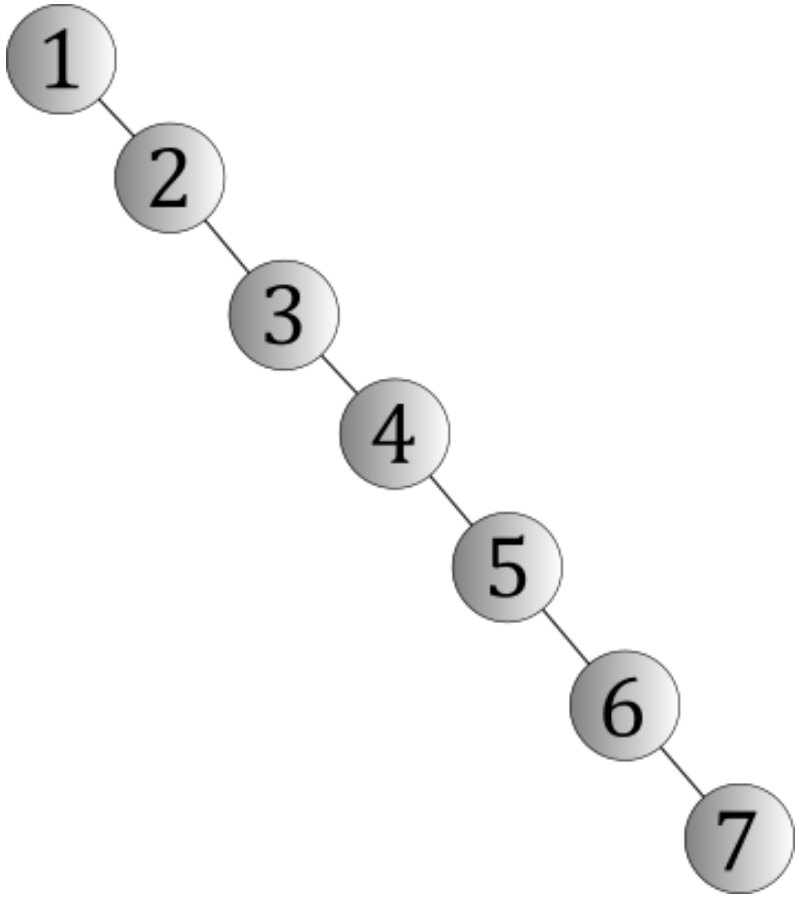
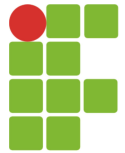
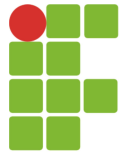


Figura: BST degenerada.

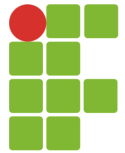


Análise

	Busca	Inserção	Remoção
Melhor caso	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(\lg n)$
Pior caso	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

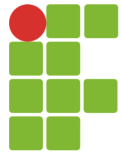


Sumário



Considerações

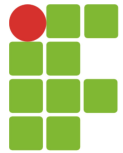
- BSTs conseguem representar elementos em um cenário dinâmico.
- A busca é muito eficiente quando a árvore está "equilibrada".



Considerações

Problemas com BSTs

- Problema: dependendo da ordem de inserções/remoções, a BST subjacente pode tornar-se degenerada.
- Operações começam a levar tanto tempo quanto em listas...



Considerações

Solução

- Árvores Binárias de Pesquisa **Balanceadas**;
- Continuam sendo BSTs, mas utilizam operações que tentam espalhar os itens da árvore de modo a deixá-la balanceada de acordo com algum critério, como altura, peso , *etc* ...
- Em nosso curso de Estruturas de Dados, veremos duas delas:
  - Árvore AVL;
  - Treap.