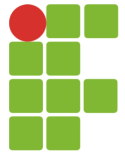
Árvores Binárias de Pesquisa

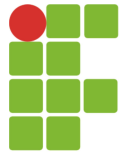
Estruturas de Dados e Algoritmos – Ciência da Computação

Prof. Daniel Saad Nogueira Nunes

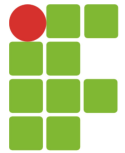
IFB – Instituto Federal de Brasília,
Campus Taguatinga



Sumário

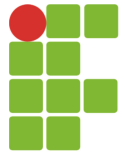


Sumário



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

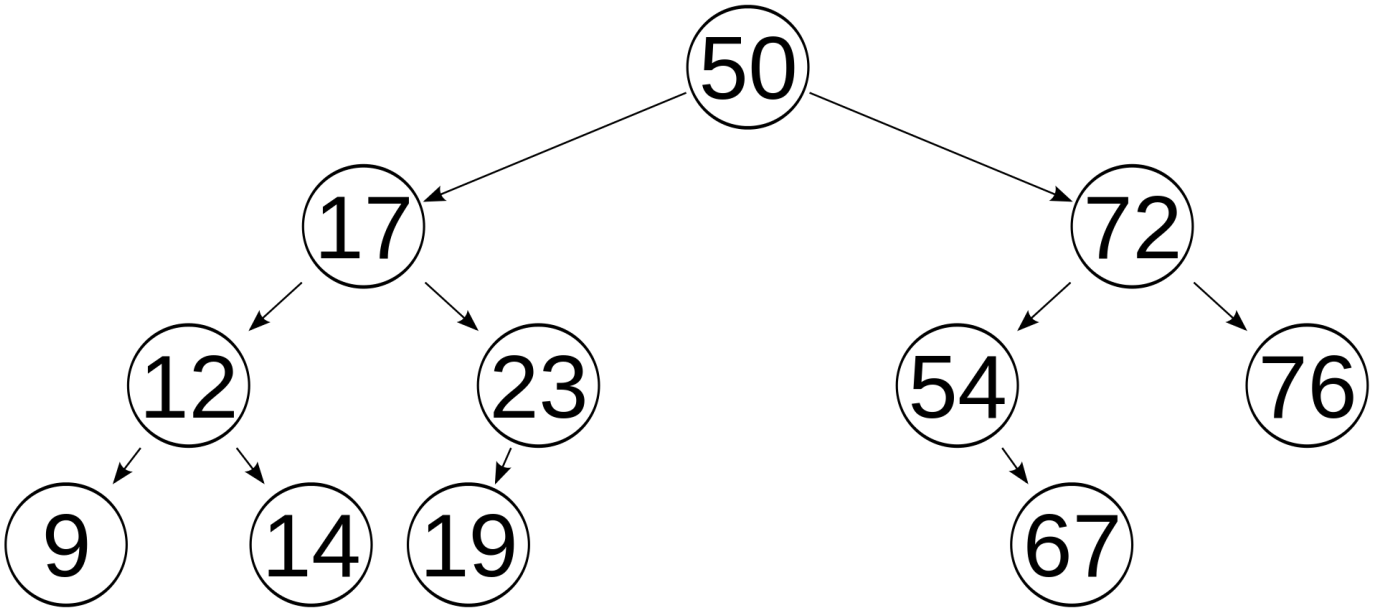
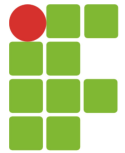
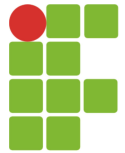


Figura: Árvores Binárias de Pesquisas

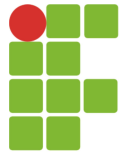


Sumário

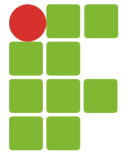


Operações

- É possível realizar buscas, inserções e remoções em uma BST.
- O único cuidado que devemos ter é não violar a propriedade de BST após uma remoção ou uma inserção.
- A busca não altera a árvore, portanto, não é necessário se preocupar com a violação da propriedade de BST nesse tipo de consulta.



Sumário



Busca em BSTs

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 em BSTs

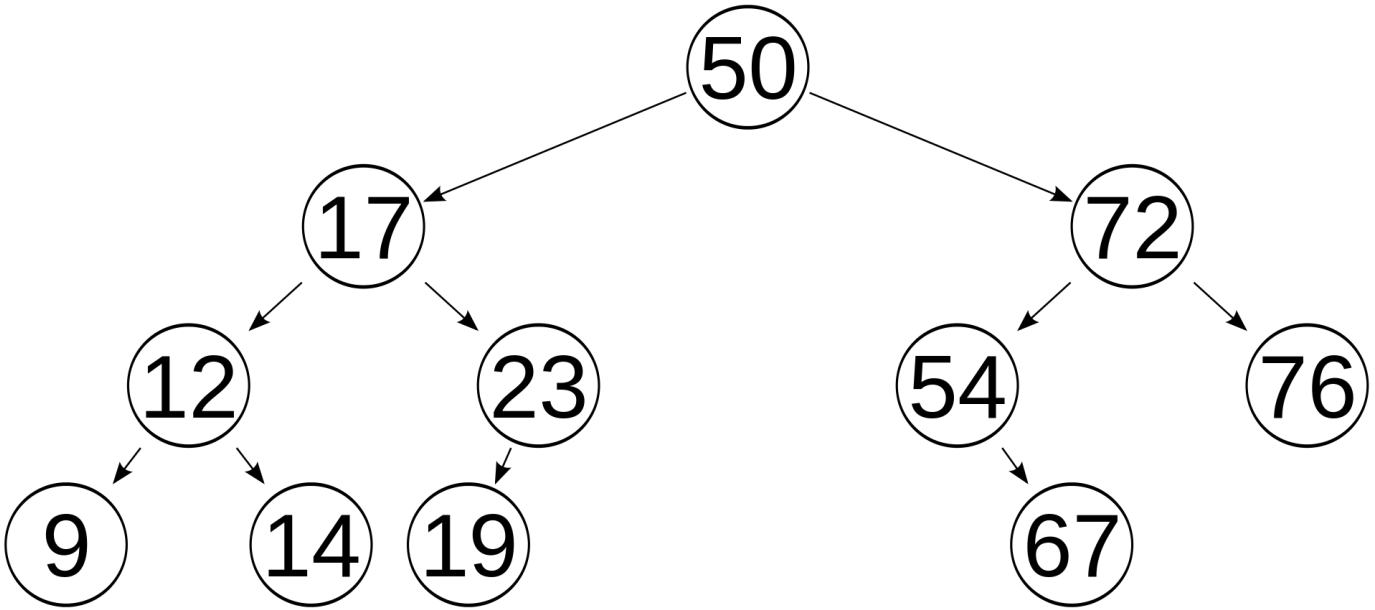


Figura: Busca Pelo 19.

Busca em BSTs

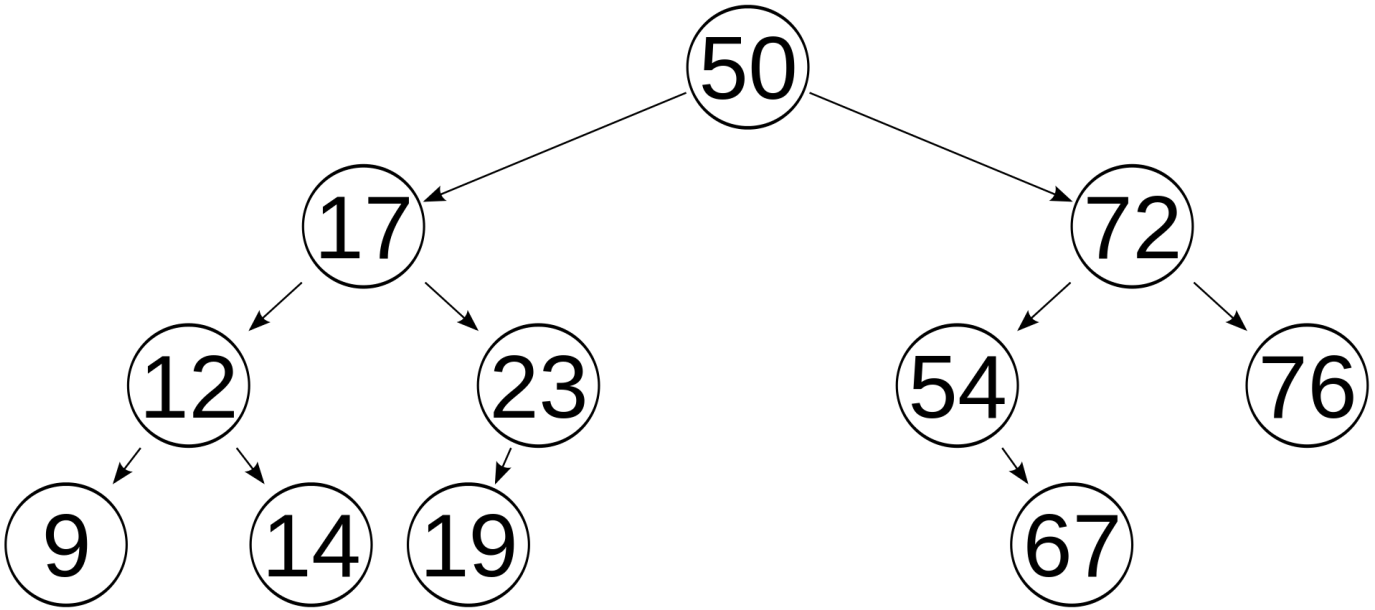


Figura: Busca Pelo 54.

Busca em BSTs

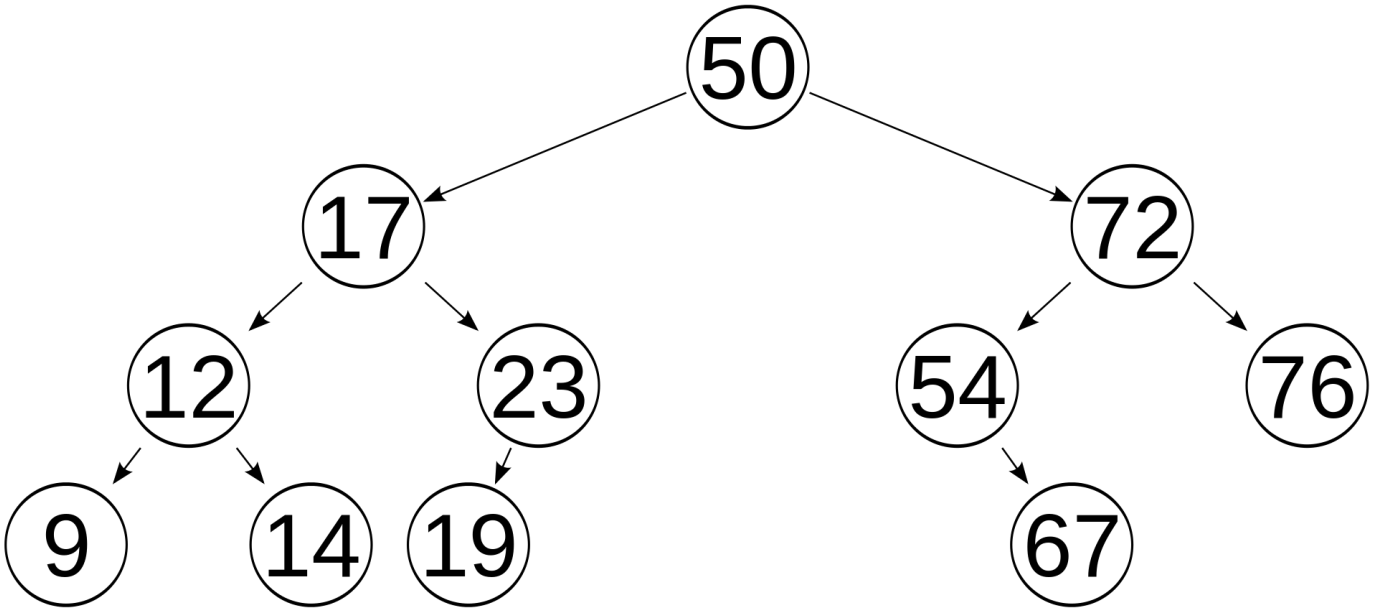
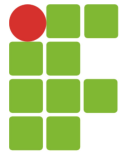
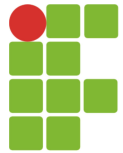


Figura: Busca Pelo 18.



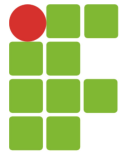
Busca em BSTs

▸ BST Applet



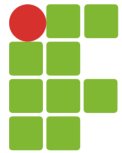
Busca em BSTs

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



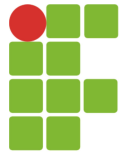
Busca em BSTs

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

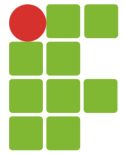


Busca em BSTs

- 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.

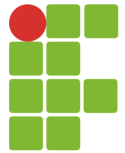


Sumário



Inserção em BSTs

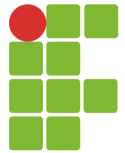
- BSTs são estruturas de dados dinâmicas, permitem inserções e remoções.
- Todas as inserções são feitas nas folhas.



Inserção em BSTs

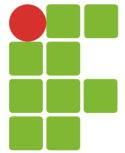
Durante a inserção de um nó com chave x em uma BST, temos os seguintes casos:

- 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.

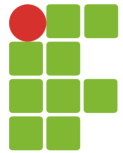


Inserção em BSTs

▸ BST Applet



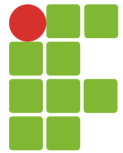
Sumário



Remoção

Remoção em BSTs

- 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?

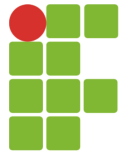


Remoção em BSTs

Caso 1

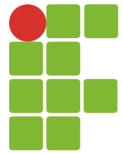
O nó a ser removido é uma folha.

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



Remoção

▸ BST Applet

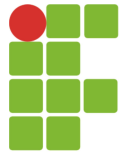


Remoção em BSTs

Caso 2

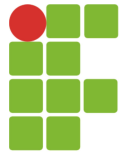
O nó a ser removido possui apenas um filho.

- 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 em BSTs

▸ BST Applet

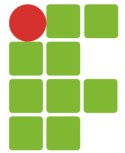


Remoção em BSTs

Caso 3

O nó a ser removido possui dois filhos.

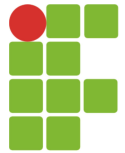
- Este é o caso mais difícil.
- Solução: transformar em um caso fácil.
- Obrigatoriamente o nó possui subárvores não vazias.
- Trocamos o nó pelo seu antecessor.
  - O antecessor encontra-se no nó mais à direita da subárvore da esquerda.
- Procede-se recursivamente na subárvore da esquerda para remover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



Remoção em BSTs

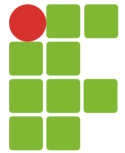
Caso 3

- Também é possível implementar de outra forma: trocando o nó pelo seu sucessor.
- O sucessor é o nó mais à esquerda na subárvore da direita.
- Nesta implementação, procedemos recursivamente na subárvore da **direita** para remover o nó desejado.

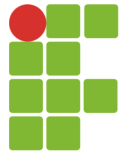


Remoção

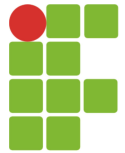
▸ BST Applet



Sumário



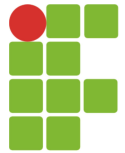
Sumário



Definição

Definição de um nó de BST.

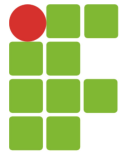
```
typedef struct bst_node_t {
    int data;
    struct bst_node_t *left;
    struct bst_node_t *right;
} bst_node_t;
```



Definição

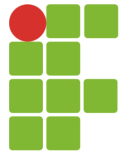
Definição da árvore. Só é necessário armazenar o ponteiro para a raiz.

```
typedef struct bst_t {
    bst_node_t *root;
    size_t size;
} bst_t;
```

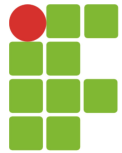


Definição

```
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;
```

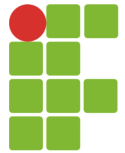


Sumário

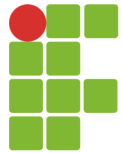


Inicialização

```
void bst_initialize(bst_t **t) {
    (*t) = mallocx(sizeof(bst_t));
    (*t)->size = 0;
    (*t)->root = NULL;
}
```



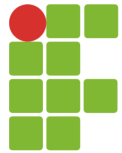
Sumário



Funções Auxiliares

Cria um novo nó e preenche com o dado.

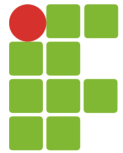
```
static bst_node_t *bst_new_node(int data) {
    bst_node_t *ptr = mallocx(sizeof(bst_node_t));
    ptr->left = NULL;
    ptr->right = NULL;
    ptr->data = data;
    return ptr;
}
```



Funções Auxiliares

Deleta um nó.

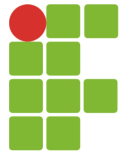
```
static void bst_delete_node(bst_node_t *node) {
    free(node);
}
```



Funções Auxiliares

Obtém o tamanho (número de nós) da árvore.

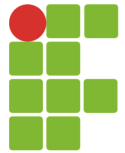
```
size_t bst_size(bst_t *t) {
    return t->size;
}
```



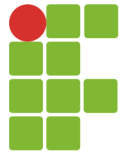
Funções Auxiliares

Devolve o nó mais à direita em uma subárvore com raiz em v.

```
static bst_node_t *bst_find_rightmost(bst_node_t *v) {
    if (v == NULL || v->right == NULL) {
        return v;
    } else {
        return bst_find_rightmost(v->right);
    }
}
```



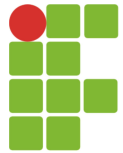
Sumário



Busca

Devolve verdadeiro se e somente se o nó com a chave estipulada encontra-se na árvore.

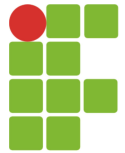
```
bool bst_find(bst_t *t, int data) {
    return bst_find_helper(t->root, data);
}
```



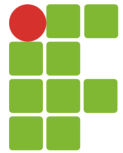
Busca

Devolve verdadeiro se e somente se o nó com a chave estipulada encontra-se na árvore.

```
static bool bst_find_helper(bst_node_t *x, int data) {
    if (x == NULL)
        return false;
    if (data == x->data)
        return true;
    if (data < x->data)
        return bst_find_helper(x->left, data);
    return bst_find_helper(x->right, data);
}
```



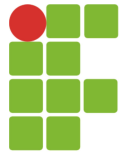
Sumário



Inserção

Insere um nó com a chave estipulada na BST. Premissa: a chave não existe na árvore.

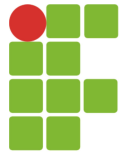
```
void bst_insert(bst_t *t, int data) {
    t->root = bst_insert_helper(t->root, data);
    t->size++;
}
```



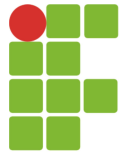
Inserção

Insere um nó com a chave estipulada na BST. Premissa: a chave não existe na árvore.

```
bst_node_t *bst_insert_helper(bst_node_t *x, int data) {
    if (x == NULL)
        return bst_new_node(data);
    assert(x->data != data);
    if (data < x->data) {
        x->left = bst_insert_helper(x->left, data);
    } else {
        x->right = bst_insert_helper(x->right, data);
    }
    return x;
}
```



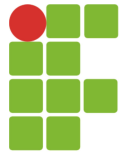
Sumário



Remoção

Removemos um nó com a chave estipulada na BST. Premissa: a chave deve existir na árvore.

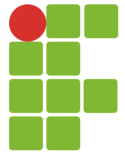
```
void bst_remove(bst_t *t, int data) {
    t->root = bst_remove_helper(t->root, data);
    t->size--;
}
```



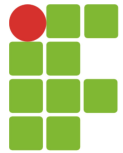
Remoção

Removemos um nó com a chave estipulada na BST. Premissa: a chave deve existir na árvore.

```
static bst_node_t *bst_remove_helper(bst_node_t *x, int data) {
    assert(x != NULL);
    if (data < x->data)
        x->left = bst_remove_helper(x->left, data);
    else if (data > x->data) {
        x->right = bst_remove_helper(x->right, data);
    } else {
        if (x->left == NULL) {
            bst_node_t *y = x->right;
            bst_delete_node(x);
            x = y;
        } else if (x->right == NULL) {
            bst_node_t *y = x->left;
            bst_delete_node(x);
            x = y;
        } else {
            bst_node_t *previous_x = bst_find_rightmost(x->left);
            int aux = x->data;
            x->data = previous_x->data;
            previous_x->data = aux;
            x->left = bst_remove_helper(x->left, data);
        }
    }
    return x;
}
```

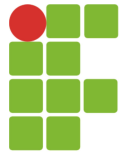


Sumário



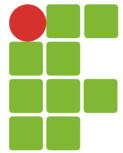
Limpeza

- 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.



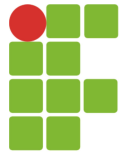
Limpeza

```
void bst_delete(bst_t **t) {
    bst_delete_helper((*t)->root);
    free(*t);
    (*t) = NULL;
}
```

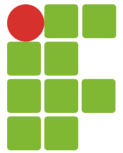


Limpeza

```
static void bst_delete_helper(bst_node_t *x) {
    if (x != NULL) {
        bst_delete_helper(x->left);
        bst_delete_helper(x->right);
        bst_delete_node(x);
    }
}
```



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

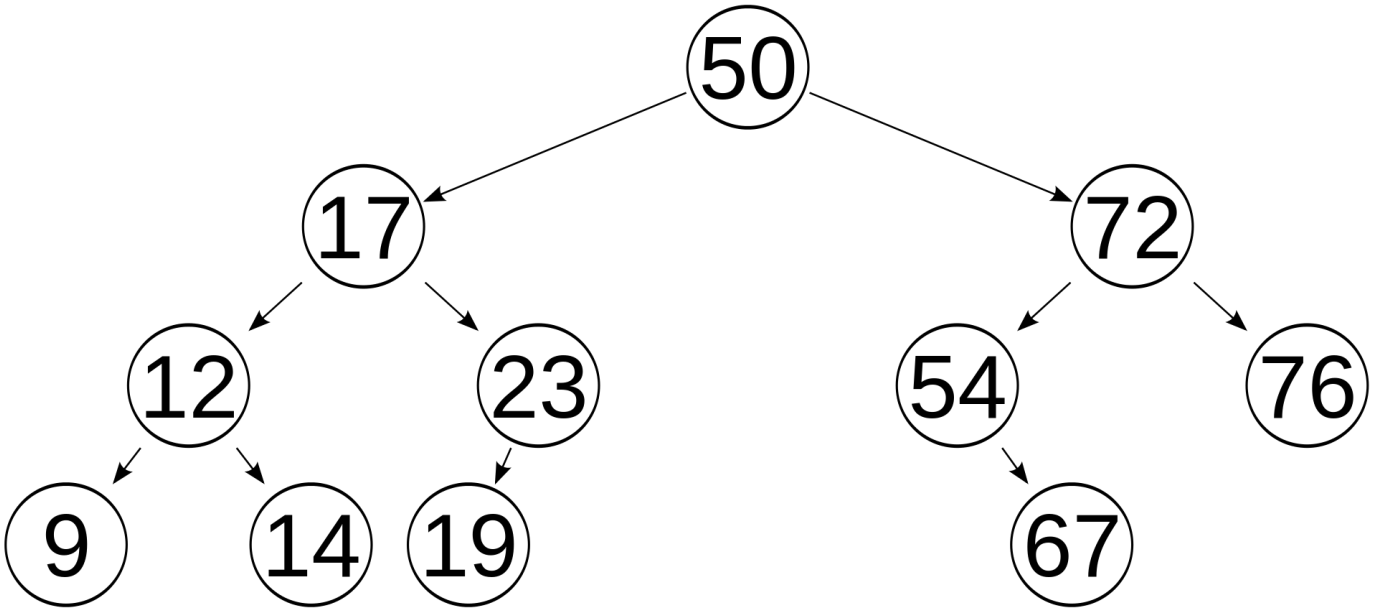
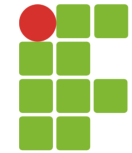


Figura: BST equilibrada.



Análise

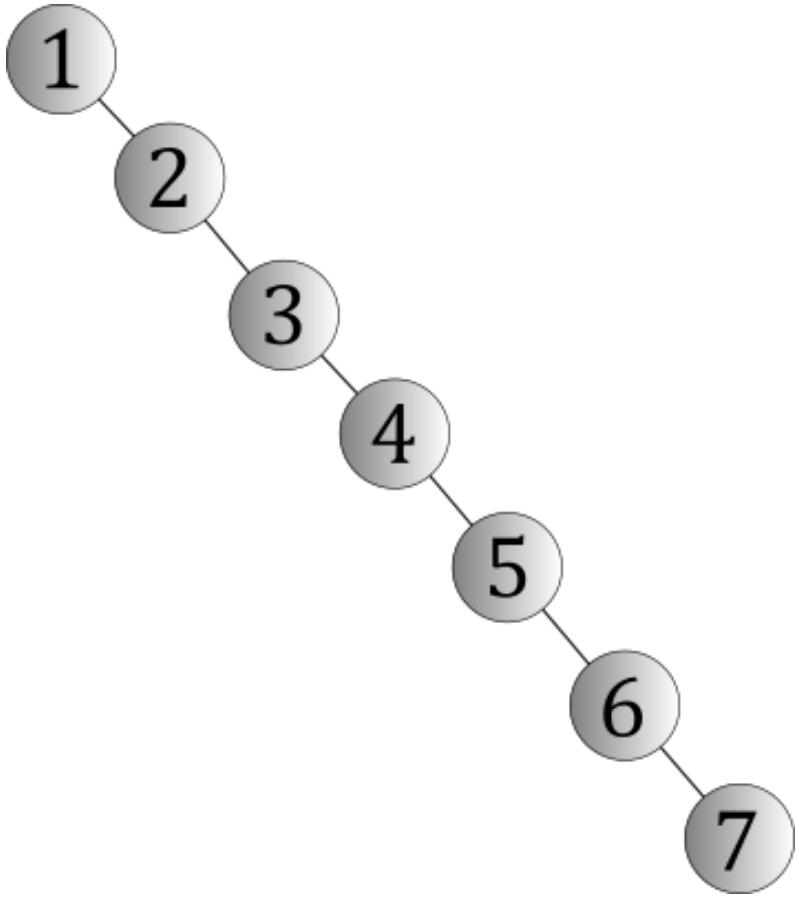
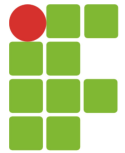
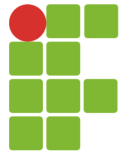


Figura: BST degenerada.

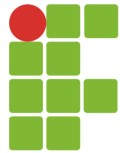


Análise

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

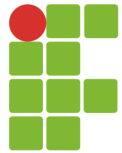


Sumário



Considerações

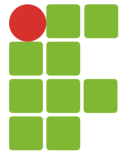
- BSTs conseguem representar elementos em um cenário dinâmico.
- A busca é muito eficiente quando a árvore está "equilibrada".



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* . . .