

Árvores

Árvores Binárias de Busca: Inserção e Remoção

Prof. Edson Alves - UnB/FGA

2018

1. Inserção de elementos em árvores binárias de busca
2. Remoção de elementos em árvores binárias de busca

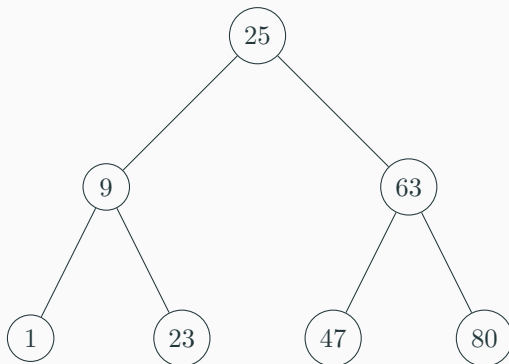
Inserção de elementos em árvores binárias de busca

Inserção em árvores binárias de busca

- O algoritmo a seguir insere um elemento x em uma árvore binária de busca:
 1. Comece no nó raiz
 2. Enquanto o nó a ser avaliado for não-nulo:
 - i. seja y a informação armazenada no nó a ser avaliado
 - ii. se x for menor do que y , vá para a raiz da subárvore da esquerda
 - iii. caso contrário, vá para a raiz da subárvore da direita
 3. Insira um novo nó com a informação igual ao valor a ser inserido como filho do último nó não-nulo, na posição adequada
- No pior caso, o algoritmo visita todos os N nós da árvore, de modo que este algoritmo tem complexidade $O(N)$

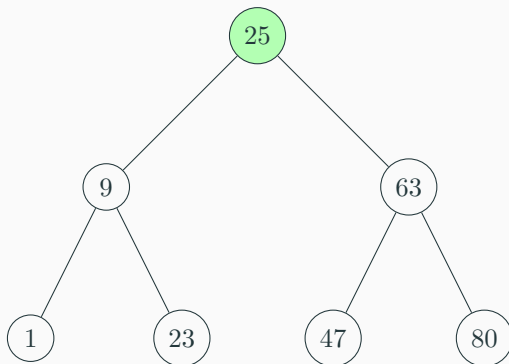
Exemplo de inserção em árvore binária de busca

Elemento a ser inserido: 14



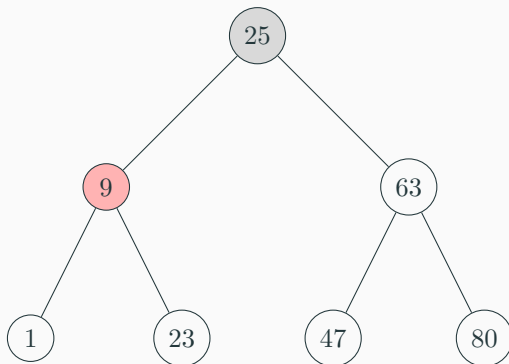
Exemplo de inserção em árvore binária de busca

Elemento a ser inserido: 14



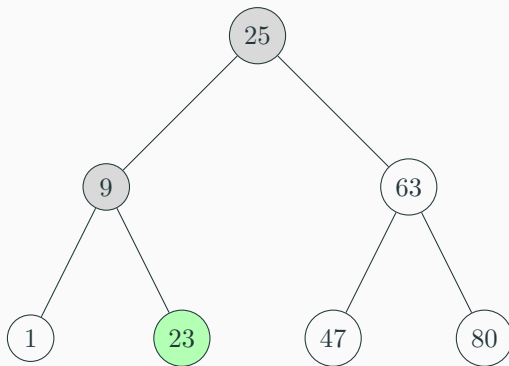
Exemplo de inserção em árvore binária de busca

Elemento a ser inserido: 14



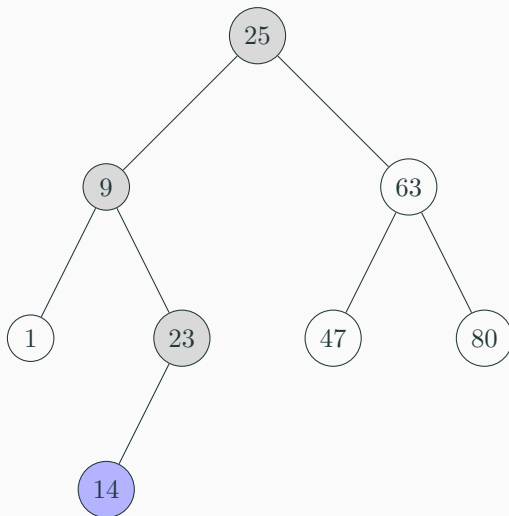
Exemplo de inserção em árvore binária de busca

Elemento a ser inserido: 14



Exemplo de inserção em árvore binária de busca

Elemento a ser inserido: 14



Implementação da inserção em uma BST

```
1 template<typename T>
2 class BST {
3 private:
4     struct Node {
5         T info;
6         Node *left, *right;
7     };
8
9     Node *root;
10
11 public:
12     BST() : root(nullptr) {}
13
14     void insert(const T& info)
15     {
16         Node *node = root, *prev = nullptr;
17
18         while (node)
19         {
20             prev = node;
21
```

Implementação da inserção em uma BST

```
22         if (node->info == info)
23             return;
24         else if (info < node->info)
25             node = node->left;
26         else
27             node = node->right;
28     }
29
30     node = new Node { info, nullptr, nullptr };
31
32     if (!root)
33         root = node;
34     else if (info < prev->info)
35         prev->left = node;
36     else
37         prev->right = node;
38 }
39 };
```

Notas sobre a inserção

- A inserção não modifica a estrutura da árvore, exceto no que se refere a acomodação do novo elemento.
- Deste modo, a propriedade da árvore binária de busca (BST) fica preservada
- O algoritmo que localiza o nó onde ocorrerá a inserção é semelhante ao código utilizado para buscar elementos na árvore
- A inserção pode desbalancear a árvore, isto é, pode fazer com que em um determinado nó, uma das subárvores tenha um número de nós significativamente maior do que a outra
- A inserção de um série de elementos em ordem crescente ou decrescente leva a uma árvore desbalanceada degenerada, que tem mesma estrutura de uma lista encadeada
- Esta árvore degenerada configura o pior caso do algoritmo

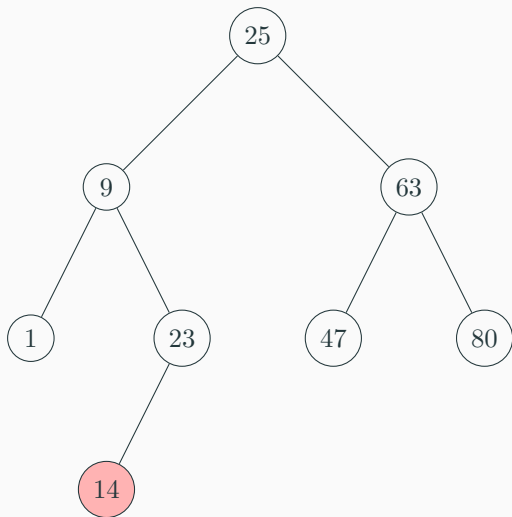
Remoção de elementos em árvores binárias de busca

Remoção em árvores binárias de busca

- A remoção em árvores binárias depende da posição do nó a ser removido
- São três casos:
 1. o nó é uma folha, isto é, não tem filhos
 2. o nó tem um filho
 3. o nó tem dois filhos
- No primeiro caso, basta remover a referência do pai e remover o nó
- No segundo caso, a referência do pai é alterada para apontar para neto, e o nó é removido
- O terceiro caso não pode ser resolvido em um único passo
- Duas possíveis soluções são a remoção por fusão ou a remoção por cópia

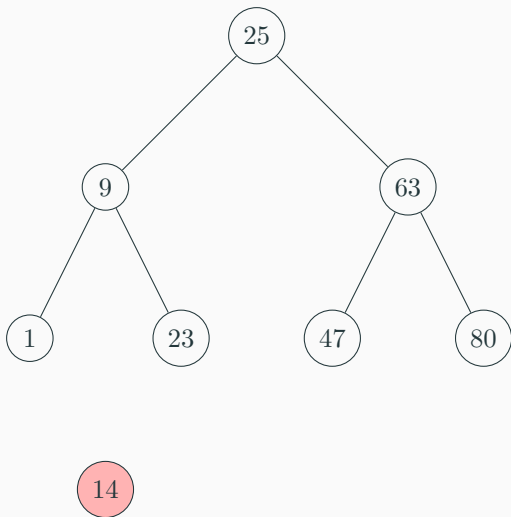
Exemplo de remoção de nó sem filhos

Elemento a ser removido: 14



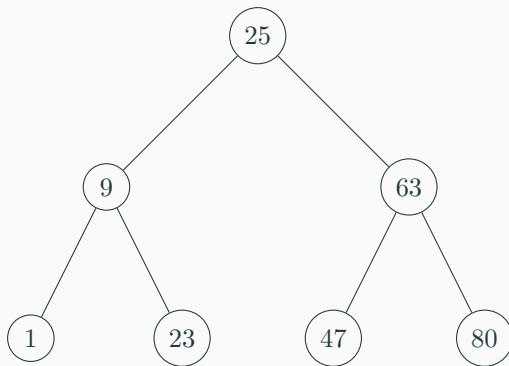
Exemplo de remoção de nó sem filhos

Elemento a ser removido: 14



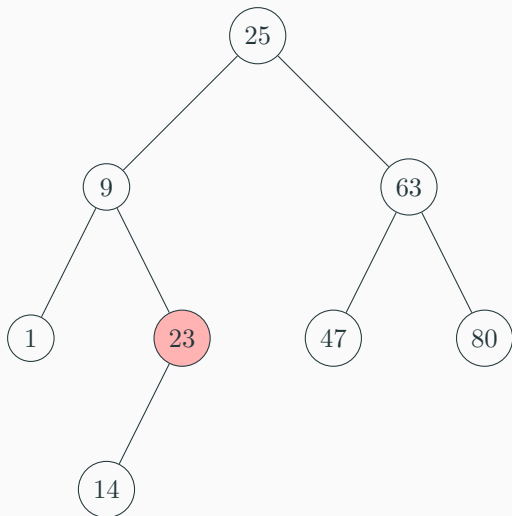
Exemplo de remoção de nó sem filhos

Elemento a ser removido: 13



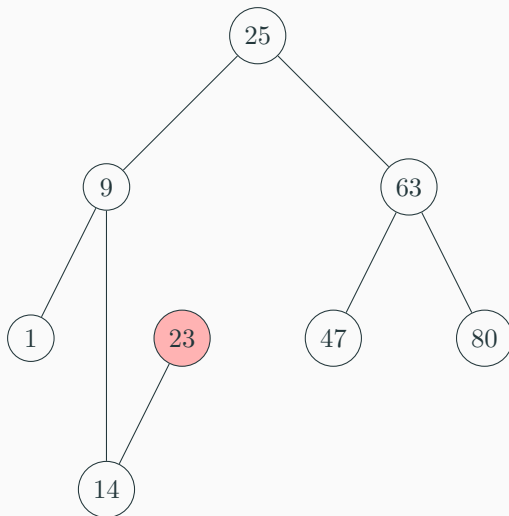
Exemplo de remoção de nó com apenas um filho

Elemento a ser removido: 23



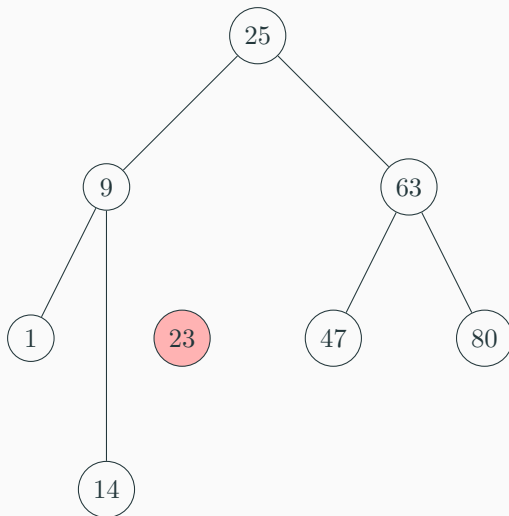
Exemplo de remoção de nó com apenas um filho

Elemento a ser removido: 23



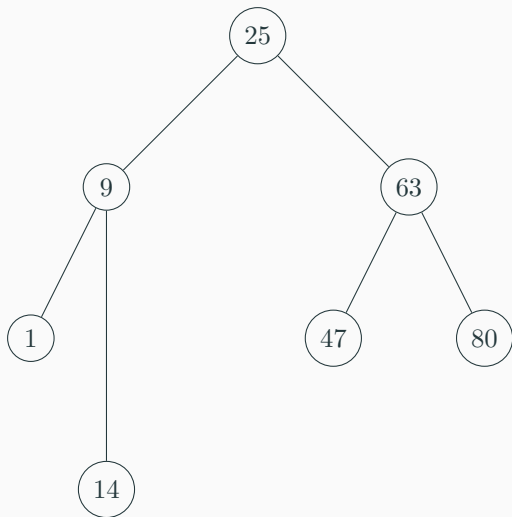
Exemplo de remoção de nó com apenas um filho

Elemento a ser removido: 23



Exemplo de remoção de nó com apenas um filho

Elemento a ser removido: 23

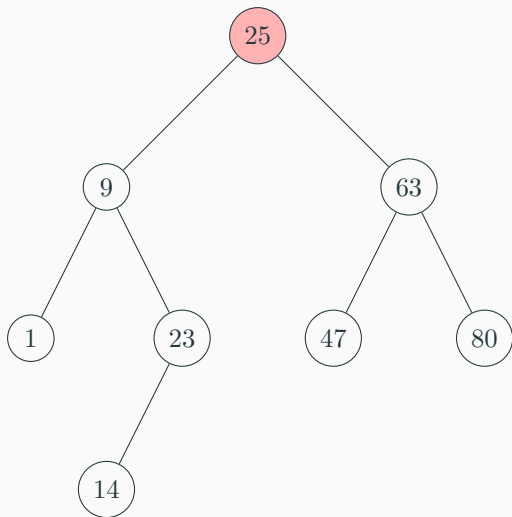


Remoção por fusão

- Esta técnica consiste em gerar uma nova árvore a partir das duas subárvores do nó a ser removido
- Numa árvore binária de busca, qualquer elemento da subárvore à direita é maior do que qualquer elemento da subárvore à esquerda
- Desta maneira, basta encontrar o nó mais à direita da subárvore à esquerda e transformá-lo no pai da subárvore à direita
- São quatro passos para a remoção por fusão:
 1. Localize o nó que deve ser removido (com dois filhos) e seu pai
 2. Na subárvore à esquerda, encontre o elemento mais à direita possível: basta mover-se sempre para a direita até que se encontre um nó nulo
 3. Torne a raiz da subárvore à esquerda o novo filho do pai do nó a ser removido
 4. Faça com que o nó mais à direita da subárvore à esquerda tenha como filho à direita a subárvore à direita do nó a ser removido
- *Corner case*: caso o nó seja a raiz, após a remoção a raiz deve apontar para a raiz da subárvore à esquerda

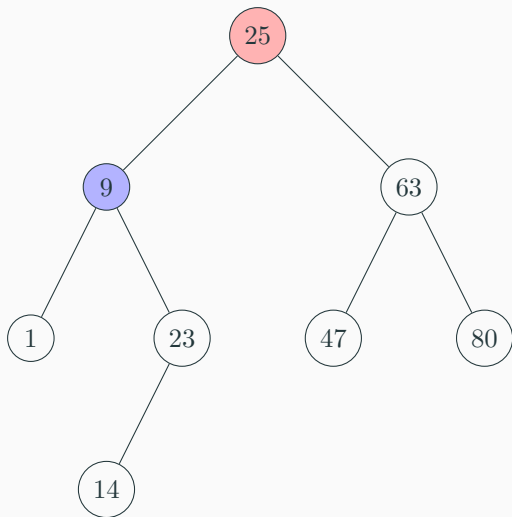
Exemplo de remoção por fusão

Elemento a ser removido: 25



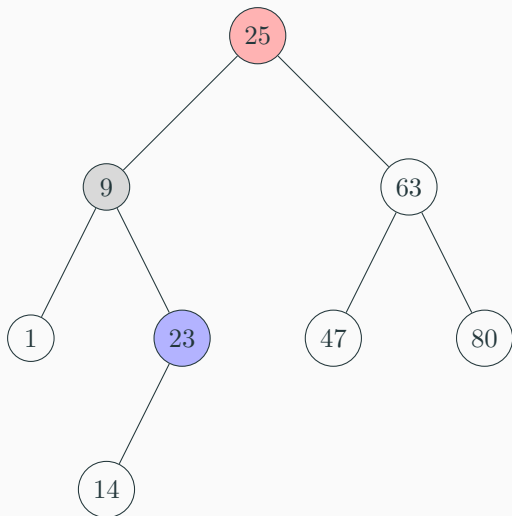
Exemplo de remoção por fusão

Elemento a ser removido: 25



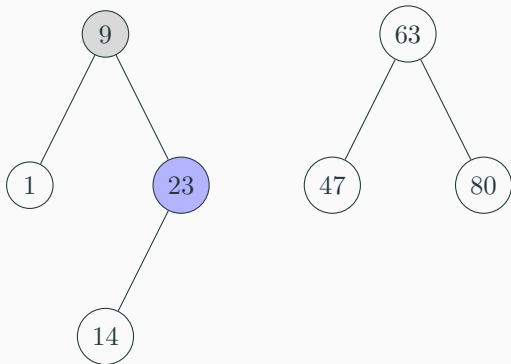
Exemplo de remoção por fusão

Elemento a ser removido: 25



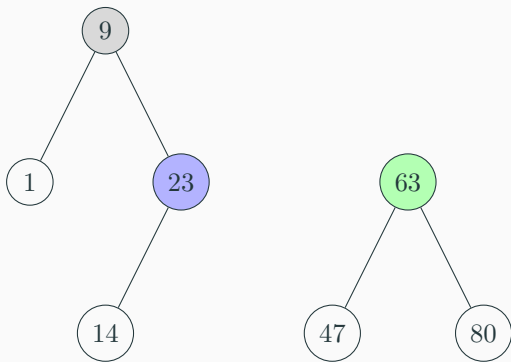
Exemplo de remoção por fusão

Elemento a ser removido: 25



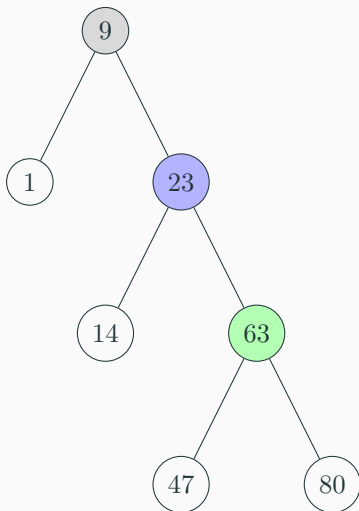
Exemplo de remoção por fusão

Elemento a ser removido: 25



Exemplo de remoção por fusão

Elemento a ser removido: 25



Implementação da remoção por fusão

```
1 template<typename T>
2 class BST {
3 private:
4     struct Node {
5         T info;
6         Node *left, *right;
7     };
8
9     Node *root;
10
11     void delete_by_merging(Node** n)
12     {
13         auto node = *n;
14
15         if (node == nullptr) return;
16
17         if (node->right == nullptr)           // Casos 1 e 2
18             *n = node->left;
19         else if (node->left == nullptr)       // Caso 2
20             *n = node->right;
```

Implementação da remoção por fusão

```
21     else {                                     // Caso 3
22         auto temp = node->left;
23
24         while (temp->right)
25             temp = temp->right;
26
27         temp->right = node->right;
28         *n = node->left;
29     }
30
31     delete node;
32 }
33
34 public:
35     BST() : root(nullptr) {}
36
```

Implementação da remoção por fusão

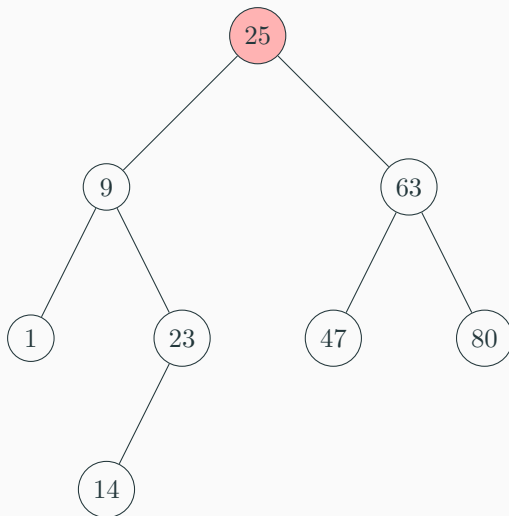
```
37 void erase(const T& info)
38 {
39     Node** node = &root;
40
41     while (*node)
42     {
43         if ((*node)->info == info)
44             break;
45
46         if (info < (*node)->info)
47             node = &(*node)->left;
48         else
49             node = &(*node)->right;
50     }
51
52     delete_by_merging(node);
53 }
54 };
```

Remoção por cópia

- Algoritmo proposto por Donald Knuth e Thomas Hibbard
- Ele reduz o caso de um nó ter dois filhos para um dos casos anteriores: ou o nó não tem filhos ou tem apenas um
- Isto é feito substituindo a informação do nó a ser deletado pela informação do nó mais à direita da subárvore à esquerda, apagando este nó em seguida
- Os quatro passos da remoção por cópia são:
 1. Localize o nó com dois filhos que deve ser removido
 2. Na subárvore à esquerda, encontre o elemento mais à direita possível
 3. Substitua a informação do nó a ser removido pela informação do nó localizado no passo anterior
 4. Remova o nó localizado no segundo passo

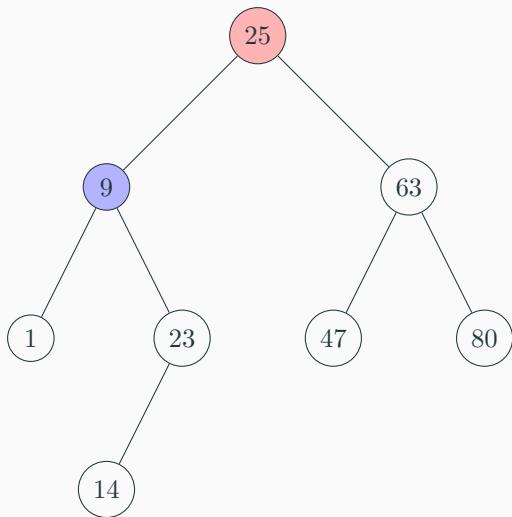
Exemplo de remoção por cópia

Elemento a ser removido: 25



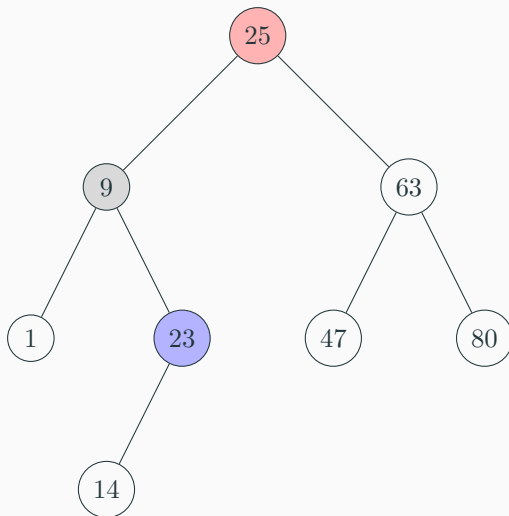
Exemplo de remoção por cópia

Elemento a ser removido: 25



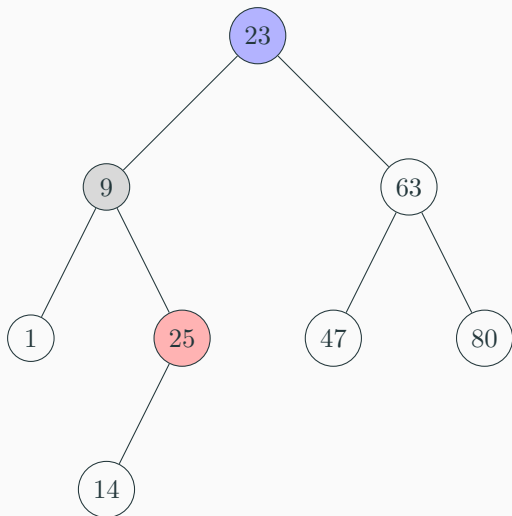
Exemplo de remoção por cópia

Elemento a ser removido: 25



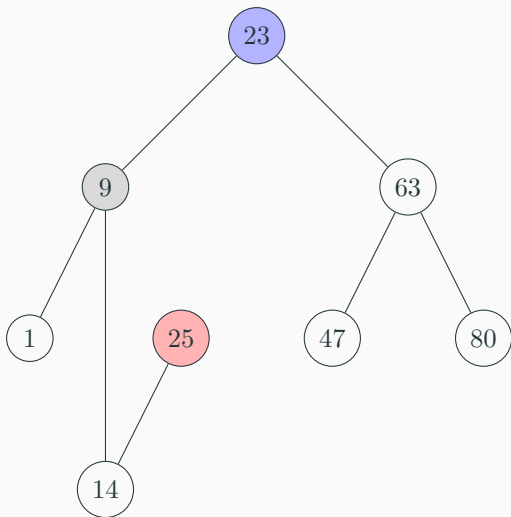
Exemplo de remoção por cópia

Elemento a ser removido: 25



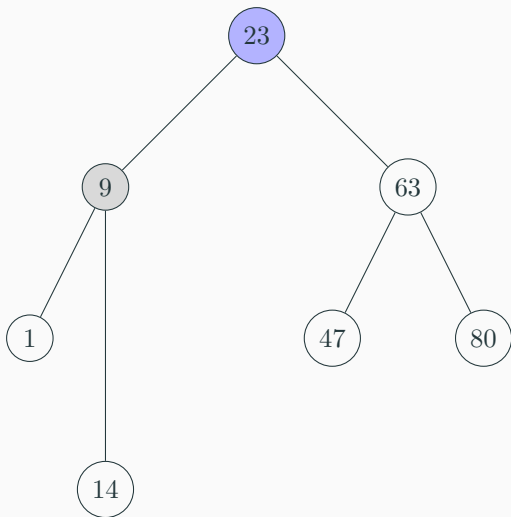
Exemplo de remoção por cópia

Elemento a ser removido: 25



Exemplo de remoção por cópia

Elemento a ser removido: 25



Implementação da remoção por cópia

```
1 template<typename T>
2 class BST {
3 private:
4     struct Node {
5         T info;
6         Node *left, *right;
7     };
8
9     Node *root;
10
11     void delete_by_copying(Node** n)
12     {
13         auto node = *n;
14
15         if (node == nullptr) return;
16
17         if (node->right == nullptr)           // Casos 1 e 2
18             *n = node->left;
19         else if (node->left == nullptr)       // Caso 2
20             *n = node->right;
```

Implementação da remoção por cópia

```
21     else {                                     // Caso 3
22         auto temp = &(*n)->left;
23
24         while ((*temp)->right)
25             temp = &(*temp)->right;
26
27         node->info = (*temp)->info;
28         return delete_by_copying(temp);
29     }
30
31     delete node;
32 }
33
34
35 public:
36     BST() : root(nullptr) {}
```


Implementação da remoção por cópia

```
37
38 void erase(const T& info)
39 {
40     Node** node = &root;
41
42     while (*node)
43     {
44         if ((*node)->info == info)
45             break;
46
47         if (info < (*node)->info)
48             node = &(*node)->left;
49         else
50             node = &(*node)->right;
51     }
52
53     delete_by_copying(node);
54 }
55 };
```

Notas sobre os algoritmos de remoção

- De forma semelhante à inserção, ambos algoritmos de remoção em complexidade $O(N)$ no pior caso (folha de uma árvore degenerada com N nós)
- Observe que a remoção por fusão aumenta ou mantém a altura da árvore
- Já a remoção por cópia mantém ou diminui a altura da árvore
- Efetivamente, a complexidade de ambos algoritmos é $O(h)$, onde h é a altura da árvore
- Em uma árvore balanceada, $h = O(\log N)$, o que melhora a complexidade de ambos algoritmos
- Para manter uma árvore balanceada, a remoção por cópia é mais adequada do que a remoção por fusão
- Contudo, como já dito, a inserção pode desbalancear a árvore
- Logo é preciso alterar a inserção para que o balanceamento seja preservado

1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. **KERNIGHAN**, Bryan; **RITCHIE**, Dennis. *The C Programming Language*, 1978.
3. **STROUSTROUP**, Bjarne. *The C++ Programming Language*, 2013.
4. C++ Reference¹.

¹<https://en.cppreference.com/w/>