Árvores Múltiplas

Árvores-B

Prof. Edson Alves - UnB/FGA

2019

Sumário

Árvores-B

Árvores múltiplas

- Segundo a definição formal de árvores, não há restrição quanto ao número de filhos que um nó pode ter
- Uma árvore múltipla de ordem m é um árvore cujos nós possuem, no máximo, m filhos
- As árvores binárias de busca que são árvores múltiplas de ordem 2 que impõem condições sobre as chaves dos nós com o intuito de agilizar o processo de busca.
- As árvores binárias de busca podem ser generalizadas como árvores de busca de ordem m

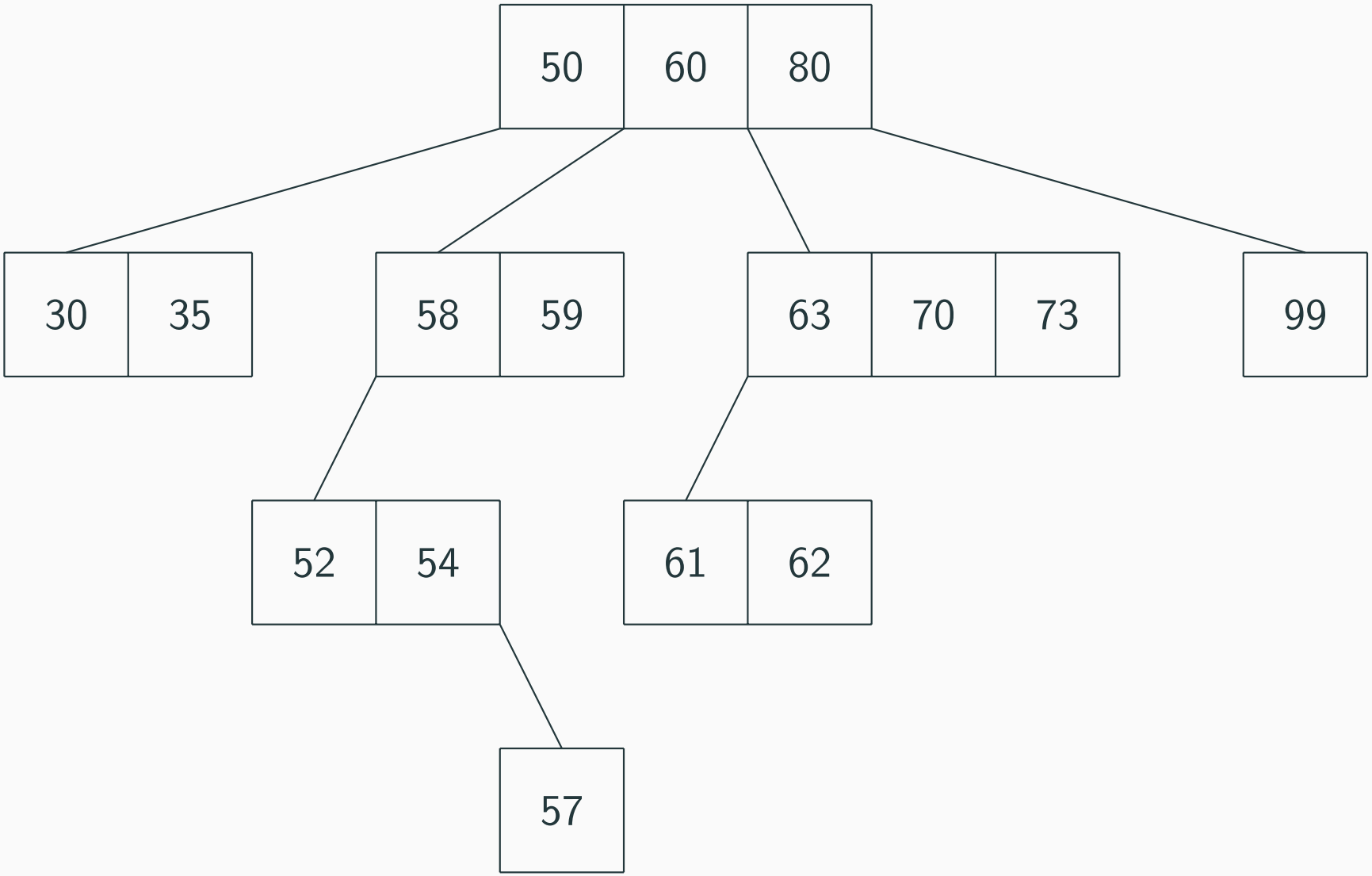
Árvores de busca de ordem m

Definição

Uma árvore de busca de ordem m é uma árvore que satisfaz as seguintes condições:

1. Cada nó tem, no máximo, m filhos e $m - 1$ chaves.
2. As chaves de cada nó são armazenadas em ordem crescente.
3. As chaves dos i primeiros filhos são menores do que a chave i.
4. As chaves dos $m - i$ últimos filhos são maiores do que a chave i.

Exemplo de árvore de busca de ordem 4



Notas sobre árvores de busca de ordem m

- As árvores de busca de ordem m tem o mesmo objetivo das árvores de busca binárias: aumentar a eficiência da rotina de busca
- Observe que, em cada nó, é preciso localizar, a partir da informação a ser encontrada e das chaves armazenadas, identificar o filho que dará sequência a busca
- A ordenação das chaves permite esta identificação em ordem $O(\log m)$, desde que o contêiner que armazena as chaves permita a busca binária
- Assim como as árvores binárias de busca, as árvores de busca de ordem m também podem ter problemas relativos ao balanceamento
- Para evitar tal problemas, existem especilizações destas árvores, como as árvores-B

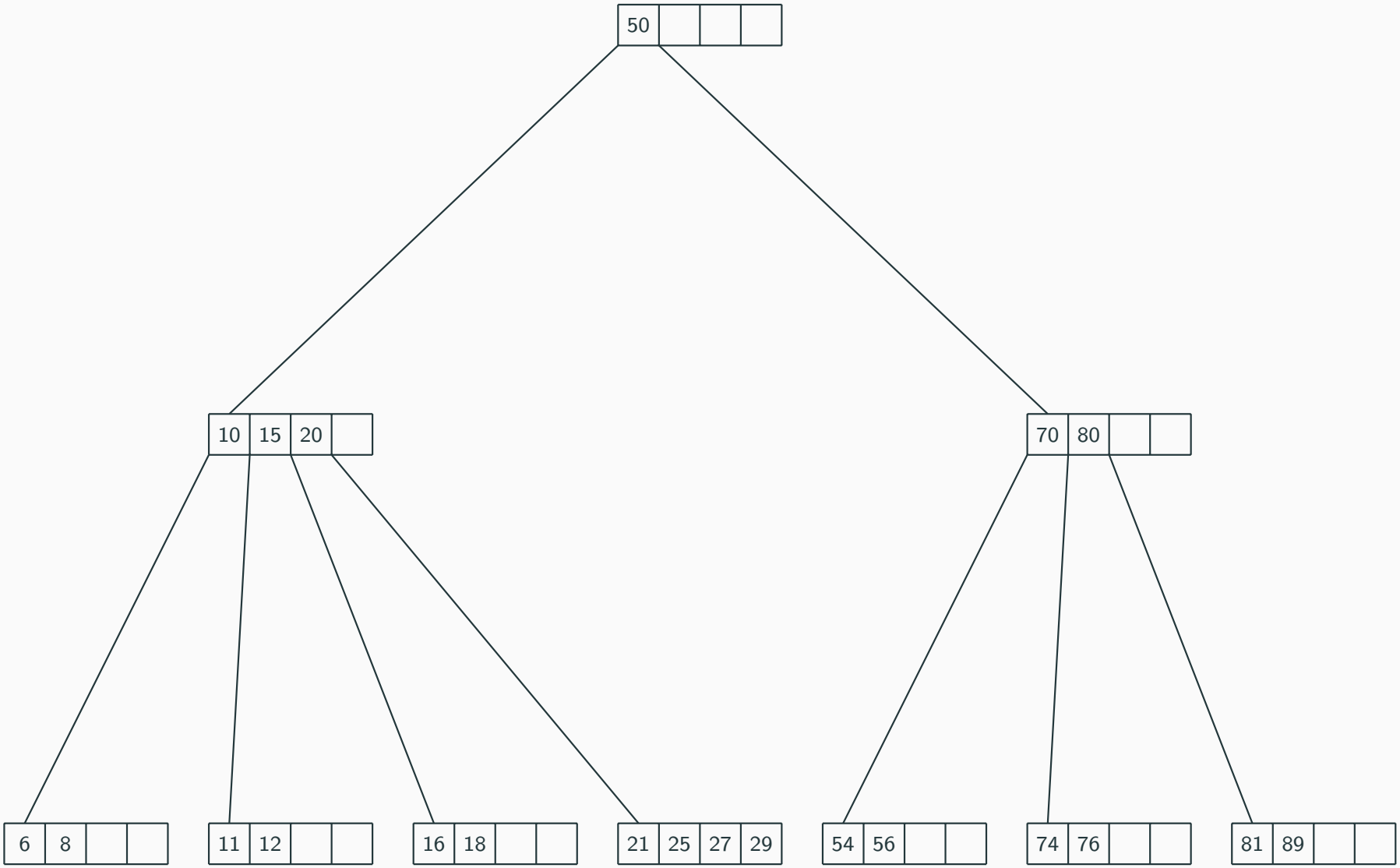
Definição de Árvores-B

Definição

Uma árvore-B de ordem m é uma árvore de busca de ordem m com as seguintes propriedades:

1. A raiz tem, no mínimo, dois filhos, caso não seja uma folha.
2. Cada nó que não é nem folha nem raiz tem $k-1$ chaves e k ponteiros para subárvores, onde $\lceil m/2 \rceil \leq k \leq m$.
3. Cada folha tem $k-1$ chaves, onde $\lceil m/2 \rceil \leq k \leq m$.
4. Todas as folhas estão no mesmo nível.

Exemplo de árvore-B de ordem 5



Árvores-B

- As árvores-B foram propostas por Bayer e McCreigh em 1972.
- O número de chaves armazenadas em uma árvore-B é proporcional a metade de sua capacidade máxima
- Devido às suas propriedades, uma árvore-B tem poucos níveis
- Uma árvore-B está sempre perfeitamente balanceada
- Um nó de uma árvore-B possui dois contêiners: um para armzenar as $m - 1$ chaves e outro para os m ponteiros para os filhos
- Na implementação dos nós de árvores-B costuma-se adicionar informações extras que facilitem a manutenção da árvore, como o número de chaves do nó e uma indicação se o nó é folha ou não

Exemplo de implementação de uma árvore-B

```cpp
#include <bits/stdc++.h>

using namespace std;

template<typename T, size_t M>
class BTree {
private:

    struct Node {
        bool leaf;
        Node *parent;
        vector<T> keys;
        vector<Node *> children;

        Node(bool is_leaf = true) : leaf(is_leaf), parent(nullptr) {}

        void sort_keys()
        {
            sort(keys.begin(), keys.end());
        }

```

Exemplo de implementação de uma árvore-B

```
22          void sort_children()
23          {
24              sort(children.begin(), children.end(),
25                  [](const Node *a, const Node *b) {
26                      if (a->keys.empty())
27                          return false;
28
29                      if (b->keys.empty())
30                          return true;
31
32                      return a->keys[0] < b->keys[0];
33                  });
34          }
35
36          // Índice da menor chave maior ou igual a key
37          size_t index(const T& key) const
38          {
39              auto it = lower_bound(keys.begin(), keys.end(), key);
40              return it - keys.begin();
41          }
42      };
```

Exemplo de implementação de uma árvore-B

```
44      Node *root;
45
46  public:
47      // A árvore é inicializada com um nó sem nenhuma chave armazenada
48      BTree() : root(new Node()) {}
49
50      // Complexidade O(log N log M)
51      bool find(const T& info) const
52      {
53          auto node = find(root, info);
54
55          return binary_search(node->keys.begin(), node->keys.end(), info);
56      }
57
58  private:
59      // Procura pelo nó onde a informação deveria estar
60      Node * find(Node *node, const T& info) const
61      {
62          if (node->leaf)
63              return node;
64
```

Exemplo de implementação de uma árvore-B

```
65          auto i = node->index(info);
66
67          if (i < node->keys.size() and node->keys[i] == info)
68              return node;
69
70          return find(node->children[i], info);
71      }
72
```

Inserção

Inserção em árvores-B

Há 3 casos a serem tratados na inserção de um elemento em uma árvore-B, uma vez localizado o nó onde deve ocorrer a inserção:

1. O nó é uma folha com espaço livre: A estrutura da árvore não é alterada. Pode ser necessário transpor algumas chaves para que se mantenha a ordem crescente das mesmas.
2. O nó é uma folha sem espaço livre: O nó deve ser dividido em dois nós. O novo nó deve receber a metade superior do antigo nó (já contabilizado o novo elemento), enquanto que a maior chave restante no antigo nó é migrada para o nó pai. Também deve-se adicionar uma referência ao novo nó no pai.
3. O nó é a raiz e ela está sem espaço livre: Deve-se proceder como no caso de uma folha cheia, dividindo o nó em dois. Deve-se criar um novo nó para ser a nova raiz, e este nó fará o papel do pai no caso anterior. Esta é a única inserção que pode alterar a altura da árvore.

Exemplo de inserção no primeiro caso, $m = 4$

Elemento a ser inserido: 50

Exemplo de inserção no primeiro caso, $m = 4$

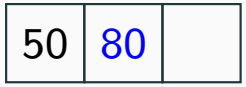
Elemento a ser inserido: 50

Exemplo de inserção no primeiro caso, $m = 4$

Elemento a ser inserido: 80

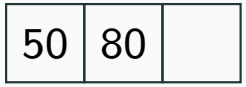
Exemplo de inserção no primeiro caso, $m = 4$

Elemento a ser inserido: 80



Exemplo de inserção no primeiro caso, $m = 4$

Elemento a ser inserido: 30



Exemplo de inserção no primeiro caso, $m = 4$

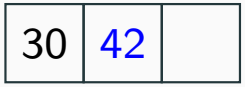
Elemento a ser inserido: 30

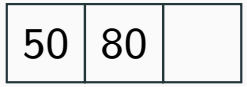
Exemplo de inserção no terceiro caso, $m = 4$

Elemento a ser inserido: 42

Exemplo de inserção no terceiro caso, $m = 4$

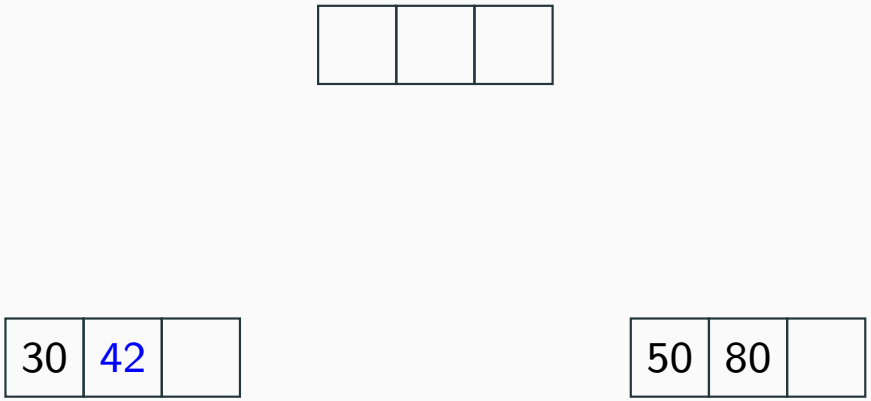
Elemento a ser inserido: 42





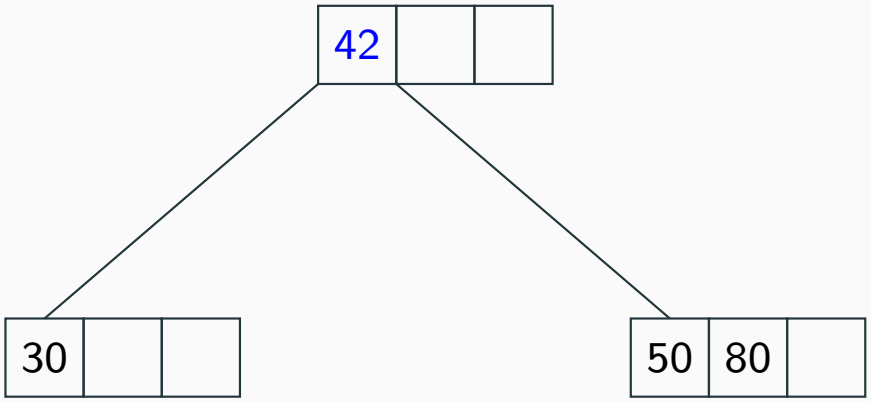
Exemplo de inserção no terceiro caso, $m = 4$

Fusão do nó dividido, nova raiz



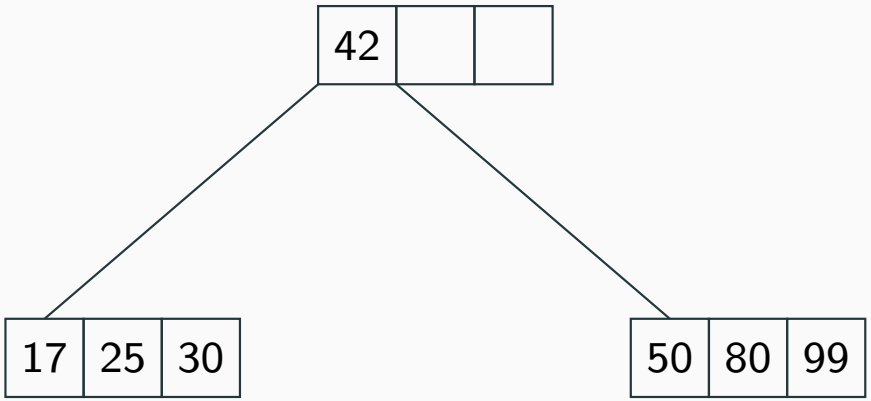
Exemplo de inserção no terceiro caso, $m = 4$

Fusão do nó dividido, nova raiz



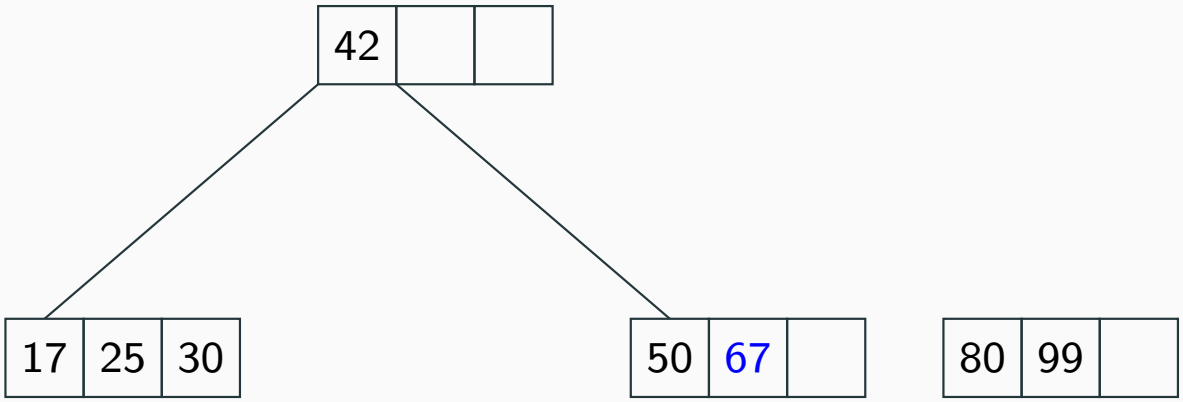
Exemplo de inserção no segundo caso, $m = 4$

Elemento a ser inserido: 67



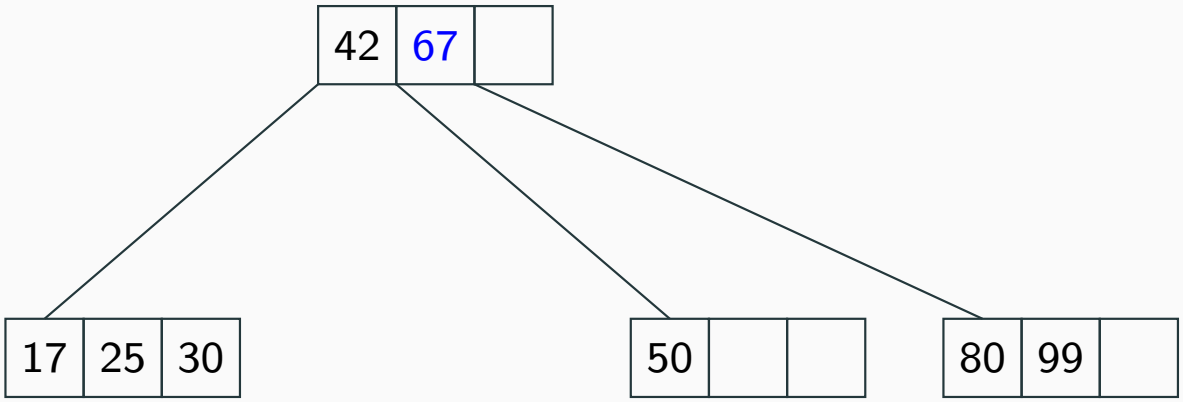
Exemplo de inserção no segundo caso, $m = 4$

Elemento a ser inserido: 67



Exemplo de inserção no segundo caso, $m = 4$

Elemento a ser inserido: 67



Implementação da inserção em uma árvore-B

```
73 public:
74     bool insert(const T& info)
75     {
76         // Não insere informações duplicadas
77         if (find(info))
78             return false;
79 
80         auto node = find(root, info);
81 
82         return insert(info, node);
83     }
84 
85 private:
86     bool insert(const T& info, Node *node, Node* child = nullptr)
87     {
88         // Insere a informação e o filho
89         node->keys.push_back(info);
90         node->sort_keys();
91 
```

Implementação da inserção em uma árvore-B

```
92          if (child)
93          {
94              node->children.push_back(child);
95              node->sort_children();
96          }
97
98          // Capacidade do nó superada: o nó deve ser dividido
99          if (node->keys.size() == M)
100         {
101             auto S = new Node(node->leaf);
102             auto half = M/2;
103
104             // Divide as chaves
105             for (size_t i = half; i < M; ++i)
106             {
107                 S->keys.push_back(node->keys.back());
108                 node->keys.pop_back();
109             }
110
111             reverse(S->keys.begin(), S->keys.end());
112
```

Implementação da inserção em uma árvore-B

```
113             // Determina o elemento do meio, que subirá para o pai
114             auto new_info = node->keys.back();
115             node->keys.pop_back();
116 
117             // Divide os filhos, se necessário
118             if (node->leaf == false)
119             {
120                 for (size_t i = 0; i <= S->keys.size(); ++i)
121                 {
122                     S->children.push_back(node->children.back());
123                     S->children.back()->parent = S;
124                     node->children.pop_back();
125                 }
126 
127                 reverse(S->children.begin(), S->children.end());
128             }
129 
```

Implementação da inserção em uma árvore-B

```
130             if (node->parent)
131             {
132                 S->parent = node->parent;
133                 return insert(new_info, node->parent, S);
134             } else
135             {
136                 root = new Node(false);
137 
138                 root->keys.push_back(new_info);
139                 root->children.push_back(node);
140                 root->children.push_back(S);
141 
142                 node->parent = root;
143                 S->parent = root;
144             }
145         }
146 
147         return true;
148     }
149 
```

Remoção

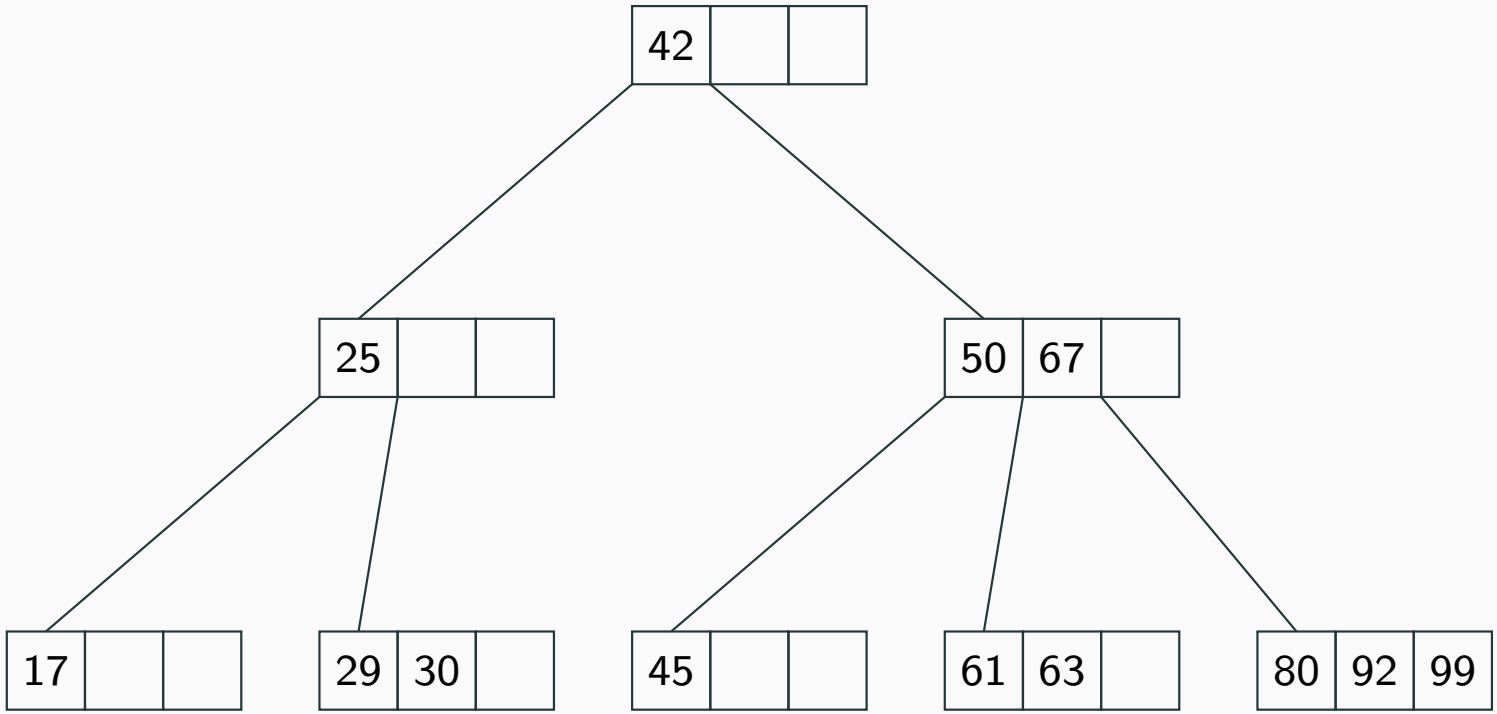
Remoção em árvores-B

Há 2 casos a serem tratados na remoção de uma chave em uma árvore-B, uma vez localizado o nó onde deve ocorrer a remoção:

1. O nó é uma folha:
   - 1.1 Após a remoção da chave, restantam ainda $M = \lceil m/2 \rceil - 1$ chaves ou mais
   - 1.2 Após a remoção da chave, a folha possui menos do que M chaves (*underflow*)
     - 1.2.1 Se o irmão à direita ou à esquerda com mais do que M chaves, as chaves são distribuídas entre o nó e seu irmão, passando a chave do pai para o nó e a chave apropriada do irmão para o pai
     - 1.2.2 Se os irmãos à direita e à esquerda possuem o mínimo M de chaves, o nó é fundido com um de seus irmãos: as chaves do nó, a chave do pai e as chaves do irmão formam um novo nó, e o irmão é descartado
2. O nó é não é uma folha: este pode ser reduzido ao caso anterior trocando o elemento a ser removido com seu sucessor (ou predecessor) que se encontra em uma folha

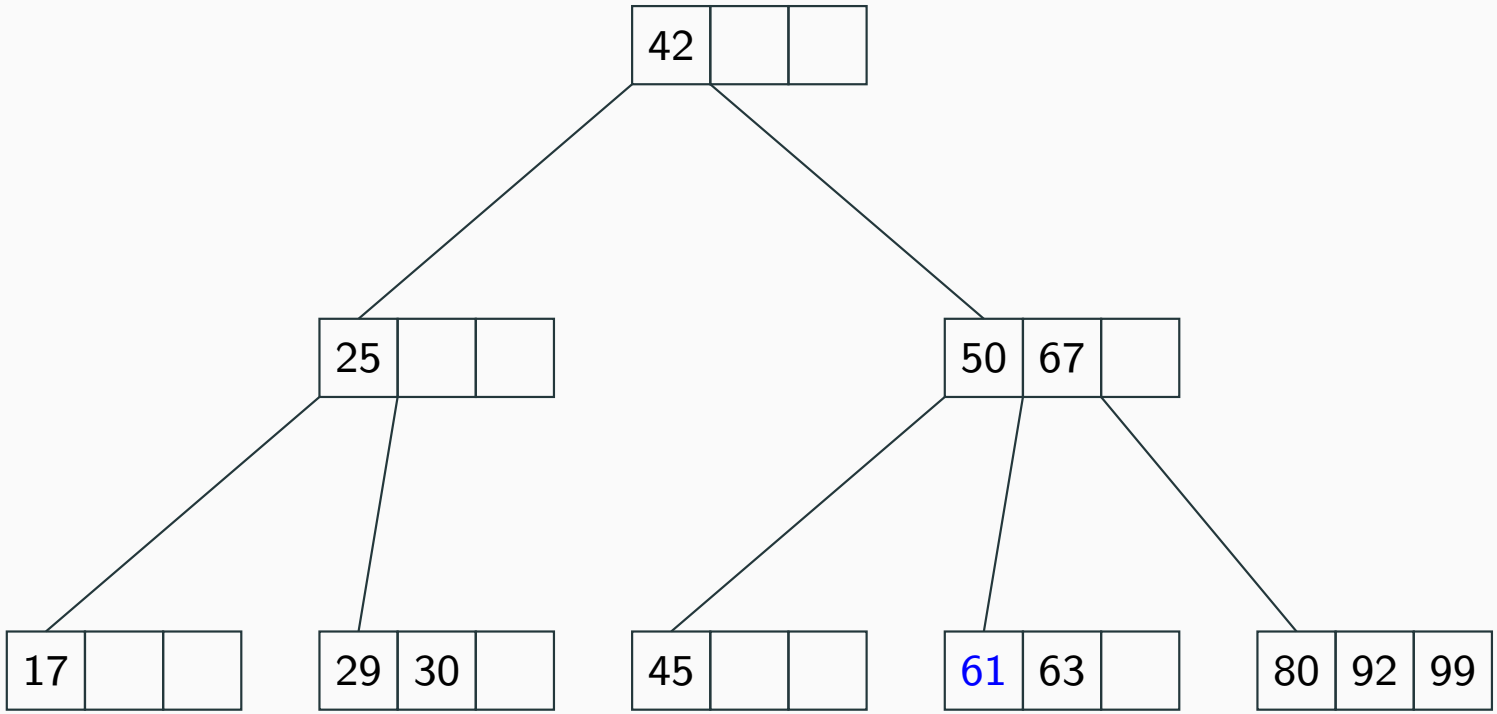
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 61



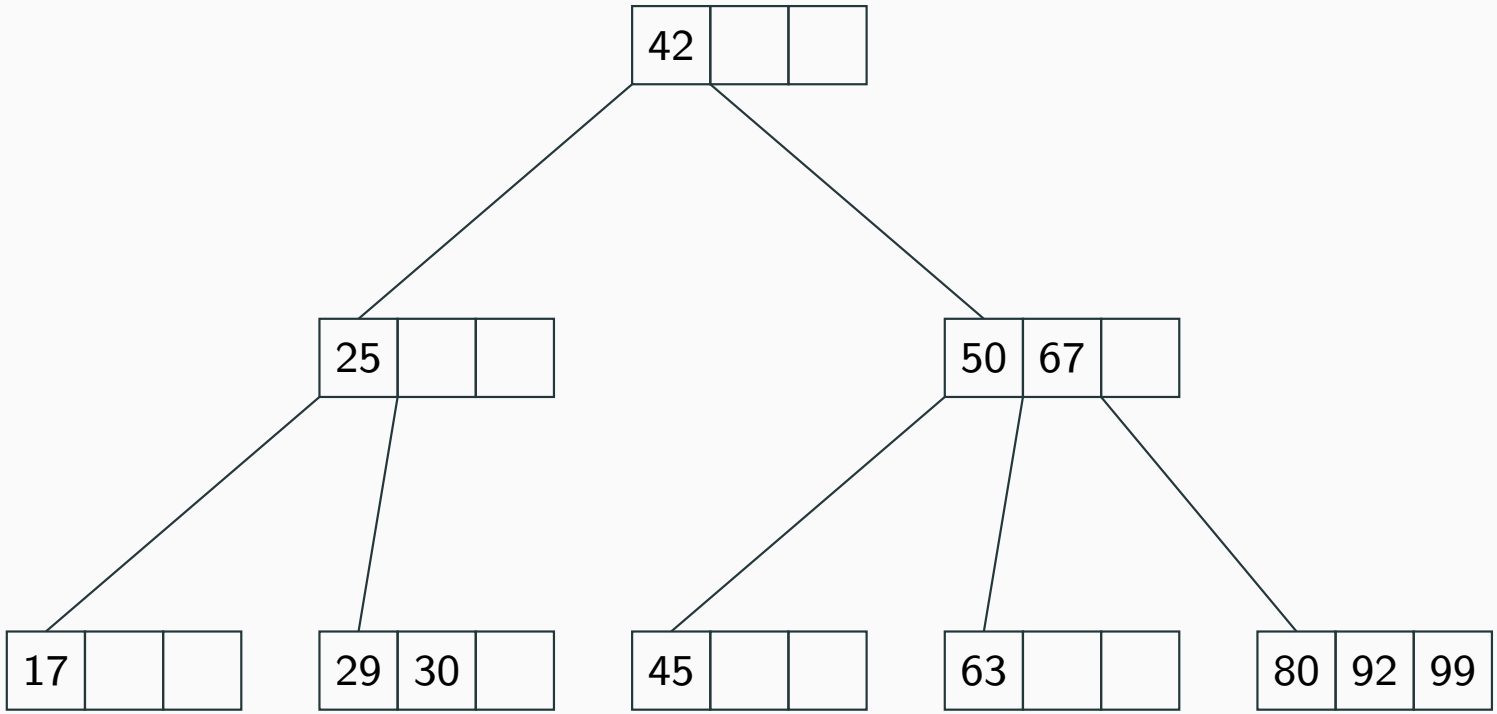
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 61



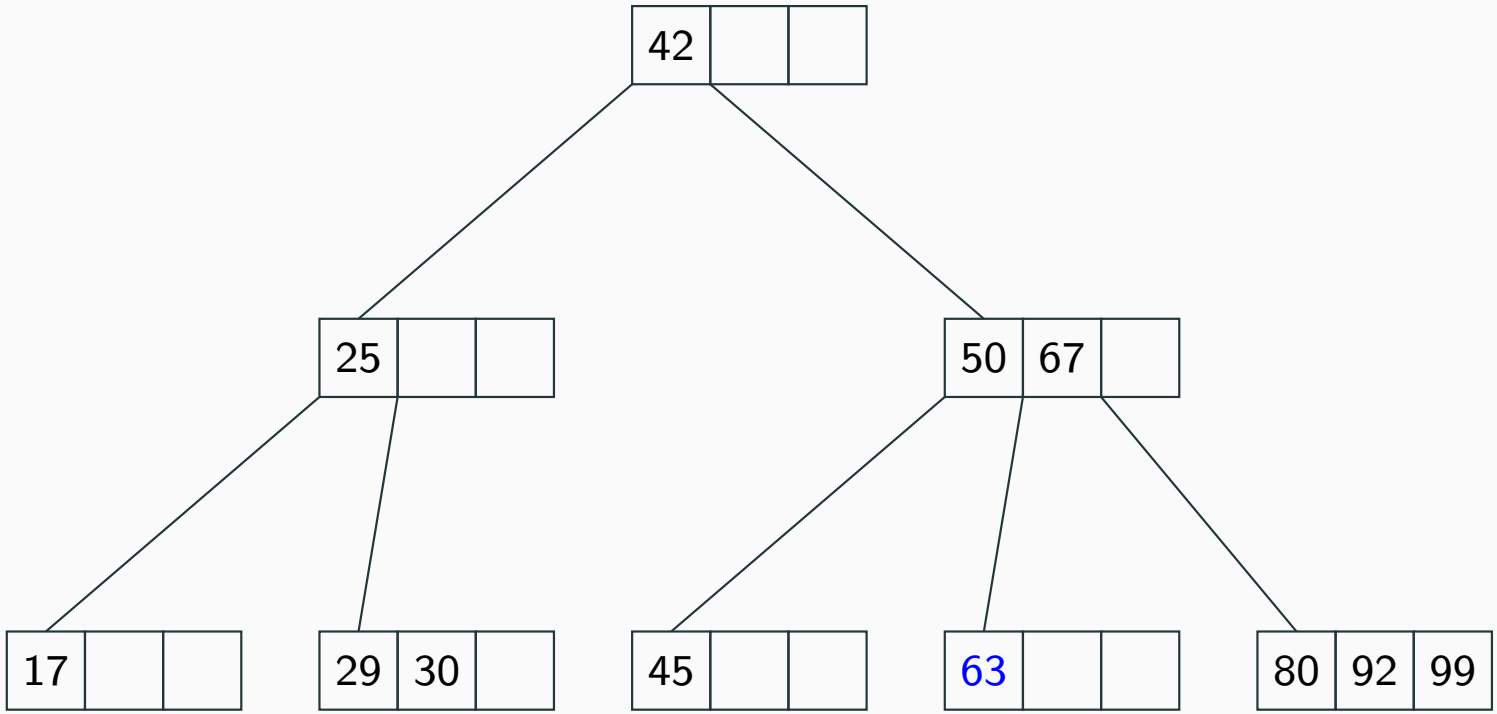
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 61



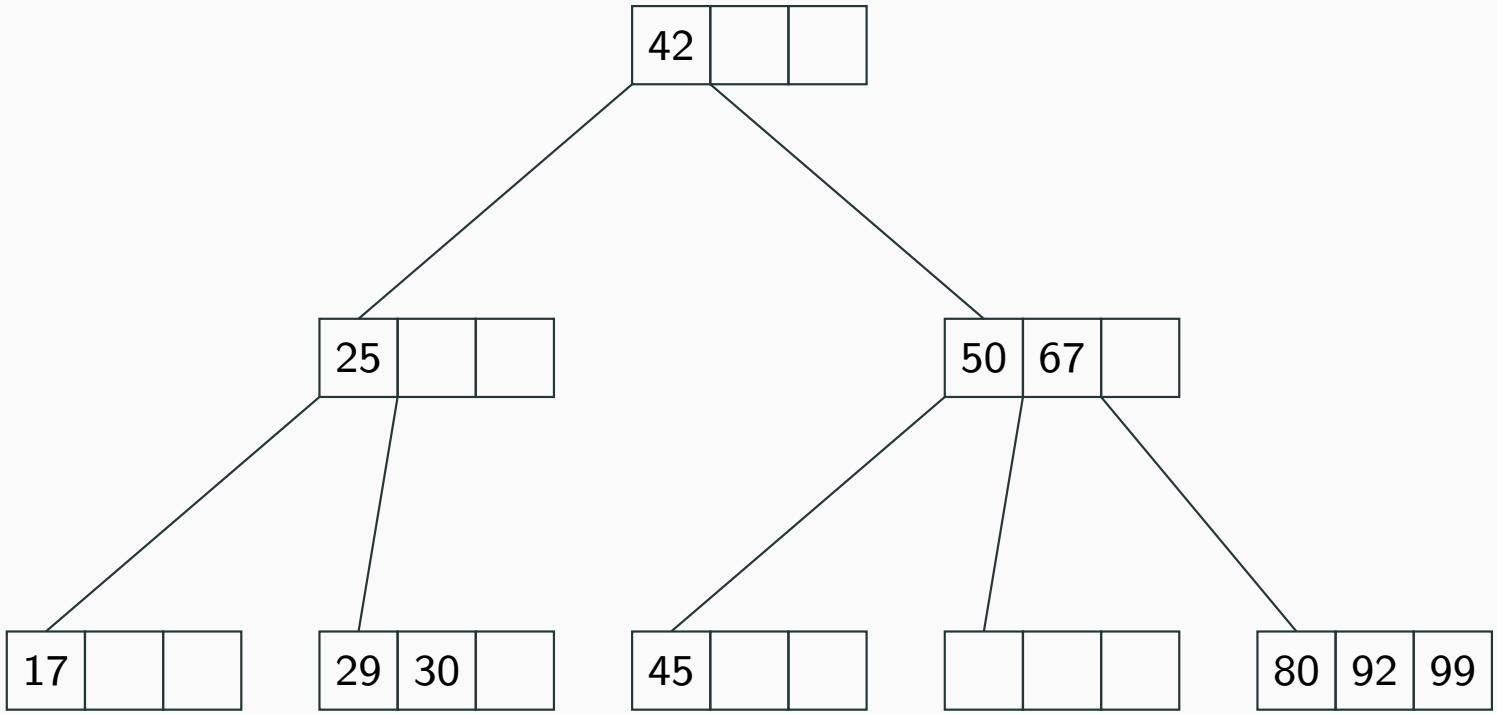
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



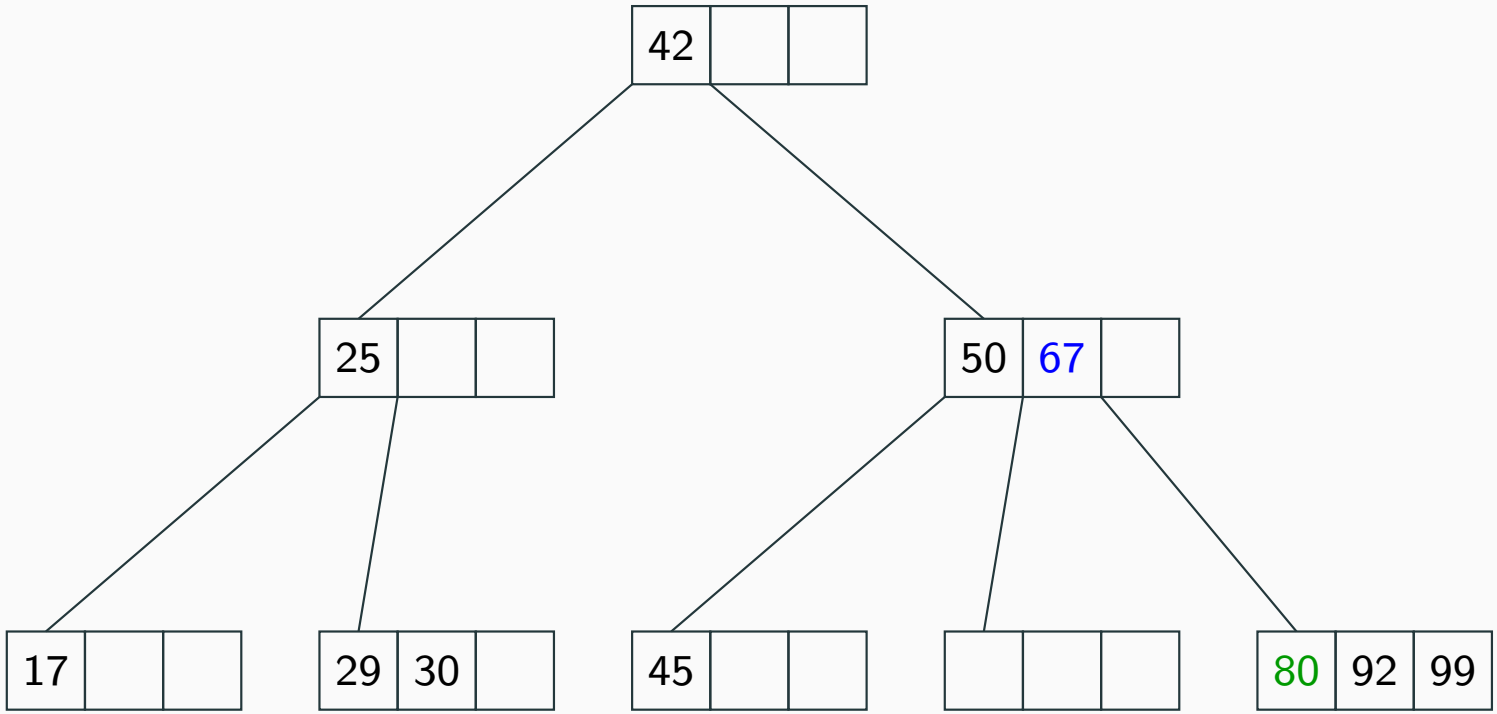
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



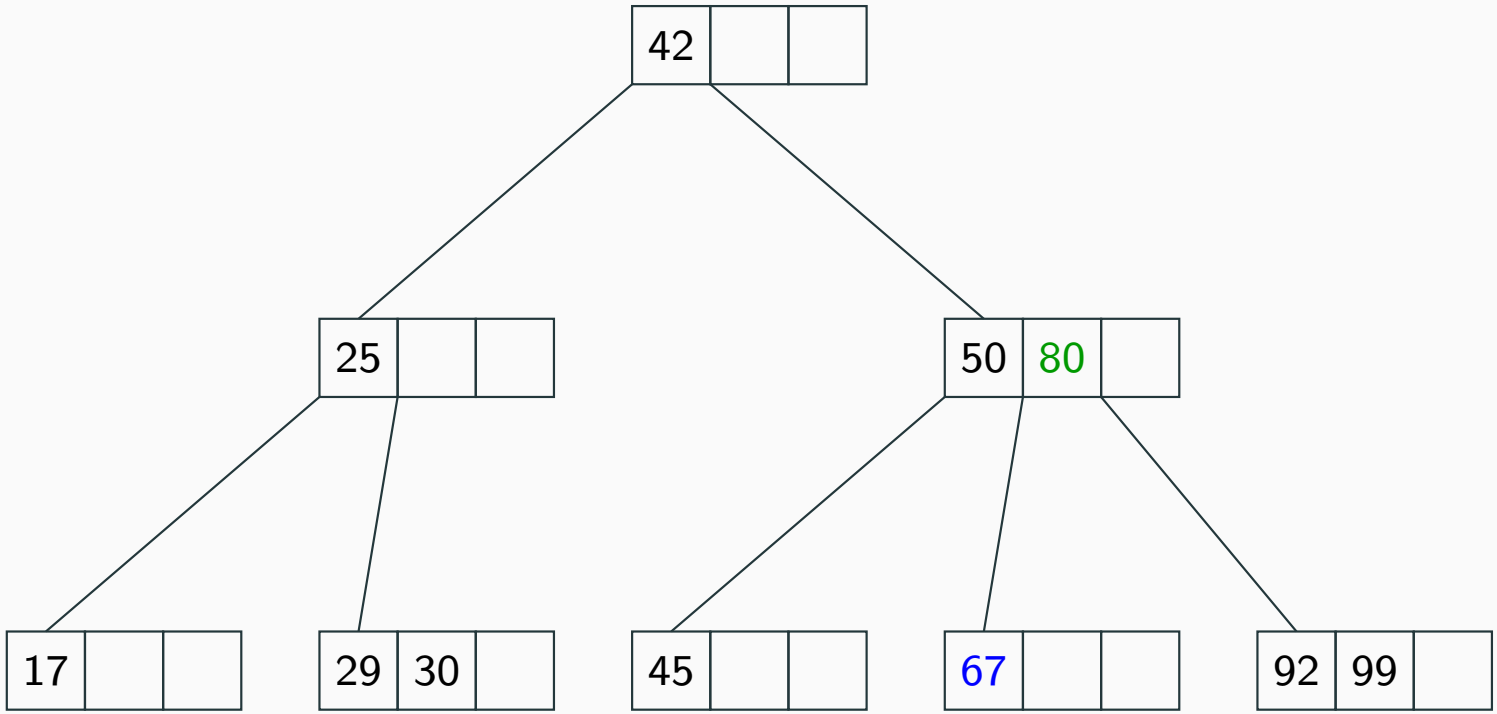
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



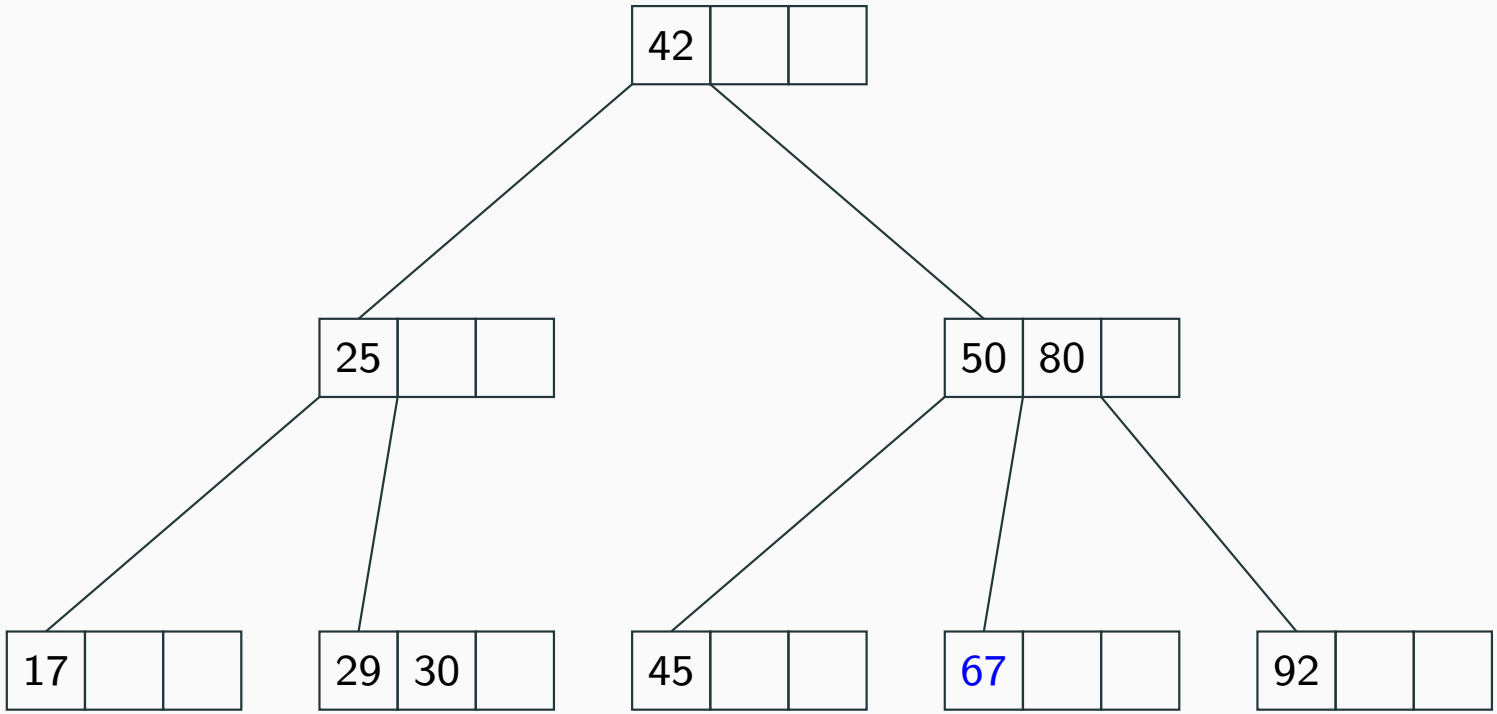
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



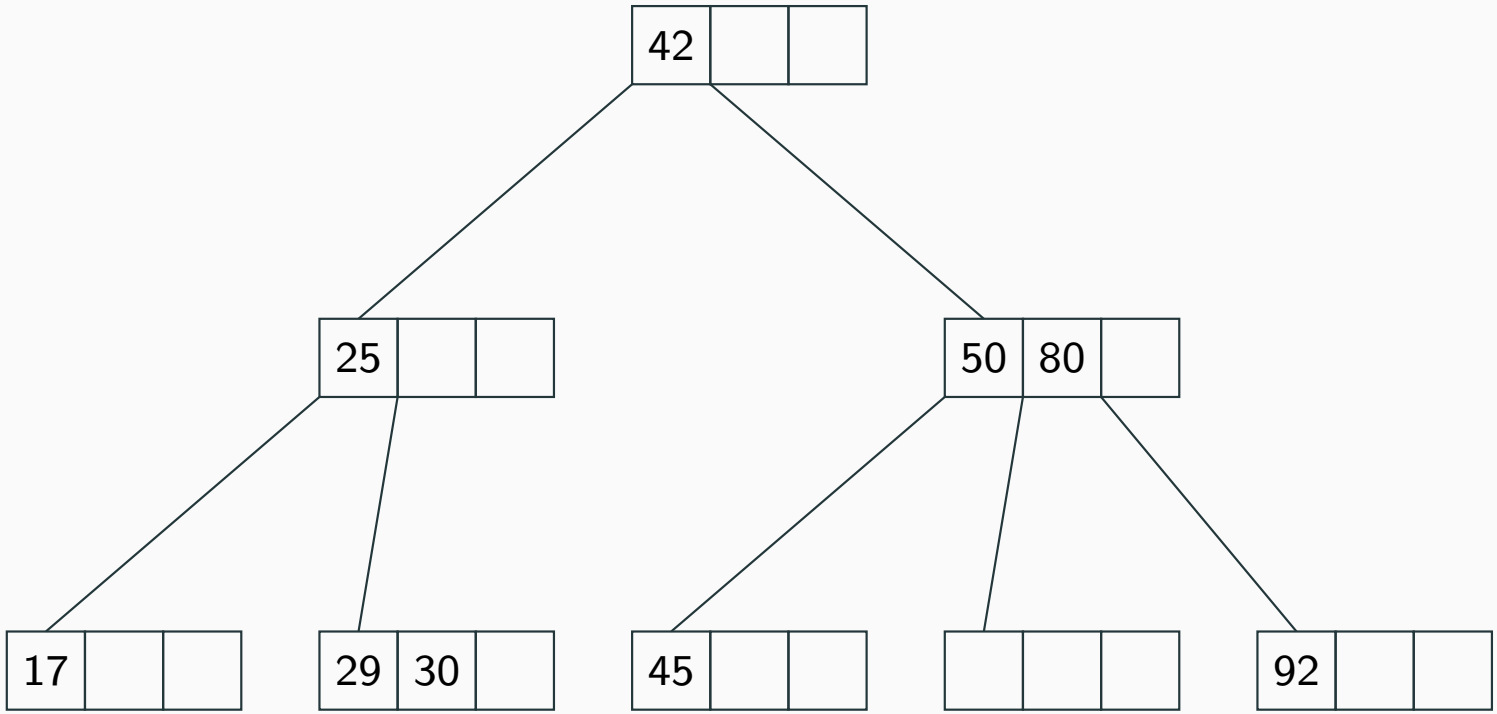
Exemplo de remoção no caso 1.2.2, $m = 4$

Elemento a ser removido: 67



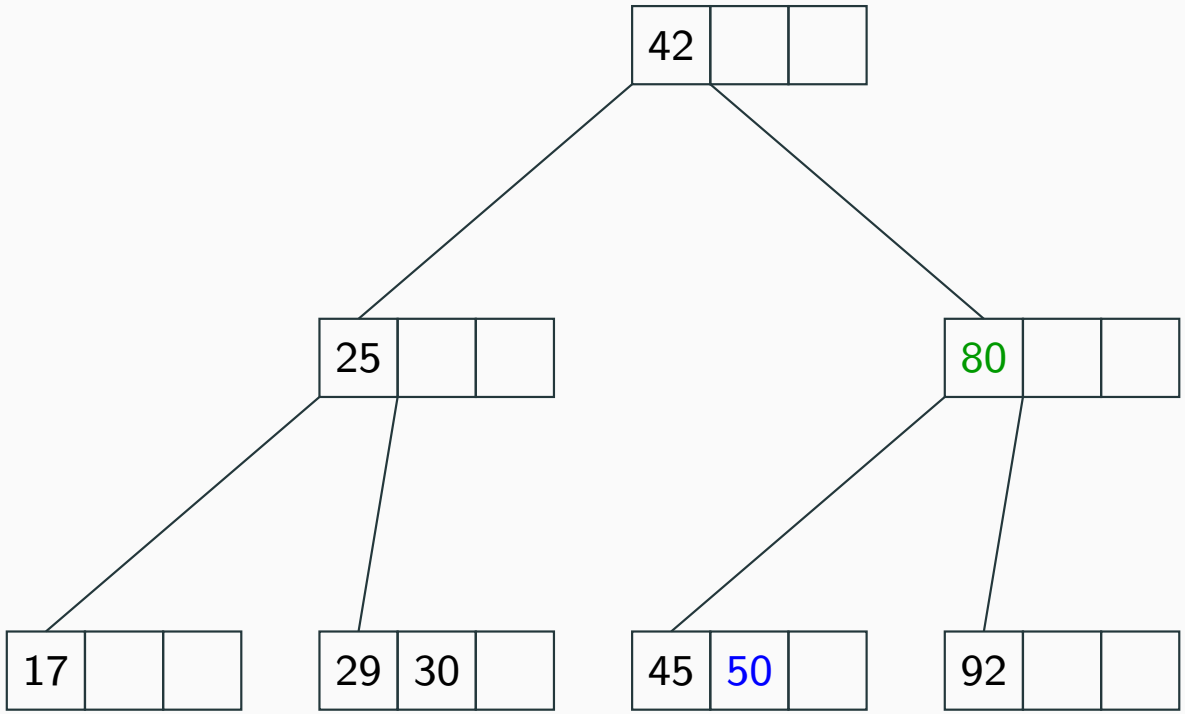
Exemplo de remoção no caso 1.2.2, $m = 4$

Elemento a ser removido: 67



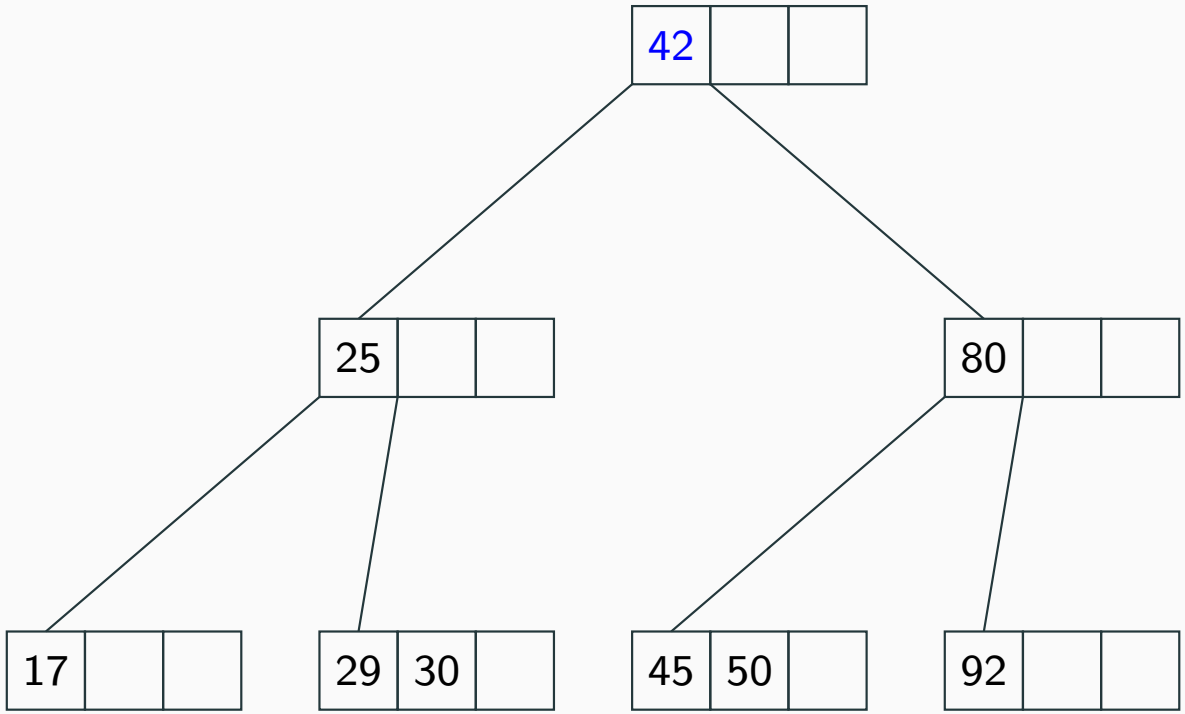
Exemplo de remoção no caso 1.2.2, $m = 4$

Elemento a ser removido: 67



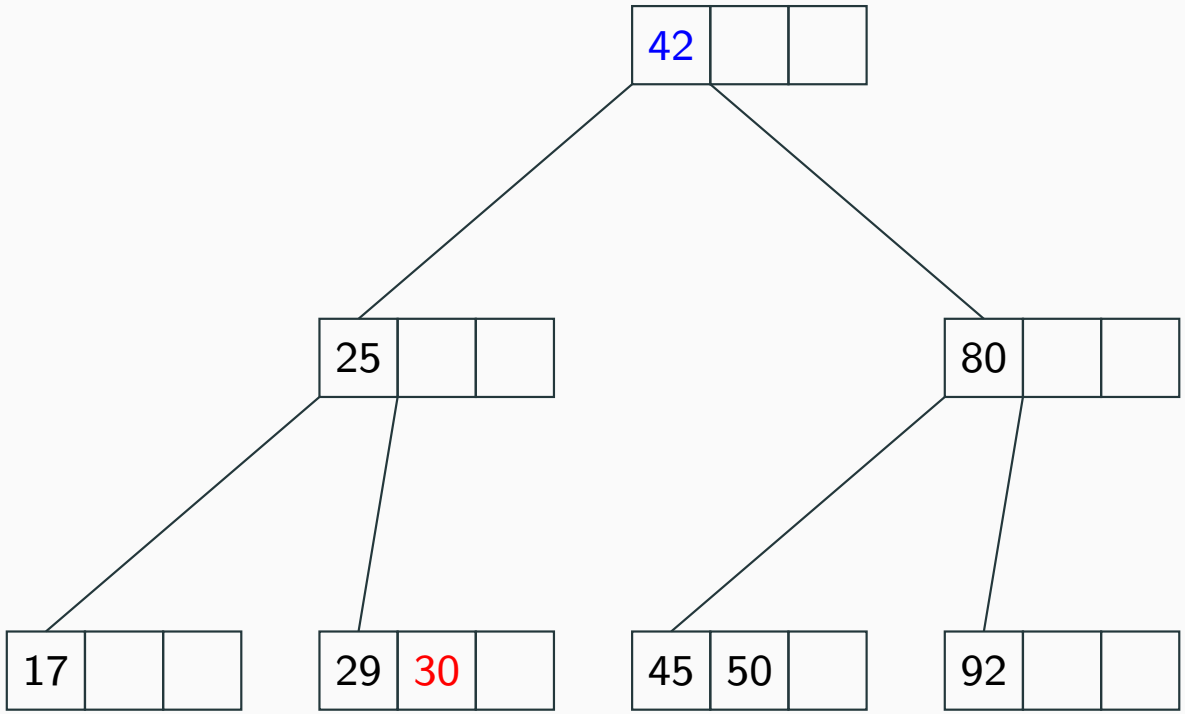
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



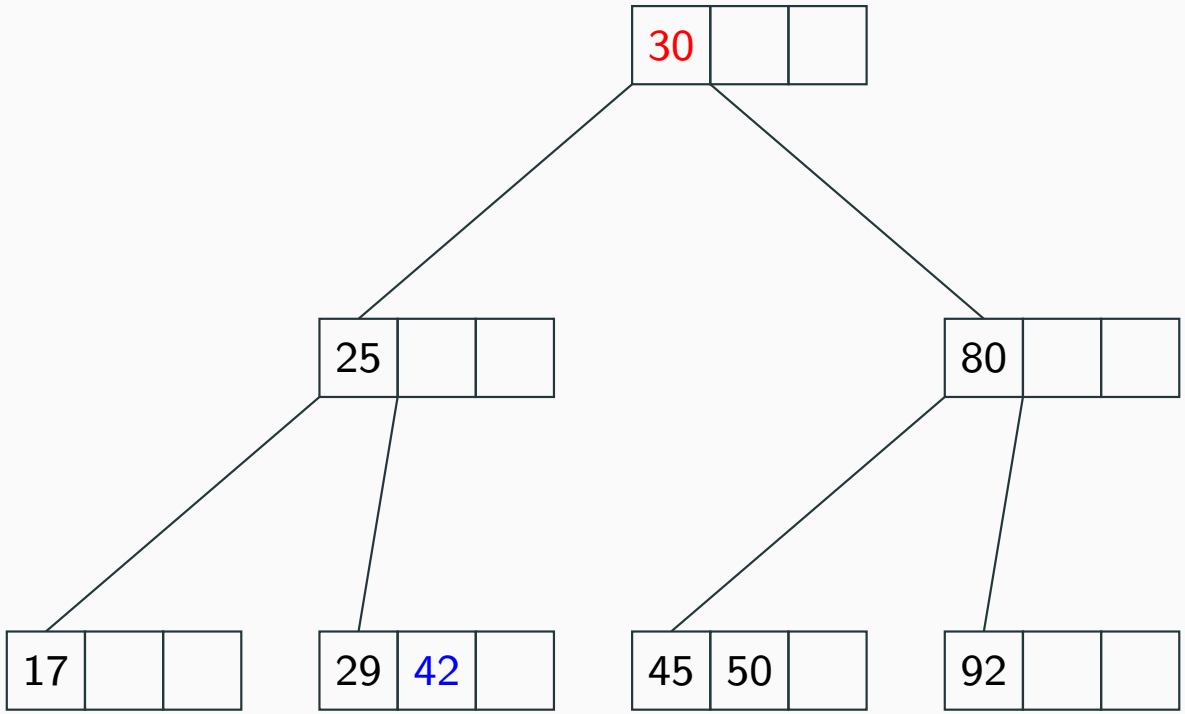
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



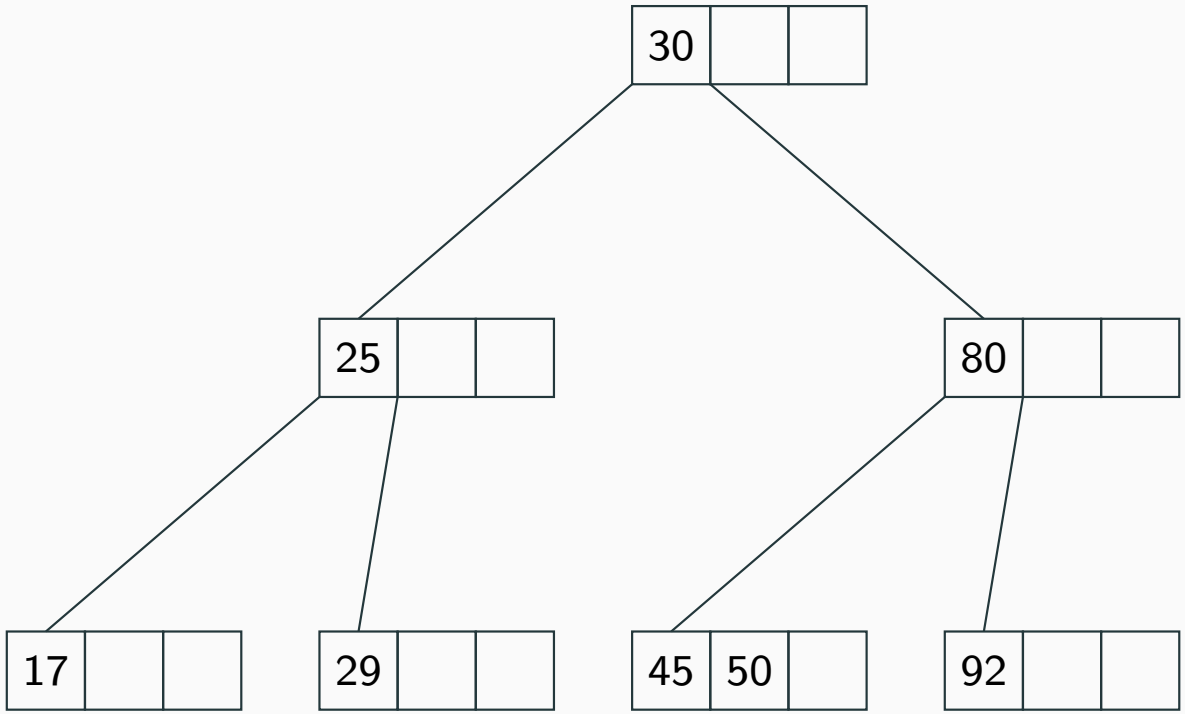
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



Implementação da remoção em uma árvore-B

```
150 public:
151     bool erase(const T& info)
152     {
153         // Não remove informações inexistentes
154         if (not find(info))
155             return false;
156 
157         auto node = find(root, info);
158         erase(info, node);
159 
160         return true;
161     }
162 
163 private:
164 
165     void erase(const T& info, Node *node)
166     {
167         if (node->leaf == false) // Remoção por cópia
168         {
169             auto i = node->index(info);
170             auto temp = node;
```

Implementação da remoção em uma árvore-B

```
172             // Procura pelo filho mais à direita da sub-árvore à esquerda
173             while (not temp->leaf)
174             {
175                 auto k = temp->index(info);
176                 temp = temp->children[k];
177             }
178
179             auto j = temp->index(info) - 1;
180
181             // Troca os nós
182             swap(node->keys[i], temp->keys[j]);
183
184             // Prossegue a remoção na folha
185             node = temp;
186         }
187
188         // Elimina a chave
189         auto it = lower_bound(node->keys.begin(), node->keys.end(), info);
190
191         node->keys.erase(it);
192         node->sort_keys();
```

Implementação da remoção em uma árvore-B

```
194         // Se o nó for a raiz ou houver chaves o suficiente, nada a fazer
195         auto P = node->parent;
196         auto limit = (M + 1)/2 - 1;
197 
198         if (not P or node->keys.size() >= limit)
199             return;
200 
201         // Irmãos
202         auto i = P->index(info);    // node é o filho i
203 
204         auto R = i == M ? nullptr : P->children[i + 1];
205         auto L = i ? P->children[i - 1] : nullptr;
206 
207         // Caso 1.1
208         if (L and L->keys.size() > limit)
209         {
210             node->keys.push_back(P->keys[i - 1]);
211             node->sort_keys();
212 
213             P->keys[i - 1] = L->keys.back();
214             L->keys.pop_back();
```

Implementação da remoção em uma árvore-B

```
216             return;
217         }
218
219         if (R and R->keys.size() > limit)
220         {
221             node->keys.push_back(P->keys[i]);
222             node->sort_keys();
223
224             P->keys[i] = R->keys.front();
225             R->keys[0] = R->keys.back();
226             R->keys.pop_back();
227
228             R->sort_keys();
229
230             return;
231         }
232
233         // Caso 2
234         auto N = L ? L : R;
235         auto k = L ? i - 1 : i;
236
```

Implementação da remoção em uma árvore-B

```
237         while (not N->keys.empty())
238         {
239             node->keys.push_back(N->keys.back());
240             N->keys.pop_back();
241         }
242
243         node->keys.push_back(P->keys[k]);
244         node->sort_keys();
245
246         P->keys[k] = P->keys.back();
247         P->keys.pop_back();
248
249         P->sort_keys();
250         P->sort_children();
251
252         delete P->children.back();
253         P->children.pop_back();
254     }
255
```

Referências

1. **DROZDEK**, Adam. *Algoritmos e Estruturas de Dados em C++*, 2002.
2. myUSF. Algorithm Visualization – B-Trees, acesso em 29/04/2019.
3. Wikipedia. B-tree, acesso em 29/04/2019.