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

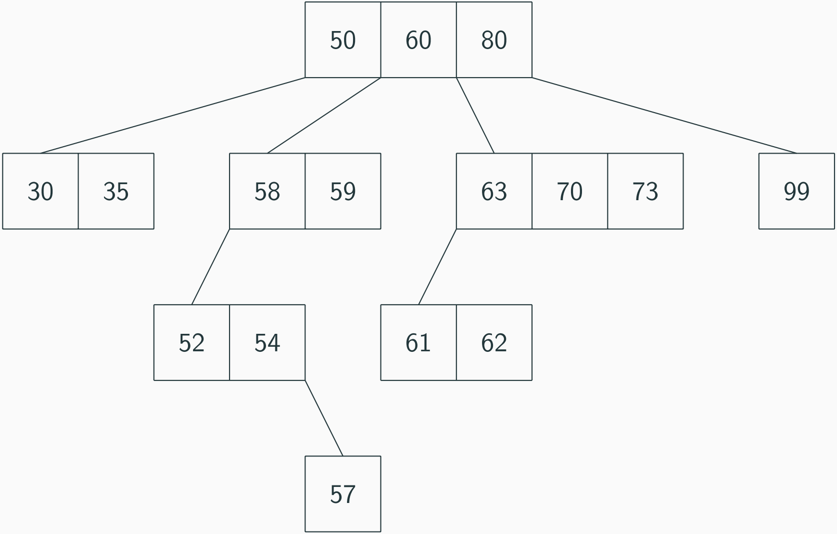
Árvores de busca de ordem m

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

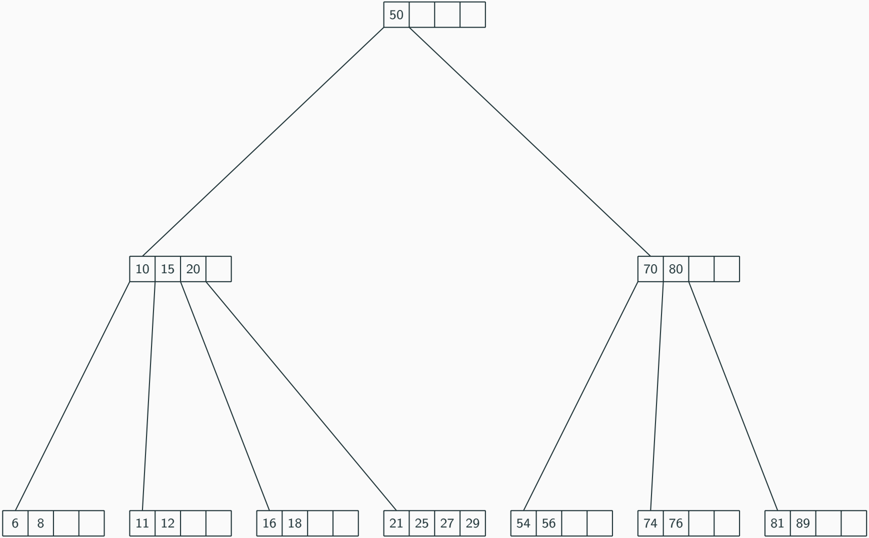
Definição de Árvores-B

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 armazenar 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

```
#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

```
        void sort_children()
        {
            sort(children.begin(), children.end(),
                [](const Node *a, const Node *b) {
                    if (a->keys.empty())
                        return false;

                    if (b->keys.empty())
                        return true;

                    return a->keys[0] < b->keys[0];
                });
        }

        // Índice da menor chave maior ou igual a key
        size_t index(const T& key) const
        {
            auto it = lower_bound(keys.begin(), keys.end(), key);
            return it - keys.begin();
        }
    };
```

Exemplo de implementação de uma árvore-B

```
    Node *root;

public:
    // A árvore é inicializada com um nó sem nenhuma chave armazenada
    BTree() : root(new Node()) {}

    // Complexidade O(log N log M)
    bool find(const T& info) const
    {
        auto node = find(root, info);

        return binary_search(node->keys.begin(), node->keys.end(), info);
    }

private:
    // Procura pelo nó onde a informação deveria estar
    Node * find(Node *node, const T& info) const
    {
        if (node->leaf)
            return node;

```

Exemplo de implementação de uma árvore-B

```
        auto i = node->index(info);

        if (i < node->keys.size() and node->keys[i] == info)
            return node;

        return find(node->children[i], info);
    }

```

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

50		

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

Elemento a ser inserido: 80

50		

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

Elemento a ser inserido: 80

50	80	

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

Elemento a ser inserido: 30

50	80	

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

Elemento a ser inserido: 30

30	50	80

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

Elemento a ser inserido: 42

42	50	80

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

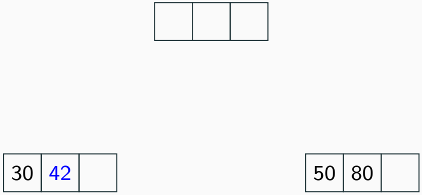
Elemento a ser inserido: 42

30	42	

50	80	

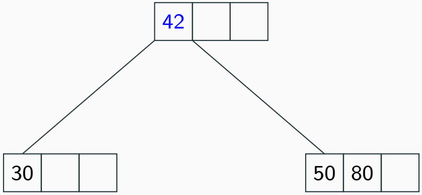
Exemplo de inserção no terceiro caso, $m = 4$

Fusão do nó dividido, nova raiz



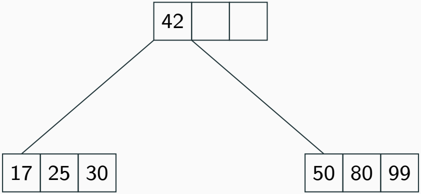
Exemplo de inserção no terceiro caso, $m = 4$

Fusão do nó dividido, nova raiz



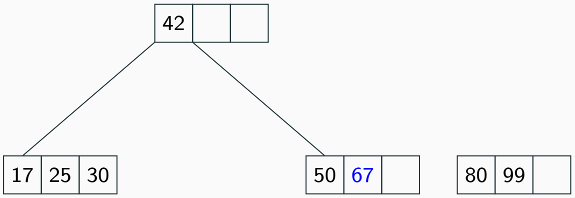
Exemplo de inserção no segundo caso, $m = 4$

Elemento a ser inserido: 67



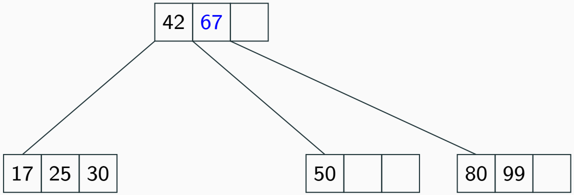
Exemplo de inserção no segundo caso, $m = 4$

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

```cpp
public:
    bool insert(const T& info)
    {
        // Não insere informações duplicadas
        if (find(info))
            return false;

        auto node = find(root, info);

        return insert(info, node);
    }

private:
    bool insert(const T& info, Node *node, Node* child = nullptr)
    {
        // Insere a informação e o filho
        node->keys.push_back(info);
        node->sort_keys();

```

Implementação da inserção em uma árvore-B

```
        if (child)
        {
            node->children.push_back(child);
            node->sort_children();
        }

        // Capacidade do nó superada: o nó deve ser dividido
        if (node->keys.size() == M)
        {
            auto S = new Node(node->leaf);
            auto half = M/2;

            // Divide as chaves
            for (size_t i = half; i < M; ++i)
            {
                S->keys.push_back(node->keys.back());
                node->keys.pop_back();
            }

            reverse(S->keys.begin(), S->keys.end());

```

Implementação da inserção em uma árvore-B

```
        // Determina o elemento do meio, que subirá para o pai
        auto new_info = node->keys.back();
        node->keys.pop_back();

        // Divide os filhos, se necessário
        if (node->leaf == false)
        {
            for (size_t i = 0; i <= S->keys.size(); ++i)
            {
                S->children.push_back(node->children.back());
                S->children.back()->parent = S;
                node->children.pop_back();
            }

            reverse(S->children.begin(), S->children.end());
        }

```

Implementação da inserção em uma árvore-B

```
            if (node->parent)
            {
                S->parent = node->parent;
                return insert(new_info, node->parent, S);
            } else
            {
                root = new Node(false);

                root->keys.push_back(new_info);
                root->children.push_back(node);
                root->children.push_back(S);

                node->parent = root;
                S->parent = root;
            }
        }

        return true;
    }

```

Remoção

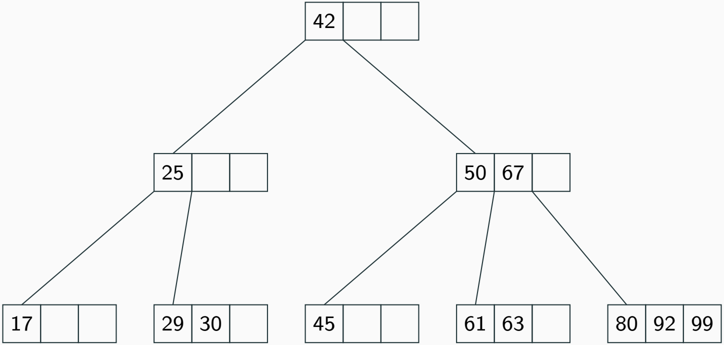
Remoção em árvores-B

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

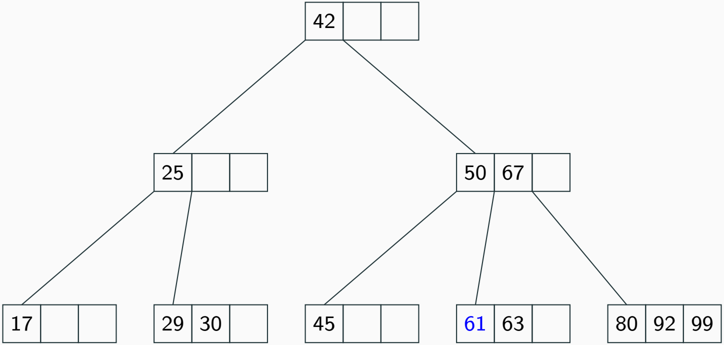
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 61



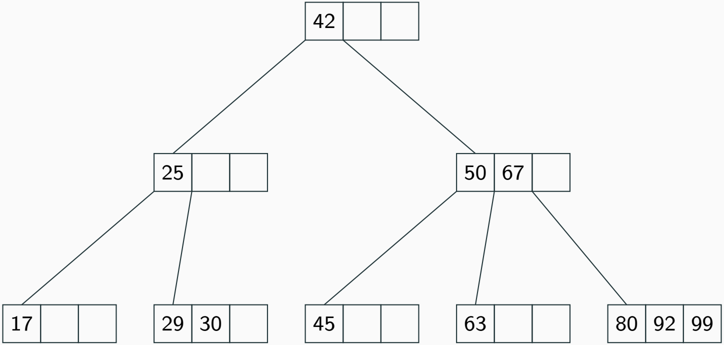
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 61



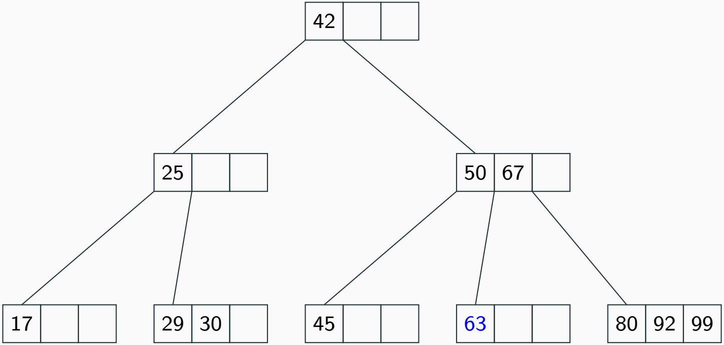
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 61



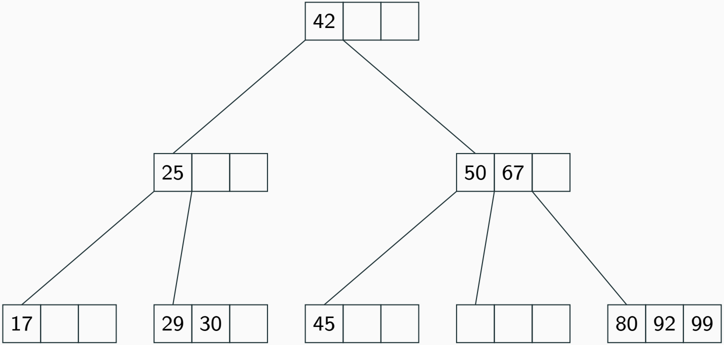
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



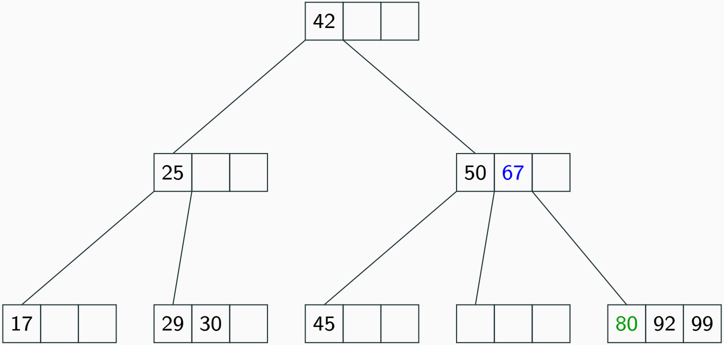
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



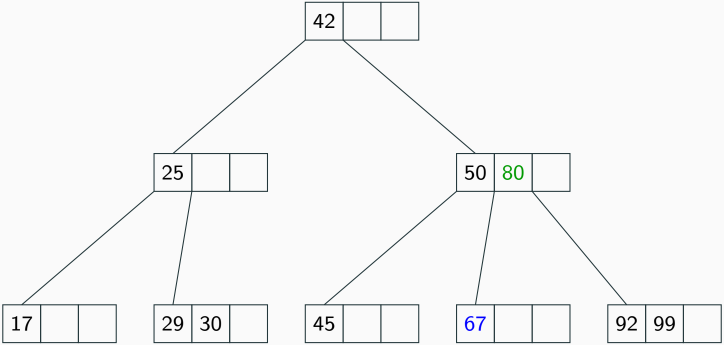
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



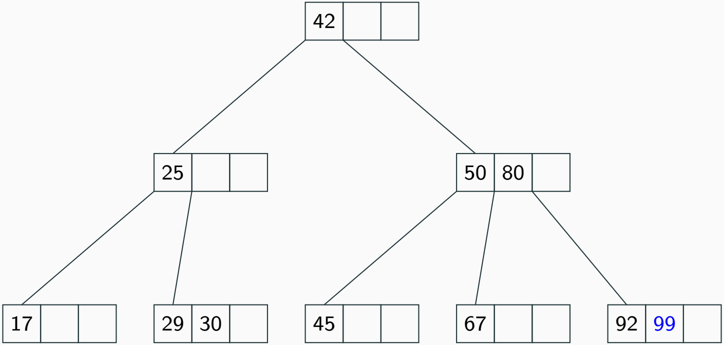
Exemplo de remoção no caso 1.2.1, $m = 4$

Elemento a ser removido: 63



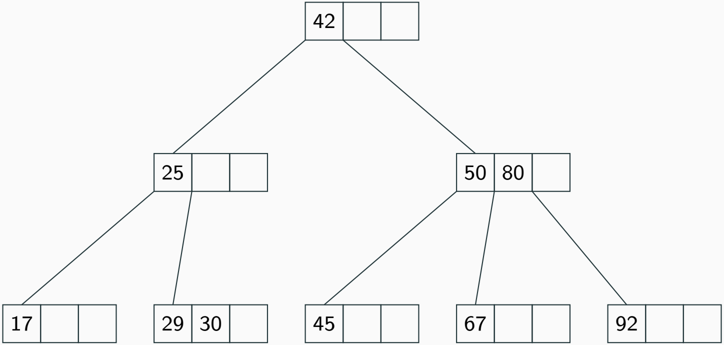
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 99



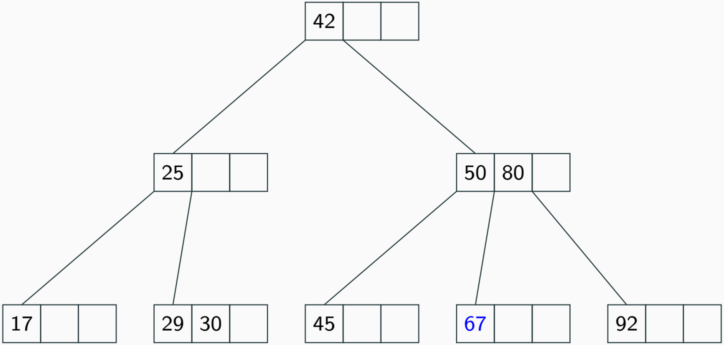
Exemplo de remoção no caso 1.1, $m = 4$

Elemento a ser removido: 99



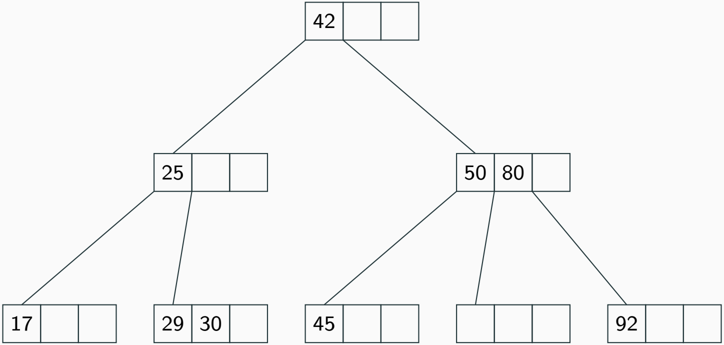
Exemplo de remoção no caso 1.2.2, $m = 4$

Elemento a ser removido: 67



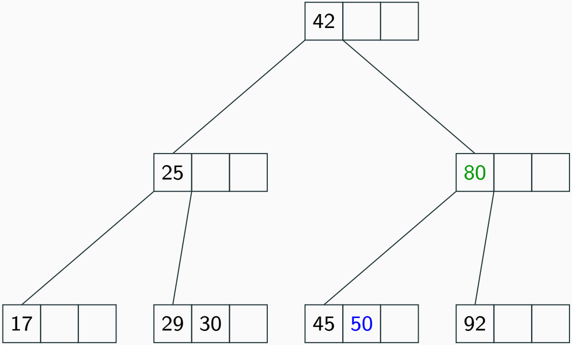
Exemplo de remoção no caso 1.2.2, $m = 4$

Elemento a ser removido: 67



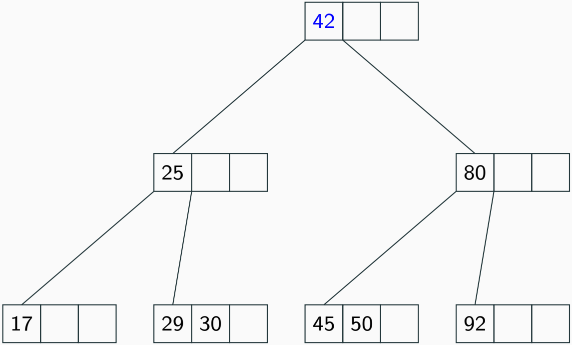
Exemplo de remoção no caso 1.2.2, $m = 4$

Elemento a ser removido: 67



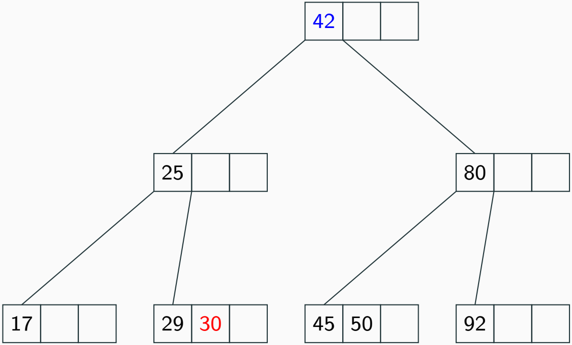
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



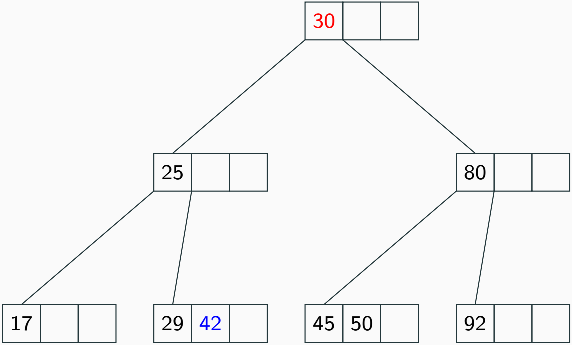
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



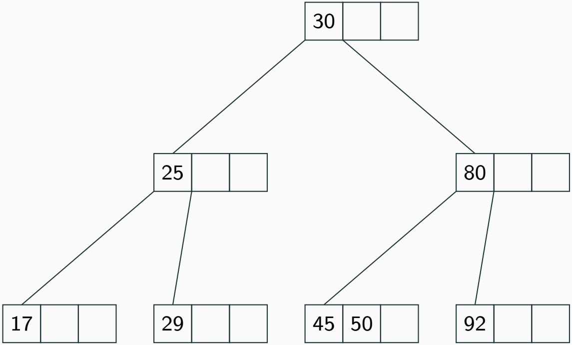
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



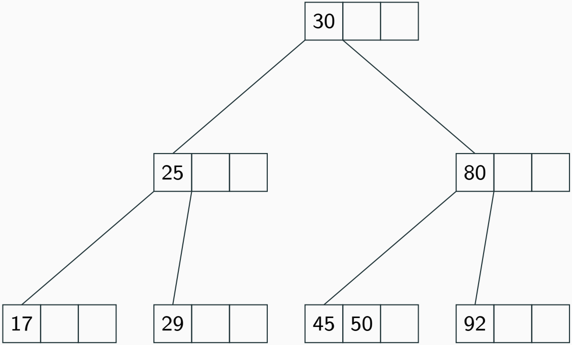
Exemplo de remoção no caso 2, $m = 4$

Elemento a ser removido: 42



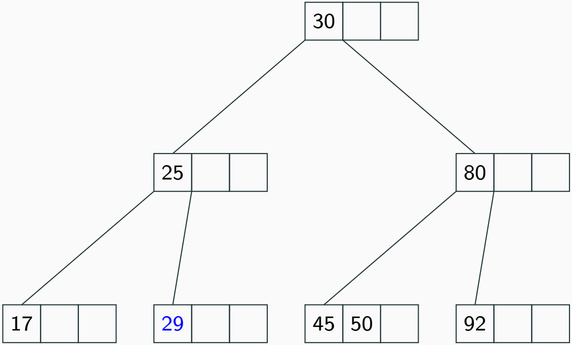
Exemplo de remoção no caso 1.2, $m = 4$

Elemento a ser removido: 29



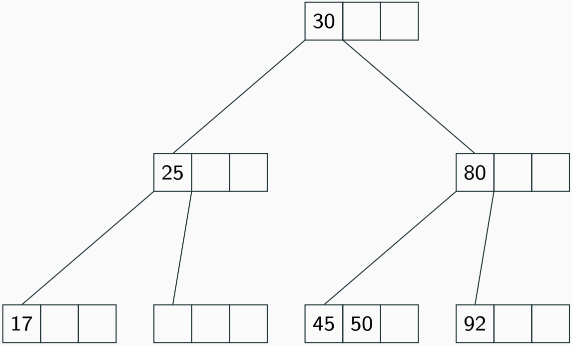
Exemplo de remoção no caso 1.2, $m = 4$

Elemento a ser removido: 29



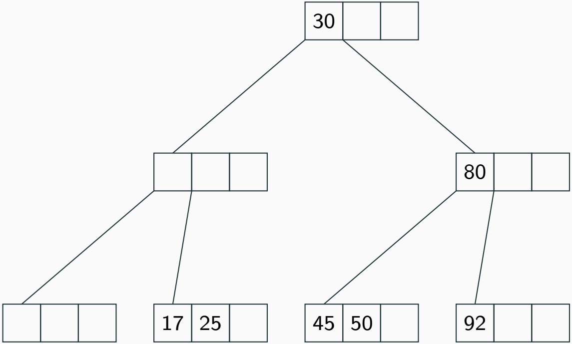
Exemplo de remoção no caso 1.2, $m = 4$

Elemento a ser removido: 29



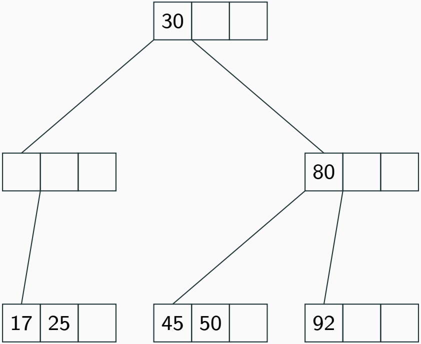
Exemplo de remoção no caso 1.2, $m = 4$

Fusão com o irmão à esquerda



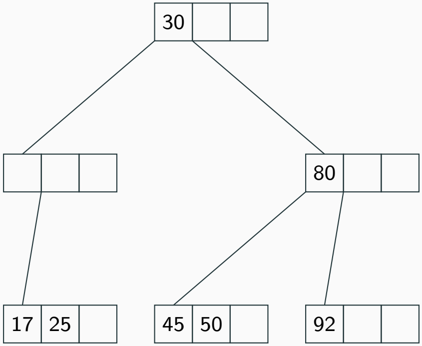
Exemplo de remoção no caso 1.2, $m = 4$

Fusão com o irmão à esquerda



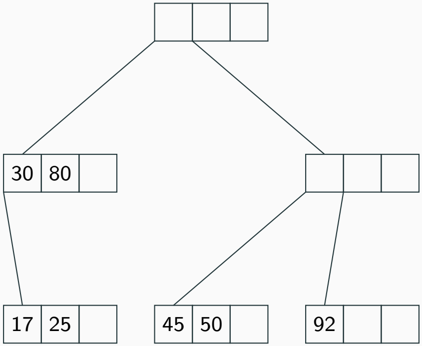
Exemplo de remoção no caso 1.2, $m = 4$

Violação das propriedades no pai: correção no nível superior



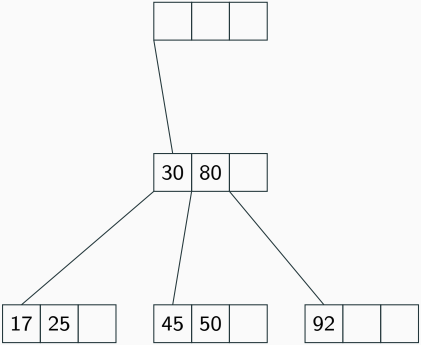
Exemplo de remoção no caso 1.2, $m = 4$

Fusão com o irmão à direita



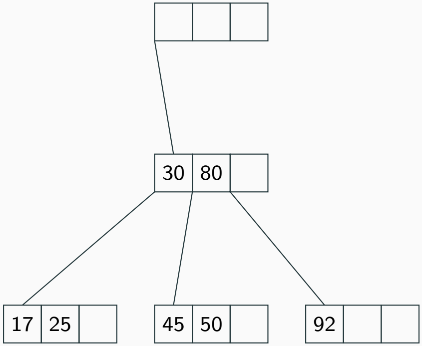
Exemplo de remoção no caso 1.2, $m = 4$

Fusão com o irmão à direita



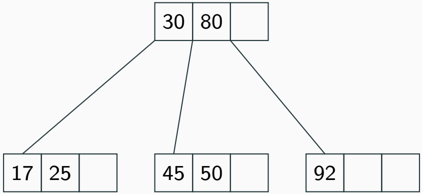
Exemplo de remoção no caso 1.2, $m = 4$

Violação na raiz: correção no nível superior



Exemplo de remoção no caso 1.2, $m = 4$

Redução da altura da árvore: nova raiz



Implementação da remoção em uma árvore-B

```
public:
    bool erase(const T& info)
    {
        // Não remove informações inexistentes
        if (not find(info))
            return false;

        auto node = find(root, info);
        erase(info, node);

        return true;
    }

private:

    void erase(const T& info, Node *node)
    {
        if (node->leaf == false) // Remoção por cópia
        {
            auto i = node->index(info);
            auto temp = node;
```

Implementação da remoção em uma árvore-B

```
            // Procura pelo filho mais à direita da sub-árvore à esquerda
            while (not temp->leaf)
            {
                auto k = temp->index(info);
                temp = temp->children[k];
            }

            auto j = temp->index(info) - 1;

            // Troca os nós
            swap(node->keys[i], temp->keys[j]);

            // Prossegue a remoção na folha
            node = temp;
        }

        // Elimina a chave
        auto it = lower_bound(node->keys.begin(), node->keys.end(), info);

        node->keys.erase(it);
        node->sort_keys();
```

Implementação da remoção em uma árvore-B

```
        // Corrige as violações, se houverem
        while (fix_node(node))
            node = node->parent;
    }

    bool fix_node(Node *node)
    {
        auto limit = (M + 1)/2 - 1;

        // Há chaves o suficiente, nada a fazer
        if (node->keys.size() >= limit)
            return false;

        auto P = node->parent;

        // Se o nó for a raiz, será a última correção pendente
        if (P == nullptr)
        {
```

Implementação da remoção em uma árvore-B

```
            // Raiz com um único filho
            if (node->children.size() == 1)
            {
                root = node->children.front();
                root->parent = nullptr;
                delete node;
            }

            // Não há mais correções pendentes
            return false;
        }

        auto missing = limit - node->keys.size();

        // Irmãos
        size_t i = std::find(P->children.begin(), P->children.end(), node)
            - P->children.begin();
        auto R = i == P->keys.size() ? nullptr : P->children[i + 1];
        auto L = i ? P->children[i - 1] : nullptr;

```

Implementação da remoção em uma árvore-B

```
        if (L and (L->keys.size() - limit) >= missing)
            borrow_from_left(node, L, P, i - 1, missing);
        else if (R and (R->keys.size() - limit) >= missing)
            borrow_from_right(node, R, P, i, missing);
        else
            merge(node, P, L, R, i);

        return true;
    }

    void borrow_from_left(Node *node, Node *L, Node *P, size_t k, size_t n)
    {
        while (n--)
        {
            node->keys.push_back(P->keys[k]);
            node->sort_keys();

            P->keys[k] = L->keys.back();
            L->keys.pop_back();
        }
    }
```

Implementação da remoção em uma árvore-B

```
void borrow_from_right(Node *node, Node *R, Node *P, size_t k, size_t n)
{
    while (n--)
    {
        node->keys.push_back(P->keys[k]);
        node->sort_keys();

        P->keys[k] = R->keys.front();
        R->keys[0] = R->keys.back();
        R->keys.pop_back();

        R->sort_keys();
    }
}

void merge(Node *node, Node *P, Node *L, Node *R, size_t i)
{
    auto N = L ? L : R;
    auto k = L ? i - 1 : i;

```

Implementação da remoção em uma árvore-B

```
        // Funde as chaves
        while (not N->keys.empty())
        {
            node->keys.push_back(N->keys.back());
            N->keys.pop_back();
        }

        node->keys.push_back(P->keys[k]);
        node->sort_keys();

        // Funde os filhos, se for o caso
        while (not N->children.empty())
        {
            node->children.push_back(N->children.back());
            node->children.back()->parent = node;
            N->children.pop_back();
        }

        node->sort_children();

```

Implementação da remoção em uma árvore-B

```cpp
        // Funde a chave de ligação do pai
        P->keys[k] = P->keys.back();
        P->keys.pop_back();
        P->sort_keys();

        // Elimina o filho sem chaves
        P->sort_children();
        delete P->children.back();
        P->children.pop_back();
    }

```

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.