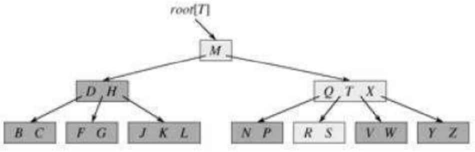
Técnicas de Busca e Ordenação (TBO)

Árvores B

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

Introdução - Árvore B

- São árvores balanceadas, desenvolvidas principalmente para otimizar o acesso a armazenamento secundário
- Os nós da árvore B podem ter muitos filhos. Esse fator de ramificação é determinante para reduzir o número de acessos a disco.
- Arvores B são balanceadas, ou seja, sua altura é O(lg(n)) - *lembrando que por ser balanceada, tem-se garantido o pior caso como a áltura da árvore*
- Arvores B são generalizações de árvores binárias balanceadas



Armazenamento Secundário

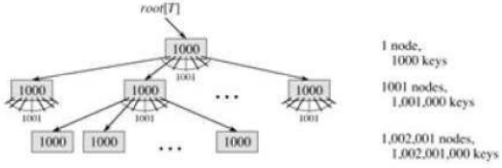
- Para o armazenamento estável feito em discos magnéticos: o custo de cada acesso (da ordem de mili segundos) é muito alto quando comparado ao acesso à memória principal (ordem de nano segundos)
- Sempre que um acesso é feito, deve-se aproveita-lo da melhor maneira possível, trazendo o máximo de informação relevante (para a MP)

Armazenamento Secundário

- A quantidade de dados utilizados numa árvore B óbviamente não cabe na memória de uma só vez, por isso é necessário paginá-la
- Especializações são feitas de acordo com as necessidades da aplicação. O fator de ramificação, por exemplo, pode variar de 3 a 2048 por exemplo (dependendo do buffer dos discos e do tamanho das páginas de memória alocados pelo SO)

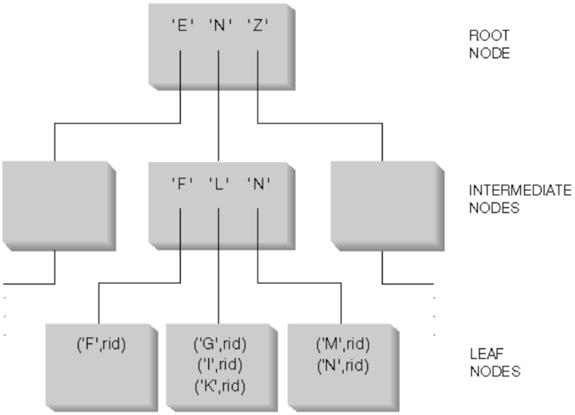
Armazenamento Secundário

- Na maioria dos sistemas o tempo de execução de um algoritmo de árvore B é determinado pelas leituras e escritas no disco - *o processo de varredura da estrutura é extremamente eficiente*
- Um fator de ramificação alto reduz drasticamente a altura da arvore. Tomemos o exemplo:



Definição de árvore B

- Na definição clássica, os dados dos registros ficam guardados junto com a chave da árvore – diferente das árvores B+ em que os registros ficam todos nos nós folha e os nós intermediários guardam somente os índices.



Definição de árvore B

Seja T uma árvore B com raíz (*root[T]*):

1. Todo nó X apresenta os seguintes campos:
   - *n[X]* - número de chaves atualmente guardadas em X
   - *n[X] chaves* - ordenadas de forma crescente, tal que $key_1[X] \leq key_2[X] \leq ... \leq key_n[X]$
   - *n[X] registros* - dados armazenados e indexados por cada respectiva chave
   - *leaf[X]* - valor booleano que indica se X é um nó interno ou folha (TRUE para folha e FALSE caso contrário)

Definição de árvore B

2. Cada nó interno x também contém *n[x] + 1* apontadores $c_1[x], c_2[x], ..., c_{n[x]+1}[x]$ para os respectivos nós filhos. Os nós folha tem os apontadores para nulo
3. As chaves $key_i[x]$ separam os intervalos de chaves guardadas em cada sub-árvore – se k_i é uma chave armazenada em uma sub-árvore com raiz $c_i[x]$, então $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq ... \leq key_{n[x]}[x] \leq k_{n[x]+1}$
4. Todas as folhas estão no mesmo nível da árvore – a altura *h*.

Definição de árvore B

5. Existem limites superiores e inferiores para o número de chaves de um nó – estes limites são definidos por um inteiro fixo $t \geq 2$ chamado *grau mínimo*:

- Todo nó que não seja raiz deve ter pelo menos $\lceil t/2 \rceil - 1$ chaves – todo nó interno tem portanto t filhos
  - Se a árvore for não vazia a raiz deve ter pelo menos uma chave
- Cada nó pode conter no máximo $t - 1$ chaves – um nó interno pode ter no máximo t filhos
  - O nó é considerado cheio quando ele tem exatamente $t - 1$ chaves

Definição de árvore B

Propriedades da árvore B:

- Comparação da árvore B com outras árvores balanceadas com altura $O(log_2(n))$: a base do logaritmo é proporcional ao fator de ramificação
  - se o fator de ramificação é 1000 e aproximadamente 1 milhão de registros – precisa-se de apenas $log_{1000}(10^6) \approx 3$ acessos ao disco

Busca por elemento

Função `B-TREE-SEARCH`:

- A função B-TREE-SEARCH recebe o apontador para o nó raiz (x) e a chave k sendo procurada
- Se a chave k pertencer à árvore o algoritmo retorna o nó ao qual ela pertence e o índice dentro do nó correspondente à chave procurada, caso contrário, retorna *NULL*

Busca por elemento

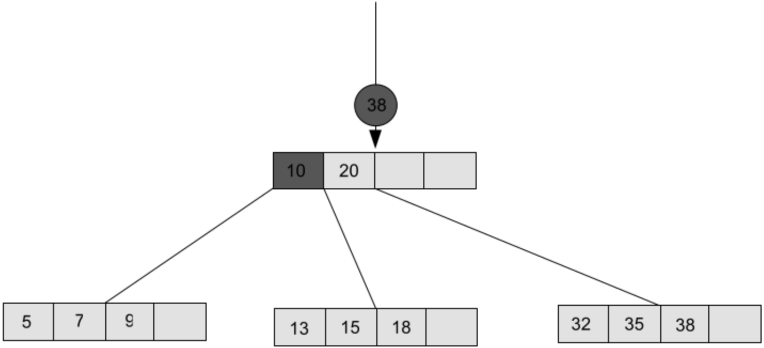
```
B-TREE-SEARCH(x, k)
        i <- 1
        while i <= n[x] and k > key_i [x] do
                i <- i +1
        if i <= n[x] and k == key_i [x] then
                return (x, i)
        if leaf[x] then
                return NULL
        else
                DISK-READ(c_i[x])
                return B-TREE-SEARCH(c_i[x], k)
```

Busca por elemento

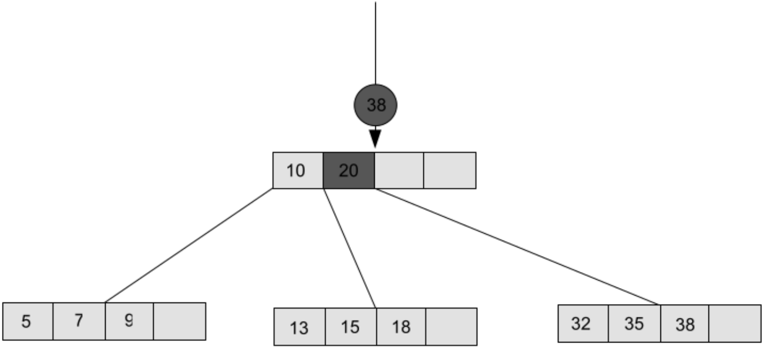
Função `B-TREE-SEARCH`:

- O número de acessos a disco é $O(log_t(n))$ onde n é o número de chaves na árvore e t é o número de filhos de cada nó
- Em cada nó realiza-se uma busca linear (mas pode ser melhorado) – custo de $O(t)$ em cada nó
- Portanto, custo total de busca é $O(t * log_t(n))$

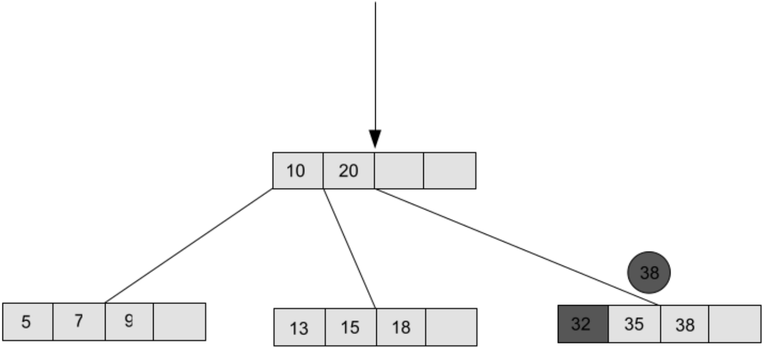
Busca por elemento: Exemplo



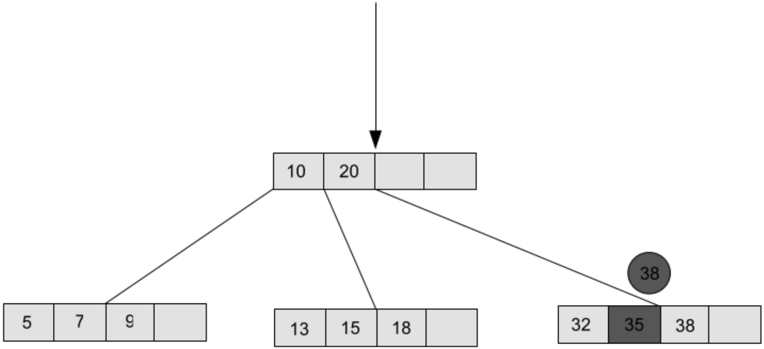
Busca por elemento: Exemplo



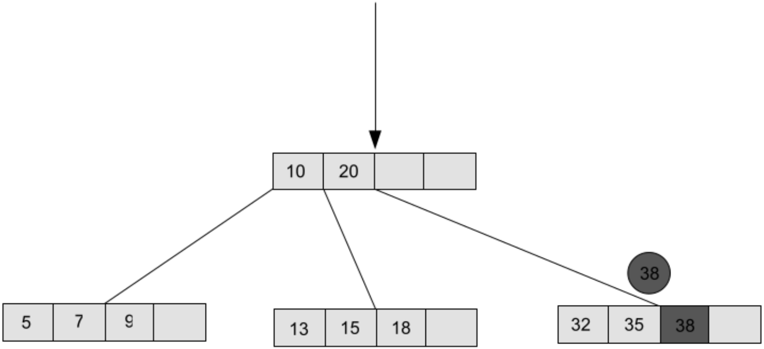
Busca por elemento: Exemplo



Busca por elemento: Exemplo



Busca por elemento: Exemplo



Inserção de elemento

- A inserção na árvore B é relativamente mais complexa, pois envolve inserir a nova chave no nó correto da árvore sem violar suas propriedades
- Como a inserção deve ser realizada se o nó estiver cheio?
  - Caso o nó esteja cheio, deve-se separar (split) o nó ao redor do elemento mediano, criando 2 novos nós que não violam as invariantes da árvore B
  - O elemento mediano é promovido (em termos de nível da árvore), passando a fazer parte do nó pai daquele nó
  - A inserção é feita em um único percurso na árvore, à partir da raiz até uma das folhas

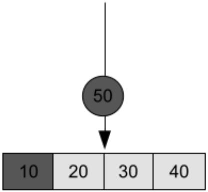
Separação de nó (split)

- A separação B-TREE-SPLIT-CHILD recebe como parâmetros um nó interno (não cheio) x um índice i e um nó y tal que $Y = c_i[x]$ é um filho de x que está cheio
- O procedimento separa o nó ao redor do elemento mediano, copiando os elementos maiores que ele em z, deixando os menores em y e ajustando o contador de elementos de z e y para $\lceil t/2 \rceil$, promovendo o elemento mediano para seu pai

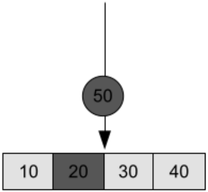
Inserção de elemento

```
B-TREE-SPLIT-CHILD(x, i, y)
        z <- ALLOCATE-NODE()
        leaf [z] <- leaf [y ]
        n[z] <- t / 2
        for j <- 1 to t - 1 do
                keyj [z] <- key_{j+t}[y]
        if not leaf [y] then
                for j <- 1 to t do
                        j [z] <- c_{j+t}[y]
        n[y ] <- t - 1
        for j <- n[x] + 1 downto i + 1 do
                c_{j+1}[x] <- cj [x]
        c_{i+1}[x] <- z
        for j <- n[x] downto i do
                key_{j+1} [x] <- key_j[x]
        key_i[x] <- key_t[y]
        n[x] <- n[x] + 1
        DISK-WRITE(y)
        DISK-WRITE(z)
        DISK-WRITE(x)
```

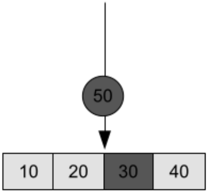
Split: Exemplo



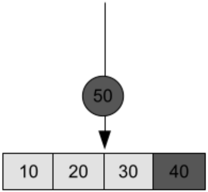
Split: Exemplo



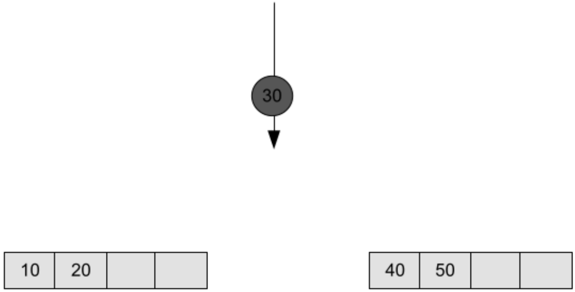
Split: Exemplo



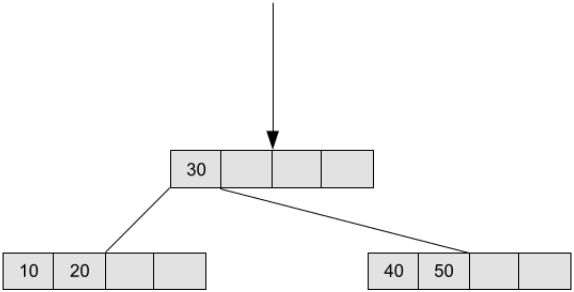
Split: Exemplo



Split: Exemplo



Split: Exemplo



Inserção com split

- Com uma única passada pela árvore, da raiz até às folhas, é possível inserir uma determinada chave, dividindo (splits) cada nó que encontrarmos no caminho, caso o nó esteja cheio
- A função B-TREE-INSERT-NONFULL insere a chave k no nó x caso este seja uma folha, e caso contrário, procura o filho adequado e desce à ele recursivamente até encontrar a folha onde k deve ser inserido

Inserção com split

```
B-TREE-INSERT-NONFULL(x, k)
        i <- n[x]
        if leaf[x] then
                while i >= 1 and k < key_i[x] do
                        key_{i+1}[x] <- key_i[x]
                        i <- i - 1
                        key_{i+1}[x] <- k
                        n[x] <- n[x] + 1
                        DISK-WRITE(x)
                else
                        while i >= 1 and k < key_i[x] do
                                i <- i - 1
                        i <- i +1
                        DISK-READ(ci [x])
                        if n[c_i[x]] = t - 1 then
                                B-TREE-SPLIT-CHILD(x, i, c_i[x])
                                if k > key_i[x] then
                                        i <- i +1
                        B-TREE-INSERT-NONFULL(c_i[x], k)
```

Inserção com split

- Portanto, a inserção tem custo $O(t * log_t(n))$, onde t representa o tamanho da página da árvore b e n representa o número total de elementos de cada nó

```
B-TREE-INSERT(T, k)
        r <- root[T]
        if n[r] = t - 1 then
                s <- ALLOCATE-NODE()
                root[T] <- s
                leaf[s] <- FALSE
                n[s] <- 0
                c1[s] <- r
                B-TREE-SPLIT-CHILD(s, 1, r)
                B-TREE-INSERT-NONFULL(s, k)
        else
                B-TREE-INSERT-NONFULL(r, k)
```

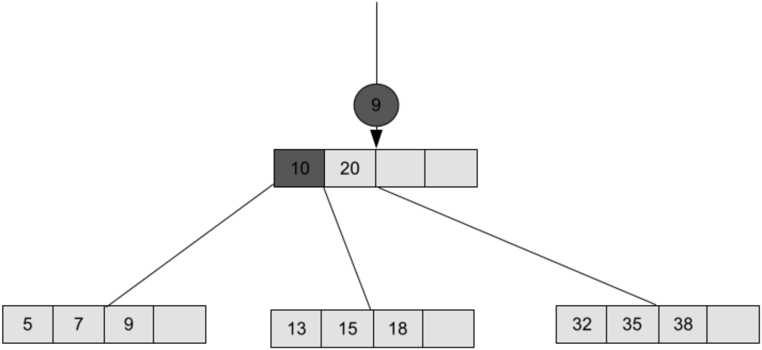
Remoção de Chaves

- A remoção de uma chave é análoga, de certa forma, à inserção, uma vez que uma chave pode ser removida de qualquer nó (seja ele raiz ou não)
- Assim como na inserção, é necessário garantir que ao remover a chave as invariantes e propriedades da árvore B não sejam violadas
- Na inserção deve-se garantir que um nó não se torne grande demais – na remoção deve-se garantir que um nó não se torne pequeno demais
  - Deve sempre ter pelo menos $\lceil t/2 \rceil - 1$ elementos

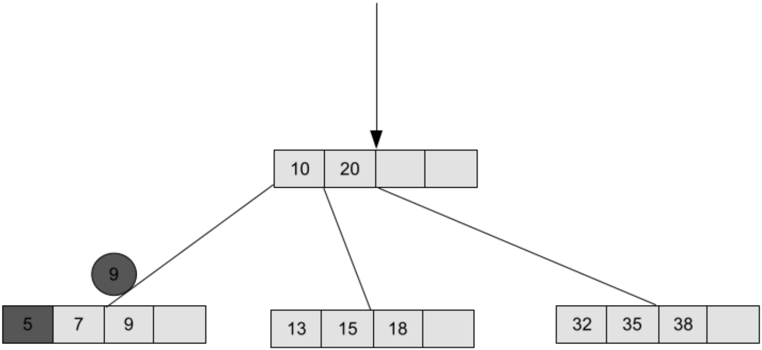
Remoção de Chaves

- Existem 6 casos possíveis para a remoção de uma chave de uma árvore B:
  - Caso 1: Se a chave *k* estiver numa folha da árvore e a folha possui pelo menos *t* chaves, remove-se a chave da árvore

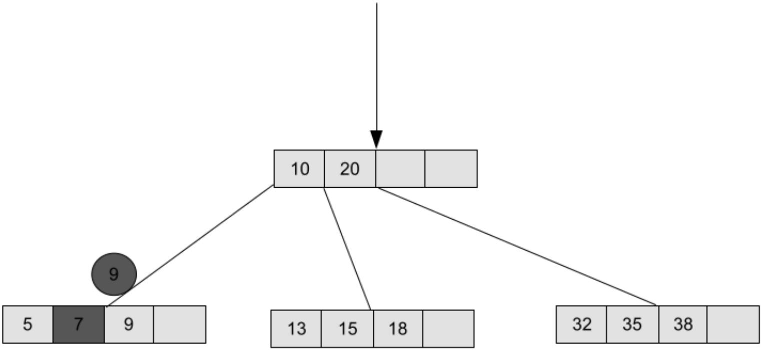
Remoção de Chaves - Caso 1



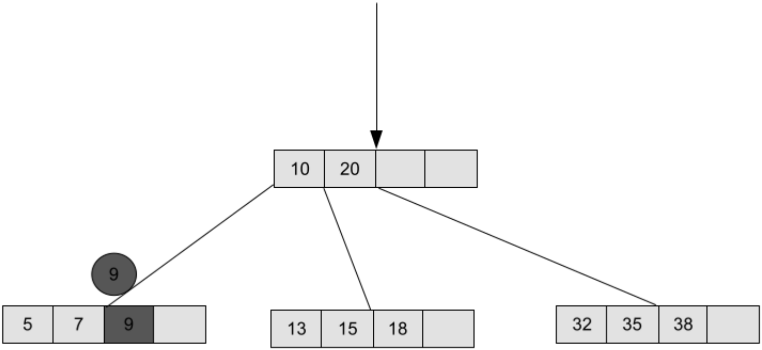
Remoção de Chaves - Caso 1



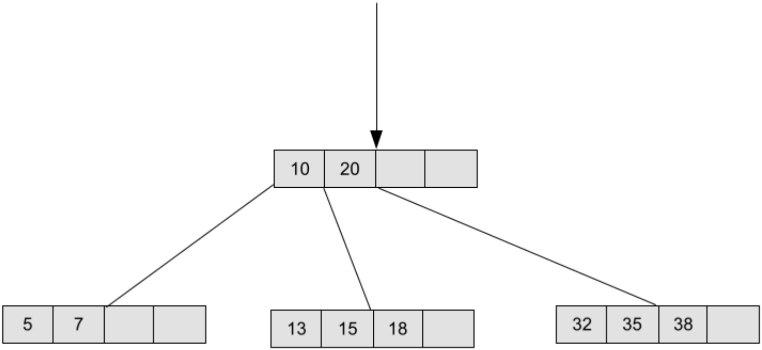
Remoção de Chaves - Caso 1



Remoção de Chaves - Caso 1



Remoção de Chaves - Caso 1

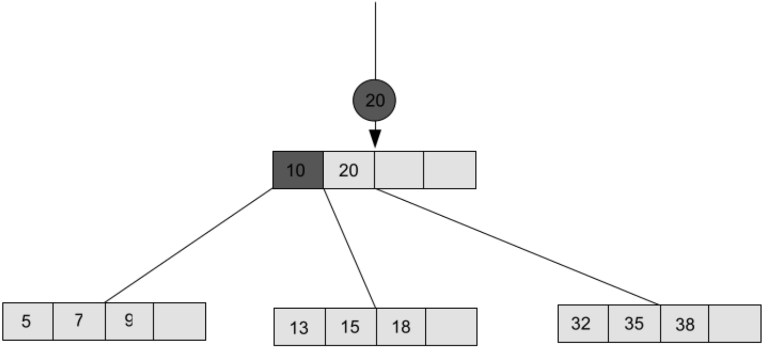


Remoção de Chaves

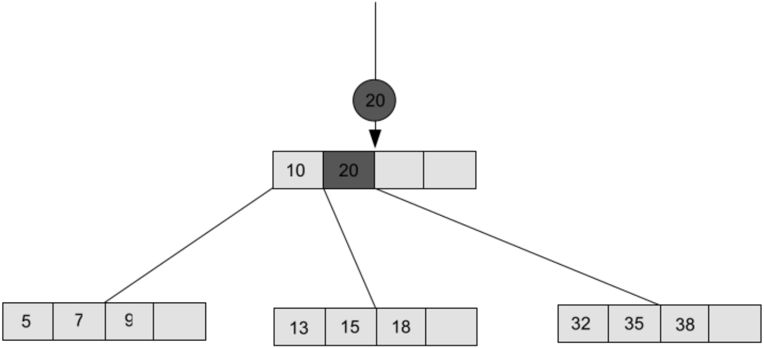
- Caso 2: Se a chave k está num nó interno x:
  - A. Se o filho y que precede k no nó x possui pelo menos $\lceil t/2 \rceil$ chaves, encontra-se o predecessor k' de k na sub-árvore com raiz em y. Remove-se k' do nó filho e substitui-se k por k' no nó atual
  - B. Simetricamente, se o filho z que sucede k no nó x possui pelo menos $\lceil t/2 \rceil - 1$ chaves, encontra-se o sucessor k' de k na sub-árvore com raiz em z. Remove-se k' do nó filho e substitui-se k por k' no nó atual
  - C. Caso ambos y e z possuam somente $\lceil t/2 \rceil - 1$ chaves, copia-se todos os elementos de z em y , libera-se a memória ocupada por z e remove-se o apontador em x. Finalmente, remove-se a chave k de x.

Remoção de Chaves - Caso 2(A/B)

,

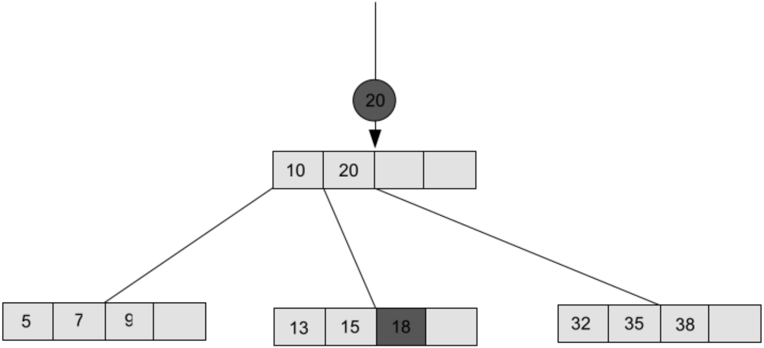


Remoção de Chaves - Caso 2(A/B)

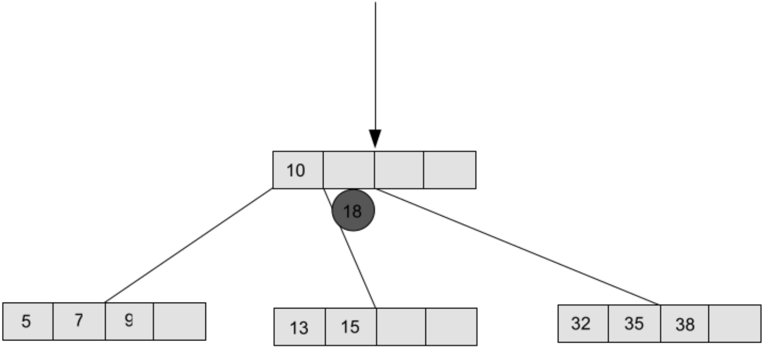


Remoção de Chaves - Caso 2(A/B)

,

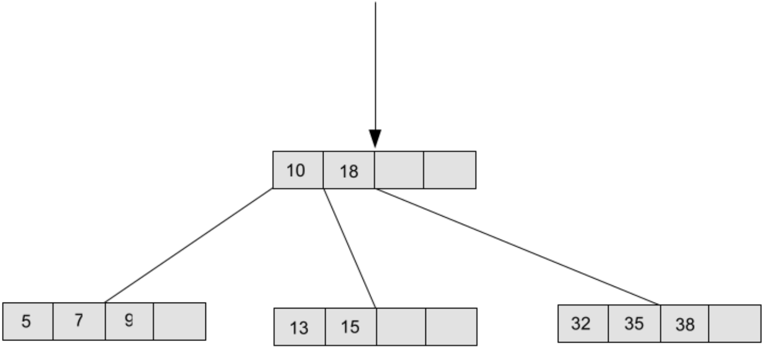


Remoção de Chaves - Caso 2(A/B)

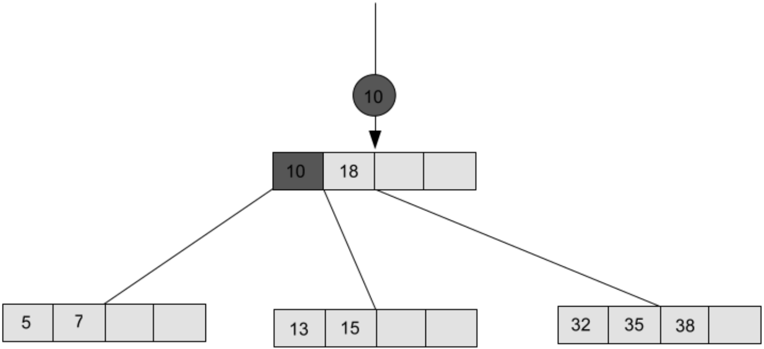


Remoção de Chaves - Caso 2(A/B)

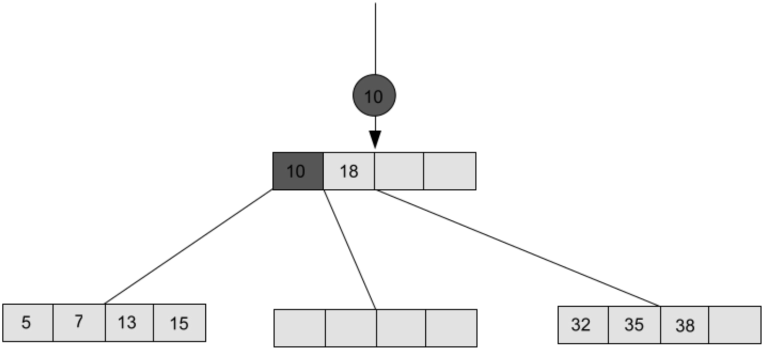
,



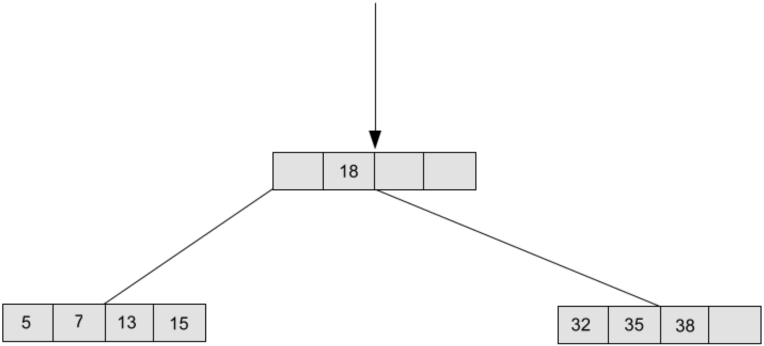
Remoção de Chaves - Caso 2(C)



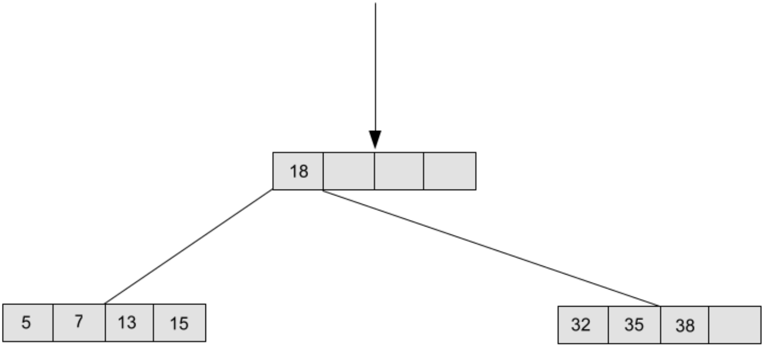
Remoção de Chaves - Caso 2(C)



Remoção de Chaves - Caso 2(C)



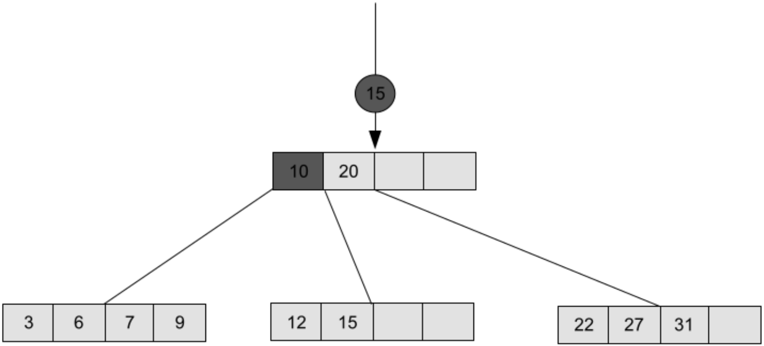
Remoção de Chaves - Caso 2(C)



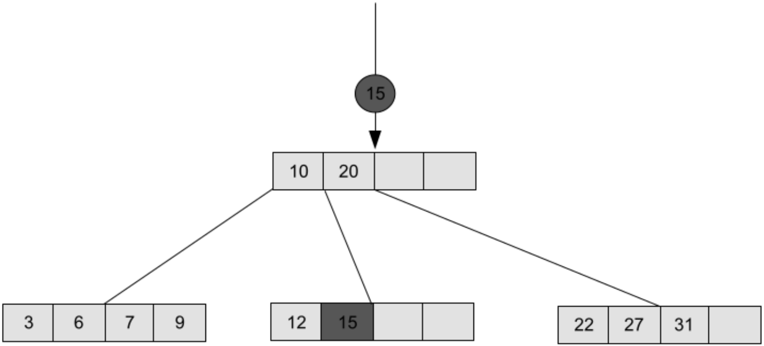
Remoção de Chaves

- Caso 3: Se a chave k não está presente no nó interno x, determina-se a sub-árvore $c_i[x]$ apropriada que deve conter k. Caso $c_i[x]$ possua somente $\lceil t/2 \rceil - 1$ chaves:
  - A. (Redistribuição) Se $c_i[x]$ possuir pelo menos $\lceil t/2 \rceil - 1$ chaves e possuir um irmão adjascente (irmão direto do mesmo pai) com pelo menos $\lceil t/2 \rceil$ chaves, copia-se para $c_i[x]$ uma chave extra, movendo uma chave de x para $c_i[x]$, em seguida movendo uma chave de um dos irmãos adjascentes de $c_i[x]$ de volta para x e ajustando o apontador para o nó correspondente
  - B. (Concatenação) Se $c_i[x]$ e ambos os seus irmãos esquerdo e direito possuem $\lceil t/2 \rceil - 1$ chaves, deve-se unir $c_i[x]$ com um dos irmãos. Isto envolve mover uma chave de x para o novo nó que acabou de ser criado, onde x é o elemento mediano daquele nó

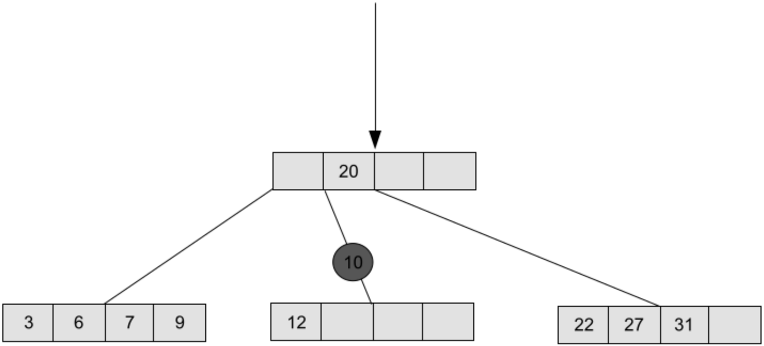
Remoção de Chaves - Caso 3(A)



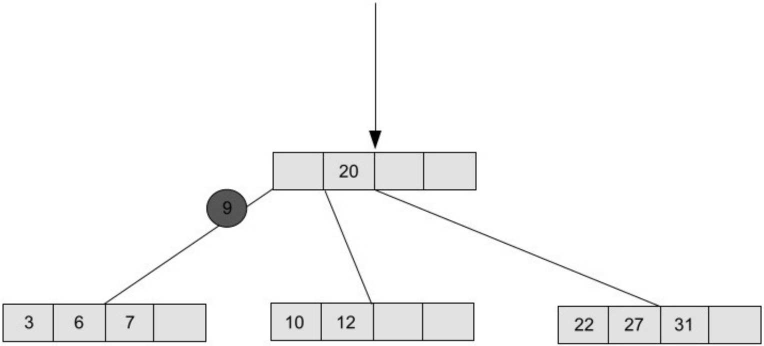
Remoção de Chaves - Caso 3(A)



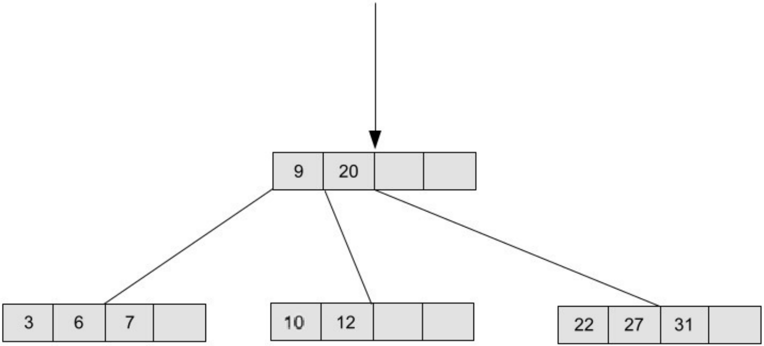
Remoção de Chaves - Caso 3(A)



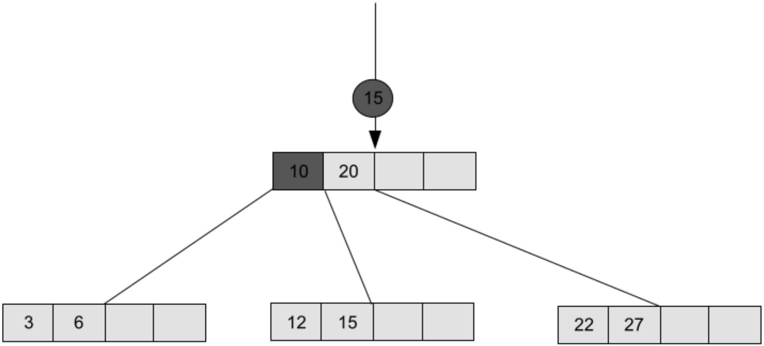
Remoção de Chaves - Caso 3(A)



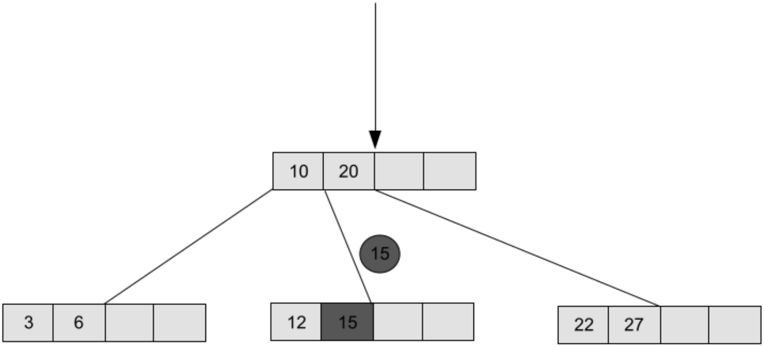
Remoção de Chaves - Caso 3(A)



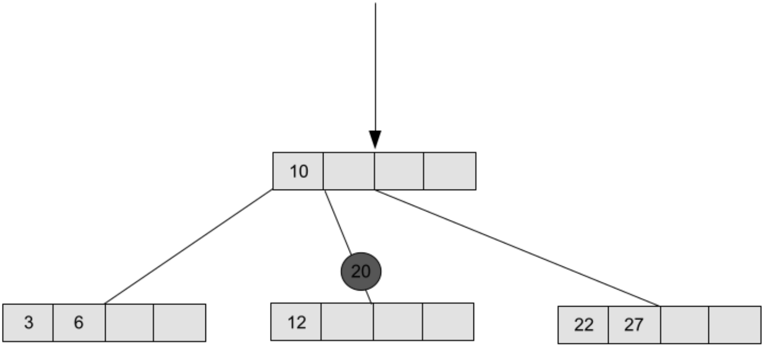
Remoção de Chaves - Caso 3(B)



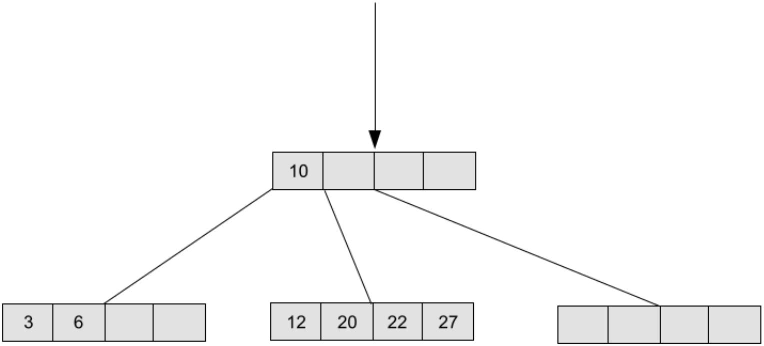
Remoção de Chaves - Caso 3(B)



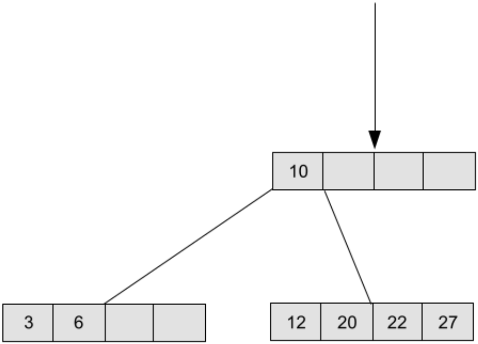
Remoção de Chaves - Caso 3(B)



Remoção de Chaves - Caso 3(B)



Remoção de Chaves - Caso 3(B)



Custo da remoção

- Antes da remoção é realizada uma busca na árvore, com custo $O(t * log_t(n))$, onde t é o tamanho da página da árvore e n é o número total de elementos
- No pior caso tem-se todas as páginas da árvore com $\lceil t/2 \rceil - 1$ elementos, já que esse é o limite inferior para cada página. Assim (e considerando que somente os casos 2C e 3B poderão ocorrer):
  - Para 2C: Tem-se o nó que perdeu a chave com $\lceil t/2 \rceil - 2$ chaves e $\lceil t/2 \rceil - 1$ filhos, pois já ocorreu um merge. O pai terá $\lceil t/2 \rceil - 2$ chaves e o elemento que perdeu o nó terá $\lceil t/2 \rceil$ chaves por causa do merge
  - Em qualquer um dos dois casos a reação disparada para corrigir a árvore será a mesma pois isso encaixa o nó com $\lceil t/2 \rceil - 2$ chaves na situação do caso 3b. Ou seja, uma chave será rebaixada da página pai para ele, e um merge dele com um dos irmãos será necessário

Custo da remoção

- (continuação) No pior caso tem-se todas as páginas da árvore com $\lceil t/2 \rceil - 1$ elementos, já que esse é o limite inferior para cada página. Assim (e considerando que somente os casos 2C e 3B poderão ocorrer):
  - Agora o nó pai possui $\lceil t/2 \rceil - 2$ chaves repetindo a situação anterior. Ou seja a operação pode propagar-se em um subconjunto de nós até chagar a raiz
  - Como as operações de merge copiam $\lceil t/2 \rceil - 1$ elementos a cada nível da árvore, tem-se um custo de $O((\lceil t/2 \rceil - 1) * log_t(n))$ para estas operações, onde $log_t(n)$ é a altura da árvore
  - Portanto, a complexidade da remoção é dada por $O(t * log_t(n)) + O((\lceil t/2 \rceil - 1) * log_t(n)) \approx O(t * log_t(n))$

Aplicativo interativo para árvore B

Para a visualização do processo de manipulação, inserção, busca e remoção em árvores B:

- https://www.cs.usfca.edu/ galles/visualization/BTree.html