Árvores B

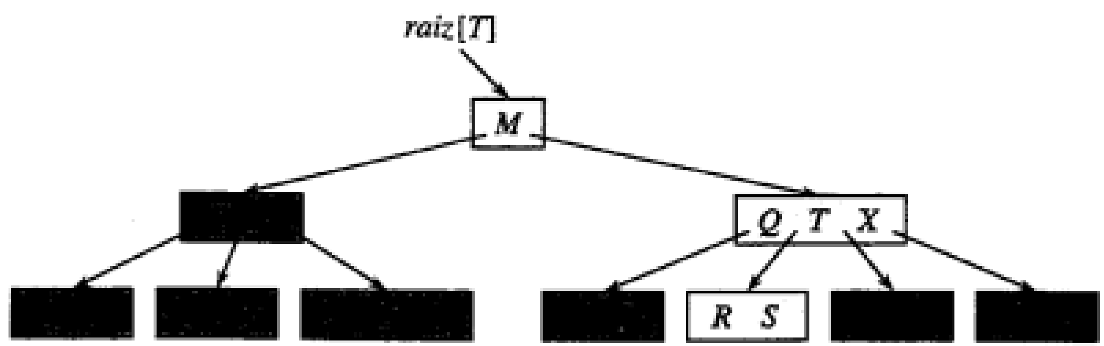
Prof. Lilian Berton

São José dos Campos, 2018

Baseado no material de Thomas Cormen

Árvores B

- **São árvores balanceadas**, projetadas para armazenamento secundário, pois **minimizam operações de E/S**. Também são utilizadas em banco de dados.
- **Os nós podem ter muitos filhos**. Seu fator de ramificação pode ser muito grande, sendo em geral determinado por características da unidade de disco.
- Se um nó interno x contém n[x] chaves, então terá n[x]+1 filhos.



Estrutura do armazenamento secundário

- Para amortizar o tempo gasto na espera por movimentos mecânicos, os discos não acessam apenas um item, mas vários de cada vez.

- As informações são divididas em páginas e cada leitura/gravação de disco inclui uma ou mais páginas.

- Muitas vezes, demora-se mais tempo para se obter acesso a uma página de informações e fazer a leitura da página que o tempo para o computador examinar todas as informações.

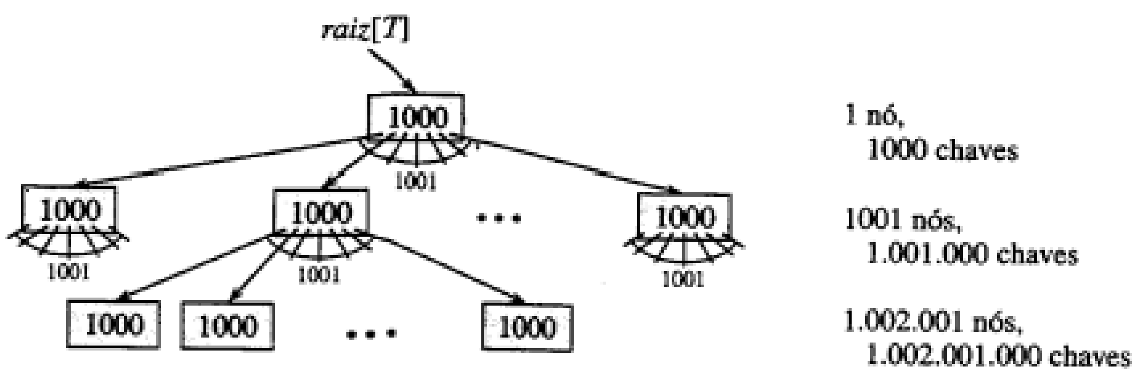
- **Em uma aplicação de árvore B, a quantidade de dados manipulada é tão grande que não cabem todos na memória principal de uma vez. Os algoritmos copiam páginas do disco para a memória principal e gravam o que for alterado.**

Árvore B no armazenamento secundário

- O tempo de execução da árvore B é determinado pelo número de **operações Disk-Read e Disk-Write**. Desse modo busca-se ler ou gravar o máximo de informações possíveis. Assim, um nó de uma árvore B é tão grande quanto uma página de disco inteira.

- **Um grande fator de ramificação reduz tanto a altura da árvore quanto o número de acessos ao disco necessários para encontrar qualquer chave.**

- A Figura abaixo mostra uma árvore B com fator de ramificação 1001 e altura 2, que pode armazenar mais de um bilhão de chaves. Como o nó raiz pode ser mantido na memória principal apenas dois acessos ao disco são exigidos para encontrar qualquer chave.



Propriedades de uma Árvore B

1. Todo nó x tem os seguintes campos:
   - n[x], o número de chaves atualmente armazenados em x;
   - As próprias n[x] chaves, tal que $chave_1[x] <= chave_2[x] <= chave_n[x]$
   - Folha[x], um valor booleano que é True se x é folha e False se x é um nó interno.

2. Cada nó interno x também contém n[x]+1 ponteiros $c_1[x]$, $c_2[x]$, ..., $c_{n+1}[x]$ para seus filhos. Os nós folhas não têm filhos e seus campos c_i são indefinidos.

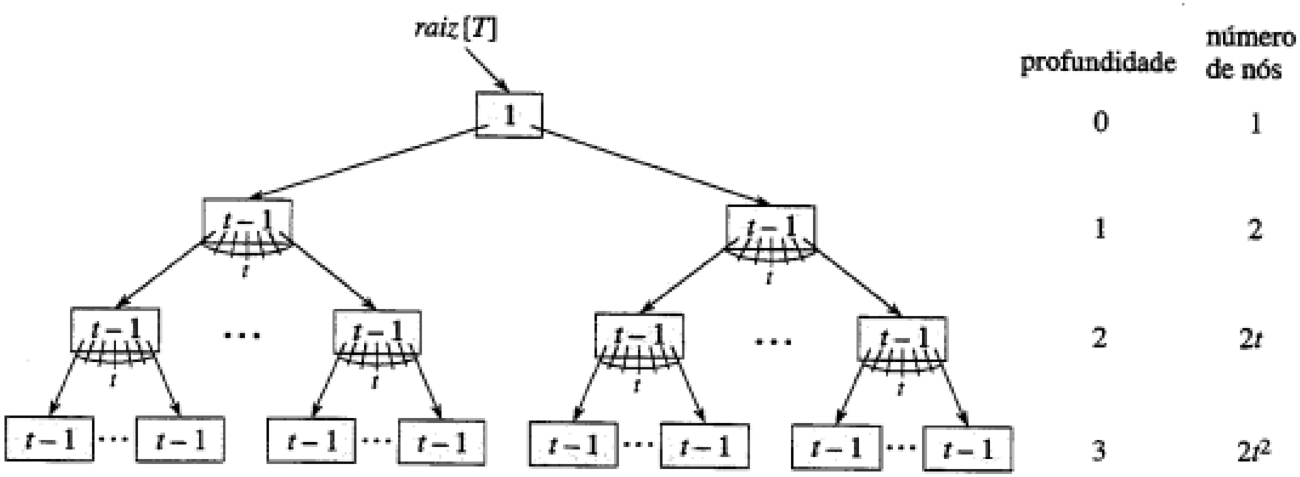
3. As chaves[x] separam os intervalos de chaves armazenados em cada subárvore: se k_i é uma chave armazenada na subárvore com raiz $c_i[x]$, temos $k_1 <= chave_1[x] <= k_2 <= chave_2[x] <= ... <= chave_n[x] <= k_{n+1}$

4. Toda folha tem a mesma profundidade que é a altura h da árvore.

5. Existem limites inferiores e superiores sobre o número de chaves que um nó pode conter expresso por um inteiro **t >= 2**.
   - Todo nó diferente da raiz deve ter ao menos t-1 chaves e t filhos.
   - Todo nó pode ter no máximo 2t-1 chaves e 2t filhos.

Altura da árvore B

- **A árvore B mais simples ocorre quando t = 2, todo nó interno tem 2, 3 ou 4 filhos e temos uma árvore 2-3-4. Na prática são utilizados valores bem maiores de t.**
- Para qualquer nó B de n nós de altura h e grau mínimo t >= 2, **$h \le \log_t (n+1)/2$**.
- A Figura abaixo mostra uma árvore B de altura 3 contendo um número mínimo possível de chaves.



Pesquisa em uma árvore B

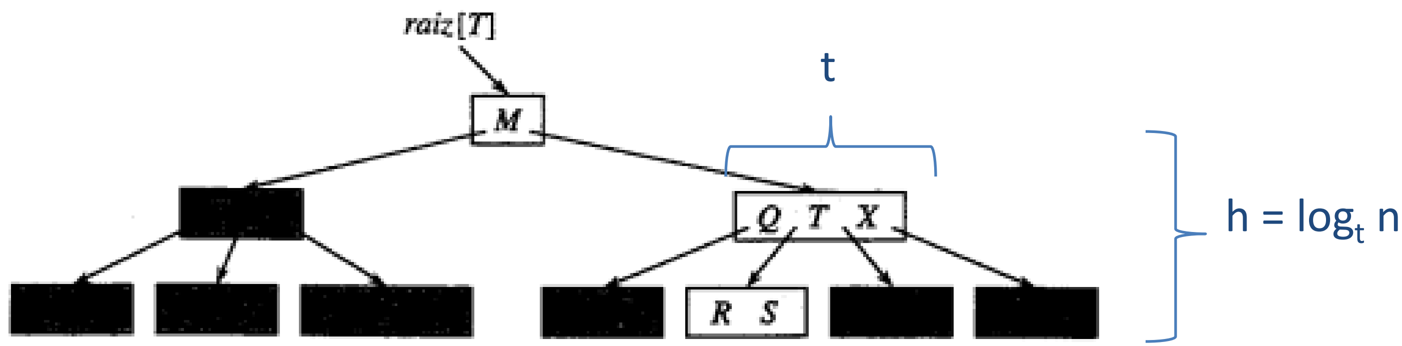
```
B-TREE-SEARCH(x, k)
1  i ← 1
2  while i ≤ n[x] e k > chave_i[x]
3     do i ← i + 1
4  if i ≤ n[x] e k = chave_i[x]
5     then return (x, i)
6  if folha[x]
7     then return NIL
8     else DISK-READ(c_i[x])
9        return B-TREE-SEARCH(c_i[x], k)
```

- A entrada é um ponteiro para o nó raiz x de uma subárvore e uma chave k a ser pesquisada nessa subárvore. Se k está na árvore B, retorna o par (x,i) que consiste no nó x e um índice i $chave_i[x] = k$, senão retorna Null.

- As linhas 1 a 3 encontram o menor i tal que k <= $chave_i[x]$, ou então definem i como n[x] +1.
- As linhas 4 e 5 verificam se a chave é encontrada.
- As linhas 6 a 9 encerram uma pesquisa malsucedida (x é folha) ou usam a recursão para pesquisar na subárvore de x, depois de executar Disk-Read sobre esse filho.

Complexidade da pesquisa em árvore B

- O número de páginas de disco as quais B-tree-search tem acesso é O(h) = O (log n) onde h é a altura da árvore B e n é o número de chaves na árvore B.
- Como n[x] < 2t, o tempo do loop while (linhas 2-3) dentro de cada nó é O(t) e **o tempo total da CPU é O(t h) = O (t $\log_t$ n).**



Criar árvore B vazia

```
B-TREE-CREATE(T)
x ← ALLOCATE-NODE( )
folha[x] ← TRUE
n[x] ← 0
DISK-WRITE(x)
raiz[T] ← x
```

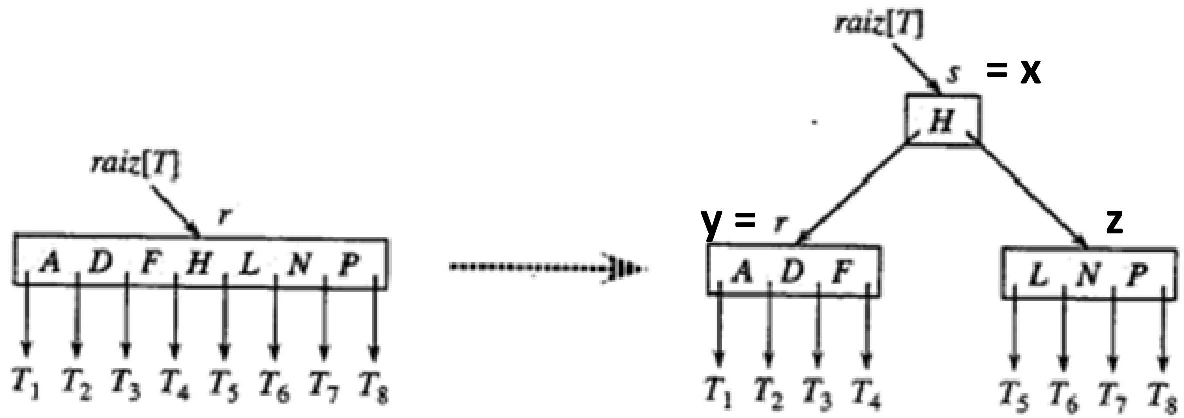
- Para construir a árvore primeiro usa-se B-tree-create para criar um nó de raiz vazio e depois o B-tree-insert para adicionar novas chaves. Ambos procedimentos usam Allocate-node que aloca uma página de disco para ser usada como um novo nó.

Partição de um nó cheio

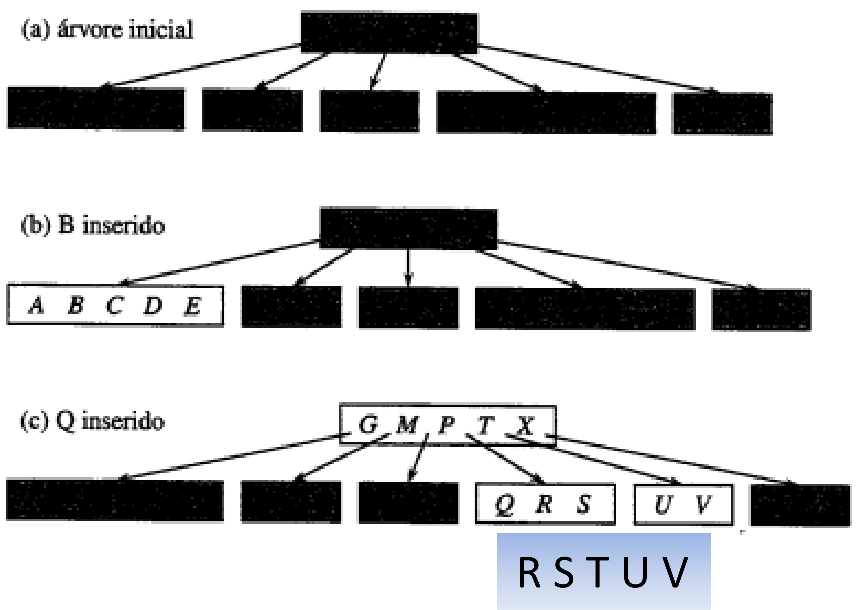
- Como não podemos inserir uma chave em um nó folha completo, dividimos um nó completo x (2t – 1 chaves) ao redor de sua chave mediana em dois nós que têm t – 1 chaves cada.
- A chave mediana se desloca para cima até o pai de x , indicando as duas novas subárvores.
- Se o pai de x também está completo ele também deve ser dividido e isso pode se propagar para cima na árvore.

Exemplo partição de um nó

• Divisão da raiz com t = 4. A raiz r é dividida em duas e cria-se uma nova raiz s.
• A nova raiz contém a chave mediana de r e tem as duas metades de r como filhos.
• A árvore B cresce em altura uma unidade quando a raiz é dividida.

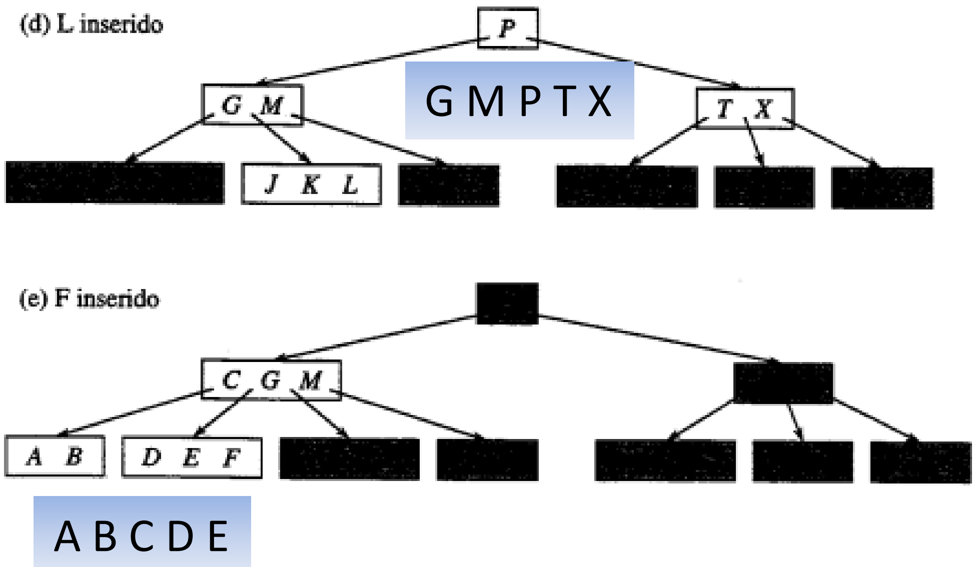


Exemplos de inserção em árvore B



- Nessa árvore t = 3. Assim um nó pode conter no máximo 5 chaves. Nós em branco estão sendo modificados.
- (a) árvore inicial.
- (b) inserção da chave B, é **uma inserção simples em um nó folha.**
- (c) inserção de Q. **O nó RSTUV é dividido em dois nós contendo RS e UV, a chave T é movida para cima até a raiz e Q é inserido na metade mais a esquerda das duas** (nó RS).

Exemplos inserção árvore B



- (d) inserção de L. **A raiz é dividida pois é completa, e a árvore B cresce uma unidade em altura**. L é inserida na folha (nó JK).
- (e) inserção de F. **O nó ABCDE é dividido antes de F ser inserido na metade mais a direita das duas partições** (nó DE).

Inserção em uma única passagem

```
B-TREE-INSERT(T, k)
1  r ← raiz[T]
2  if n[r] = 2t − 1
3     then s ← ALLOCATE-NODE( )
4          raiz[T] ← s
5          folha[s] ← FALSE
6          n[s] ← 0
7          c_1[s] ← r
8          B-TREE-SPLIT-CHILD(s, 1, r)
9          B-TREE-INSERT-NONFULL(s, k)
10    else B-TREE-INSERT-NONFULL(r, k)
```

- As linhas 3-9 tratam o caso no qual o nó raiz r é completo: a raiz é dividida e um novo nó s se torna raiz.
- O procedimento termina chamando B-tree-insert-nonfull para executar a inserção da chave k no nó r que se presume ser não cheio quando é chamado.

A inserção de uma chave k exige O(h) acessos ao disco. O tempo de CPU é $O(th) = O(t \log_t n)$.

Partição de um nó em árvore B

(s, 1, r)

```
B-TREE-SPLIT-CHILD(x, i, y)
 1  z ← ALLOCATE-NODE( )
 2  folha[z] ← folha[y]
 3  n[z] ← t – 1
 4  for j ← 1 to t – 1
 5      do chave_j[z] ← chave_{j+t}[y]
 6  if not folha[y]
 7      then for j ← 1 to t
 8               do c_j[z] ← c_{j+t}[y]
 9  n[y] ← t – 1
10  for j ← n[x] + 1 downto i + 1
11      do c_{j+1}[x] ← c_j[x]
12  c_{i+1}[x] ← z
13  for j ← n[x] downto i
14      do chave_{j+1}[x] ← chave_j[x]
15  chave_i[x] ← chave_t[y]
16  n[x] ← n[x] + 1
17  DISK-WRITE(y)
18  DISK-WRITE(z)
19  DISK-WRITE(x)
```

- As linhas 1-8 criam o nó z e dão a ele as t–1 chaves maiores e os t filhos correspondentes de y.
- A linha 9 ajusta a contagem de chaves para y.
- As linhas 10-16 inserem z como um filho de x, movem a chave mediana de y para cima até x, a fim de separar y de z e ajustam a contagem de chaves de x.
- As linhas 17-19 gravam todas as páginas de disco modificadas.
- O tempo de CPU é t e de operações de disco é O(1) .

Inserção árvore B

```
B-TREE-INSERT-NONFULL(x, k)
1  i ← n[x]
2  if folha[x]
3     then while i ≥ 1 e k < chave_i[x]
4              do chave_{i+1}[x] ← chave_i[x]
5                 i ← i − 1
6          chave_{i+1}[x] ← k
7          n[x] ← n[x] + 1
8          DISK-WRITE(x)
9     else while i ≥ 1 e k < chave_i[x]
10             do i ← i − 1
11         i ← i + 1
12         DISK-READ(c_i[x])
13         if n[c_i[x]] = 2t − 1
14            then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                 if k > chave_i[x]
16                    then i ← i + 1
17         B-TREE-INSERT-NONFULL(c_i[x], k)
```

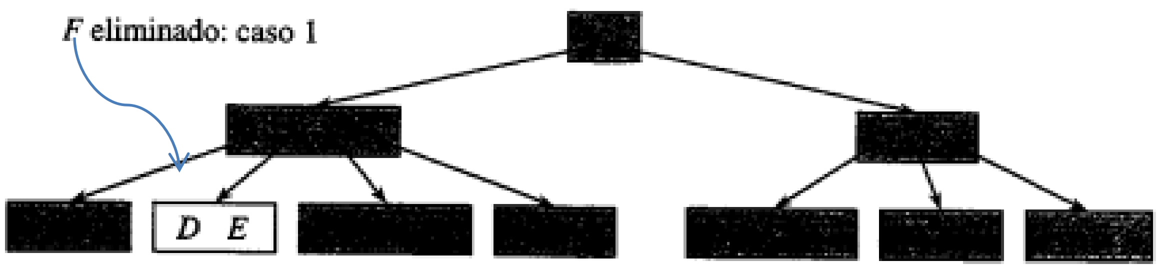
- As linhas 3-8 tratam o caso no qual x é um nó folha, inserindo a chave k em x.
- Se x não é um nó folha então devemos inserir k no local apropriado. Nesse caso as linhas 9-11 determinam o filho de x para o qual a recursão é descendente.
- A linha 13 detecta se a recursão desceria até um filho completo. Nesse caso a linha 14 chama B-tree-split-child para dividir esse filho.
- As linhas 15-16 determinam qual dos dois filhos é agora o filho correto para o qual se deve descer.
- A linha 17 utiliza a recursão para inserir k na subárvore apropriada.

Eliminação em uma árvore B – caso 1

- Assim como tivemos que assegurar que um nó não ficasse grande demais na inserção, devemos assegurar que um nó não fique pequeno demais durante a eliminação.

- A eliminação pode ter os seguintes casos:

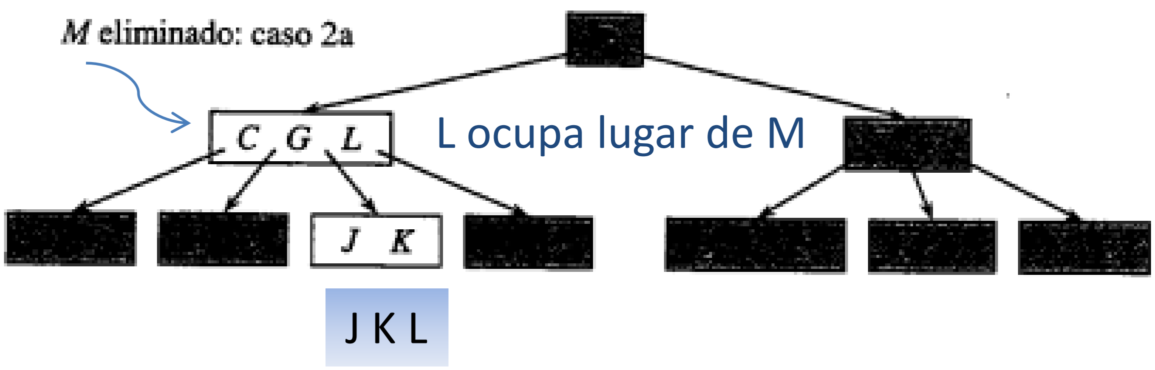
1. Se a chave k está no nó x e x é uma folha, elimine a chave k de x.



Eliminação em uma árvore B – caso 2a e 2b

2. se a chave k está no nó x e x é um nó interno, faça:

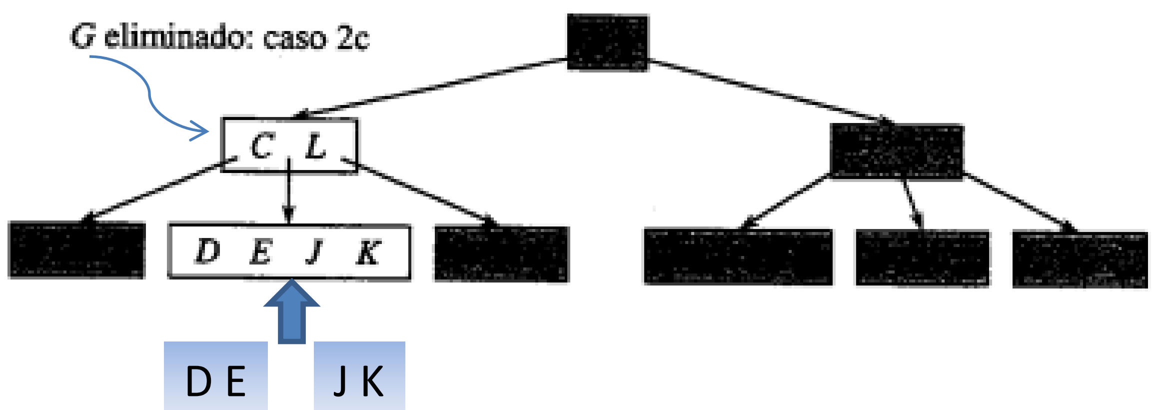
- (a) se o filho y que precede k no nó x **tem pelo menos t chaves**, então encontre o **predecessor** k´ de k na subárvore com raiz em y. Elimine recursivamente k´ e substitua k por k´ em x.
- (b) se o filho z que segue k no nó x **tem pelo menos t chaves**, então encontre o **sucessor** k´ de k na subárvore com raiz em z. Elimine recursivamente k´ e substitua k por k´ em x.
- (c) ...



Eliminação em uma árvore B – caso 2c

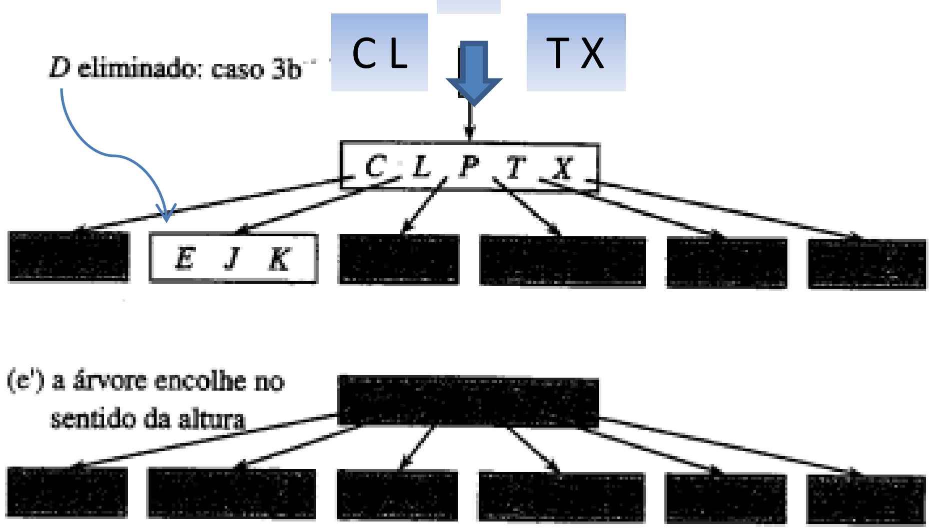
2. se a chave k está no nó x e x é um nó interno, faça:

- (a) ...
- (b) ...
- (c) caso contrário, se tanto y quanto z **têm apenas t – 1 chaves**, faça a intercalação de k e todos os itens z em y, de modo que **x perca tanto k quanto o ponteiro para z**, e **y contenha agora 2t – 1 chaves**. Libere z e elimine recursivamente k de y.



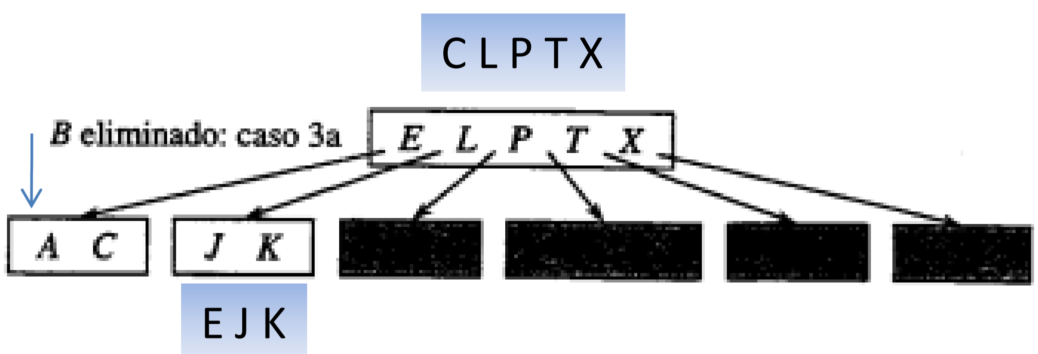
Eliminação em uma árvore B – caso 3b

3. **se a chave k não estiver presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k, se k estiver na árvore. Se $c_i[x]$ tiver somente t - 1 chaves, execute o passo 3(a) ou 3(b)** conforme necessário para garantir que **desceremos até um nó contendo pelo menos t chaves**. Encerre efetuando uma recursão sobre o filho apropriado de x.
   – (a) ...
   – (b) Se $c_i[x]$ e todos os irmãos têm t – 1 chaves, faça a intercalação de $c_i[x]$ com um único irmão, o que envolve mover uma chave de x para baixo até o novo nó intercalado, a fim de se tornar a chave mediana para esse nó.



Eliminação em uma árvore B – caso 3a

3. **se a chave k não estiver presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k, se k estiver na árvore. Se $c_i[x]$ tiver somente t – 1 chaves, execute o passo 3(a) ou 3(b) conforme necessário** para garantir que desceremos até um nó contendo pelo menos t chaves. Encerre efetuando uma recursão sobre o filho apropriado de x.
   – (a) se $c_i[x]$ tiver somente t – 1 chaves, mas tiver um irmão com t chaves, forneça a $c_i[x]$ uma chave extra, movendo uma chave de x para baixo até $c_i[x]$, movendo uma chave do irmão esquerdo ou direito imediato de $c_i[x]$ para dentro de x, e movendo o ponteiro do filho apropriado do irmão para $c_i[x]$.
   – (b) ...



Comparação entre as árvores

- Embora a altura das árvores cresça na proporção O (log n), para as árvores B, a base do logaritmo pode ser muitas vezes maior.

- As árvores B poupam um fator de log t sobre as árvores vermelho-preto no número de nós examinados para a maioria das operações.

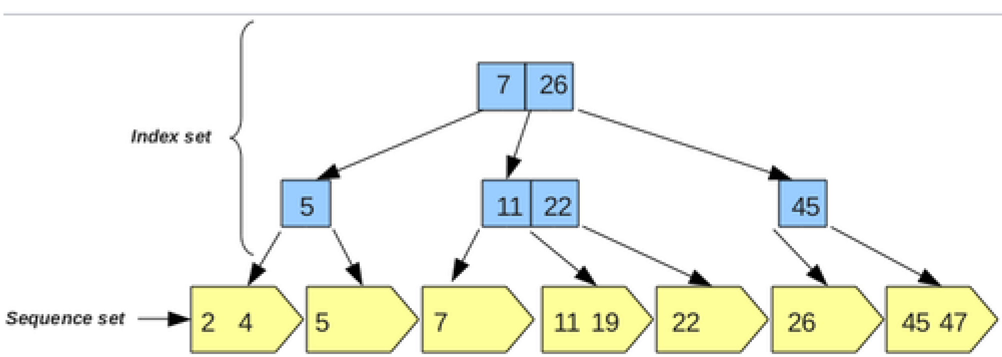
Árvores B+

- Assim como as árvores B, as **árvores B+** visam reduzir as operações de leitura e escrita em memória secundária, uma vez que, essas operações são demoradas para um sistema computacional e devem ser minimizadas sempre que possível.

- As **árvores B+** possuem seus dados armazenados somente em seus nós folha e, seus nós internos e raiz, são apenas referências para as chaves que estão em nós folha. Assim é possivel manter ponteiros em seus nós folha para um acesso sequencial ordenado das chaves contidas no arquivo.

Árvore B+

- Como nas árvore B, as chaves estão ordenadas tanto em suas páginas internas quanto em páginas folha.
- Cada nó folha contém apontadores para quais nós são seus predecessores ou sucessores na sequência de chaves
- Dessa forma, quando realizamos uma busca por uma chave k e para encontrarmos a chave k+1, ou seja sua sucessora na ordem, basta carregar a próxima página contida na lista de páginas para verificar qual chave sucede k.



Inserção

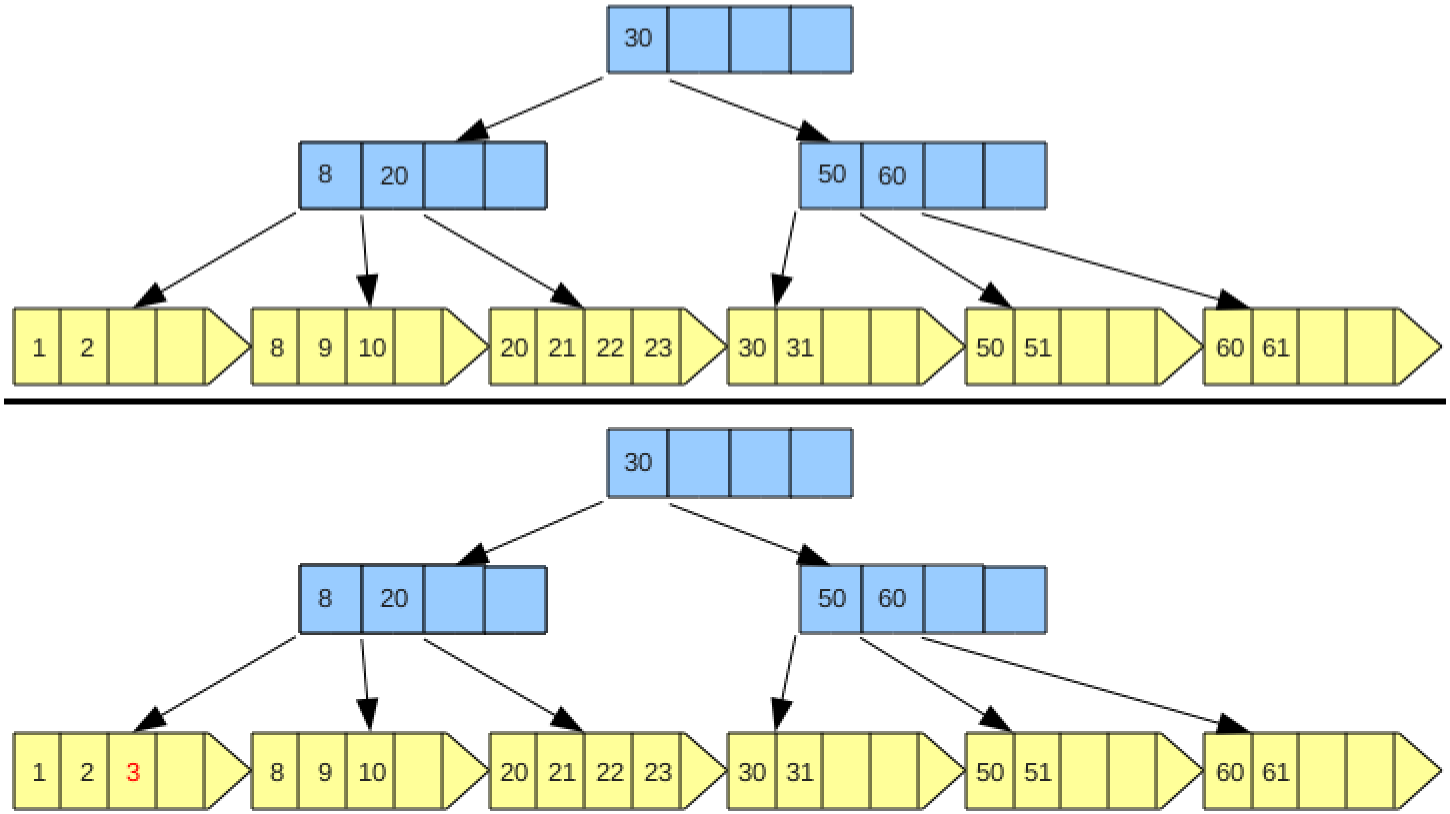
- Após buscar a página folha que uma chave deve ser inserida, devemos analisar dois casos:

- **Página folha incompleta**: inserimos a chave de maneira a manter a ordenação das chaves.

- **Página folha completa**: A página folha em questão deve sofrer uma operação de *split*. Tal operação cria uma nova página em arquivo dividindo as chaves entre a nova página e a anterior.
- Após isso a chave intermediária deve ser inserida no *index set* semelhante ao processo de inserção em árvore B.

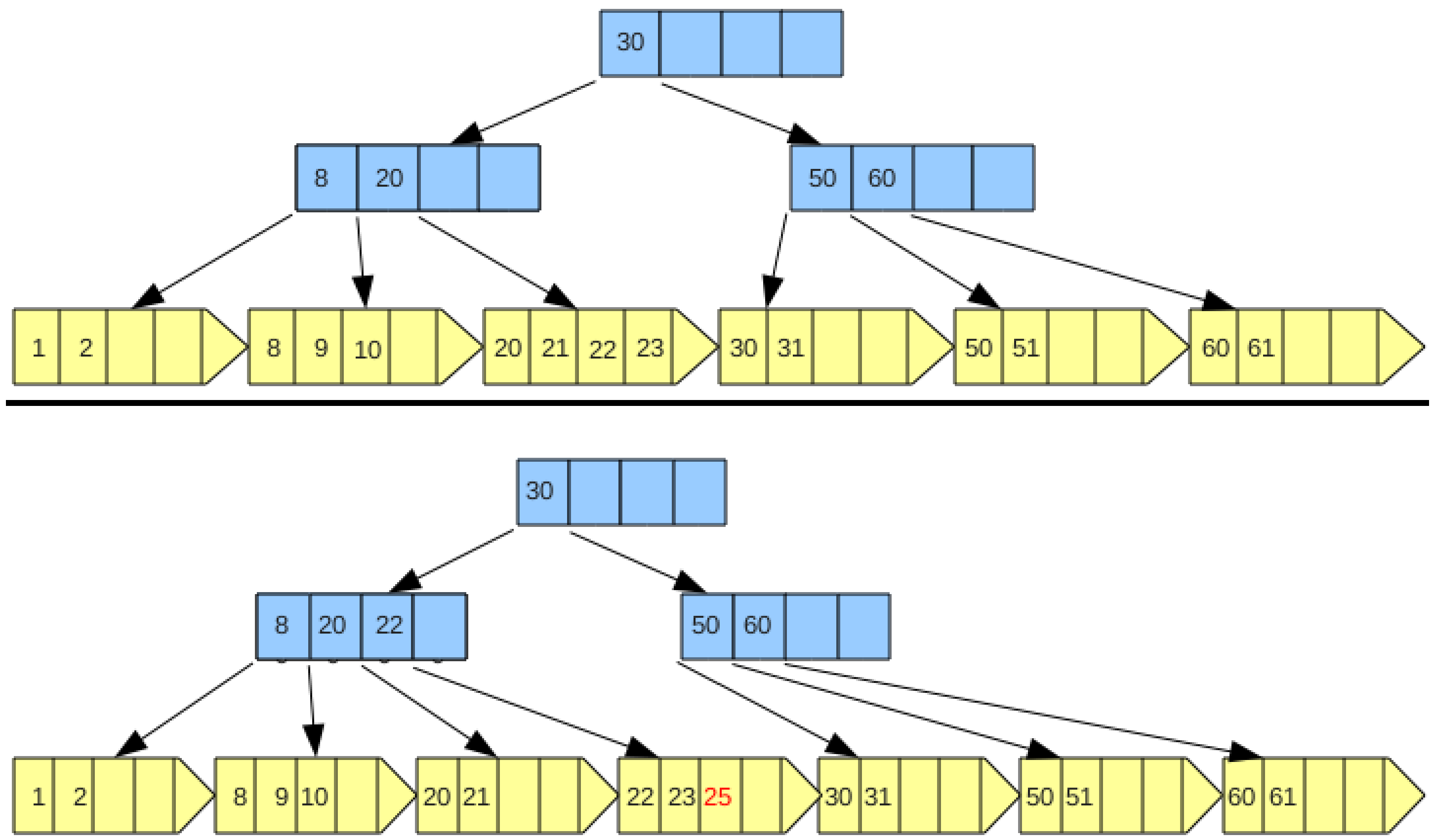
Inserção: página folha incompleta

- Inserir elemento 3:



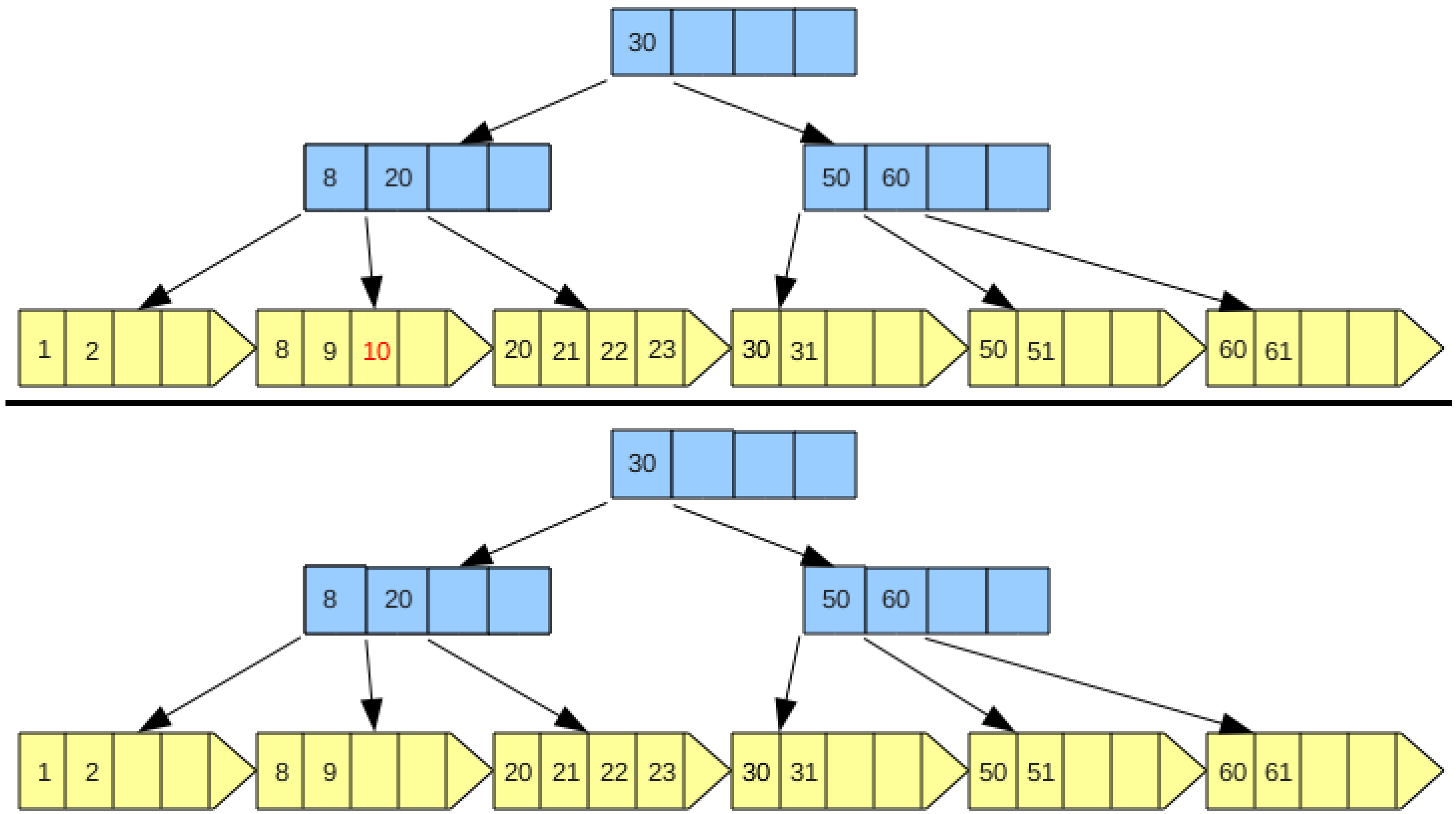
Inserção: página folha completa

- Inserir elemento 25:



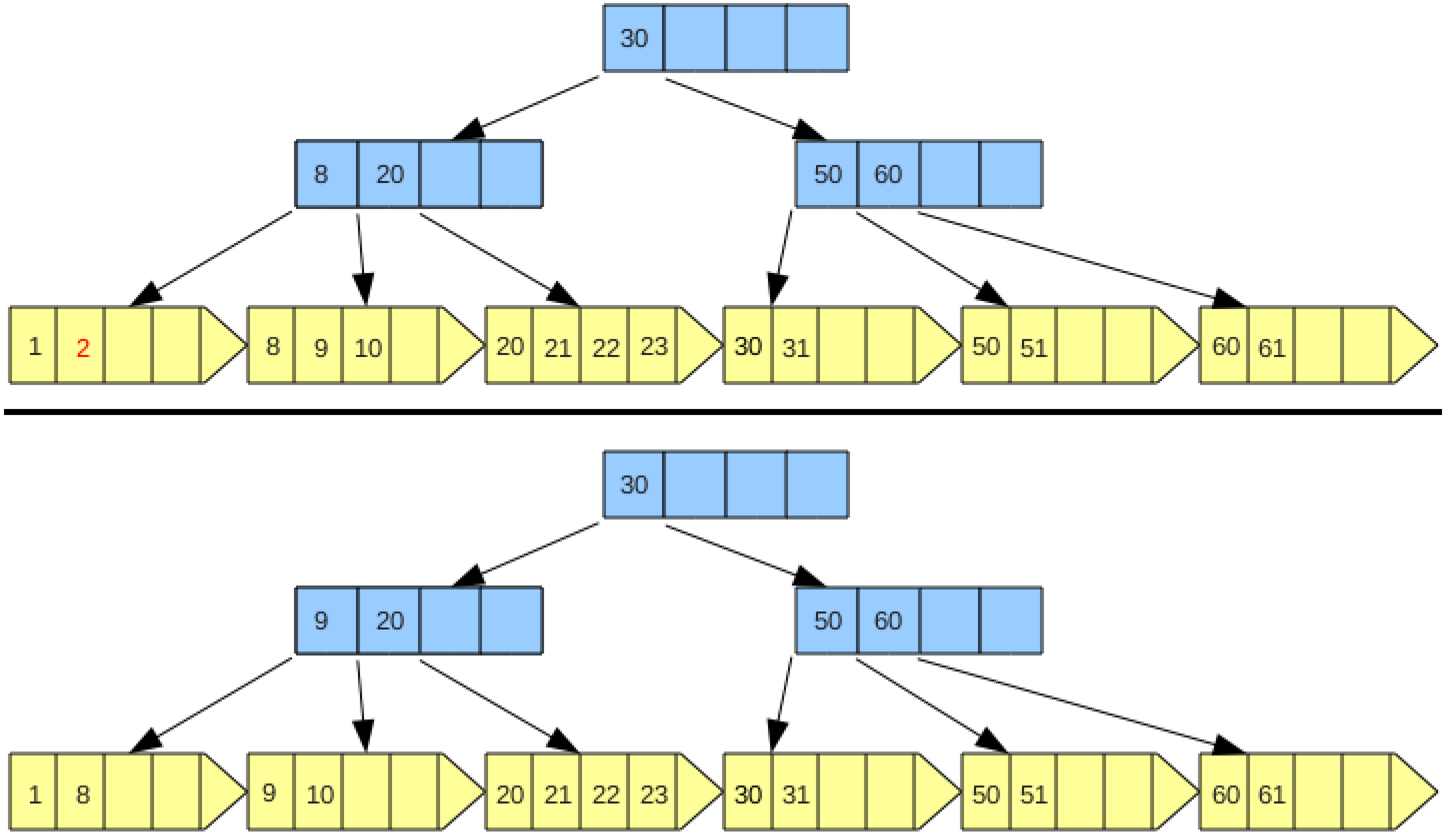
Remoção: simples

- Remoção da chave 10.



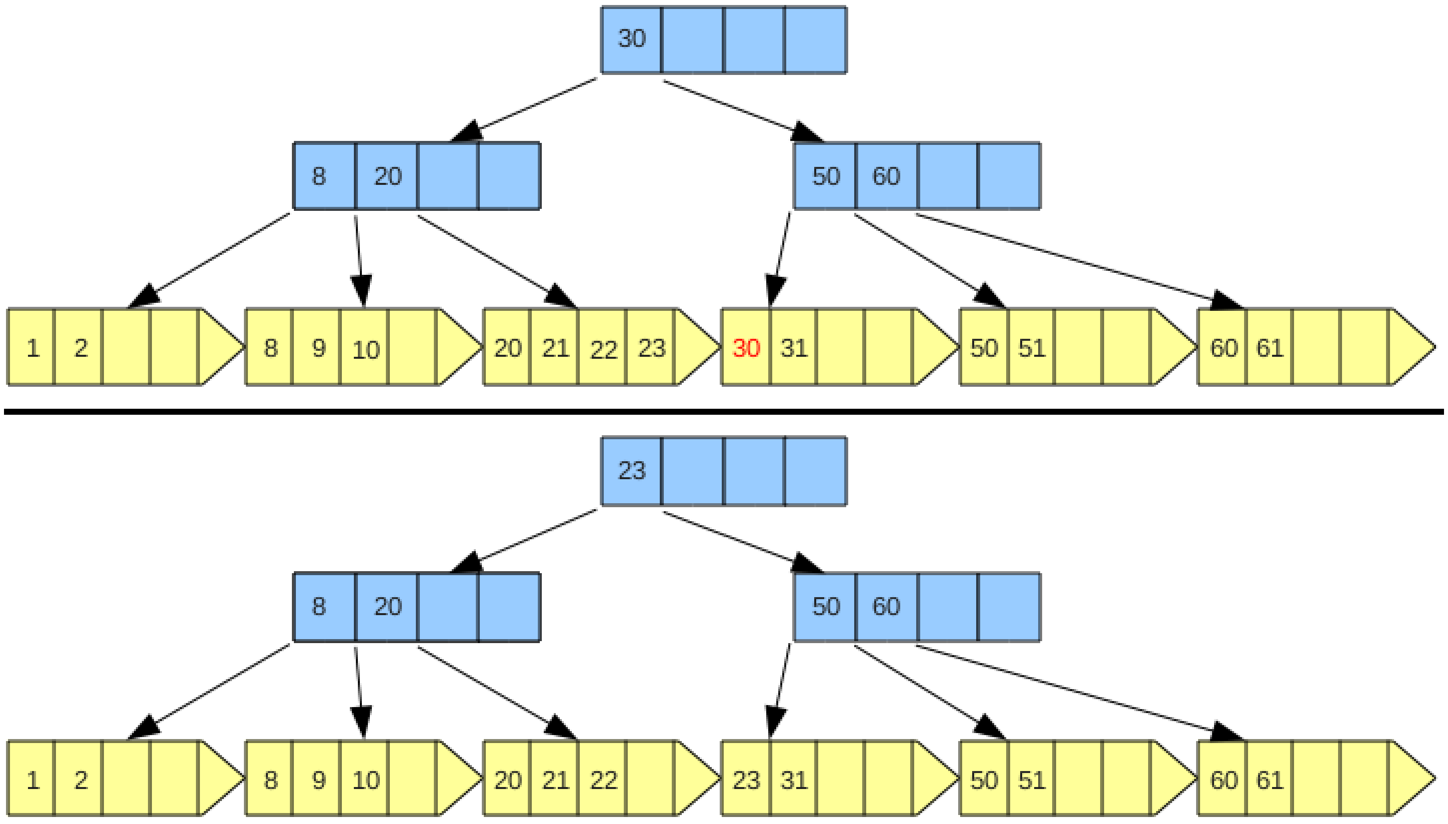
Remoção: nó incompleto empresta elemento

- Remoção da chave 2:



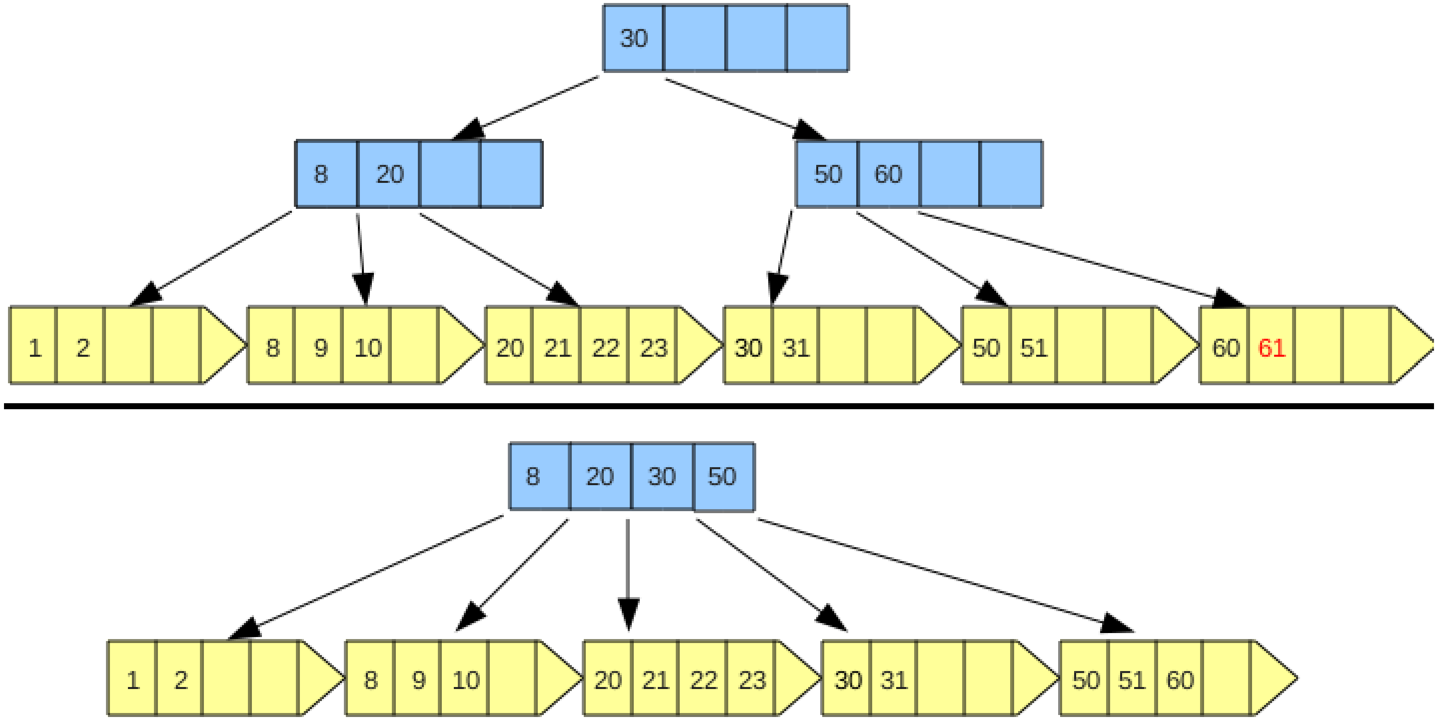
Remoção: nó incompleto empresta elemento

- Remoção da chave 30:



Remoção: junção de chaves

- Remoção da chave 61:



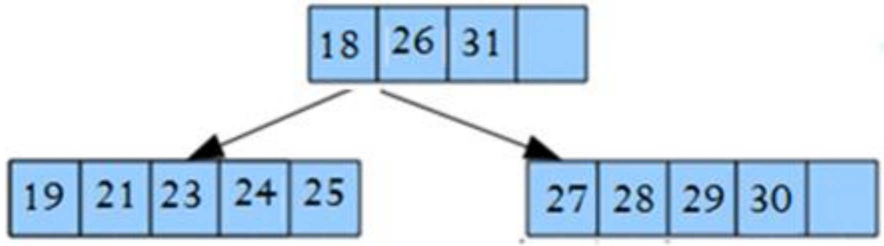
Árvores B*

- As chaves numa **árvore B*** ficam armazenadas nas páginas internas, folha e raiz.

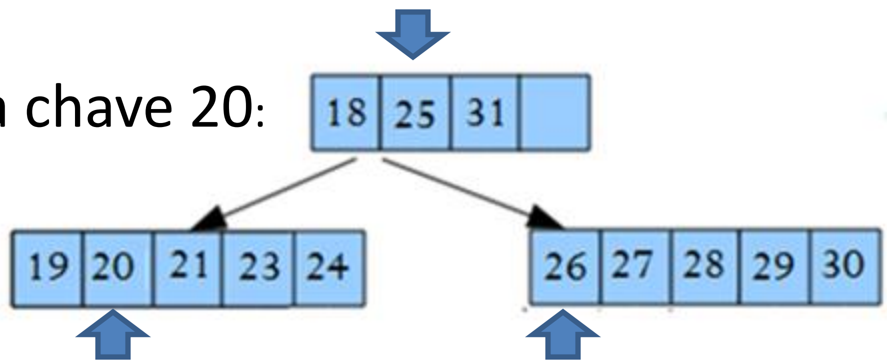
- A principal diferença é relacionada ao momento de inserção de chaves, na qual utiliza a redistribuição de chaves entre páginas irmãs até que estas estejam completamente cheias.

- Desse modo a operação de split é atrasada até que duas páginas irmãs estejam completamente cheias, conferindo maior aproveitamento de espaço em arquivo.

Inserção

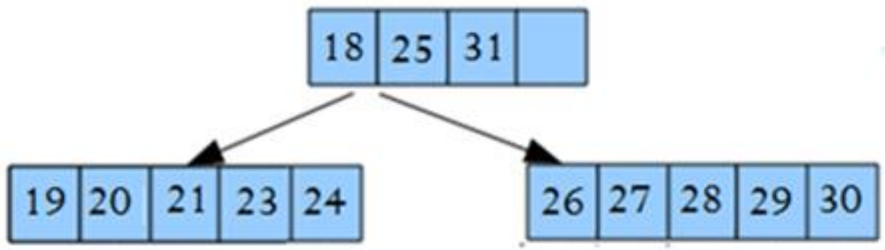


Inserção da chave 20:

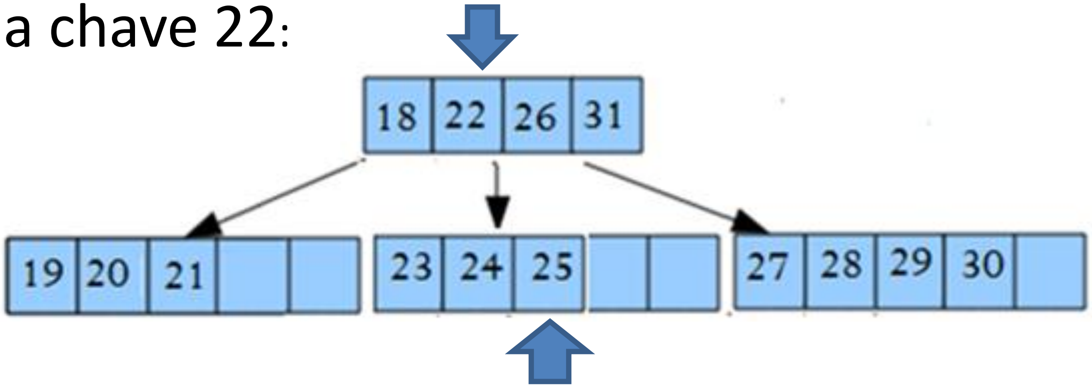


O elemento mais a direita (**25)** sobe ocupando a posição de quem apontava para esse nó (**26**), que passa para o nó irmão

Inserção



Inserção da chave 22:



O nó onde o novo item seria inserido e seu irmão já estão cheios, o nó é então dividido ao meio. O elemento mais a direita (**22)** sobe, o **25** desce e o elemento mais a esquerda do nó irmão (**26)** ocupa o lugar do **25** no nó pai.

Comparações

- **Árvore B+:**
- A principal característica proporcionada por esta variação é o fato de permitir o acesso sequencial das chaves por meio de seu *sequence set* de maneira mais eficiente do que o realizado em árvores B .
- Além do mais, as páginas utilizadas em seu *index set* podem conter mais apontadores para páginas filha permitindo reduzir a altura da árvore.
- **Árvore B*:**
- A principal vantagem decorrente dessa variação é o fato desta apresentar suas páginas com no mínimo 2/3 do número máximo de chaves, ou seja, esta variação apresenta no pior caso um índice de utilização do arquivo de 66%, enquanto em árvores B esse índice de pior caso cai para 50%.

Exercícios

1. Fazer um resumo sobre árvores AVL, Vermelho-Preto, B e suas variações.

2. Insira as seguintes chaves numa árvore B e depois em uma B+ e B* com t = 2:

   {45, 66, 1, 5, 99, 41, 10, 11, 7, 15, 21, 75, 80}

3. Remova as chaves 41 e 11 nas árvores B anteriores.

4. Estude uma implementação de árvore B. Faça upload do código estudado.

Sugestão: ver implementação do Ziviane ou outro autor.

Os analfabetos do século 21 não serão os que não conseguem ler e escrever, mas os que não conseguem aprender, desaprender e reaprender.

Alvin Toffler