UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
São José do Rio Preto - SP

JOGI SUDA NETO

ÁRVORES B, B+, B* E APLICAÇÕES

São José do Rio Preto - SP
2020

JOGI SUDA NETO

ÁRVORES B, B+, B* E APLICAÇÕES

Dissertação apresentada como parte dos requisitos para a disciplina de Estrutura de Dados, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista "Júlio de Mesquita Filho", Câmpus de São José do Rio Preto.

Prof. Dra. Renata Spolon Lobato

São José do Rio Preto - SP
2020

RESUMO

Neste trabalho serão apresentadas as estruturas de dados do tipo árvore B, B+ e B*, assim como suas aplicações, algoritmos de busca, inserção e remoção.

Palavras-chave: Estrutura de Dados, Árvores, árvores B, árvores B+, árvores B*.

Lista de Figuras

Lista de Tabelas

Sumário

1 Árvores B

1.1 Introdução

As árvores B foram projetadas para serem eficazes com acesso a discos magnéticos ou quaisquer dispositivos de armazenamento secundários (Cormen et al., 2009). Foram inventadas em 1979 por Rudolf Bayer e Edward Meyers McCreight (Bayer e McCreight, 2002), pesquisadores da Boeing Scientific Labs. Aqui, temos uma grande vantagem que, nas árvores B, a complexidade de tempo no caso médio e pior caso, para todas as operações de inserção, remoção e busca, são **sempre** $O(logn)$!

Importante dizer que, para as árvores binárias, após operações de inserção/remoção, estas estruturas podem ficar desbalanceadas e, no pior caso, degeneradas, ou seja, se transformarem em uma lista simples encadeada. A beleza das árvores B (e, por consequência, também as B+ e B*) está justamente no fato de que a árvore **sempre** estará balanceada dado a maneira como as operações de inserção e remoção serão feitas!

1.2 Definição

Dizemos que uma árvore B é uma definição mais geral das árvores binárias de busca, pois as árvores binárias comportam apenas um valor de chave por nó, enquanto as árvores B comportam naturalmente mais. Também dizemos que são árvores que crescem de baixo para cima, ou seja, *bottom-up*.

Uma árvore é do tipo B, se atende alguns requisitos:

- A raiz tem pelo menos duas subárvores, a não ser que seja uma folha.
- Seja m o número máximo de filhos por nó: dizemos que m é a ordem da árvore.
- Naturalmente, cada nó terá no máximo m-1 chaves.
- Cada nó não-folha e não-raiz deve ter $k - 1$ chaves e k ponteiros para seus filhos, onde $\lceil m/2 \rceil \leq k \leq m$.
- Cada folha deve ter $k - 1$ ponteiros para seus filhos, onde $\lceil m/2 \rceil \leq k \leq m$.

- Todas as folhas estão sempre no **mesmo** nível!

O algoritmo abaixo define uma estrutura que representa um nó dessa árvore, e logo abaixo, uma função para criar de fato um novo nó:

```
1 #define MAX 3
2 #define MIN 2
3
4 struct BTreeNode {
5   int val[MAX + 1], count;
6   struct BTreeNode *link[MAX + 1];
7 };
8
9 struct BTreeNode *raiz;
10
11 // Cria n
12 struct BTreeNode *criarNo(int val, struct BTreeNode *filho) {
13   struct BTreeNode *novoNo;
14   novoNo = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
15   novoNo->val[1] = val;
16   novoNo->count = 1;
17   novoNo->link[0] = raiz;
18   novoNo->link[1] = filho;
19   return novoNo;
20 }
```

Listing 1: Definição da TAD Árvore B (em C)

1.3 Busca

A busca nas árvores B segue uma lógica parecida com as árvores de busca binária, com a exceção de que agora temos mais chaves e filhos por página.

Considerando a busca por uma chave K, começamos pela raiz: iteramos todas as chaves enquanto forem maiores que K (ou menores, se percorrermos em ordem crescente as chaves). A busca pára se a próxima chave for maior ou igual a K, ou se chegarmos ao final da página. Se a chave for igual a K, encontramos o elemento; caso contrário, devemos buscar na subárvore-filha correspondente ao índice atual. Se a busca atingir uma folha e a chave não for encontrada, então K não está na árvore. O algoritmo a seguir mostra isto:

```
1  // Busca a chave
2  void busca(int val, int *pos, struct BTreeNode *myNode) {
3    if (!myNode) {
4      return;
5    }
6    if (val < myNode->val[1]) {
7      *pos = 0;
8    } else {
9      for (*pos = myNode->count;
10         (val < myNode->val[*pos] && *pos > 1); (*pos)--);
11     if (val == myNode->val[*pos]) {
12       printf("Chave %d encontrada.\n", val);
13       return;
14     }
15   }
16   busca(val, pos, myNode->link[*pos]);
17   return;
18 }
```

Listing 2: Algoritmo de busca (em C)

1.4 Inserção

A inserção nessas árvores será sempre feita nas folhas: começamos com apenas uma página, que será a raiz, e assim vamos inserindo de maneira sempre ordenada. Quando esta estiver lotada e formos inserir o próximo elemento, realizaremos a **cisão** desta página em outras duas (uma página irmã e outra página pai destas duas), de maneira que uma das chaves (a mediana caso número de chaves seja ímpar, ou quaisquer uma das duas chaves-centro, caso o número de chaves seja par) será promovida para a página pai, tornando-se a nova página separadora. A metade maior irá para a página irmã direita, e a outra metade (menor), ficará na irmã esquerda. Assim continuamos a inserção, até que a página-pai esteja lotada e uma nova cisão deva ser realizada, e assim por diante, lembrando sempre que, após uma inserção/cisão, as chaves das páginas afetadas devem ser sempre reorganizadas (ordenadas).

Segue o exemplo (Drozdek, 2013):

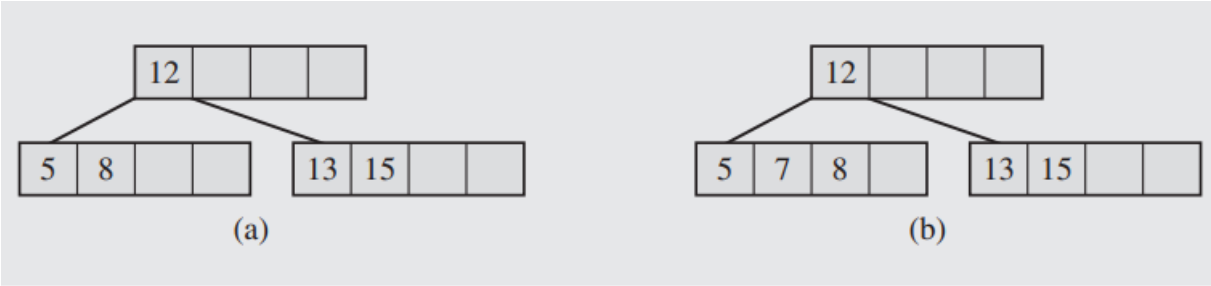

Figura 1: Elemento '2' é inserido na árvore.

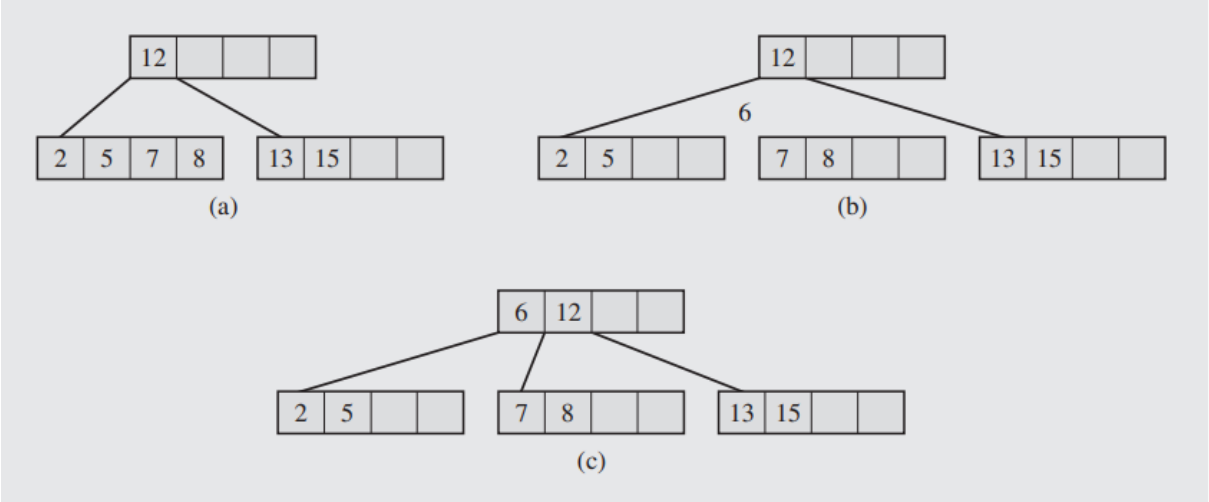

Figura 2: Elemento '6' é inserido na árvore, mas a página está cheia. A redistribuição então é aplicada, '6' é promovido à página pai (chave separadora) e as chaves das páginas afetadas são reorganizadas. Lembrando que poderíamos ter escolhido o '7' para ser promovido ao invés do '6' (a critério do programador).

A seguir é apresentado um algoritmo para inserção nas árvores B (Drozdek, 2013):

```
1 arvoreBInsercao(K)
2  encontre a folha certa para inserir K;
3    enquanto (verdade)
4     encontre a posicao adequada no vetor de chaves para K;
5      se a pagina nao esta cheia
6       insira K e incremente o contador de chaves;
7       retorne;
8      se nao, faca a cisao da pagina em no1 e no2;// no1 =
          pagina, no2   novo;
9        redistribua as chaves igualmente entre no1 e no2;
10       atualize os contadores;
11       K = chave mediana;
12       se a pagina era a raiz
13          crie uma nova raiz como pai de no1 e no2;
14          coloque K e ponteiros para no1 e no2 na raiz, e
```

```
            atualize seu contador de chaves para 1;
15          retorne;
16      se nao pagina = seu pai;// e agora processe o pai desta
            pagina;
```

Listing 3: Algoritmo de inserção

1.5 Remoção

Para a remoção nas árvores B, devemos analisar os seguintes casos:

- Caso 1: a chave a ser excluída está em um nó folha, e após a remoção desta chave, o número mínimo de chaves ainda é respeitado (ou seja, não há underflow).
- Caso 2: a chave a ser excluída está em um nó não-folha.
- Caso 3: A eliminação causou underflow na página (deve-se tentar **redistribuir**: será explicado adiante).
- Caso 4: temos underflow na página (caso 3), mas não foi possível aplicar redistribuição (deve-se aplicar **concatenação**, também a ser explicado adiante).
- Caso 5: a concatenação se propagou até a raiz: ocorrerá então uma diminuição da altura da árvore.

Para o caso 1, a solução é bem trivial, bastando remover a chave da página (folha) em si.

Para o caso 2, situação em que queremos excluir uma chave de uma página não-folha, basta trocarmos a chave a ser removida com sua sucessora imediata (ou predecessora imediata, fica a critério do programador) que está em uma folha. Logo depois, excluímos então a chave que agora está na folha.

Para o caso 3, deve-se primeiramente checar se alguma das páginas irmãs possui chaves sobrando para redistribuir (ou seja, possuem mais que $\lceil m/2 \rceil - 1$ chaves!) igualmente. Pode ser que a redistribuição acarrete em uma alteração na chave pai.

Para o caso 4, em que há underflow na página após eliminação da chave, caso não consigamos aplicar redistribuição, devemos aplicar concatenação: juntaremos os conteúdos das páginas irmãs com a chave pai delas em uma página só.

Para o caso 5, não há muito a ser explicado: veremos que a concatenação é um processo **propagável**, ou seja.pode ocorrer underflow na página pai, sendo que neste caso deve-se repetir a concatenação nesse nó. Eventualmente, se esse processo atingir a raiz. a árvore diminui de altura!

1.6 Remoção com concatenação

Diremos que duas páginas **R** e **S** serão adjacentes se tiverem o mesmo pai T em comum, e são apontadas por ponteiros adjacentes em **T**. Então, se **R** e **S**, juntas, possuírem um total de chaves menor que $m - 1$, podemos economizar complexidade de espaço concatenando-as, ou seja, concatenando as chaves de R e S em apenas uma página. Além disso, o nó pai de **R** e **S** simplesmente deixará de existir. Segue o exemplo:

Note que, caso o nó pai de **R** e **S**, após o processo de concatenação, tiver menos que $(m - 1)/2$ chaves, esse processo se repetirá até que o número de chaves resultante esteja de acordo com a definição da árvore B+. Por este motivo, também se diz que o processo de concatenação é propagável, sendo que, se este atingir a raiz, a árvore diminuirá de altura.

Além disso, vale lembrar que a concatenação é o processo **inverso** da cisão de uma página que utiliza-se na inserção, quando há overflow. Assim sendo, o nó pai das páginas irmãs adjacentes "desce".

1.7 Remoção com redistribuição

Devemos considerar o caso em que **R** e **S** possuem, juntas, m-1 ou mais chaves: neste caso, basta concatenarmos ambas e, após, realizar a cisão, redistribuindo as chaves de **R** **S** e atualizando a chave separadora da página pai **T**. Lembrando que esta última etapa é igual à que fazemos na inserção.

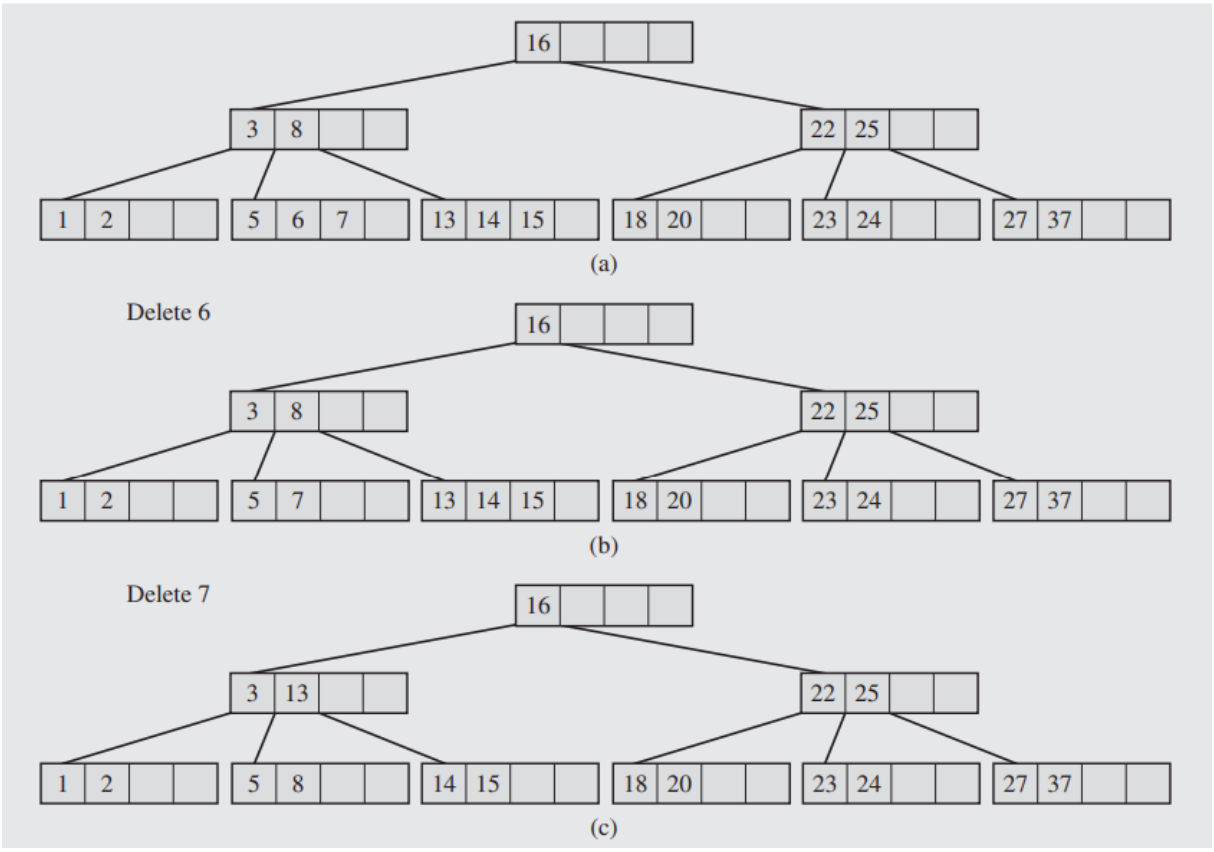

Figura 3: Elementos '6' e '7' são removidos da árvore: para o '6', lembre-se do caso 1. Para o '7', observe a redistribuição sendo aplicada (caso 4)!

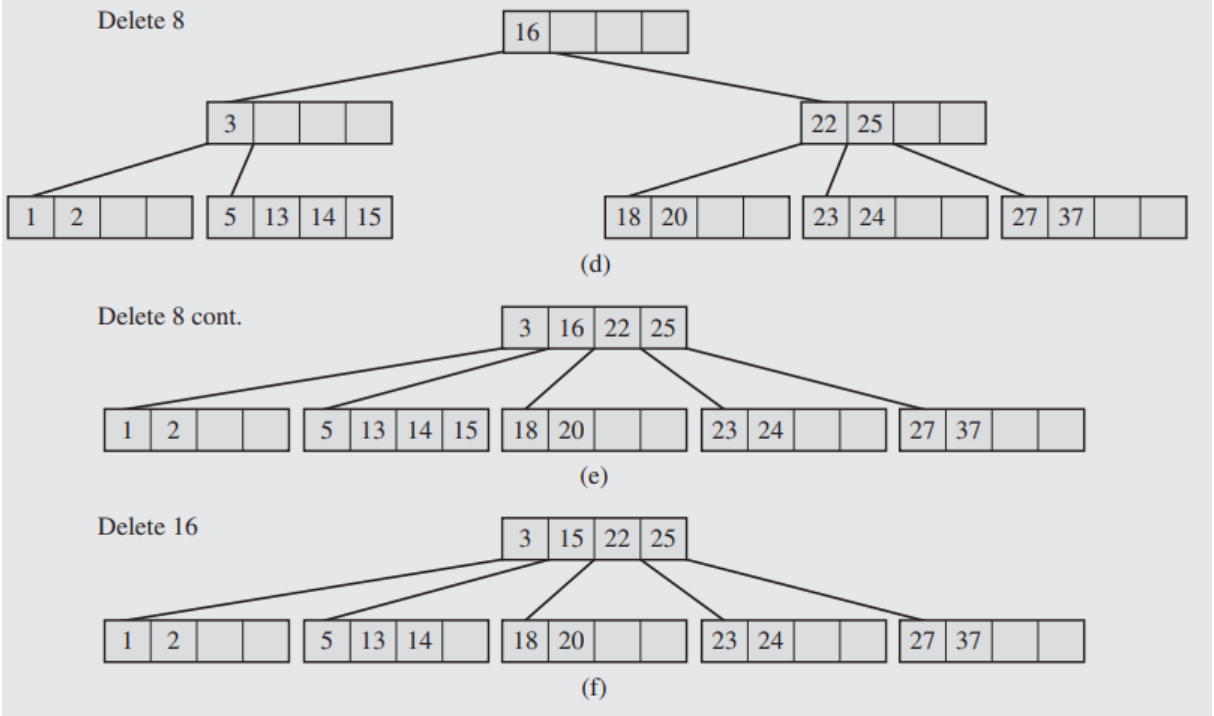

Figura 4: Elementos '8' e '16' são removidos: no primeiro cenário, observe a propagação da concatenação. No segundo cenário, observe a aplicação do caso 5!

Para exemplificar o algoritmo de remoção, (Drozdek, 2013) apresenta também:

```
1 removerArvoreB(K)
2   nodo = BTreeSearch(K, raiz);
3   se (nodo != NIL)
4     se o no nao e uma folha
5       encontre uma folha com o predecessor imediato S de K;
6       copie S no lugar de K;
7       nodo = folha contendo S;
8       remova S do nodo;
9     se nao remova K do nodo;
10    enquanto (verdadeiro)
11      se nao ha underflow em nodo
12        retorn;
13      se nao, se ha uma pagina irma com chaves suficientes
14        aplique redistribuicao;
15        retorne;
16      se nao, se o pai deste nodo for raiz
17        se o pai tem apenas uma chave
18          concatene o nodo, sua irma e seu pai para formar uma
                nova raiz;
19        se nao concatene nodo e sua irma;
20        retorne;
21      se nao concatene nodo e sua irma;
22        nodo = seu pai;
```

Listing 4: Algoritmo de remoção

2 Árvores B+

2.1 Introdução

Aqui, temos uma estrutura que é uma variação das árvores B, onde conseguimos manter a eficiência da busca e inserção das árvores B, e aumentarmos a eficiência da localização do próximo registro na árvore de $O(\log_2 N)$ para $O(1)$! Além disso, não será necessário manter nenhum ponteiro de registro em nenhum nó não-folha.

Sobre as aplicações que podemos citar, as árvores B+ são utilizadas em uns dos principais bancos de dados utilizados hoje, como o SQLServer e a própria Oracle.

2.2 Definição

Para as características dessas árvores, temos:

- Todas as chaves são mantidas na folha.
- As chaves continuam organizadas em ordem crescente.
- Algumas chaves se repetem em nós internos, de maneira que formamos um conjunto de índices.
- Estes nós internos servem apenas para guiar o processo de busca, pois eles contém apenas índices, e não ponteiros para os dados em si. Estes último estão presentes apenas nas folhas.
- As folhas são ligadas (como uma lista encadeada) para oferecer um caminho sequencial para percorrer as chaves.
- A busca nunca para até se atingir uma folha que contenha a chave buscada, mesmo que durante o percurso encontremos esta chave em algum nó interno.

2.3 Busca

A única diferença que teremos nas árvores B+ para a busca é que mesmo que a chave desejada seja encontrada em algum nó interno da árvore, a pesquisa deve apenas retornar

a chave em questão se a mesma se encontrar em algum nó-folha.

2.4 Inserção

Para a inserção, o modo de funcionamento se parece muito com o das árvores B, com a diferença sendo quando inserirmos alguma chave em uma página cheia: devemos realizar a cisão dela. A regra é:

- Quando a página estiver lotada, faremos uma cisão da página, e as $(m-1)/2$ menores chaves irão para a nova folha da esquerda.
- As $(m-1)/2$ maiores chaves irão para a página da direita.
- A maior chave da página esquerda (ou a menor da direita) é **copiada** (e não movida, como nas árvores B) para o nó pai, como sendo a nova chave separadora.

Os nós internos são iguais a uma árvore B, e portanto quando o overflow de alguma folha se propagar para estes nós, faremos a cisão igual nas árvores B (as chaves serão **movidas**, não copiadas!).

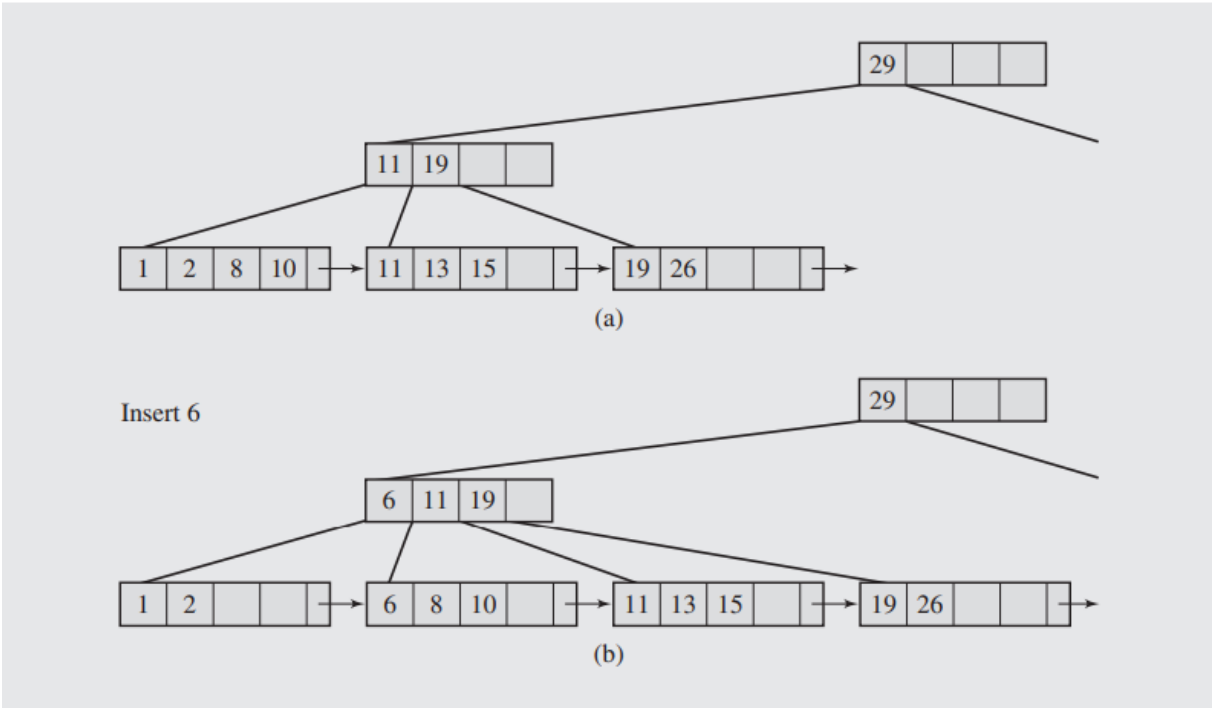

Figura 5: Observe a cisão sendo realizada após a inserção de '6': a chave mediana é copiada (e não somente movida) para a chave pai.

2.5 Remoção

Para a remoção, seguiremos a mesma lógica das árvores B, porém, a remoção dos elementos será feita sempre em uma folha! Considerando dois casos, temos:

- No primeiro, caso a chave apareça apenas em um nó folha: basta removê-lo e reorganizar o nó folha da mesma maneira feita para as árvores B.

- No segundo caso, a chave a ser excluída aparece em algum nó interno também: aqui, seguiremos os mesmos passos do caso anterior, com a única diferença que esta chave **não** será excluída dos nós internos. Isto é interessante pois a página interna que contém esta chave serve como índice que ainda pode guiar o processo de busca corretamente, além de economizar complexidade de tempo: remover dos nós internos pode acarretar em underflow, exigindo mais operações de concatenação/redistribuição.

Caso, após removermos uma chave (de alguma folha), o número de chaves restantes for menor que **(m-1)/2**, devemos aplicar o tratamento adequado: ou **redistribuir** as chaves da folha e sua irmã, atualizando a chave separadora (por cópia, não por promoção!), ou apagar a folha e juntar todas suas chaves na sua irmã, removendo após isso, a chave separadora.

Perceba como esses dois processos se parecem muito com a redistribuição e a concatenação apresentados nas árvores B, com a diferença que na redistribuição consideramos apenas as chaves das páginas irmãs (não levamos em conta a chave pai: é apenas um índice!), e após isto, a chave central (ou uma das chaves centrais) será copiada (e não movida) para a chave pai. Para a concatenação, também juntamos apenas o conteúdo das irmãs, não levando em conta a chave pai, que é simplesmente removida após o processo!

As duas nova maneiras de redistribuir e concatenar, porém, só são válidas para as folhas, que contém as chaves de fato. Para os nó internos, que contém apenas os índices, incluindo a raiz, caso o underflow se propague, devemos aplicar a concatenação a redistribuição antiga, pois esta parte da árvore B+ é simplesmente uma árvore B!

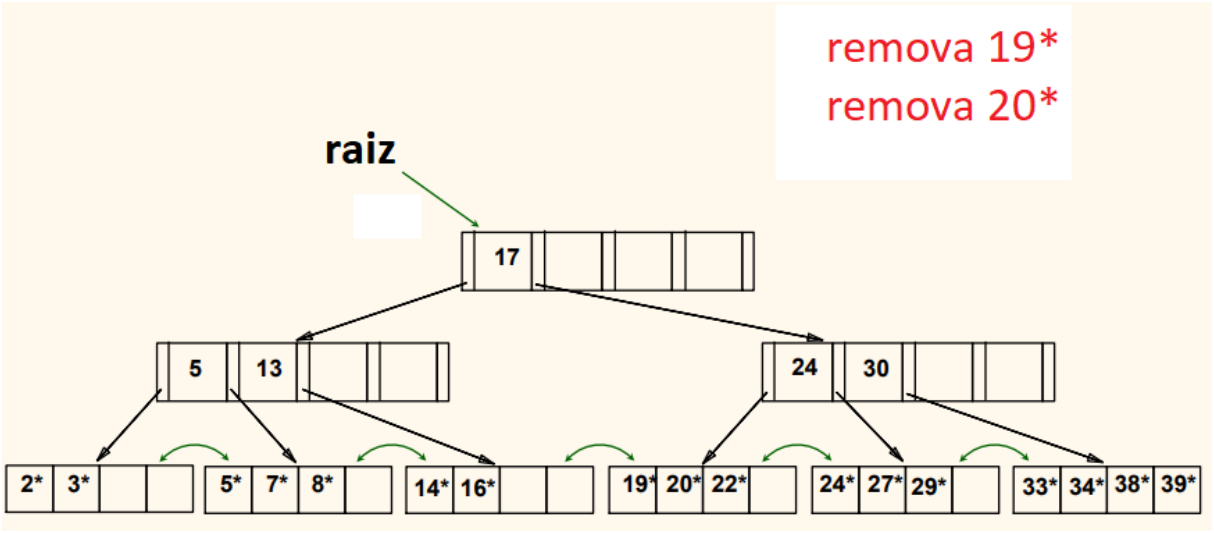

Figura 6: Elementos '19*' e '20*' serão removidos: para o primeiro caso, apenas removeremos a chave. Após remover '20*', o underflow ocorre, e sua página-irmã contém mais do que $\lceil m \rceil - 1$ chaves: aplicaremos a nova redistribuição.

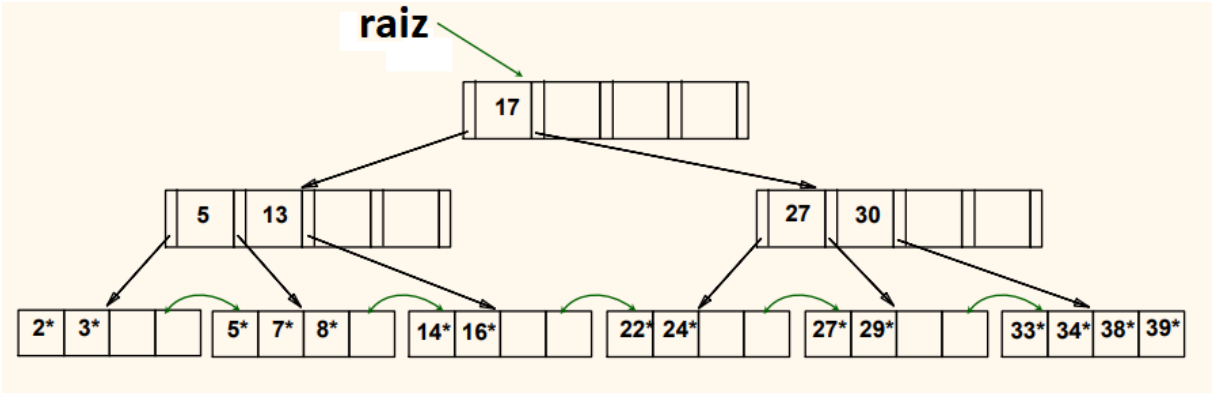

Figura 7: Observe a nova redistribuição sendo aplicada.

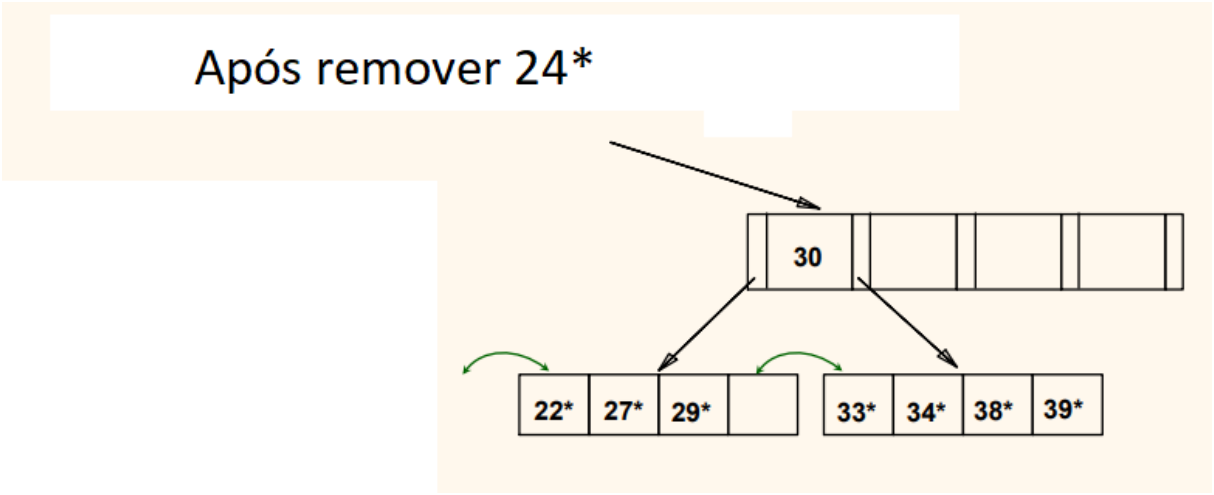

Figura 8: Após a remoção do 24*, a redistribuição é impossível dado o número de chaves da página irmã: a nova concatenação será aplicada, juntando o conteúdo da folha em sua irmã e simplesmente apagando a chave pai!

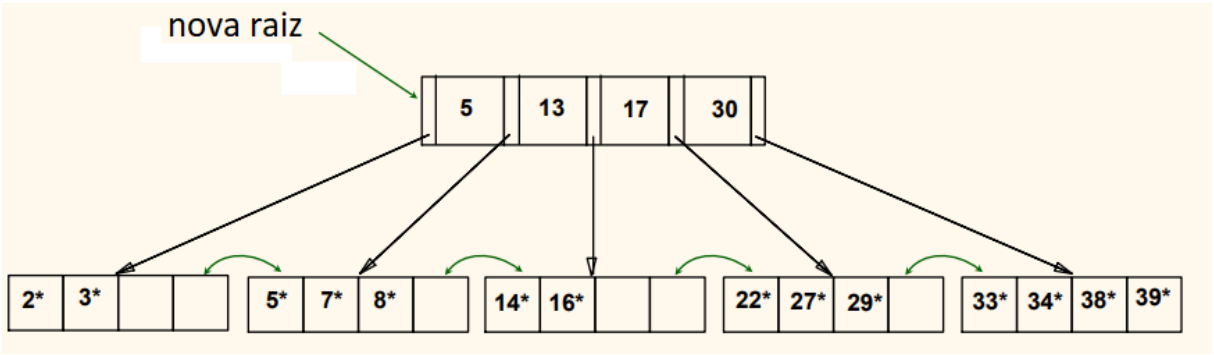

Figura 9: Agora o underflow se propagou para o nó pai (interno): devemos aplicar as operações da árvore B. Como a redistribuição é impossível dado o número insuficiente de chaves da página irmã, a concatenação antiga foi aplicada (observe que a chave pai de 30 agora não foi simplesmente apagada, mas ela 'desceu', juntando-se com o conteúdo de suas filhas em um único nó (além disso, temos o caso 5 das árvores B - diminuição da altura da árvore).

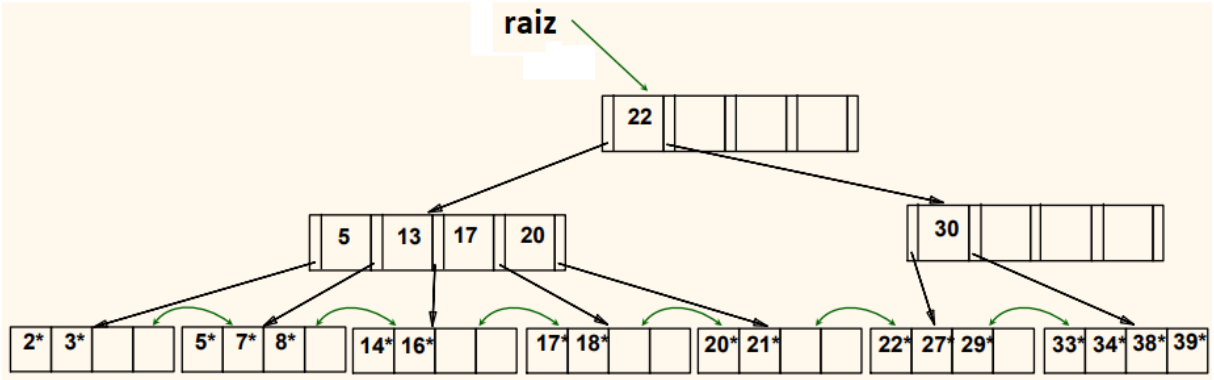

Figura 10: Aqui está outra situação ilustrando um caso onde aplicamos a nova redistribuição nos nós internos.

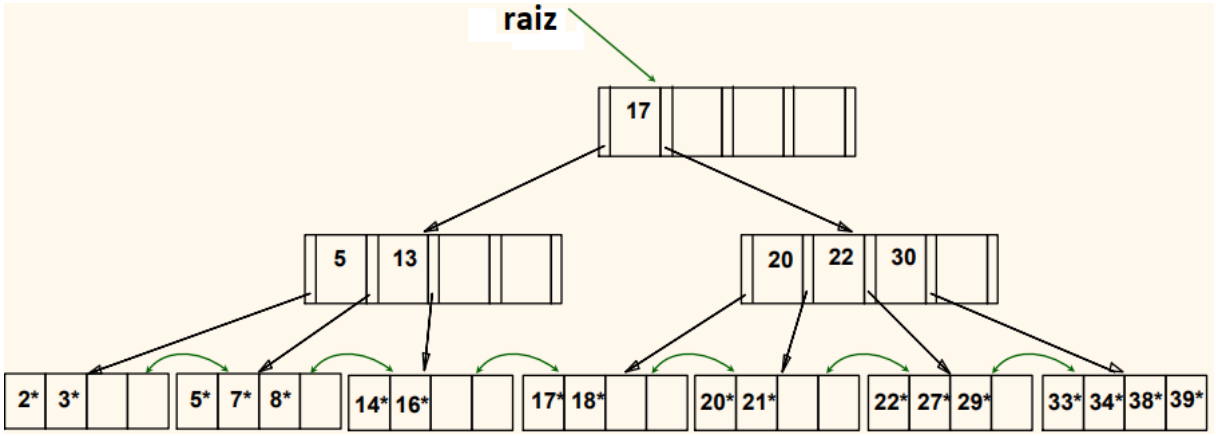

Figura 11: Como são nós internos, a redistribuição antiga foi aplicada.

3 Árvores B*

Na verdade, o termo árvore B* é um pouco confundido na literatura, sendo que Knuth em 1973 (Knuth, 1973) propõe o modelo de árvore B* como um tipo de árvore B em que cada nó deve ocupar pelo menos 2/3 de sua capacidade, e não apenas 1/2. Além disso, a diferença é que na inserção, este tipo de árvore adia a cisão da página até que 2 nós adjacentes estejam cheios, aplicando **redistribuição** enquando apenas um nó estiver com overflow (sim, durante a inserção, não apenas na remoção)!

Assim, quando ambas estiverem lotadas, a árvore B* redistribuirá as chaves igualmente entre 3 páginas novas, sendo que cada página ocupará, então, 2/3 de sua capacidade. Isto garante que a utilização de memória seja de pelo menos 66 % (Comer, 1979). (LEUNG, 1984) mostra que a utilização média das árvores B* é de 81%. Devemos notar que aumentar a ocupação mínima da árvore garante melhora no tempo de pesquisa, já que a altura tende a ser menor, exigindo menos acessos por busca.

São árvores utilizadas no método de acesso a arquivos da IBM (estratégias de SGBD's).

3.1 Definição

De maneira mais formal, uma árvore B* de ordem **m** é definida assim:

- Cada página pode ter no máximo m filhos.
- Toda página, com exceção da raiz e das folhas, deve ter pelo menos **(2m-1)/3** filhos.
- A raiz deve ter pelo menos 2 filhos, a não ser que seja uma folha.
- Todas as folhas devem estar no mesmo nível (mesma propriedade das árvores B).
- Uma página não-folha com k filhos contém k-1 chaves.
- Uma folha contém no mínimo $\lfloor$**(2m-1)/3**$\rfloor$ chaves e no máximo **m-1** chaves.

Assim, é perceptível que a maioria das características da própria árvore B é mantida, sendo que as principais diferenças estão da segunda e na última regra. Por consequência, as operações de remoção e redistribuição também serão alteradas. Além disso,

como a raiz não possui nenhuma página irmã, deve-se tomar cuidado em sua implementação, sendo que uma possível estratégia é, para a raiz, utilizar a cisão tradicional em 2 páginas. Outra estratégia é permitir que a raiz tenha um tamanho maior (ver http://wiki.icmc.usp.br/images/8/8e/SCC578920131-B.pdf).

3.2 Inserção

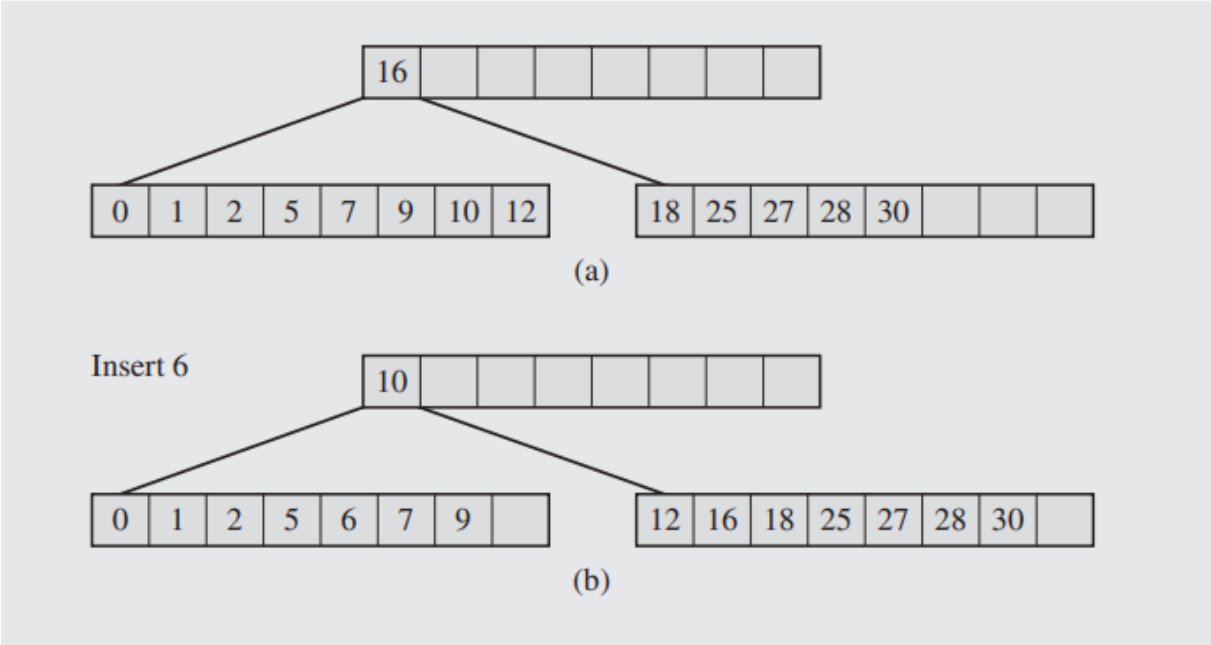

Figura 12: Elemento '6' é inserido na árvore.

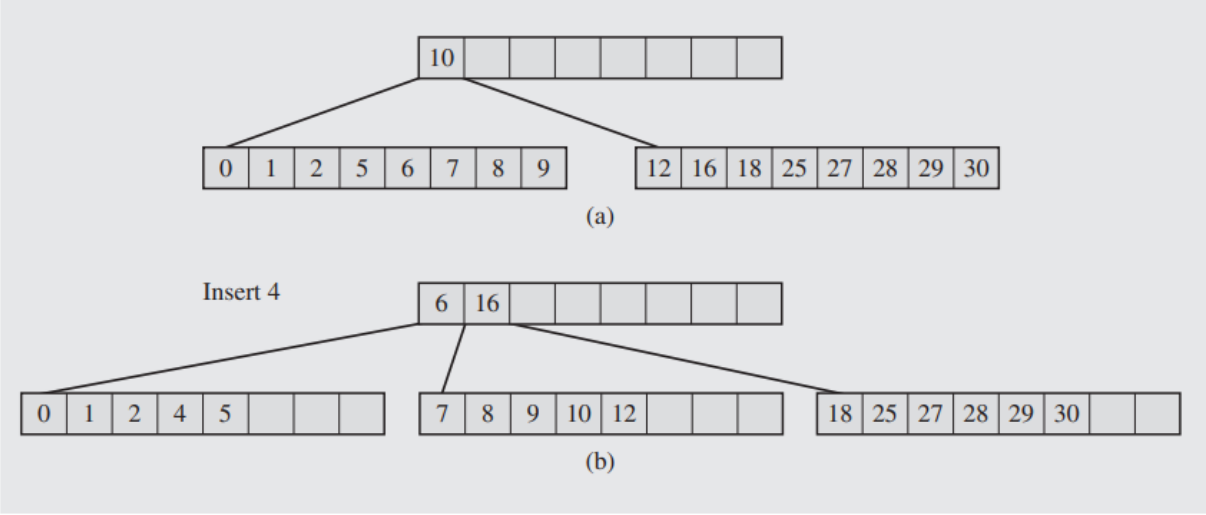

Figura 13: Elemento '4' é inserido na árvore: como a página-alvo da inserção e sua irmã estão cheias, agora sim a cisão em 3 partes será efetuada (observe que as 3 páginas todas ocupam agora pelo menos 2/3 da capacidade máxima).

3.3 Remoção

Para a remoção, segue-se o mesmo critério para as árvores B, o que muda é a condição de chaves mínimas que, caso seja desrespeitada, acarretará em underflow.

Referências Bibliográficas

Bayer, R. e McCreight, E. (2002). Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer.

Comer, D. (1979). The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., e Stein, C. (2009). *Introduction to algorithms*. MIT press.

Drozdek, A. (2013). *Data Structures and Algorithms in C++*. Cengage Learning, 4 edition.

Knuth, D. E. (1973). The art of computer programming, volume 3: Searching and sorting. *Addison-Westley Publishing Company: Reading, MA*.