Gerardo Valdisio Rodrigues Viana
Glauber Ferreira Cintra

Pesquisa e Ordenação de Dados

2011

Capítulo

4

Técnicas de pesquisa

Objetivos:

- Definir as técnicas de pesquisa sequencial e binária
- Apresentar as tabelas de dispersão
- Apresentar as árvores de busca balanceadas
- Mostrar as técnicas de manutenção das árvores de pesquisa
- Analisar a complexidade dos algoritmos de pesquisa

Apresentação

Neste capítulo apresentamos diversas técnicas e estruturas de dados que permitem fazer pesquisa com bastante eficiência. Inicialmente, apresentamos o procedimento de busca sequencial que, apesar de ser ineficiente, serve para efeito de comparação com os demais métodos. A seguir abordamos a busca binária, que permite encontrar um valor contido num vetor ordenado com extrema eficiência. No entanto, as operações de inserção e remoção têm custo computacional elevado, por isso é comum utilizar outras estruturas de pesquisa em que os dados estejam implicitamente ordenados, como as tabelas de dispersão onde, sob certas hipóteses, essas operações podem ser feitas em tempo esperado constante. No final do capítulo abordamos três tipos de árvores de busca balanceadas: árvores AVL, árvores B e árvores B+. Descrevemos as operações de inserção, busca e remoção nessas estruturas e mostramos que tais operações são feitas em tempo logarítmico, no pior caso.

1. Técnicas de Pesquisa

Uma das tarefas de maior importância na computação é a pesquisa de informações contidas em coleções de dados. Em geral, desejamos que essa tarefa seja executada sem que haja a necessidade de inspecionar toda a coleção de dados.

Neste capítulo apresentamos diversas técnicas e estruturas de dados que permitem fazer pesquisa com bastante eficiência. Discutimos as operações de inserção, busca e remoção nessas estruturas, analisando sua complexidade temporal e espacial.

Inicialmente, nas seções 2 e 3, descrevemos respectivamente os procedimentos de *busca sequencial* que é trivial e ineficiente, e de *busca binária*, que permite encontrar um valor contido num vetor ordenado com extrema eficiência. Sabe-se, no entanto, que as operações de inserção e remoção num vetor ordenado têm custo computacional elevado.

Em seguida, na seção, apresentamos as *tabelas de dispersão*, discutindo as técnicas de *hashing fechado* e *hashing aberto*. Tais técnicas podem ser usadas para resolver o problema da colisão de chaves que é inerente ao uso dessas estruturas. Mostramos que, sob certas hipóteses, as operações de inserção, busca e remoção em tabelas de dispersão são feitas em tempo esperado constante.

Finalmente, na seção 5 abordamos três tipos de árvores de busca balanceadas: árvores AVL, árvores B e árvores B+. Nessas estruturas as operações de inserção, busca e remoção são feitas em tempo logarítmico.

2. Busca Sequencial

Podemos procurar, ou buscar, um valor x num vetor L inspecionando em sequência as posições de L a partir da primeira posição. Se encontrarmos x, a busca tem sucesso. Se alcançarmos a última posição de L sem encontrar x, concluímos que x não ocorre no vetor L. Esse tipo de busca é chamado de busca *sequencial*.

Considerando que o vetor L contém n elementos, ordenados ou não, é fácil verificar que a busca sequencial requer tempo linearmente proporcional ao tamanho do vetor, ou seja, *O*(n); por isso, é comum dizer que a busca sequencial é uma busca *linear*. Observa-se que, no melhor caso, o elemento x é encontrado logo na primeira tentativa da busca, através de uma comparação; no pior caso, x encontra-se na última posição e são feitas n comparações, logo, no caso médio, o elemento é encontrado após (n+1)/2 comparações. Para vetores de médio ou grande porte, esse tempo é considerado inaceitável, dado que existem técnicas mais eficientes descritas nas seções seguintes.

Antes apresentamos dois algoritmos para a busca sequencial.

```
PROCEDIMENTO BUSCA_SEQUENCIAL ( L , X , POS )

   ENTRADA: UM VETOR L e UM VALOR X
   SAÍDA: POS = i, SE X OCORRE NA POSIÇÃO i DE L // SUCESSO
          POS = 0, CASO CONTRÁRIO.

   POS = 0

   PARA i = 1 até N
       SE L[i] = X
          POS = i
          ESCAPE
   // fim para
   DEVOLVA POS
FIM {BUSCA_SEQUENCIAL}
```

Algoritmo 4.1: Busca Sequencial Simples

```
PROCEDIMENTO BUSCA_SEQUENCIAL_REC ( L , N, X )

   ENTRADA: UM VETOR L DE TAMANHO N e UM VALOR X
   SAÍDA: i, SE X OCORRE NA POSIÇÃO i DE L    // SUCESSO
          0, CASO CONTRÁRIO.

   SE N = 1
         SE L[1] = X
              DEVOLVA 1
         SENÃO
              DEVOLVA 0
           SENÃO
         SE L[N] = X
            DEVOLVA N
   SENÃO
         BUSCA_SEQUENCIAL_REC ( L , N-1, X )

FIM {BUSCA_SEQUENCIAL_REC}
```

Algoritmo 4.2: Busca Sequencial Recursiva

3. Busca Binária

Quando o vetor L estiver em ordem crescente, podemos determinar se x ocorre em L de forma mais rápida da seguinte maneira: inspecionamos a posição central de L; se ela contém x a busca para, tendo sido bem sucedida, caso contrário, se x for menor do que o elemento central passamos a procurar x, recursivamente, no intervalo de L que está à esquerda da posição central. Se x for maior do que o elemento central continuamos a procurar x, recursivamente, no intervalo de L que está à direita da posição central. Se o intervalo se tornar vazio, a busca para, tendo sido mal sucedida.

O procedimento que acabamos de descrever é chamado de *busca binária*. Facilmente, podemos adaptar a busca binária para procurar valores em vetores que estejam em ordem decrescente. A seguir fornecemos uma descrição recursiva em pseudocódigo desse procedimento.

```
PROCEDIMENTO BUSCA_BINARIA
  ENTRADA: UM VETOR L EM ORDEM CRESCENTE, UM VALOR X, E AS
           POSIÇÕES INICIO E FIM.
  SAÍDA: SIM, SE X OCORRE ENTRE AS POSIÇÕES INICIO E FIM DE
         L; NÃO, CASO CONTRÁRIO.

  SE INICIO > FIM
    DEVOLVA NÃO E PARE
  MEIO = (INICIO+FIM)/2              // DIVISÃO INTEIRA
  SE X = L[MEIO]
    DEVOLVA SIM E PARE
  SE X < L[MEIO]
    DEVOLVA BUSCA_BINARIA(L, X, INICIO, MEIO - 1)
  SENAO
    DEVOLVA BUSCA_BINARIA(L, X, MEIO + 1, FIM)
FIM {BUSCA_BINARIA}
```

Algoritmo 4.3: Busca Binária

A Figura 4.1 ilustra o funcionamento do Algoritmo Busca Binária ao procurar o valor 34 num vetor ordenado. Observe que apenas as posições 8, 13 e 10 do vetor (nessa ordem) são inspecionadas. A tabela que aparece após o vetor mostra os valores dos parâmetros de entrada (exceto o vetor L, que é passado por referência e é compartilhado por todas as chamadas), o conteúdo da variável MEIO e o valor devolvido por cada chamada.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L	2	5	5	8	10	13	16	17	29	31	34	36	43	49	51	56	65	69
									1ª		3ª			2ª				

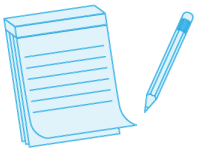
Chamada	x	INICIO	FIM	MEIO	Valor devolvido
1	34	0	17	8	Sim
2	34	9	17	13	Sim
3	34	9	12	10	Sim

Figura 4.1: Exemplo de busca binária

Se estivéssemos procurando o valor 7 no vetor da Figura 4.1, iríamos inspecionar apenas as posições 8, 3, 1 e 2 do vetor (nessa ordem). A tabela a seguir mostra os valores dos parâmetros de entrada, o conteúdo da variá-

vel MEIO e o valor devolvido por cada chamada. Naturalmente, como 7 não ocorre no vetor, o valor devolvido é *Não*.

Chamada	x	INICIO	FIM	MEIO	Valor devolvido
1	7	0	17	8	Não
2	7	0	7	3	Não
3	7	0	3	1	Não
4	7	2	2	2	Não
5	7	3	2	-	Não



ANOTE

Na verdade, como o Algoritmo Busca Binária utiliza recursão de cauda, o compilador/interpretador pode gerar uma versão desse algoritmo em linguagem de máquina que tenha complexidade espacial constante.

Vamos denotar por T(n) o tempo requerido pelo Algoritmo Busca Binária para procurar um valor num intervalo com n posições. Note que se x não estiver na posição central do intervalo será feita uma chamada recursiva para procurar x num intervalo que terá aproximadamente n/2 posições. Dessa forma, é correto afirmar que:

$T(n) \leq T(n/2) + c$

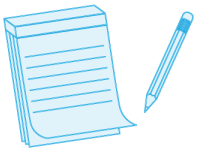
$T(0) = c$

Resolvendo essa fórmula de recorrência concluímos que $T(n) \in O(\log n)$. Sendo assim, a complexidade temporal do Algoritmo Busca Binária pertence a O(logn).

É fácil perceber que a complexidade espacial desse algoritmo também é dada pela mesma fórmula de recorrência. Concluímos que tal complexidade também pertence a O(logn). Vale salientar que se implementarmos o procedimento de busca binária ser usar recursividade, a complexidade espacial passa a ser constante.

É possível adaptar o procedimento de busca binária para encontrar um valor mais próximo de x num vetor ordenado. Por exemplo, no vetor da Figura 4.1, o valor mais próximo do 40 é 43. Esse tipo de busca é conhecida como busca binária aproximada. Tal adaptação é deixada como exercício para o leitor no final deste capítulo.

Apesar da busca binária ser extremamente eficiente, as operações de inserção e remoção são feitas em tempo linearmente proporcional ao tamanho do vetor e isso, em geral, é considerado inaceitável. Por esse motivo, nas situações em que as operações de inserção e remoção são frequentes, é preferível utilizar uma estrutura de dados onde os dados estejam implicitamente ordenados e que também permitam fazer inserções e remoções com bastante eficiência. Na Seção 4 descrevemos alguns tipos de árvores que possuem tais características.



ANOTE

O operador **mod** denota a operação de resto da divisão, inteira.

4. Tabelas de Dispersão

As *tabelas de dispersão*, também conhecidas como *tabelas de espalhamento* ou *tabelas de hashing*, armazenam uma coleção de valores, sendo que cada valor está associado a uma chave. Tais chaves têm que ser todas distintas e são usadas para mapear os valores na tabela. Esse mapeamento é feito por uma *função de hashing*, que chamaremos de h.

Por exemplo, podemos implementar uma tabela de dispersão usando um vetor com oito posições e utilizar h(x) = x **mod** 8 como função de **hashing**. Dizemos que h(k) é a *posição original* da chave k e é nessa posição da tabela

que a chave k deve ser inserida. A figura a seguir ilustra a inserção de uma coleção de valores com suas respectivas chaves numa tabela de dispersão.

Função de hashing:
h(x) = x mod 8

	Chave	Valor
0		
1	25	Lia
2	18	Ana
3		
4		
5	5	Rui
6		
7	31	Gil

Figura 4.2: Tabela de dispersão

Observe que o valor Gil foi colocado na posição 7 da tabela, pois a chave associada a Gil é 31 e h(31) = 7. O que aconteceria se tentássemos inserir nessa tabela o valor Ivo com chave 33 ? Observe que h(33) = 1, mas a posição 1 da tabela já está ocupada. Dizemos que as chaves 25 e 33 *colidiram*. Mais precisamente, duas chaves x e y *colidem* se h(x) = h(y).

Note que o problema da colisão de chaves ocorre porque, na maioria dos casos, o domínio das chaves é maior que a quantidade de posições da tabela. Sendo assim, pelo *princípio da casa de pombos*, qualquer função do conjunto das chaves para o conjunto das posições da tabela não é injetora.

Não é aceitável recusar a inserção de uma chave que colida com outra já existente na tabela se ela ainda tiver posições livres. Precisamos, portanto, de alguma estratégia para lidar com as colisões de chaves. Existem diversas técnicas para lidar com as colisões. Nas próximas duas subseções apresentaremos duas dessas técnicas.

4.1 Hashing Fechado

Na técnica conhecida como *hashing fechado*, quando tentamos inserir uma chave y e ela colide com uma chave x já existente na tabela, colocamos y na primeira posição livre após a posição h(y). Uma tabela de dispersão que utilize essa estratégia para resolver as colisões de chaves costuma ser chamada de *tabela de hashing fechado*.

Na tabela da Figura 4.2, a chave 33 deve ser inserida na posição 3 visto que ela é a primeira posição livre após a posição h(33). Consideramos que após a última posição da tabela vem a posição 0. Sendo assim, na tabela da Figura 4.2, a chave 23 deve ser inserida na posição 0. A figura a seguir mostra a tabela da Figura 4.2 após a inserção das chaves 33 e 23. Dizemos que as chaves 23 e 33 estão *deslocadas* de suas posições originais e denotamos tal fato na figura usando o asterisco.

Função de hashing:

h(x) = x mod 8

	Chave	Valor
0	23*	Edu
1	25	Lia
2	18	Ana
3	33*	Ivo
4		
5	5	Rui
6		
7	31	Gil

Figura 4.3: Inserções em hashing fechado

Obviamente, se a tabela estiver cheia e tentarmos inserir mais uma chave, tal inserção não será possível. Nesse caso, dizemos que ocorreu um evento conhecido como estouro da tabela de dispersão (*hash overflow*).

Ao usarmos essa estratégia para resolver as colisões, a seguinte propriedade é mantida pelas operações de inserção: se uma chave k foi inserida numa posição j, então as posições da tabela desde h(k) até j têm que estar ocupadas. Como decorrência dessa propriedade, para buscar uma chave k numa tabela de hashing fechado devemos inspecionar em sequência as posições da tabela a partir da posição h(k). A busca termina quando ocorrer um dos seguintes eventos:

- A posição inspecionada contém k. Nesse caso, a busca foi bem sucedida.
- A posição inspecionada está livre. Nesse caso, podemos concluir que k não ocorre na tabela.
- Todas as posições foram inspecionadas. Se esse evento ocorrer antes dos dois anteriores significa a tabela não tem posições livres e que a chave k não ocorre na tabela.

Por exemplo, se procurarmos a chave 33 na tabela da Figura 4.3, teremos que inspecionar as posições 1, 2 e 3 e a busca será bem sucedida. Por outro lado, se procurarmos a chave 15, inspecionaremos as posições 7, 0, 1, 2 e 3 e a busca será mal sucedida.

A remoção numa tabela de hashing fechado é um pouco mais complicada que a inserção e a busca. Precisamos garantir que a propriedade decorrente das inserções, que enunciamos anteriormente, também seja preservada pelas remoções.

Para remover uma chave k, primeiramente precisamos encontrá-la. Obviamente, se k não ocorre na tabela a remoção é mal sucedida. Suponha que a chave k foi encontrada numa posição j. Tal posição deve ser liberada. Em seguida, inspecionamos as posições subsequentes em busca de uma chave que esteja deslocada. Se encontrarmos uma chave deslocada cuja posição original seja a posição que foi liberada ou uma posição anterior a ela, devemos movê-la para a posição que está livre. Note que a posição onde estava a chave deslocada agora está livre. O processo é então repetido até que uma posição livre seja alcançada.

Por exemplo, para remover a chave 18 da tabela da Figura 4.3 devemos liberar a posição 2 e em seguida mover a chave 33 para a posição 2. A remoção

para quando a posição 4 é atingida. Se após essa remoção quisermos remover a chave 31, teremos que mover a chave 23 para a posição 7. Com isso, a posição 0 fica livre. Observe que chave 33, apesar de estar deslocada, não deve ser movida para a posição 0, pois sua posição original é a posição 1. A Figura 4.4 mostra como ficaria a tabela da Figura 4.3 após essas duas remoções.

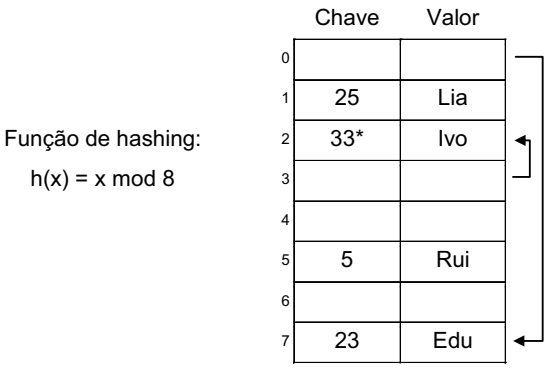
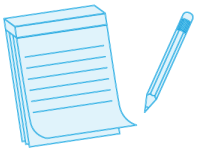


Figura 4.4: Remoções em hashing fechado



ANOTE

Naturalmente existe um limite para a quantidade de chaves que podem ser armazenadas numa tabela de hashing aberto. No entanto, esse limite é determinado pela memória física do computador e não pelo tamanho da tabela.

4.2 Hashing Aberto

A técnica de *hashing aberto* é bem mais simples que hashing fechado. Numa tabela de hashing aberto cada posição da tabela contém um ponteiro para uma lista encadeada que contém todas as chaves mapeadas pela função de hashing naquela posição. Note que usando essa técnica o espaço de armazenamento das chaves não fica restrito à tabela, daí o nome hashing aberto, e não há limitação quanto à quantidade de chaves que podem ser armazenadas na **tabela**. Dessa forma, ao utilizar essa estratégia para resolver as colisões, não precisamos nos preocupar com hash overflow.

Para inserir uma chave k numa tabela de hashing aberto, calculamos h(k) e então inserimos tal chave (e o valor associado a ela) no final da lista encadeada apontada pela posição h(k) da tabela, caso essa chave não ocorra nessa lista (lembre-se de que numa tabela de dispersão as chaves têm que ser todas distintas).

Ilustramos na figura a seguir a inserção das chaves 25, 18, 31, 5, 23 e 33, nessa ordem, numa tabela de hashing aberto com 5 posições e que utiliza a função de hashing h(x) = x mod 5.

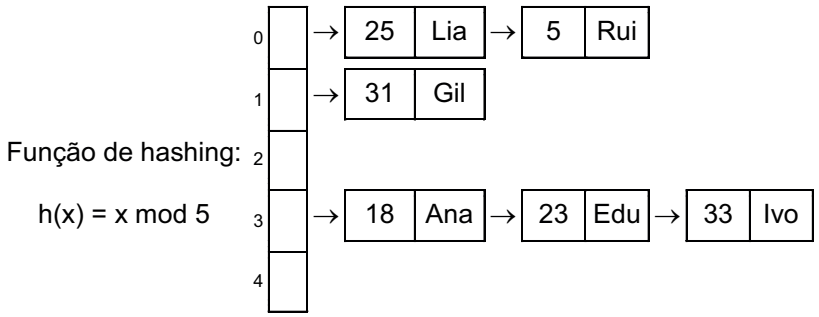


Figura 4.5: Inserções em Hashing aberto

Para encontrar uma chave k numa tabela de hashing aberto basta procurar tal chave na lista encadeada apontada pela posição h(k) da tabela.

A remoção em hashing aberto também é bastante simples. Após encontrar a chave que se deseja remover, basta remover o nó que a contém. A Figura 4.6 mostra a tabela da Figura 4.5 após a remoção das chaves 18 e 31.

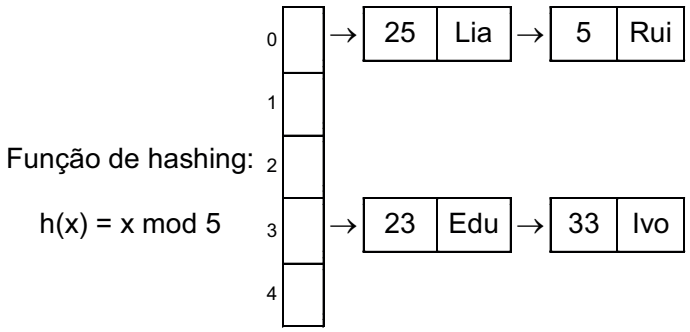


Figura 4.6: Remoções em Hashing aberto

Vamos agora discutir a eficiência das operações de inserção, busca e remoção em tabelas de dispersão. Antes, precisamos definir alguns termos.

Dizemos que uma função de hashing é *boa* se ela pode ser computada em *tempo constante* e se ela espalha as chaves na tabela de maneira uniforme. Por exemplo, dada uma tabela de dispersão com 37 posições, a função h(x) = 2x mod 37 não é boa, pois ela espalha as chaves somente nas posições ímpares da tabela. A função h(x) = x! mod 37 também não é boa, pois não pode ser computada em tempo constante. Já a função h(x) = x mod 37 é boa, se o domínio das chaves é o conjunto dos números naturais. Caso o domínio das chaves seja o conjunto dos naturais pares, h(x) = x mod 37 deixaria de ser uma boa função de hashing, pois mapearia as chaves somente nas posições ímpares da tabela. Nesse caso, a função h(x) = x/2 mod 37 seria considerada boa.

A *carga* de uma tabela de dispersão é a razão entre a quantidade de chaves e a quantidade de posições da tabela. Por exemplo, a carga da tabela da Figura 4.6 é 0,8. A carga de uma tabela de hashing fechado é considerada *baixa* se ela é menor ou igual a 0,5. No caso de uma tabela de hashing aberto, a carga é considerada baixa se ela é limitada por uma constante.

É possível mostrar que se a função de hashing é boa e a carga é baixa, as operações de inserção, busca e remoção são feitas em *tempo esperado constante*.

Observe que no caso de hashing fechado, as operações de inserção, busca e remoção sempre terminam quando uma posição livre é atingida. Se a carga é baixa, pelo menos a metade das posições da tabela estarão livres. Sendo assim, se a função de hashing espalha as chaves de maneira uniforme, a cada duas posições consecutivas é esperado que pelo menos uma esteja livre e consequentemente a quantidade esperada de posições que devem ser inspecionadas é constante.

No caso de um hashing aberto, se a carga é baixa e a função de hashing espalha as chaves de maneira uniforme, o tamanho esperado de cada lista encadeada será constante. Note que as operações de inserção, busca e remoção gastam tempo limitado pelo tamanho da maior lista encadeada. Isso implica que tais operações irão requerer tempo esperado constante.

Nas situações práticas nas quais a quantidade máxima de chaves que serão armazenadas na tabela pode ser estimada *a priori*, é possível determinar o tamanho que a tabela deve ter de modo a garantir que sua carga

seja sempre baixa. Em tais situações, o uso de tabelas de dispersão pode ser bastante adequado.

5. Árvores de Busca Balanceadas

Nessa seção abordaremos as árvores de busca balanceadas. Nessas árvores, as chaves armazenadas são mantidas implicitamente ordenadas. Isso permite que a operação de busca seja feita percorrendo-se um ramo da árvore, desde a raiz até, no máximo, chegar a uma folha. Além disso, tais árvores possuem propriedades que garantem que sua altura seja muito pequena se comparada à quantidade de chaves contidas na árvore. Como veremos mais adiante, isso garante que as operações de inserção, busca e remoção sejam feitas com muita eficiência.

Discutiremos três tipos de árvores de busca balanceadas: árvores AVL, árvores B e árvores B+.

5.1 Árvores AVL

As árvores AVL são árvores binárias propostas por Adelson-Velski e Landis em 1962 que se caracterizam por duas propriedades:

- Se um nó da árvore contém uma chave x, as chaves contidas na subárvore à esquerda desse nó são todas menores do que x e as chaves contidas na subárvore à direita desse nó são todas maiores do que x.
- Para cada nó da árvore, a diferença de altura entre a subárvore à esquerda desse nó e a subárvore à direita desse nó é de no máximo 1.

A primeira propriedade garante que as chaves contidas na árvore estejam implicitamente ordenadas. A segunda propriedade garante o balanceamento da árvore.

A estrutura de um nó de uma árvore AVL pode ser constituída dos seguintes campos:

- chave: armazena uma chave.
- filhoesq: ponteiro para o filho esquerdo.
- filhodir: ponteiro para o filho direito.
- bal: a altura da subárvore à esquerda menos a altura da subárvore à direita do nó.

O campo bal indica como está o balanceamento das subárvores apontadas pelo nó e auxilia a manter o balanceamento da árvore nas operações de inserção e remoção. Observe que os únicos valores aceitáveis para o campo bal são –1, 0 e 1. A Figura 4.7 exibe um exemplo de árvore AVL. O valor do campo bal aparece acima de cada nó.

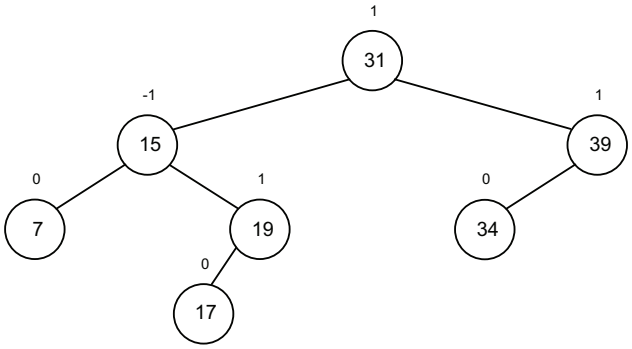
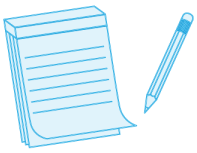


Figura 4.7: Exemplo de árvore AVL

A operação de busca numa árvore AVL é similar à busca numa árvore binária de busca qualquer. Para procurar uma chave k, primeiramente verificamos se a árvore é vazia. Em caso afirmativo, a busca para, tendo sido mal sucedida. Caso contrário, verificamos se a chave contida na raiz é k. Nesse caso, a busca tem sucesso. Caso contrário, se k for menor do que a chave contida na raiz, repetimos o processo recursivamente na subárvore à esquerda da raiz. Se k for maior do que a chave contida na raiz, repetimos o processo recursivamente na subárvore à direita da raiz. Na árvore da Figura 4.7, para chegar na chave 17 teremos que passar antes pelos nós que contém as chaves 31, 15 e 19. Se buscarmos a chave 42, passaremos pelos nós que contém as chaves 31 e 39 até atingir a subárvore à direita do 39, que é vazia. A busca será então mal sucedida.



ANOTE

Se permitirmos a ocorrência de chaves repetidas numa árvore AVL podemos chegar numa situação onde é impossível preservar a propriedade do balanceamento. Por exemplo, imagine como seria uma árvore AVL com as chaves 15, 15 e 15.

Já a operação de inserção é mais complicada. Para inserir uma chave k numa árvore AVL devemos percorrê-la, a partir da raiz, como se estivéssemos procurando k. Se a chave k for encontrada, a inserção deve ser abortada pois árvores AVL não podem ter **chaves repetidas**. Caso contrário, atingiremos um ponteiro nulo. Devemos então criar uma nova folha contendo a chave k e fazer o ponteiro que era nulo apontar para tal folha. Naturalmente, os campos filhoesq e filhodir dessa nova folha devem ser nulos e o campo bal deve receber o valor 0.

Precisamos ainda atualizar o bal dos ancestrais dessa nova folha. Isso deve ser feito da seguinte maneira. Se a folha foi inserida à esquerda de seu pai, o bal do pai deve ser incrementado. Se ela foi inserida à direita de seu pai, o bal do pai deve ser decrementado.

Durante uma inserção, se o bal de um nó tornar-se -1 ou 1 será preciso propagar a atualização do bal para o seu nó pai. Se esse nó estiver à esquerda de seu pai, o bal do pai deve ser incrementado, caso contrário, o bal do pai deve ser decrementado. Se o bal de um nó tornar-se 0, a inserção é finalizada.

Por exemplo, ao inserir a chave 3 na árvore da Figura 4.7 teremos que incrementar o bal do nó que contém a chave 7. Como o bal desse nó passará a ser 1, teremos que atualizar o bal do seu nó pai, que contém a chave 15. O bal do nó pai passará a ser 0 e a inserção será finalizada.

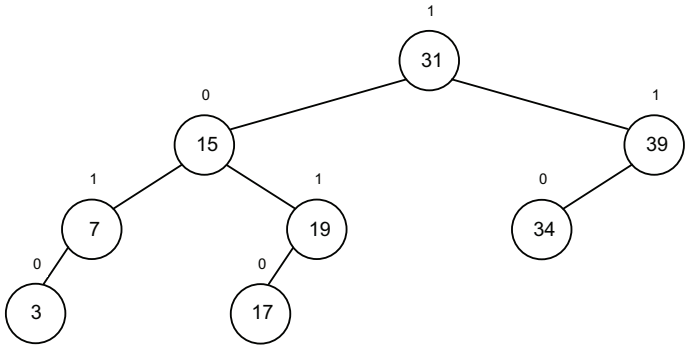


Figura 4.8: Árvore da Figura 4.7 após a inserção da chave 3

Existe ainda mais uma possibilidade a tratar. Se o bal de um nó tornar-se –2 ou 2, o que é inaceitável, será necessário realizar um procedimento nesse nó de modo a restaurar o balanceamento da árvore. Tal procedimento é chamado de *rotação*.

Em árvores AVL, existem essencialmente dois tipos de rotação: simples e dupla. Essas rotações podem ser à esquerda ou à direita. A figura a seguir ilustra graficamente a rotação *simples à esquerda*, também chamada de rotação *left-left* ou simplesmente rotação LL. Nessa figura, B e C são subárvores de altura H e A é uma subárvore de altura H + 1. Observe que o nó n_1 toma o lugar do nó n que então é movido para a direita. A subárvore B é posicionada à esquerda do nó n.

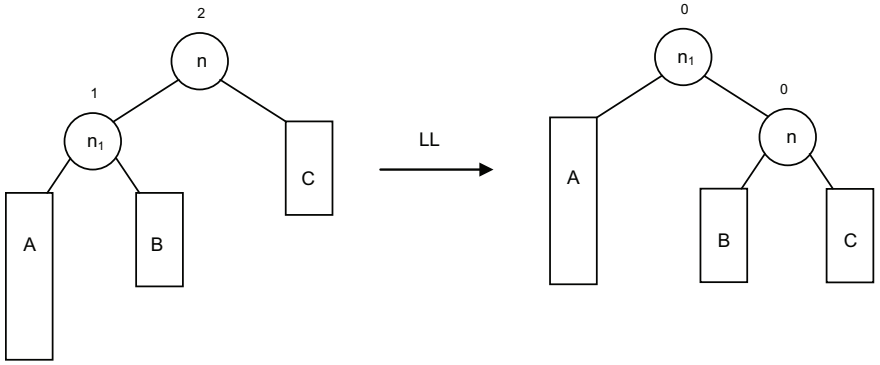


Figura 4.9: Rotação simples à esquerda (LL)

A rotação *simples à direita* (*right-right* ou RR) é análoga à rotação LL.

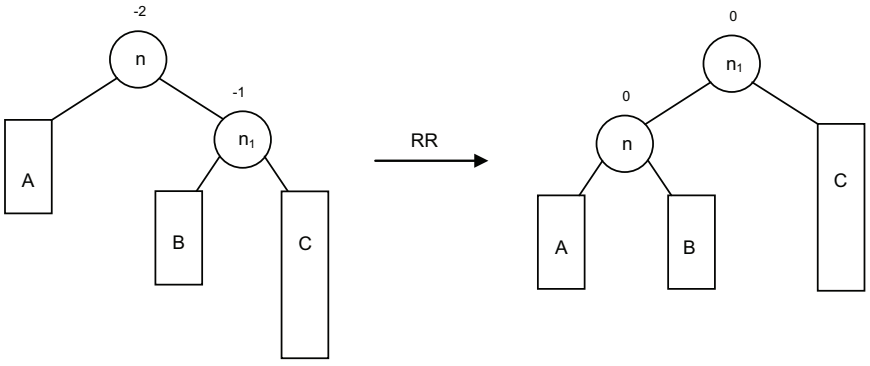


Figura 4.10: Rotação simples à direita (RR)

Temos ainda a rotação *dupla à esquerda*, também conhecida como rotação *left-right* ou LR. Na Figura 4.11 as subárvores A, B e D têm altura H + 1 e a subárvore C tem altura H. O nó n_2 toma o lugar do nó n que então é movido para a direita. A subárvore B é colocada à direita do nó n_1 e a subárvore C é colocada à esquerda do nó n. O nome dessa rotação deve-se ao fato de que ela equivale a fazer uma rotação simples à direita em torno de n_2 e depois fazer uma rotação simples à esquerda em torno de n.

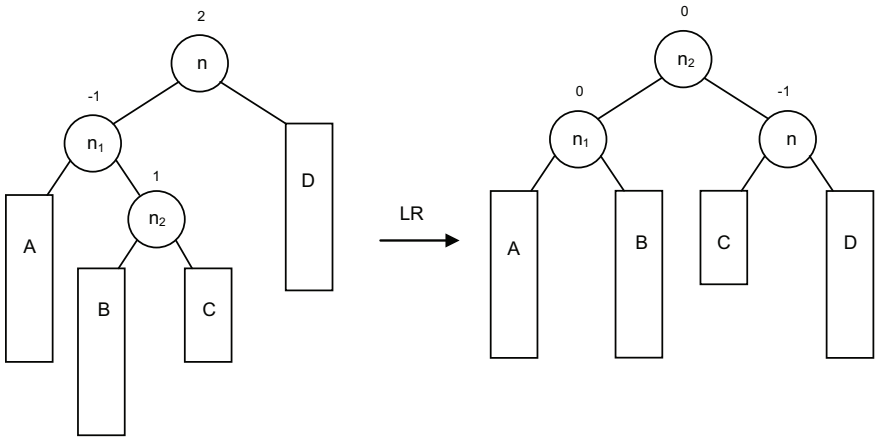


Figura 4.11: Rotação dupla à esquerda (LR)

Na rotação LR a subárvore B poderia ter altura H e a subárvore C poderia ter altura H + 1. A rotação seria feita da mesma forma que foi explicada no parágrafo anterior. Uma única diferença é que ao final da rotação o bal de n_1 seria 1 e o bal de n seria 0. Temos ainda o caso especial em que as subárvores B e C são vazias (e portanto H é igual a 1). Mais uma vez, nada muda no procedimento de rotação, exceto pelo fato de que ao final da rotação os nós n_1 e n terão bal 0.

A figura a seguir ilustra a rotação *dupla à direita* (*right-left* ou RL) que é análoga à rotação LR.

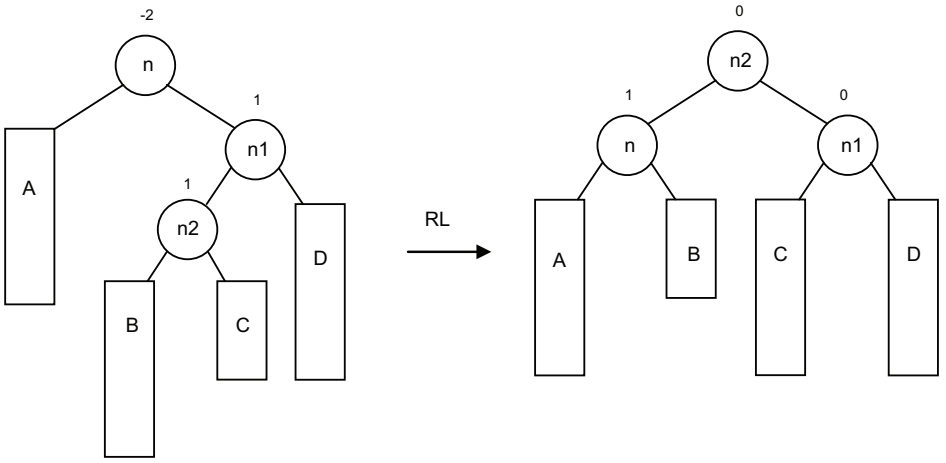


Figura 4.12: Rotação dupla à direita (RL)

Todas essas rotações podem ser feitas em tempo constante, pois exigem apenas o redirecionamento de uma quantidade limitada de ponteiros

e a atualização do bal de no máximo três nós. Uma análise cuidadosa das rotações nos permite concluir que além de restaurar o balanceamento, as rotações mantém a ordenação existente na árvore.

Note que em todas as rotações a raiz da subárvore na qual foi feita a rotação passa a ter bal zero. Isso significa que numa operação de inserção será feita no máximo uma rotação.

Na Figura 4.13 mostramos como ficaria a árvore da Figura 4.8 após a inserção da chave 16. Note que o bal do nó que contém a chave 17 seria atualizado para 1 e o bal do nó que contém o 19 passaria a ser 2. Para restaurar o balanceamento foi necessária uma rotação LL em torno desse nó.

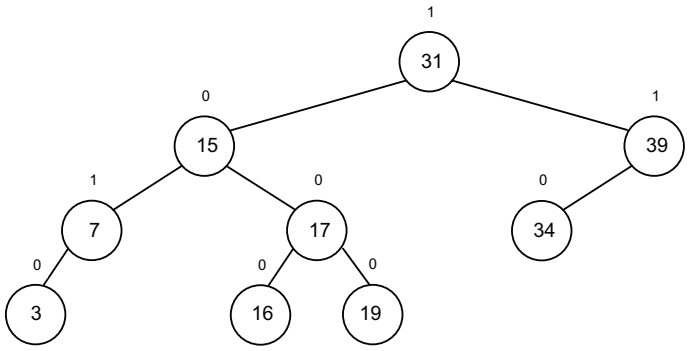


Figura 4.13: Árvore da Figura 4.8 após a inserção da chave 16

Resta-nos discutir a operação de remoção. Infelizmente essa operação é ainda mais complicada que a inserção, razão pela qual vamos tratá-la fazendo distinção entre dois casos.

Abordaremos primeiro a remoção de uma chave k contida numa folha. Naturalmente, tal folha deve ser removida. Em seguida, devemos atualizar o bal dos ancestrais dessa folha. Se a folha estava à esquerda de seu pai, o bal do pai deve ser decrementado. Se ela estava à direita, o bal do pai deve ser incrementado.

Durante uma remoção, se o bal de um nó tornar-se 0 será preciso propagar a atualização do bal para o seu nó pai. Se esse nó estiver à esquerda de seu pai, o bal do pai deve ser decrementado, caso contrário, o bal do pai deve ser incrementado. Se o bal de um nó tornar-se -1 ou 1, a remoção é finalizada. Se o bal de um nó tornar-se –2 ou 2, será necessário realizar uma das quatro rotações explicadas anteriormente.

Convém salientar que é possível que o bal de um nó seja atualizado para 2 e o bal de seu filho esquerdo seja 0. Nesse caso, podemos fazer uma rotação simples ou dupla à esquerda. Analogamente, é possível que o bal de um nó torne-se -2 e o bal de seu filho direito seja 0. Teremos a opção de fazer uma rotação simples ou dupla à direita. Em geral escolhemos fazer a rotação simples.

Note que em todas as rotações a raiz da subárvore na qual foi feita a rotação passa a ter bal zero. Isso significa que na remoção, após uma rotação será preciso propagar a atualização do bal para os seus ancestrais. Isso pode levar à necessidade de múltiplas rotações.

A Figura 4.14 mostra como ficaria a árvore da Figura 4.13 após a remoção da chave 34. Como essa chave estava à esquerda de seu pai, teremos que decrementar o bal de seu pai passará a ser 0. Em seguida, teremos que atualizar o bal da raiz, que passa a ser 2. Teremos então que fazer uma ro-

tação à esquerda, que pode ser simples ou dupla, visto que o filho esquerdo da raiz tem bal 0. Optamos por fazer uma rotação LR para que possamos fazer uma remoção mais interessante em seguida.

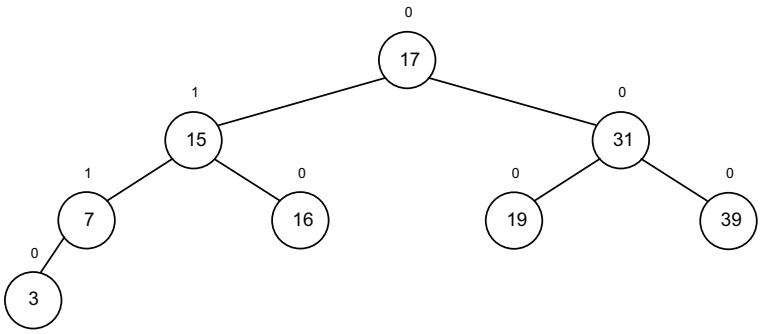
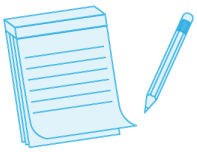


Figura 4.14: Árvore da Figura 4.13 após a remoção da chave 34



ANOTE

Se o nó que contém k não tiver filho esquerdo, devido à propriedade do balanceamento, seu filho direito será folha. Ele deve então tomar o lugar do nó que contém k.

Vamos agora analisar a remoção de uma chave k contida em um nó interno. Nesse caso, devemos substituir k por sua ***chave antecessora***, que é a maior chave da subárvore à esquerda de k. Encontramos tal chave percorrendo a subárvore à esquerda de k sempre para a direita até atingir um nó que não tenho filho direito.

Se o nó que contém a chave antecessora for uma folha, recaímos no caso que tratamos anteriormente, pois teremos que remover a chave antecessora. Se a chave antecessora não estiver numa folha, devemos substituí-la por seu filho esquerdo, que necessariamente será folha, visto que a chave antecessora não pode ter filho à direita. Como o filho esquerdo da chave antecessora é uma folha, recaímos novamente no caso de remover uma folha e devemos proceder como explicado anteriormente.

Na figura a seguir mostramos como ficaria a árvore da Figura 4.14 após a remoção da chave 17. Observe que essa chave deve ser substituída por sua chave antecessora, que é o 16. Ao remover a folha que contém o 16, precisamos incrementar o bal de seu pai, que passará a ser 2. Precisaremos de uma rotação LL em torno do nó que contém o 15. Após essa rotação, teremos que decrementar o bal da raiz.

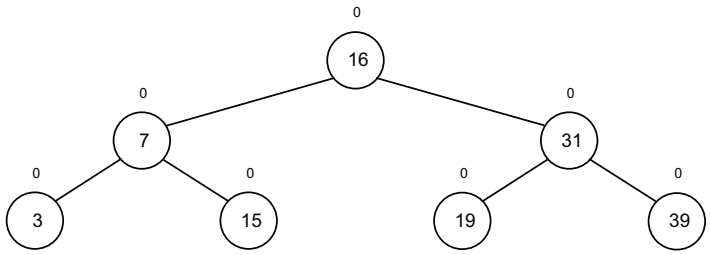


Figura 4.15: Árvore da Figura 4.14 após a remoção da chave 17

Vamos agora analisar a complexidade das operações de busca, inserção e remoção em árvores AVL. No pior caso da operação de busca, é preciso percorrer a árvore a partir da raiz até atingir uma folha que esteja no último nível. Na inserção e na remoção pode ser necessário percorrer a árvore desde a raiz até atingir o último nível e então subir na árvore atualizando o bal dos ancestrais do nó que foi inserido ou removido e, eventualmente, fazendo rotações. Torna-se claro, portanto, que essas operações gastam tempo limi-

tado pela altura da árvore. O seguinte teorema relaciona a altura de uma árvore AVL com a quantidade de chaves contidas na árvore.

Teorema AVL: Seja H a altura de uma árvore AVL que contém n chaves. Então:

$$\log_2(n + 1) \leq H \leq 1{,}44 \log_2(n + 2) - 0{,}328.$$

Esse teorema implica que a complexidade das operações de busca, inserção e remoção em árvores AVL é O(logn). São, portanto, estruturas de pesquisa bastante eficientes.

5.2 Árvores B

Nessa e na próxima seção estudaremos as árvores B e as árvores B+. Tais estruturas são muito utilizadas na prática devido à extrema eficiência com que são feitas as operações de busca, inserção e remoção. A maioria dos sistemas gerenciadores de bancos de dados utiliza essas estruturas, especialmente as árvores B+, para criar arquivos de índices. Elas também são utilizadas por diversos sistemas de arquivos.

As árvores B foram propostas em 1972 por Rudolf Bayer e Edward McCreight, pesquisadores da Boeing Research Labs, e constituem uma generalização das árvores 2-3.

Ao contrário das árvores AVL, cada nó de uma árvore B pode armazenar diversas chaves. Uma característica importante de uma árvore B é a sua *ordem*. A ordem da árvore determina a quantidade de chaves que um nó pode armazenar e também a quantidade de filhos que um nó pode ter. Uma árvore B de ordem m possui as seguintes propriedades:

- Cada nó da árvore armazena de m a 2m chaves, exceto a raiz, que armazena de 1 a 2m chaves.
- Se um nó interno armazena k chaves então ele tem que ter k + 1 filhos.
- Todas as folhas estão contidas no último nível da árvore.
- A subárvore à esquerda de uma chave x contém apenas chaves menores do que x e a subárvore à direita de x contém apenas chaves maiores do que x.
- Em cada nó da árvore as chaves são mantidas em ordem estritamente crescente.

As três primeiras propriedades garantem o balanceamento da árvore. De fato, as árvores B são extremamente bem balanceadas. Surpreendentemente, não há necessidade de rotações para manter o balanceamento. Isso decorre do fato de que tais árvores crescem “para cima”, na direção da raiz, fato incomum entre as árvores.

As outras duas propriedades garantem que as chaves contidas na árvore estejam implicitamente ordenadas. Elas também implicam que árvores B não podem armazenar chaves repetidas.

A estrutura de um nó de uma árvore B de ordem m pode ser constituída dos seguintes campos:

- c: um vetor de chaves com 2m posições.
- p: um vetor de ponteiros com 2m + 1 posições.
- numchaves: indica a quantidade de chaves contidas no nó.
- pai: ponteiro para o nó pai.

Os campos numchaves e pai, embora não sejam imprescindíveis, facilitam sobremaneira a implementação de diversas operações em árvores B. Em um nó da árvore, o ponteiro armazenado em p[i] aponta para a subárvore à esquerda da chave armazenada em c[i] e p[i + 1] aponta para a subárvore à direita de c[i]. A figura a seguir mostra como poderia ser a estrutura de um nó de uma árvore B de ordem 2.

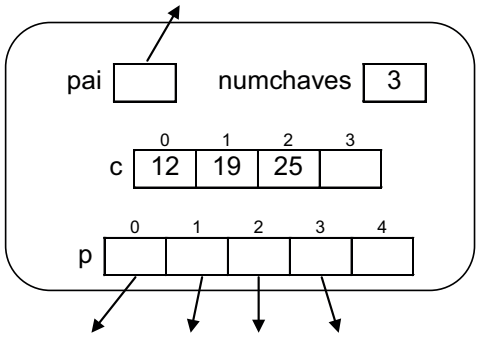


Figura 4.16: Estrutura de um nó de uma árvore B

Exibimos a seguir um exemplo de árvore B de ordem 1.

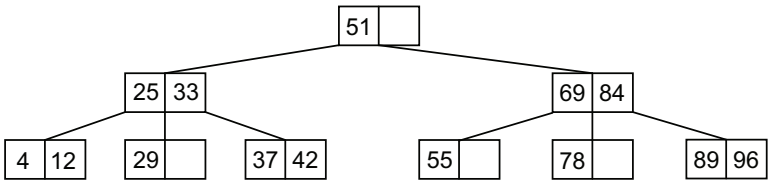


Figura 4.17: Exemplo de árvore B

A busca de uma chave em árvores B é simples. Para procurar uma chave k, primeiramente verificamos se a árvore é vazia. Em caso afirmativo, a busca para, tendo sido mal sucedida. Caso contrário, verificamos se k ocorre na raiz da árvore. Nesse caso, a busca tem sucesso. Caso contrário, determinamos a menor chave contida na raiz que é maior do que k. Se essa chave existir, repetimos o procedimento, recursivamente, na subárvore à esquerda dessa chave. Se k for maior do que todas as chaves contidas na raiz, repetimos o procedimento, recursivamente, na subárvore à direita da maior chave contida na raiz.

Na árvore da Figura 4.17, para buscar a chave 37 teremos que inspecionar a raiz. Como a chave 37 é menor do que 51, devemos seguir o ponteiro para a subárvore à esquerda do 51, chegando ao nó que contém as chaves 25 e 33. Como 37 é maior do que essas chaves, devemos seguir o ponteiro para a subárvore à esquerda do 33, chegando ao nó que contém o 37. Se quisermos procurar a chave 30, começaremos inspecionando a raiz. Como a chave 30 é menor do que 51, seguiremos para o nó que contém as chaves 25 e 33. Como 33 é a menor chave desse nó que é maior do que 30, devemos seguir o ponteiro para a subárvore à esquerda do 33, chegando ao nó que contém o 29. Visto que 30 é maior do que 29, devemos seguir o ponteiro para a subárvore à direita do 29. Observe que tal subárvore é vazia e, portanto, a busca será mal sucedida.

Vamos agora explicar como fazer a inserção de chaves em árvores B. Para inserir uma chave k numa árvore B de ordem m devemos percorrê-la,

a partir da raiz, como se estivéssemos procurando k. Se a chave k for encontrada, a inserção deve ser abortada, pois árvores B não podem ter chaves repetidas. Caso contrário, atingiremos uma folha. Se essa folha tiver menos do que 2m chaves, basta inserir k de modo que as chaves contidas nessa folha continuem em ordem crescente.

Suponha agora que a folha contenha 2m chaves. Nesse caso, a folha precisará ser *subdividida*, criando-se uma nova folha. Distribuiremos a chave k e as chaves contidas nessa folha da seguinte maneira. As m menores chaves continuarão na folha, as m maiores chaves serão movidas para a nova folha e a chave *central* (aquela que não está entre as m menores nem entre as m maiores chaves) deve "subir" para o nó pai. Note que se o nó pai tiver 2m chaves, ele também precisará ser subdividido.

Esse processo de subdivisão pode propagar-se até a raiz da árvore. Se a raiz for subdividida, a chave central será armazenada numa nova raiz e a altura da árvore aumentará. A figura a seguir mostra como seria a subdivisão da folha esquerda ao inserir a chave 12.

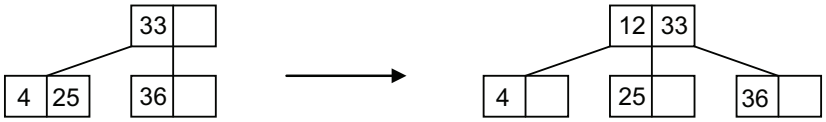


Figura 4.18: Subdivisão de uma folha numa árvore B

Na Figura 4.19 exibimos a árvore da figura 4.17 após a inserção das chaves 73 e 45, nessa ordem. Note que a chave 73 deve ser inserida na folha que contém o 78. Já a chave 45 deve ser inserida na folha que contém as chaves 37 e 42. Como essa folha está cheia (armazena 2m chaves) ela precisará ser subdividida. A chave 42 (chave central) deverá ser movida para o nó pai. Como o nó pai também está cheio, ele também precisará ser subdividido. A chave 33 será movida para a raiz da árvore e a inserção será finalizada.

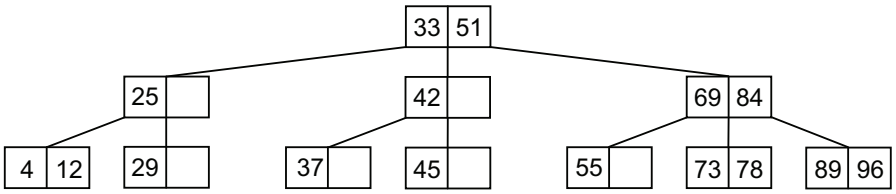


Figura 4.19: Árvore B da Figura 4.17 após a inserção das chaves 73 e 45

Observe que se inserirmos a chave 80 na árvore da Figura 4.19, teremos que subdividir a folha que contém o 73 e o 78, seu nó pai, que contém o 69 e o 84, e a raiz da árvore. Com isso, a árvore passará a ter altura 4.

Vamos agora a discutir operação de remoção. Para remover uma chave k de uma árvore B de ordem m precisamos encontrar o nó da árvore que contém k. Vamos tratar primeiramente o caso em que tal nó é uma folha. Se essa folha tiver mais do que m chaves ou for a raiz da árvore, basta remover k apropriadamente dessa folha.

Suponha agora que a folha que contém k não é a raiz da árvore e armazena exatamente m chaves. Após remover k dessa folha, devemos tentar obter mais uma chave para essa folha de modo que ele continue armazenando m chaves. Para isso, a folha irmã à esquerda (preferencialmente) ou

a folha irmã à direita deve doar uma chave. Essa chave é movida para o nó pai e a chave do nó pai que está entre as folhas envolvidas na doação deve ser movida para a folha que continha k. A Figura 4.20 ilustra a doação de uma chave da folha irmã à esquerda.

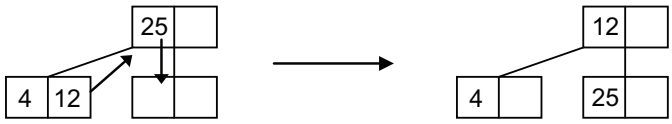


Figura 4.20: Doação de uma chave numa remoção em árvore B

Se nem a folha irmã à esquerda nem a folha irmã à direita tiver mais do que m chaves, será necessário *fundir* a folha que contém k com a folha irmã à sua esquerda (preferencialmente) ou com a folha irmã à sua direita. A chave do nó pai que está entre as folhas que estão se fundindo deve ser movida para a nova folha que será obtida com a fusão. A figura a seguir ilustra a fusão de duas folhas.

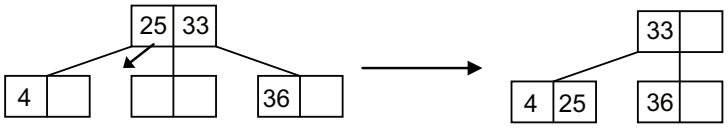


Figura 4.21: Fusão de duas folhas de uma árvore B

Observe que ao fundir duas folhas o nó pai dessas folhas perde uma chave. Se ele não for a raiz e tiver exatamente m chaves será necessário obter a doação de uma chave do seu nó irmão à esquerda (preferencialmente) ou do seu nó irmão à direita. Se nenhum desses dois nós irmãos puder doar uma chave, devemos fundir o nó pai com o seu nó irmão à esquerda (preferencialmente) ou com o seu nó irmão à direita.

Esse processo de subdivisão pode propagar-se até os filhos da raiz da árvore. Se for necessário fundir os únicos dois filhos da raiz, a raiz antiga deixará de existir, o nó obtido com a fusão passará a ser a nova raiz e a altura da árvore diminuirá.

Na Figura 4.22 exibimos a árvore da figura 4.19 após a remoção das chaves 4 e 55, nessa ordem. Note que a folha que contém a chave 4 tem mais do que m chaves e, portanto, basta remover tal chave dessa folha. Já a remoção da chave 55 leva à necessidade de obter a doação de uma chave da folha irmã à sua direita.

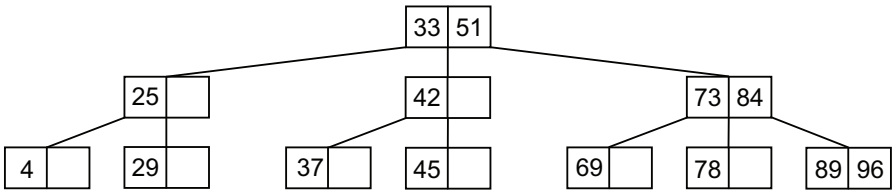


Figura 4.22: Árvore B da Figura 4.19 após a remoção das chaves 4 e 55

Ao removermos a chave 45 da árvore da Figura 4.22 teremos que fundir a folha que contém tal chave com à folha irmã à esquerda. O nó pai das folhas que serão fundidas precisará obter uma doação do nó irmão à direita. A árvore resultante é mostrada na figura a seguir.

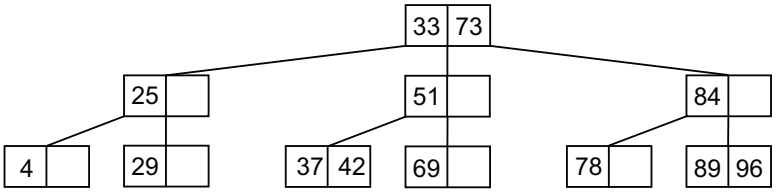


Figura 4.23: Árvore B da Figura 4.22 após a remoção da chave 45

Vamos agora tratar o caso em que a chave k, a ser removida, está em um nó interno. Nesse caso, devemos substituir k por sua *chave antecessora*, que é a maior chave da subárvore à esquerda de k. Observe que tal chave necessariamente estará numa folha e, portanto, recaímos no caso de remoção de uma chave. Devemos então proceder como explicado nos parágrafos anteriores.

Por exemplo, se quisermos remover a chave 33 da árvore mostrada na Figura 4.23, teremos que substituí-la pela chave 29, que é sua chave antecessora. A folha que continha o 29 ficará vazia e terá que ser fundida com sua folha irmã à esquerda e a chave 25 descerá para a folha obtida com a fusão. Precisaremos fazer mais uma fusão envolvendo o nó onde estava a chave 25 e o nó que contém o 51. Com isso, a chave 29 que foi para a raiz deverá descer para o nó obtido nessa última fusão.

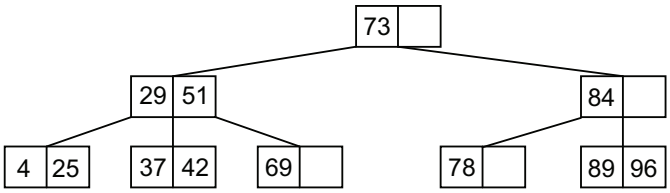


Figura 4.24: Árvore B da Figura 4.23 após a remoção da chave 33

Deve ter ficado claro que a operação de busca requer que a árvore seja percorrida a partir da raiz até, no máximo, atingir uma folha. Dessa forma, tal operação gasta tempo linearmente proporcional à altura da árvore, no pior caso. Na inserção e na remoção é necessário percorrer a árvore da raiz até atingir o último nível e, eventualmente, subir na árvore realizando subdivisões ou fusões. Fica claro, portanto, que essas operações gastam tempo limitado pela altura da árvore. O teorema a seguir relaciona a altura de uma árvore B com a quantidade de chaves contidas na árvore.

Teorema de Bayer-McCreight: Seja H a altura de uma árvore B de ordem m que contém n chaves. Então:

$$\lfloor \log_{2m}(n+1) \rfloor \leq H \leq \lceil \log_{m+1}(n+1) \rceil.$$

Esse teorema implica que a complexidade temporal das operações de busca, inserção e remoção em árvores B é O(logn). Vemos ainda que quanto maior a ordem da árvore, menor será sua altura. Na prática, em comum utilizar ordens bem maiores do que 1, fazendo com que a altura da árvore seja muito pequena. Por exemplo, se a ordem da árvore B for 50 e ela armazenar 1 bilhão de chaves, sua altura será no máximo 5. São, portanto, estruturas de pesquisa extremamente eficientes.

5.3 Árvores B+

As árvores B+, também propostas por Bayer e McCreight, são bem parecidas com as árvores B, tendo apenas duas diferenças mais significativas:

- Todas as chaves válidas contidas na árvore têm que aparecer em alguma folha da árvore.
- Cada folha possui um ponteiro que aponta para a folha imediatamente à sua direita.

Convém salientar que as chaves contidas nos nós internos servem apenas para orientar o caminhamento na árvore. A primeira diferença é mais importante e requer o relaxamento de uma das propriedades das árvores B, descrita no início da Subseção 5.2:

- A subárvore à esquerda de uma chave x contém apenas chaves *menores ou iguais a* x e a subárvore à direita de x contém apenas chaves maiores do que x.

A figura a seguir mostra uma árvore B+ de ordem 1 com as mesmas chaves contidas na árvore B da Figura 4.24.

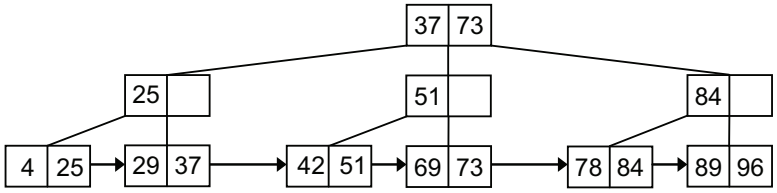


Figura 4.25: Exemplo de árvore B+

Como numa árvore B+ uma chave só é válida se aparece em alguma folha, em toda operação de busca devemos percorrer a árvore a partir da raiz até atingir uma folha. Para procurar uma chave k, primeiramente verificamos se a árvore é vazia. Em caso afirmativo, a busca para, tendo sido mal sucedida. Caso contrário, verificamos se a raiz da árvore é folha. Se esse for o caso, verificamos se x ocorre na raiz. Se for este o caso, a busca tem sucesso; caso contrário, a busca é mal sucedida. Se a raiz não for uma folha, determinamos a menor chave contida na raiz que é maior ou igual a k. Se essa chave existir, repetimos o procedimento, recursivamente, na subárvore à esquerda dessa chave. Se k for maior do que todas as chaves contidas na raiz, repetimos o procedimento, recursivamente, na subárvore à direita da maior chave contida na raiz.

Para buscar a chave 29 na árvore da Figura 4.25 teremos que inspecionar inicialmente a raiz. Como 29 é menor do que 37, devemos seguir o ponteiro para a subárvore à esquerda do 37, chegando ao nó que contém a chave 25. Visto que 29 é maior do que 25, devemos seguir o ponteiro para a subárvore à direita do 25, chegando à folha contém o 29. Se quisermos procurar a chave 75, começaremos inspecionando a raiz. Como 75 é maior do que 73, seguiremos para o nó que contém a chave 84. Visto que 75 é menor do que 84, devemos seguir o ponteiro para a subárvore à esquerda do 84, chegando à folha que contém o 78 e o 84. Tal folha não contém o 75 e, portanto, a busca é mal sucedida.

A inserção em árvores B+ é similar à inserção em árvores B, com apenas uma diferença significativa: ao subdividir uma *folha*, a chave central deve ser *copiada* para o nó pai. Dessa maneira, a chave central continua a ser uma chave válida, pois ela permanece numa folha. Por exemplo, ao

inserir a chave 75 na árvore da Figura 4.25, a folha que contém as chaves 78 e 84 precisará ser subdividida. A chave 78 será então copiada para o nó pai. Eis a árvore obtida com essa inserção.

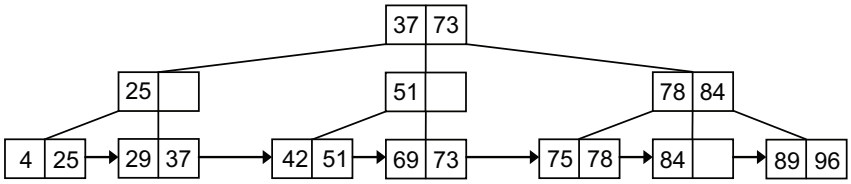


Figura 4.26: Árvore B+ da Figura 4.25 após a inserção da chave 75

A remoção em árvores B+ também é similar à remoção em árvores B, com apenas duas diferenças mais significativas. Uma delas está na forma com é feita a doação de uma chave contida numa folha.

Por exemplo, ao fazer a doação de uma chave da folha irmã à esquerda, a segunda maior chave da folha que está doando é copiada para o nó pai e ocupa o lugar da chave que estava entre as folhas envolvidas na doação. A maior chave da folha que está doando é então movida para a folha que está recebendo a doação. A figura a seguir ilustra essa diferença. Note que a chave 75 foi copiada para o nó pai e o 78 foi movido para a folha que antes continha o 84. Observe que o 84 ainda aparece na árvore, mas não numa folha. Dessa forma, o 84 não é mais uma chave válida da árvore.

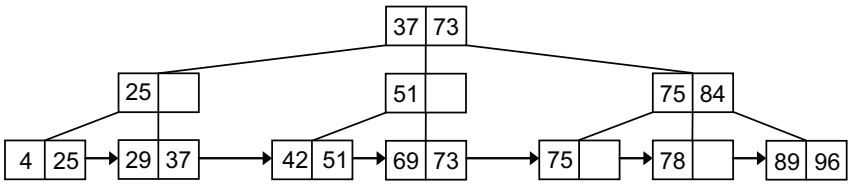


Figura 4.27: Árvore B+ da Figura 4.26 após a remoção da chave 84

A outra diferença ocorre na fusão de duas *folhas*. Nesse caso, a chave do nó pai que está entre elas simplesmente é *eliminada*. Observe na figura a seguir como fica a árvore da Figura 4.27 após a remoção da chave 75.

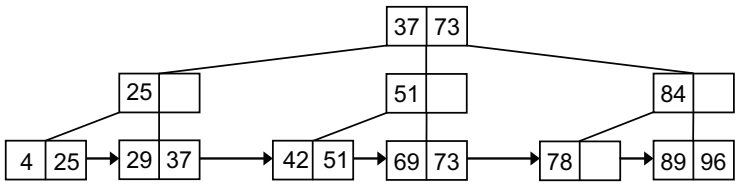
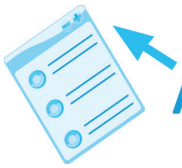


Figura 4.28: Árvore B+ da Figura 4.27 após a remoção da chave 75

É possível mostrar que uma árvore B+ que contenha as mesmas chaves válidas que uma árvore B terá no máximo um nível a mais do que a árvore B. Sendo assim, pelo Teorema de Bayer-McCreight, concluímos que a altura de uma árvore B+ é logarítmica na quantidade de chaves contidas na árvore.

Claramente, as operações de busca, inserção e remoção em árvores B+ gastam tempo linearmente proporcional à altura da árvore. Sendo assim, tais operações são feitas em tempo $\Theta(\log n)$. Assim como as árvores B, as árvores B+ também são estruturas de pesquisa extremamente eficientes.



ATIVIDADES DE AVALIAÇÃO

1. Escreva uma versão não recursiva do algoritmo de busca binária.

2. Escreva uma função que receba um vetor em ordem crescente e um valor x e devolva o índice da posição do vetor cujo conteúdo é mais próximo de x. Sua função deverá requerer tempo logarítmico no tamanho do vetor (*sugestão*: adapte o algoritmo de busca binária).

3. Mostre como ficaria uma tabela de hashing fechado com 13 posições, após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem (nessa e na próxima questão, os valores associados às chaves devem ser ignorados). Utilize a seguinte função de *hashing*: $h(x) = x \bmod 13$. Em seguida, remova as chaves 46, 8 e 74, nesta ordem, e mostre como ficaria a tabela.

4. Mostre como ficaria uma tabela de hashing aberto com 7 posições, após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem. Utilize a seguinte função de *hashing*: $h(x) = x \bmod 7$. Em seguida, remova as chaves 15, 23 e 51, nesta ordem, e mostre como ficaria a tabela.

5. Explique o que é a *carga* de uma tabela de hashing e diga quando ela é considerada *baixa*. Explique também o que é uma *boa* função de hashing.

6. Escreva um algoritmo que receba uma tabela de hashing fechado (passada por referência) e uma chave k e então insira essa chave na tabela, se ela ainda não existir na tabela.

7. Insira as chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nessa ordem, numa árvore AVL. Em seguida, remova as chaves 83, 69 e 9, nessa ordem. Após cada inserção ou remoção desenhe como ficou a árvore, incluindo o bal de cada nó. Mencione também as rotações utilizadas nas inserções e remoções.

8. Escreva um algoritmo que receba um ponteiro para a raiz de uma árvore AVL e imprima o conteúdo dos nós da árvore em ordem decrescente.

9. Escreva uma função que receba um ponteiro para a raiz de uma árvore AVL e devolva a altura da árvore. Sua função deverá requerer tempo logarítmico no tamanho da árvore.

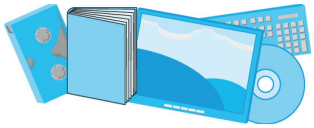
10. Escreva um algoritmo que receba um ponteiro para a um nó de uma árvore AVL e realize uma rotação simples à direita em torno desse nó.

11. Mostre como ficaria uma árvore B de ordem 1 após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem. Em seguida, remova as chaves 24, 33, e 12, nessa ordem, e mostre como ficaria a árvore.

12. Mostre como ficaria uma árvore B+ de ordem 1 após a inserção das chaves 46, 8, 74, 15, 23, 51, 83, 69, 9, 24, 33 e 12, nesta ordem. Em seguida, remova as chaves 46, 8 e 74, nessa ordem, e mostre como ficaria a árvore.

13. Escreva uma função que receba um ponteiro para a raiz de uma árvore B+ e devolva a quantidade de chaves contidas na árvore.

14. Discorra sobre a eficiência das operações de inserção, remoção, busca, busca aproximada e listagem em ordem em árvores B e B+.



LEITURAS, FILMES E SITES

Sites

- http://pt.wikipedia.org/wiki/Bubble_sort
- http://pt.wikipedia.org/wiki/Insertion_sort
- http://pt.wikipedia.org/wiki/Selection_sort
- http://pt.wikipedia.org/wiki/Shell_sort
- http://pt.wikipedia.org/wiki/Merge_sort
- http://pt.wikipedia.org/wiki/Quick_sort
- http://pt.wikipedia.org/wiki/Heapsort
- http://pt.wikipedia.org/wiki/Count_sort
- http://pt.wikipedia.org/wiki/Bucket_sort
- http://pt.wikipedia.org/wiki/Radix_sort
- http://pt.wikipedia.org/wiki/Busca_binária
- http://pt.wikipedia.org/wiki/Tabela_de_dispersão
- http://pt.wikipedia.org/wiki/Árvore_AVL
- http://pt.wikipedia.org/wiki/Árvore_B
- http://pt.wikipedia.org/wiki/Árvore_B+
- http://pt.wikipedia.prg/wiki/Sort-merge_utility
- http://www.cs.pitt.edu/~kirk/cs1501/animations/Sort1.html
- http://slady.net/java/bt/view.php?w=750&h=500

REFERÊNCIAS

Ascencio A.F.G., Aplicações de **Estrutura de Dados em Delphi**. São Paulo: Pearson Prentice Hall, 2005.

Cormen. T.H., C.E. Leiserson, R.L. Rivest, and C. Stein, **Algoritmos: Teoria e Prática**. Rio de Janeiro: Editora Campus, 2002.

Feofiloff, P. **Algoritmos em Linguagem C**. Elsevier, 2008.

Horowitz. E. and S. Sahni. **Fundamentos de Estrutura de Dados**. Rio de Janeiro: Editora Campus, 1987.

Tenenbaum, A.M., Y. Langsam and M.J. Augenstein. **Estrutura de Dados usando C**. São Paulo: Pearson Makron Books, 1996.

Ziviani, N. **Projeto de Algoritmos**. 2.ed., São Paulo: Thomson, 2004.

_____. **Projeto de Algoritmos com Implementações em Java e C++**. Cengage Learning, 2006.