

Notas de aula

Pesquisa e Ordenação de Dados “A”

João Vicente Ferreira Lima
Departamento de Linguagens e Sistemas de Computação
Universidade Federal de Santa Maria

6 de maio de 2018

Versão compilada em 6 de maio de 2018. Esta obra está licenciada sob uma licença [Creative Commons](#) (Atribuição-NãoComercial-SemDerivações 4.0 Internacional).

Material baseado nas aulas do prof. Sérgio Luis Sardi Mergen (UFSM) e slides do livro “Projeto de Algoritmos” do prof. Nivio Ziviani.

Sumário

1 Introdução

As mídias de armazenamento podem ser analisadas por diversos aspectos como:

- velocidade de acesso aos dados.
- custo unitário.
- confiabilidade nos casos de perda de dados por desligamento de energia ou falha no sistema, ou ainda falha física.

As mídias podem ser classificadas de duas formas:

- **Armazenamento volátil** - perde o conteúdo quando a energia é desligada. Ex.: memória principal (RAM).
- **Armazenamento não volátil** - o conteúdo permanece (persiste) mesmo após o desligamento da energia. Ex.: memórias secundários (como disco rígido) e terceária.

1.1 Memória principal

Texto.

1.2 Memória secundário

Texto.

1.3 Memória terciária

Texto.

2 Introdução a análise e complexidade de algoritmos

2.1 Definição de algoritmo

- Procedimento bem definido que:
 - recebe como entrada um valor, ou conjunto de valores
 - produz como saída um valor ou conjunto de valores
- Conjunto de passos que transforma a entrada na saída.

Exemplo do **problema de ordenação**:

- **Entrada:** uma sequência de n números $\langle a_1, a_2, \dots, a_n \rangle$.
- **Saída:** Uma permutação $\langle a'_1, a'_2, \dots, a'_n \rangle$ da entrada tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.
- Essa entrada do algoritmo é uma **instância** do problema.
- Para um algoritmo estar **correto**, para cada instância, ele termina com saída correta. Um algoritmo correto *resolve* um problema.

Exemplos de problemas:

- O Projeto do Genoma Humano para identificar todos os genes do DNA humano.
- Internet com descoberta de rotas.
- E-commerce.

2.2 Análise de algoritmos

- Medida de custo de execução de um programa.
- Importante separar entre uma medida **real** e **generalizada**.
 - real depende do compilador, hardware, memória, etc.
 - generalizada pode usar um computador abstrato para análise.
 - * Ex: máquina teóricas (Turing, lambda).
- Grande parte dos casos utiliza-se o número de **elementos de entrada**, denotado n , para determinar o tempo de um algoritmo.
- Se fosse um grafo $G = (V, E)$, poderia-se usar na análise $|V|$, ou seja, o número de vértices ou o número de arestas $|E|$.

- O **tempo de execução** é o número de passos executados, sendo o mais independente de máquina possível.

Vamos considerar dois exemplos de algoritmos de ordenação com n elementos de entrada:

- **Ordenação por inserção** com custo $c_1 n^2$ sendo c_1 uma constante que não depende de n .
- **Ordenação merge sort** com custo $c_2 n \log n$ sendo c_2 outra constante que não depende de n .

Considere um exemplo onde vamos executar a ordenação por inserção em uma máquina mais rápida (A) e a ordenação por mergesort em uma máquina mais lenta (B). A entrada será de 10 milhões de números (10^7 ou equivalente a 80 MB). Suponha que a máquina A executa 10^{10} instruções por segundo, enquanto que a máquina B executa somente 10^7 instruções por segundo.

Ainda, dadas as implementações dos algoritmos, o código da inserção tem custo $2n^2$ enquanto que o mergesort tem $50n \log n$ de custo.

Para ordenar 10^7 números, o computador A com ordenação por inserção demora

$$\frac{2(10^7)^2 \text{ instrucoes}}{10^{10} \text{ instrucoes/segundo}} = 20.000 \text{ segundos.}$$

Depois, para ordenar os 10^7 números no computador B com mergesort custa

$$\frac{50(10^7) \log 10^7 \text{ instrucoes}}{10^7 \text{ instrucoes/segundo}} = 1.163 \text{ segundos (20 minutos).}$$

2.3 Função de complexidade

O custo de um algoritmo é definido em geral por uma função de complexidade f . Sendo assim, $f(n)$ é a medida de tempo de execução de um algoritmo para uma instância do problema de tamanho n . Podemos ter uma função $f(n)$ que descreve:

- Complexidade de **tempo** onde $f(n)$ mede o tempo para executar um problema n .
- Complexidade de **espaço** onde $f(n)$ mede o espaço de memória ocupado pelo algoritmo com entrada n .

Caso nada for mencionado, estamos falando sobre complexidade de tempo em grande parte do material.

2.4 Analise simples - maior elemento

A figura ?? mostra o algoritmo que encontra o maior elemento em um vetor. Seja $f(n)$ a função de complexidade que determina o número de comparações entre elementos de A . Logo, $f(n) = n - 1$, para $n > 0$.

No caso do algoritmo de maior elemento, o custo de execução é uniforme sobre todos os problemas de tamanho n .

```

1 int Max( int *A, int n ){
2     int i, temp;
3     temp = A[0];
4     for( i = 1; i < n; i++ ){
5         if( temp < A[i] )
6             temp = A[i];
7     }
8     return temp;
9 }
```

Figura 2.1: Algoritmo que encontra o maior elemento.

```

1 void Insercao( int *A, int n ) {
2     int i, j;
3     int x;
4     for( j = 1; j < n; j++ ){
5         x = A[j];
6         i = j - 1;
7         while( i >= 0 && x < A[i] ){
8             A[i+1] = A[i];
9             i--;
10        }
11        A[i+1] = x;
12    }
13 }
```

Figura 2.2: Algoritmo de ordenação por inserção.

2.5 Analise da ordenação por inserção

O código da figura ?? mostra o algoritmo da ordenação por inserção.

A ordem dos elementos na ordenação por inserção depende do tamanho da entrada n . Além disso, o tempo de execução pode variar conforme a ordem dos elementos. O tempo de execução depende da soma dos tempos de execução de cada passo do algoritmo executado: se o passo tem custo c_i e executa n vezes contribui $c_i n$ para o tempo total. Assumindo o pior caso, quando a entrada está em ordem reversa, o custo de execução é $an^2 + bn + c$ para constantes a , b e c .

2.6 Melhor caso, caso médio e pior caso

Em alguns algoritmos, o custo de execução também é uma função da entrada de dados, não somente do tamanho n da entrada. No exemplo anterior, demostramos apenas o pior caso. Os possíveis casos para análise são:

- **Pior caso** – quando os elementos estão em ordem inversa e todos os elementos são trocados. Sabendo o pior caso, o custo do algoritmo nunca é maior que $f(n)$. O custo seria $an^2 + bn + c$ para constantes a , b e c .
- **Melhor caso** – onde os elementos já estão ordenados e nenhuma troca é necessária. O custo é $an + b$ para constantes a e b .
- **Caso médio** – pode ser tanto quanto o pior caso. Supondo que aplicamos a ordenação por inserção para elementos em ordem aleatória n . O resultado de uma execução de caso médio será quadrática ($an^2 + bn + c$) tal como no pior caso.

2.7 Ordem de crescimento

A **ordem de crescimento** é uma forma de simplificar a análise de algoritmos e desconsiderar detalhes de custo. Na ordenação por inserção, consideramos apenas o termo maior (an^2) e desconsideramos sua constante (a) e os termos menores (n). Para grandes valores de n , os termos menores são relativamente insignificantes.

Na tabela abaixo, é possível comparar as diferentes funções de crescimento e o tempo de execução de acordo com a entrada.

descrição	função	2x	10x	tempo	tempo $10n$ (10× rápido)
constante	1				
linear	n	2	10	um dia	algumas horas
linear-logarítmico	$n \log n$	2	10	um dia	algumas horas
quadrático	n^2	4	100	algumas semanas	um dia
cúbico	n^3	8	1.000	meses	algumas semanas
exponencial	2^n	2^n	2^{9n}	nunca	nunca

2.8 Notação assintótica

A notação assintótica é uma forma de denotar o crescimento de uma função, e uma notação para comparar algoritmos.

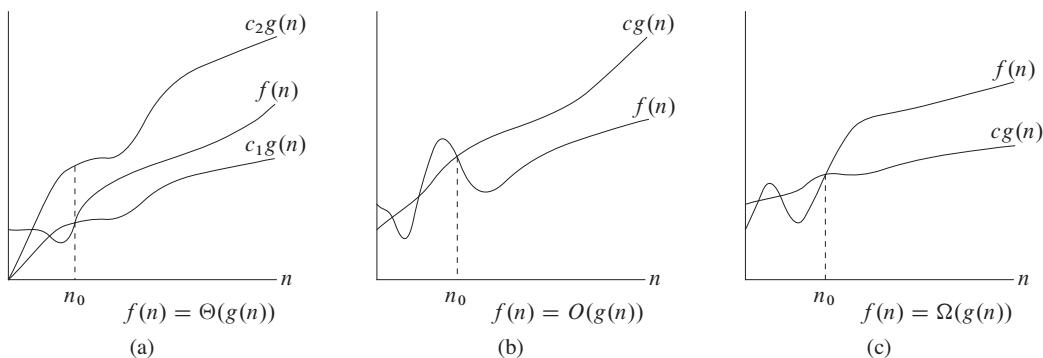


Figura 2.3: Comparação das notações Θ , O e Ω .

- O (*big-oh*) define o **limite assintótico superior**, ou seja, limite para o tempo de execução para o pior caso. Escrevemos $g(n) = O(f(n))$ para expressar que $f(n)$ domina assintoticamente $g(n)$. Para uma função $g(n)$ define-se $O(g(n))$ o conjunto de funções $O(g(n)) = \{f(n) : \text{existe constantes positivas } c \text{ e } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}$.
- Θ é um **limite assintótico firme** onde para uma função $g(n)$ define-se $\Theta(g(n))$ o conjunto de funções $\Theta(g(n)) = \{f(n) : \text{existe constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tal que } c_1g(n) \leq f(n) \leq c_2g(n) \text{ para todo } n \geq n_0\}$.
- Ω é um **limite inferior** onde para uma função $g(n)$ define-se $\Omega(g(n))$ o conjunto de funções $\Omega(g(n)) = \{f(n) : \text{existe constantes positivas } c \text{ e } n_0 \text{ tal que } cg(n) \leq f(n) \text{ para todo } n \geq n_0\}$.

No caso da ordenação por inserção, podemos afirmar que o tempo de execução fica entre $\Omega(n)$ (melhor caso) e $O(n^2)$ (pior caso).

Nesse curso vamos focar na notação O , ou seja, onde $f(n) = O(g(n))$ tal que $f(n) \leq cg(n)$ para $n \geq n_0$. Exemplos:

- $2n + 10$ é $O(n)$:
 - $2n + 10 \leq cn$
 - $(c - 2)n \geq 10$
 - $n \geq 10/(c - 2)$
 - Vale para $c = 3$ e $n_0 = 10$.
- n^2 não é $O(n)$:
 - $n^2 \leq cn$
 - $n \leq c$
 - Não pode ser satisfeito porque c precisa ser constante.
- $7n - 2$ é $O(n)$.
- $3n^3 + 20n^2 + 5$ é $O(n^3)$.
- $3 \log(n) + 5$ é $O(\log(n))$.

Alguma regras para a notação O :

- Se $f(n)$ tem grau d então $f(n)$ é $O(n^d)$, ou seja:
 - Ignora items de ordem menor.
 - Ignora constantes.
- Use a menor ordem possível.
 - “ $2n$ é $O(n)$ ” invés de “ $2n$ é $O(n^2)$ ”.
- Use a expressão mais simples da ordem.
 - “ $3n + 5$ é $O(n)$ ” invés de “ $3n + 5$ é $O(3n)$ ”.

2.9 Principais classes de problemas

- $f(n) = O(1)$ são algoritmos de **complexidade constante** independentes de n com número de instruções fixas.
- $f(n) = O(\log n)$ são algoritmos de **complexidade logarítmica**. Classe típica de problemas que dividem um problema em outros menores (divisão e conquista).
- $f(n) = O(n)$ são algoritmos de **complexidade linear**. Em geral um pequeno trabalho é realizado sobre cada elemento.
- $f(n) = O(n \log n)$ são algoritmos que em geral quebram um problema maior em outros menores, resolvem cada um deles e juntam as soluções depois.
- $f(n) = O(n^2)$ são algoritmos de **complexidade quadrática**. Ocorrem quando os dados são processados aos pares, muitas vezes em laços aninhados.
- $f(n) = O(n^3)$ são algoritmos de **complexidade cúbica**. Um exemplo desses algoritmos é o caso de três laços aninhados.
- $f(n) = O(2^n)$ são algoritmos de **complexidade exponencial**. Eles não são úteis do ponto de vista prático (problemas **NP-complete**). Em geral ocorrem em soluções que usam *força bruta*. Exemplo: quando n é 20, o tempo de execução chega a 1 milhão.
- $f(n) = O(n!)$ são algoritmos de complexidade exponencial, apesar de $O(n!)$ ser muito pior que $O(2^n)$. Também ocorrem em casos de força bruta.

2.10 Algoritmos polinomiais e exponenciais

- **Algoritmo polinomial** em tempo de execução tem complexidade $O(p(n))$, onde $p(n)$ é um polinômio.
- **Algoritmo exponencial** em tempo de execução tem complexidade $O(c^n)$, $c > 1$.

Um algoritmo é eficiente quando seu tempo de execução é polinomial.

Um problema é dito **intratável** se não existe um algoritmo polinomial para resolvê-lo.

2.11 Exercícios

1. Dê o conceito de algoritmo.
2. O que significa dizer que uma função $g(n)$ é $O(f(n))$?
3. Qual algoritmo você prefere: um algoritmo que requer n^5 passos ou um que requer 2^n passos ?

4. Suponha que um algoritmo A e um algoritmo B com funções de complexidade de tempo $a(n) = n^2 + n + 549$ e $b(n) = 49n + 49$, respectivamente. Determine quais são os valores de n pertencentes ao conjunto dos números naturais para os quais A leva menos tempo para executar do que B.
5. Qual é o menor valor de n tal que um algoritmo cujo tempo de execução é $100n^2$ funciona mais rápido que um algoritmo cujo tempo de execução é 2^n na mesma máquina ?
6. Indique, para cada par de expressões (A, B) na tabela a seguir, se A é O , Ω ou Θ de B. Suponha que $k \geq 1$, $\epsilon > 0$ e $c > 1$ são constantes. Sua resposta deve estar na forma da tabela, com “sim” ou “não” escrito em cada espaço.

A	B	O	Ω	Θ
$\log^k(n)$	n^ϵ			
n^k	c^n			
\sqrt{n}	$\sin(n)$			
2^n	$2^{n/2}$			
$n^{\log(c)}$	$c^{\log(n)}$			
$\log(n!)$	$\log(n^n)$			

Parte I

Classificação de dados

3 Introdução

Os algoritmos deste capítulo resolvem o **problema de ordenação**:

- **Entrada:** uma sequência de n números $\langle a_1, a_2, \dots, a_n \rangle$.
- **Saída:** Uma permutação $\langle a'_1, a'_2, \dots, a'_n \rangle$ da entrada tal que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Ordenação pode ser usado em diversos outros algoritmos. Ela pode ser necessária devido a requisitos do usuário, ou para a otimização de pesquisa como na pesquisa binária.

Em geral os dados são mantidos em um vetor onde cada objeto possui um atributo **chave** que deve ser mantido ordenado. Para fins de exemplo, utiliza-se números inteiros como elementos.

Um algoritmo de ordenação possui duas características principais:

- **Estabilidade** – relativo a manutenção da ordem original dos itens com chaves iguais.
 - Um algoritmo de ordenação é **estável** se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação.
- **Uso de memória** – quanto ao uso de memória pelo algoritmo.
 - **Com cópia de dados** – utiliza um vetor temporário para realizar a ordenação. As trocas são feitas entre o vetor original e o temporário.
 - **In-place** – as trocas são feitas dentro do próprio vetor original.

O critério de avaliação, sendo n o número de registros, pode ser por:

- $C(n)$ – número de comparações.
- $M(n)$ – número de movimentações de elementos.

Em grande parte dos casos, nos concentramos no número de comparações.

Os métodos de ordenação podem ser *interno* (em memória primária) ou *externo* (em memória secundária). Na *interna* o arquivo de entrada cabe todo na memória principal, enquanto que na *externa* o arquivo não cabe na memória principal.

A maioria dos métodos é baseada em *comparações* de chaves. Porém, existem outros métodos que utilizam o princípio da **distribuição**. Um exemplo é ordenar um baralho com 52 cartas na ordem numérica e ordem de naipes. O algoritmo seria:

1. Distribuir cartas em treze montes: ases, dois, três,, reis.
2. Coletar os montes na ordem especificada.
3. Distribuir novamente as cartas em quatro montes: paus, ouros, copas e espadas.
4. Coletar os montes na ordem especificada.

Alguns desses métodos são o *radixsort* e o *bucketsort*.

4 Classificação em memória primária

Podem ser classificados como:

- **Métodos simples** – adequado para vetores pequenos, requerem $O(n^2)$ comparações. Ex.: bolha, inserção, seleção, shellsort.
- **Métodos eficientes** – adequados para vetores grandes, requerem $O(n \log n)$ comparações. Ex.: quicksort, mergesort, heapsort.
- **Métodos mais eficientes** – ou por distribuição, requerem $O(n)$ atribuições. Ex.: radixsort.

4.1 Métodos de ordenação simples

4.1.1 Seleção (*selection sort*)

A ordenação por seleção coloca o menor elemento sempre no começo do vetor. O algoritmo é ilustrado na figura ??.

```
1 void Selecao( int *A, int n ){
2     int i, j, min;
3     for( i = 0; i < n; i++ ){
4         min = i;
5         for(j= i+1; j < n; j++)
6             if( A[j] < A[min] )
7                 min = j;
8         troca( A[min], A[i] );
9     }
10 }
```

Figura 4.1: Algoritmo de ordenação por seleção.

O número de comparações é $(n - 1) + (n - 2) + \dots + 2 + 1$ com complexidade (qualquer caso)

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

Vantagens:

- custo linear para movimentações.
- interessante para vetores pequenos.

Desvantagens:

- vetor ordenado não ajuda, pois o custo continua quadrático.
- algoritmo **não é estável**.

4.1.2 Bolha (*bubble sort*)

A ordenação por bolha ordena os elementos ao colocar o maior sempre no fim do vetor. O algoritmo é ilustrado na figura ??.

```

1 void Bolha( int *A, int n ){
2     int i;
3     bool trocado;
4     do{
5         trocado = false;
6         for( i = 1; i < n; i++){
7             if( A[i-1] > A[i] ){
8                 troca( A[i-1], A[i] );
9                 trocado = true;
10            }
11        }
12    } while(trocado);
13 }
```

Figura 4.2: Algoritmo do bubblesort.

O número de comparações é $(n - 1) + (n - 2) + \dots + 2 + 1$ com complexidade (para pior caso)

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

As principais vantagens:

- Algoritmo simples.
- Algoritmo estável.

4.1.3 Inserção (*insertion sort*)

A figura ?? da seção ?? ilustra o algoritmo. O número de comparações é $(n - 1) + (n - 2) + \dots + 2 + 1$ no pior caso (ordem reversa), com complexidade

$$C(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2).$$

Por outro lado, o número de comparações no melhor caso (vetor ordenado) é $1+1+1+1+\dots+1 = n - 1$ com complexidade

$$C(n) = n - 1 = O(n).$$

Vantagens:

- ideal quando o vetor está “quase” ordenado.
- ordenação estável.

Desvantagens:

- custo médio é quadrático.
- alto custo para inserir elemento na posição correta.

4.1.4 Shellsort

Proposto por Shell em 1959, o algoritmo é uma extensão do algoritmo por inserção. Ele é eficiente quando a entrada está parcialmente ordenada. Porém, ele é ineficiente no caso geral pois troca os valores apenas uma posição por vez.

A proposta geral do shellsort é trocar elementos de posições distantes para vetores muito desordenados, e trocar elementos próximos para entradas parcialmente ordenados. O algoritmo é ilustrado na figura ???. Quando $h = 1$ shellsort corresponde ao algoritmo de inserção.

```

1 void Shellsort( int *A, int n ){
2     int i, j;
3     int h = 1;
4     while( h < n/3 )
5         h = 3*h + 1;
6     while( h >= 1 ) {
7         for( i = h; i < n; i++ ){
8             for( j = i; j >= h && A[j] < A[j-h]; j -= h )
9                 troca( A[j], A[j-h] );
10        }
11        h = h/3;
12    }
13 }
```

Figura 4.3: Algoritmo de ordenação Shellsort.

A eficiência depende do intervalo (gap) usado. Exemplos:

- $gap = \frac{n}{2^k} = \frac{n}{2}, \frac{n}{4}, \dots, 1 = O(n^2)$.
- $gap = 2^k - 1 = 1, 3, 7, 15, 31, \dots = O(n^{\frac{3}{2}})$.

Vantagens:

- podem ser mais eficiente que os demais algoritmos de ordem quadrática.

Desvantagens:

- não é estável.

4.1.5 Comparação dos métodos simples

A tabela ?? sumariza os métodos simples e suas complexidades.

Tabela 4.1: Comparação dos métodos simples.

	Melhor caso	Caso médio	Pior caso	Memória	Estável
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	1	sim
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	1	não
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	1	sim
Shellsort	$O(n)$	$O(n^{3/2})$	$O(n^{3/2})$	1	não

4.2 Métodos eficientes

Os métodos eficientes são baseados na estratégia de **divisão e conquista** (D&C) onde:

- **divisão** – divide um problema maior em problemas menores (subproblemas) recursivamente até que não seja mais possível dividir (caso base).
- **conquista** – a solução dos problemas menores leva a solução do problema maior.

Diversos métodos de ordenação aplicam divisão e conquista. Os algoritmos detalhados aqui são o *quicksort* e o *mergesort*.

4.2.1 Quicksort

Proposto em 1960 e publicado em 1962, a ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Os problemas menores são ordenados independentemente, e combinados para produzir a solução final.

O Quicksort funciona da seguinte forma:

1. selecionar um elemento como **pivô**.
2. particionar os elementos em dois subvetores:
 - (a) elementos menores que o pivô.
 - (b) elementos maiores que o pivô.
3. executar o quicksort recursivamente em cada subvetor.

Os passos do **particionamento** são:

1. escolher um **pivô** *pivo*.

2. percorrer o vetor a partir da esquerda até que $A[esq] \geq pivo$.
3. percorrer o vetor a partir da direita até que $A[dir] \leq pivo$.
4. trocar $A[esq]$ com $A[dir]$.
5. continuar até que esq e dir se cruzem, ou seja, enquanto $esq >= dir$.

O algoritmo do Quicksort é ilustrado na figura ?? enquanto que o particionamento na figura ??.
Nota-se que o pivô escolhido em cada particionamento é o número na primeira posição do subvetor (linha 4).

```

1 void Quicksort( int *A, int esq, int dir ){
2     int i;
3     if( esq < dir ){
4         i = Partition( A, esq, dir );
5         Quicksort( A, esq, i - 1 );
6         Quicksort( A, i + 1, dir );
7     }
8 }
```

Figura 4.4: Algoritmo de ordenação Quicksort.

```

1 int Partition( int *A, int inicio, int fim ){
2     int esq = inicio;
3     int dir = fim + 1;
4     int pivo = A[inicio];
5     do {
6         while( pivo > A[++esq] )
7             if( esq == fim ) break;
8         while( pivo < A[--dir] )
9             if( dir == inicio ) break;
10            if( esq < dir )
11                troca( &A[esq], &A[dir] );
12    } while( esq < dir );
13    troca( &A[inicio], &A[dir] );
14    return dir;
15 }
```

Figura 4.5: Algoritmo de partição do Quicksort.

A análise do Quicksort depende da **escolha do pivô** para balanceamento do particionamento, essencial na eficiência do algoritmo.

O melhor caso do Quicksort acontece quando os dois subvetores possuem tamanho $n/2$ e aplica-se Quicksort recursivamente. Dado sua estrutura de árvore, a altura da árvore será $O(\log n)$ e o número de comparações em cada nível da árvore é $O(n)$. Logo, o custo no **melhor caso** é $O(n \log n)$.

Para o pior caso, vamos supor que o pivô é o primeiro elemento e que o vetor já está ordenado. O particionamento resulta em subvetores de tamanho 1 e $n - 1$ onde executa-se Quicksort em ambos recursivamente. Dessa forma, a altura da árvore é $O(n)$ (pode-se imaginar todos os nós com 1 filho) e comparações por nível de $O(n)$. O custo no **pior caso** será $O(n^2)$.

O problema do pior caso no Quicksort pode ser contornado ao escolher o ponto médio de três elementos do vetor. Por exemplo, das posições $A[0]$, $A[n/2]$ e $A[n - 1]$, escolhe-se o que tem valor médio entre eles.

4.2.2 Mergesort

A ideia do Mergesort é dividir um conjunto com n itens em duas partes iguais. Os problemas menores são ordenados independentemente, e mesclados para produzir a solução final.

O algoritmo do Mergesort é ilustrado na figura ?? enquanto que a mescla na figura ??.

```

1 void Mergesort( int *A, int inicio, int fim, int *aux ){
2     int meio;
3     if( inicio < fim ){
4         meio = (inicio + fim - 1)/2;
5         Mergesort( A, inicio, meio, aux );
6         Mergesort( A, meio + 1, fim, aux );
7         Merge( A, inicio, meio, fim, aux );
8     }
9 }
```

Figura 4.6: Algoritmo de ordenação Mergesort.

O algoritmo de Mergesort ilustrado **não é in-place** e utiliza um vetor temporário (ou auxiliar) para a operação de mescla. Dessa forma, pode-se mesclar os elementos no vetor temporário e depois copiar de volta ao vetor original. O uso do vetor auxiliar simplifica o algoritmo, mas necessita do dobro de espaço para fazer a ordenação.

A análise do Mergesort deve considerar que ele divide o problema em dois subvetores de tamanho $n/2$ e aplica Mergesort recursivamente. O custo de comparações em cada nível é cn sendo o primeiro nível $c(n/2) + c(n/2)$, o segundo $c(n/4) + c(n/4) + c(n/4) + c(n/4)$, etc. A árvore de recursão terá $\log n + 1$ níveis, cada um com custo cn . Portanto, o custo total será $cn(\log n + 1) = cn \log n + cn$. O custo no **pior caso** do algoritmo é $O(n \log n)$.

No melhor caso, o vetor de entrada já está ordenado. Uma forma eficiente de descobrir é testar o último elemento do primeiro subvetor e o primeiro elemento do segundo subvetor. As chamadas recursivas ainda são necessárias mas não será preciso fazer chamadas ao Merge. O custo no **melhor caso** é $O(n)$.

```

1 void Merge( int *A, int inicio, int meio, int fim, int *aux ){
2     int i = inicio, j = meio + 1, k;
3     for(k = inicio; k <= fim; k++)
4         aux[k] = A[k];
5     for(k = inicio; k <= fim; k++){
6         if(i > meio)           A[k] = aux[j++];
7         else if(j > fim)      A[k] = aux[i++];
8         else if(aux[j] < aux[i]) A[k] = aux[j++];
9         else                   A[k] = aux[i++];
10    }
11 }

```

Figura 4.7: Algoritmo para mesclar subvetores do Mergesort.

4.2.3 Heapsort

Heapsort possui o mesmo princípio da ordenação por seleção onde:

1. selecionar o maior item do vetor.
2. trocar com o item da primeira posição do vetor.
3. repetir essa operação para os $n - 1$ elementos, os $n - 2$, e assim sucessivamente.

O custo de encontrar o menor elemento é de $n - 1$ comparações, porém esse custo pode ser reduzido com uma estrutura de dados **heap**.

Definição de heap

O heap é uma árvore binária balanceada e justificada a esquerda onde todos os nós satisfazem a *propriedade do heap*: seu valor é menor ou igual ao valor do nó pai. Uma árvore balanceada significa que todos os níveis estão completos com exceção do nível folha. Uma árvore justificada a esquerda significa que os nós do último nível estão mais a esquerda possível. A partir da propriedade do heap, conclui-se que o maior elemento de todos na árvore está no topo.

A inserção no heap sempre ocorre no último nível, a direita do último nó. Se o último nível estiver cheio, começa um novo nível. Na remoção o elemento removido é trocado pelo mais a direita da árvore. As operações de inserção e remoção podem fazer com que a árvore perca a propriedade do heap. A correção é feita por transformações locais, com a troca do nó pai pelo nó filho de maior valor.

Heaps podem ser mapeados para vetores com operações sobre vetores. No caso de um nó de índice i :

- o pai está em $\lceil i/2 \rceil - 1$ ($\lfloor i/2 \rfloor$).
- o filho à *esquerda* está em $2*i + 1$ ($2i$).
- o filho à *direita* está em $2*i + 2$ ($2i + 1$).

A **raiz** é o primeiro elemento do vetor e o nó mais à direita é o último elemento do vetor. A figura ?? ilustra um exemplo de vetor e seu heap correspondente.

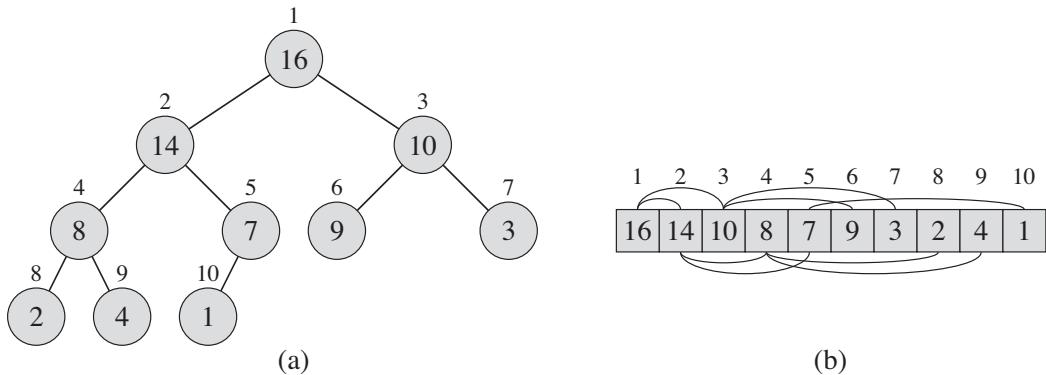


Figura 4.8: Exemplo de vetor e representação como um heap.

Heap para ordenação

O primeiro algoritmo (figura ??) garante a principal propriedade do heap a partir de uma posição i do vetor.

```

1 void Refaz( int *A, int i, int n ){
2     int esq = 2 * i + 1;
3     int dir = 2 * i + 2;
4     int maior;
5
6     if( esq < n && A[esq] > A[i] )      maior = esq;
7     else                                     maior = i;
8     if( dir < n && A[dir] > A[maior] )  maior = dir;
9
10    if( maior != i ){
11        troca( &A[i], &A[maior] );
12        Refaz( A, maior, n );
13    }
14}
```

Figura 4.9: Algoritmo que transforma a árvore em um heap de uma posição i do vetor.

A figura ?? ilustra os passos do algoritmo `Refaz` em um heap inválido devido ao elemento 4.

O segundo algoritmo (figura ??) constroi um heap a partir de um vetor. Para cada posição do vetor, ele usa a função `Refaz()` e transforma o vetor em um heap.

A figura ?? ilustra os passos do algoritmo `Constroi` em um vetor.

O algoritmo Heapsort (figura ??) inicia com a construção de um heap com o vetor de entrada. Como o maior elemento do heap fica na posição $A[0]$, o algoritmo coloca o elemento no final do

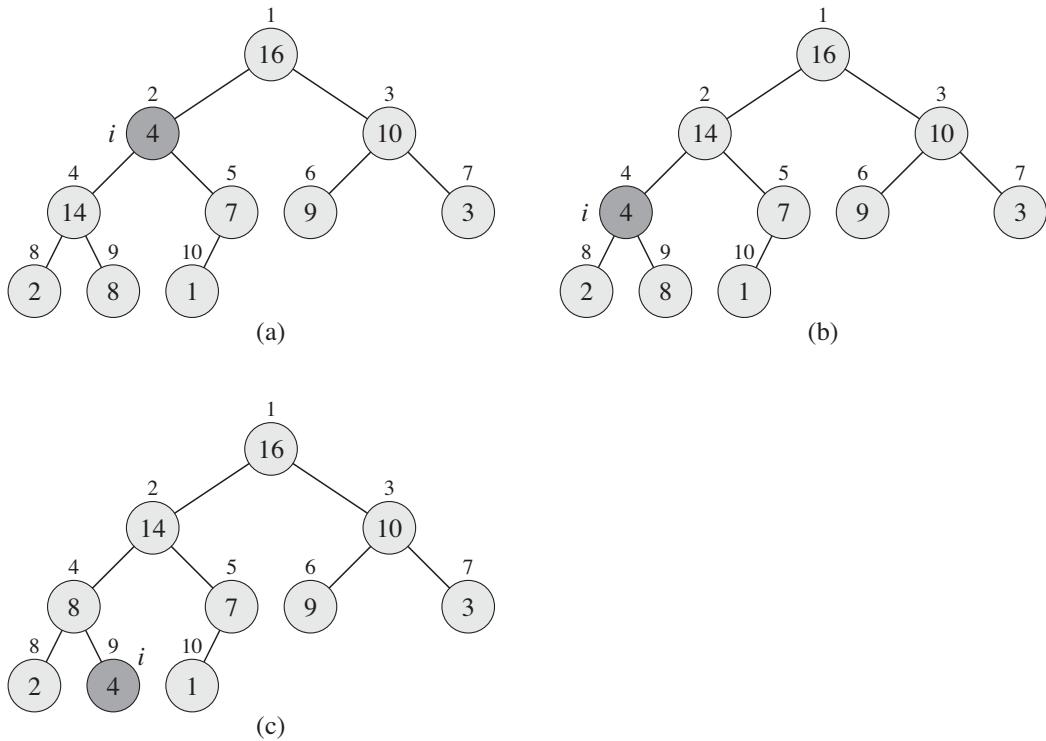


Figura 4.10: Passos do algoritmo `Refaz(A, 2, 10)` na reconstrução de um heap. O elemento da posição 2 faz o heap inválido e é trocado pelo filho de maior valor, no caso A[2] e A[4]. A chamada recursiva com `Refaz(A, 4, 10)` detecta um heap inválido e leva a troca entre os elementos A[4] e A[9].

```

1 void Constroi( int *A, int n ){
2     int k;
3     for(k = n/2; k >= 0; k-- ){
4         Refaz( A, k, n );
5     }
6 }
```

Figura 4.11: Algoritmo que constroi o heap de um vetor n .

vetor ao trocar com $A[n - 1]$. Em seguida ele ignora o último elemento (o maior) e usa `Refaz()` para reconstruir o heap. O algoritmo repete esses dois passos $n - 1$ vezes.

Análise

A análise envolve o custo da construção do heap e da leitura dos elementos em ordem. A construção do heap adiciona um elemento ao final e pode ser deslocado até a raiz. Como a árvore é balanceada, o deslocamento tem custo $O(\log n)$ e isso é feito n vezes. O custo de construir o heap é $O(n \log n)$.

A leitura em ordem a troca do primeiro com o último com custo $O(1)$. Após a troca, o elemento pode precisar ser deslocado até o nível folha. Em uma árvore balanceada o custo de

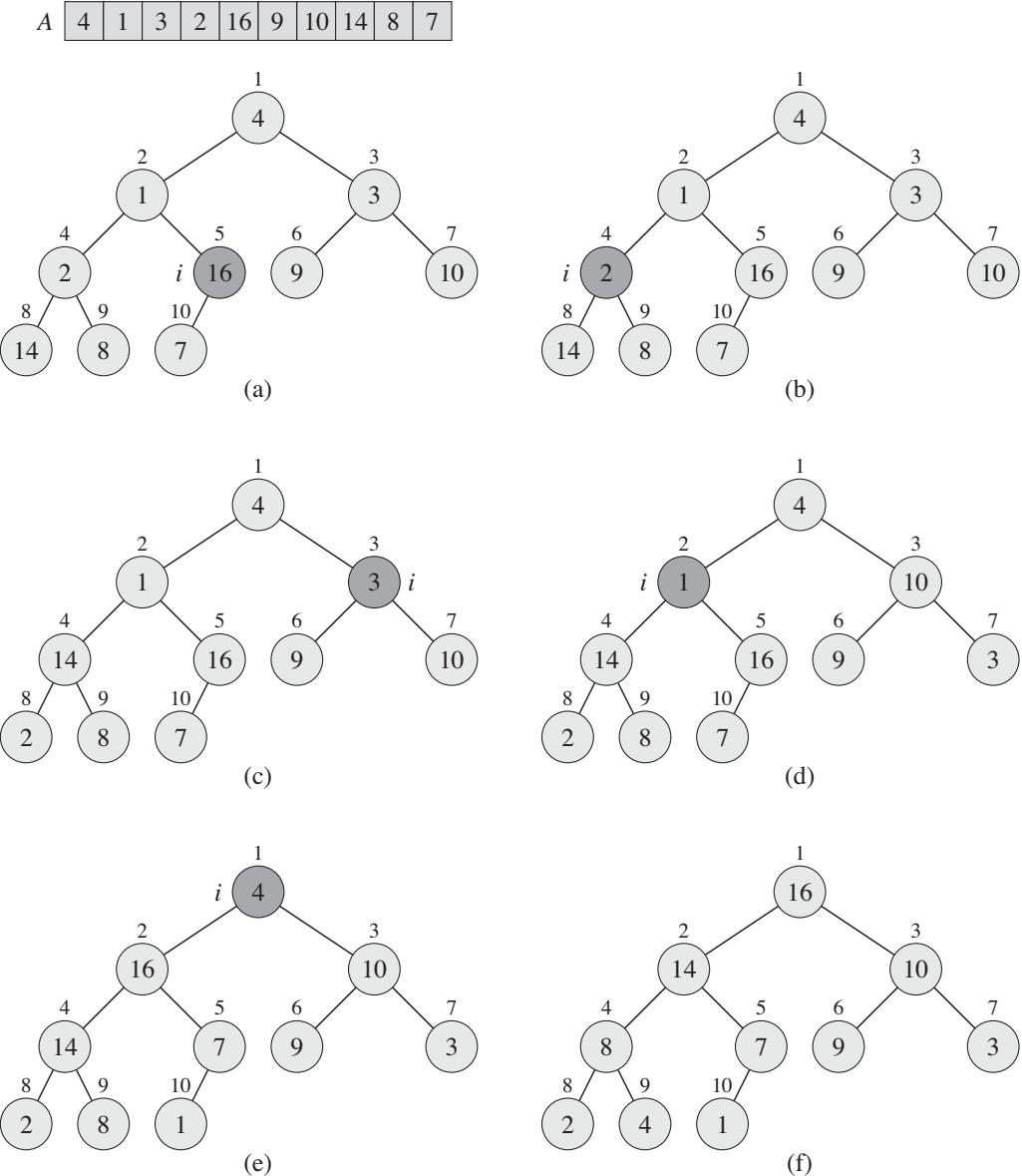


Figura 4.12: Passos do algoritmo `Constroi(A, 10)` na construção de um heap. Note que a função `Refaz` é usada para todos os nós da árvore que não são folhas.

mover é $O(\log n)$. Como isso é feito n vezes, o custo de leitura em ordem é $O(n \log n)$.

O custo total é $2n \log n = O(n \log n)$.

4.2.4 Comparação entre métodos eficientes

A tabela ?? sumariza o Mergesort e Quicksort com suas complexidades.

Em casos típicos o Quicksort tem melhor desempenho que Heapsort, que é melhor que Mergesort. O Heapsort é melhor para aplicações de tempo crítico, ou ordenações parciais (apenas 10 elementos por exemplo).

Há implementações estáveis do Quicksort, porém o desempenho pode ser afetado. Existem algoritmos do Mergesort com memória in-place, mas o desempenho pode ser afetado.

```

1 void Heapsort( int *A, int n ){
2     int i;
3     Constroi( A, n );
4     for( i = n-1; i > 0; i-- ) {
5         troca( &A[0], &A[i] );
6         Refaz( A, 0, i );
7     }
8 }
```

Figura 4.13: Algoritmo do Heapsort: passa o maior elemento para o fim e refaz o heap.

Tabela 4.2: Comparação dos métodos simples.

	Melhor caso	Caso médio	Pior caso	Memória	Estável
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	1	não
Mergesort	$O(n)$	$O(n \log n)$	$O(n \log n)$	n	sim
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	1	não

4.3 Métodos de ordenação por distribuição

Os algoritmos discutidos até agora tem custo $O(n \log n)$ em casos médios. Porém, caso o custo de espaço não é problema e somente a comparação lexicográfica é necessária, é possível usar estratégias sem comparação.

Os algoritmos de ordenação por distribuição distribuem os elementos de entrada em estruturas intermediárias com base nos seus valores. Essas estruturas, por si só, já auxiliam na ordenação dos elementos.

4.3.1 Countingsort

Countingsort assume que os elementos de entrada n são inteiros e os valores (chaves) estão no intervalo de 0 até k para um inteiro k . Esse algoritmo requer um vetor de entrada A , um vetor de saída B e um temporário C de tamanho k sendo k o maior valor armazenado. O algoritmo da figura ?? mostra a implementação do Countingsort.

O custo do Countingsort depende de zerar o vetor C – $O(k)$ –, contar em C o número de ocorrências dos valores de A – $O(n)$ –, somar as ocorrências – $O(k)$ – e ordenar o vetor em B – $O(n)$ –. Assim, o custo será $k + n + k + n = O(k + n)$. O custo em espaço é $O(n + k)$ para usar B e C .

O Countingsort custa menos que $O(n \log n)$ porque não é um algoritmo baseado em comparação. Ele usa o valor dos elementos para indexar o vetor. O algoritmo é estável e não é in-place, assim como só funciona com inteiros.

```

1 void Countingsort( int *A, int* B, int n, int k ){
2     int C[k];
3     int i;
4
5     for( i = 0; i < k; i++)    C[i] = 0;
6     for( i = 0; i < n; i++)    C[A[i]]++;
7     for( i = 1; i < k; i++)    C[i] = C[i] + C[i-1];
8     for( i = n-1; i >= 0; i-- ){
9         B[C[A[i]] - 1] = A[i];
10        C[A[i]]--;
11    }
12 }
```

Figura 4.14: Algoritmo do Countingsort para n elementos de valor até k .

4.3.2 Bucketsort

O Bucketsort assume que a entrada está uniformemente distribuída e que tem um tempo de execução médio de $O(n)$. Ele assume que os números de entrada são distribuídos em um intervalo $[0, 1)$ (por exemplo) e são divididos em r grupos chamados **buckets**.

O algoritmo distribui os n elementos entre os buckets. Para gerar a saída, ele ordena cada bucket e depois copia para saída cada elemento dos buckets em ordem.

O algoritmo da figura ?? mostra a implementação do Bucketsort. Ele usa r como o número de buckets usados e $B[r]$ um vetor de listas para cada bucket. Ao distribuir os números, ele ordena cada bucket com uma função que ordena a lista. A ordenação da lista pode ser feita por Insertionsort. Por fim, ele percorre cada lista para copiar os elementos na saída A .

O custo depende do número de buckets usados. A distribuição é mais uniforme se os buckets serem equivalentes ao número de possíveis valores. Para uma entrada n e r buckets, assumindo uma distribuição uniforme, tem-se n cópias para os buckets, r passagens pelos buckets e n cópias de volta para o vetor original. A ordenação de cada bucket é desprezível pois contém 1 ou poucos elementos. O custo no **caso médio** será $O(2n + r) = O(n + r)$. Por outro lado, o pior caso ocorre quando um bucket tem todos os elementos. O custo no **pior caso** é $O(n^2 + r)$ (usando Insertionsort para ordenar um bucket).

Para uma entrada n e r buckets, o custo em espaço é $O(n + r)$.

4.3.3 Radixsort

O Radixsort é uma extensão do Bucketsort onde o número de buckets depende do tamanho do alfabeto de entrada. O algoritmo realiza diversas rodadas para ordenar os buckets. Na primeira rodada o mapeamento é feito com o dígitos menos significativo. Em seguida o algoritmo ordena pelo próximo dígito mais a esquerda, e assim por diante. O número de rodadas será d sendo d o dígito mais significativo.

Se o maior número da entrada for 999.999, apenas 10 buckets são necessários ao invés de 1 milhão no Countingsort e algumas versões do Bucketsort. No caso de palavras são necessários 26 buckets ou mais, dependendo do tamanho do alfabeto.

```

1 void BucketSort( float *A, int n, int r ){
2     lista_t* B[r];
3     int i, j;
4     for( j = 0; j < r; j++ )    B[i] = lista_cria();
5     for( i = 0; i < n; i++ ){
6         int j = (int)floor(A[i]*10);
7         lista_insere( B[j], A[i] );
8     }
9     for( j = 0; j < r; j++ )    OrdenaLista(B[j]);
10
11    i = 0;
12    for( j = 0; j < r; j++ ){
13        while(!lista_vazia(B[j]))
14            A[i++] = lista_remove(B[j]);
15    }
16 }
```

Figura 4.15: Algoritmo do BucketSort para n com k buckets.

O algoritmo da figura ?? mostra a implementação do Radixsort para números de base 10 com 2 dígitos. Essa implementação é baseada no Countingsort para ordenar os números para um certo dígito d .

O custo do Radixsort, caso o número de rodadas seja fixo, é $O(n)$ para todos os casos. Para um número arbitrário de rodadas k , o custo é $O(kn)$. Para ordenar números, por exemplo, $k = \lfloor \log_{10} m \rfloor + 1$ onde m é o maior valor do vetor.

4.3.4 Comparação entre métodos por distribuição

A tabela ?? ilustra a comparação entre os métodos onde:

- n = número de elementos.
- r = número de divisões dos valores.
- s = tamanho do alfabeto.

Tabela 4.3: Comparação dos métodos por distribuição.

	Melhor caso	Caso médio	Pior caso	Memória	Estável
Countingsort	$O(n + r)$	$O(n + r)$	$O(n + r)$	$n + r$	sim
Bucketsort	$O(n + r)$	$O(n + r)$	$O(n^2 + r)$	$n + r$	sim
Radixsort	$O(kn)$	$O(kn)$	$O(kn)$	$n + s$	sim

A partir dessa tabela, podemos afirmar que o Countingsort e Bucketsort podem ser inviáveis se r é grande. Mais especificamente, o Bucketsort tem complexidade relativa pois depende do quão uniforme é a entrada.

```

1 void Radixsort( int *A, int* B, int n, int digitos ){
2     int C[10];
3     int k = 10;
4     int i, d;
5     int base = 10;
6     int div = 1;
7
8     for(d = 1; d <= digitos; d++){
9         for( i = 0; i < k; i++) C[i] = 0;
10        for( i = 0; i < n; i++) C[(A[i]/div)%base]++;
11        for( i = 1; i < k; i++) C[i] = C[i] + C[i-1];
12        for( i = n-1; i >= 0; i--){
13            B[C[(A[i]/div)%base] - 1] = A[i];
14            C[(A[i]/div)%base]--;
15        }
16        for( i = 0; i < n; i++) A[i] = B[i];
17        div = div * base;
18    }
19 }
```

Figura 4.16: Algoritmo do Radixsort para números de base 10 (2 dígitos).

4.4 Comparação geral de métodos

A tabela ?? mostra uma comparação geral de métodos de ordenação.

Tabela 4.4: Comparação geral dos métodos de ordenação.

	Melhor caso	Caso médio	Pior caso	Memória	Estável
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	1	não
Mergesort	$O(n)$	$O(n \log n)$	$O(n \log n)$	n	sim
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	1	não
Radixsort	$O(kn)$	$O(kn)$	$O(kn)$	$n + s$	sim

Na comparação geral, o Radixsort é melhor para números inteiros. Porém, ele não é genérico.

4.5 Exercícios

1. Invente um exemplo de entrada (vetor) para demonstrar que a ordenação por Seleção é um método instável. Mostre os passos da execução até que a estabilidade seja violada.
2. Ordene os elementos 3, 7, 1, 4, 9, 2 usando os métodos: bolha, seleção, inserção, shellsort (com gaps 3, 2, 1). Não esqueça de exibir o estado do vetor a cada troca de elementos.

3. Ordene os elementos do vetor $3, 10, 8, 9, 5, 4, 1, 2$ usando os métodos: Quicksort (primeiro elemento como pivô) e Mergesort. Exibir o estado do vetor toda vez que ocorrer uma troca.
4. Sobre Heaps
 - (a) Quais são os elementos mínimo e máximo de um Heap com altura h ?
 - (b) Um vetor ordenado um Heap mínimo ?
 - (c) Um vetor com valores $A = [23, 17, 14, 6, 13, 10, 1, 5, 7, 12]$ é um Heap máximo ?
 - (d) Usando o algoritmo da figura ??, ilustre as operações da operação `Refaz(A, 3, 14)` para o vetor $A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0]$.
 - (e) Reescreva o algoritmo da figura ??, que cria um Heap máximo, para gerar um Heap mínimo.
5. Ordene os vetores abaixo usando Heapsort:
 - (a) $A = [5, 3, 17, 10, 8, 9]$.
 - (b) $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.
6. Qual é o custo de execução do Heapsort para um vetor A de tamanho n já classificado em ordem crescente ? E em ordem decrescente ?
7. Qual é o curso do Quicksort quando todos os elementos de A tem o mesmo valor.
8. Suponha que cada um dos elementos em $A[1..n]$ possua um valor entre três valores distintos.
 - (a) Descreva um algoritmo eficiente para ordenar os elementos. (Dica: uma boa solução é utilizar a função principal do Quicksort.).
 - (b) Apresente a análise do pior caso para número de comparações.
9. Que algoritmo de ordenação você usaria para cada um dos seguintes casos, e justifique:
 - (a) A ordem original de elementos com chave idêntica precisa ser mantida.
 - (b) O tempo de execução não pode ter grandes variações para nenhum caso.
 - (c) A lista a ser ordenada já está bem próxima da ordem final.
 - (d) Os elementos a serem ordenados são muito grandes se comparados ao tamanho das chaves.
10. Sobre Quicksort:
 - (a) Descreva uma (ou mais) maneira para manter o tamanho da pilha de recursão (chamadas recursivas) o menor possível na implementação do Quicksort.
 - (b) Sem este artifício, qual seria o tamanho da pilha de chamadas para o pior caso do Quicksort ?
11. Ordene os números abaixo usando:
 - Countingsort: $1, 5, 3, 9, 7, 3, 2, 1, 4, 6, 6, 5, 4, 1$. Mostre o valor das estruturas temporárias.

- Radixsort: 2, 17, 324, 45, 41, 86, 26, 91, 118, 99, 19. Mostre o estado do vetor após cada rodada de ordenação.
- Bucketsort: .79, .13, .16, .64, .39, .20, .89, .53, .71, .42. Mostre o estado do vetor após cada rodada de ordenação.

5 Classificação em memória secundária

Classificação em memória secundária, ou ordenação em memória externa, consiste em ordenar arquivos de tamanho maior que a memória principal (interna) disponível. Nos algoritmos de ordenação externa deve-se reduzir o número de acessos ao disco, responsável por grande parte do custo do algoritmo.

Os três principais fatores que diferenciam ordenação externa de interna são:

1. O custo para acessar um registro.
2. Restrições ao acesso físico, como em fitas magnéticas.
3. Tecnologia empregada.

A maioria dos métodos de ordenação externa utiliza a seguinte estratégia:

1. Quebrar o arquivo em blocos do tamanho da memória interna disponível.
2. Ordenar cada bloco.
3. Intercalar os blocos ordenados, fazendo várias passadas.
4. A cada passada são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

Esses passos são agrupados em **passos**. O **estágio de classificação** envolve os passos 1 e 2, enquanto que o **estágio de intercalação** consiste nos passos 3 e 4. Os métodos para ordenação externa devem reduzir o número de passadas no arquivo.

Os conceitos usados neste capítulo são:

- **Arquivo** - estrutura física de armazenamento.
- **Partição** - estrutura lógica de armazenamento, que pode ser composto de diversas partições.

5.1 Classificação

Considera-se que a memória principal tem capacidade para armazenar M registros do arquivo a classificar.

5.1.1 Classificação interna

A **classificação interna** consiste na leitura de M registros para a memória, ordenação desses registros por qualquer outro algoritmo de classificação interna e gravação desses registros classificados em uma partição. Todas as partições ordenadas contêm M registros, exceto (talvez) a última.

A **classificação interna** envolve os seguintes passos:

1. leitura de M registros para a memória.
2. ordenação desses registros por um algoritmo de ordenação interna.
 - quicksort, mergesort, etc.
3. gravação desses registros classificados.
 - em uma partição nova de um arquivo existente (ou em novo arquivo).

A classificação interna possui a desvantagem de não explorar a ordem parcial existente no arquivo de entrada.

5.1.2 Seleção por substituição

A **seleção por substituição** aproveita a possível classificação parcial do arquivo de entrada. Em média, o tamanho das partições obtidas pelo processo de seleção com substituição é de $2 * M$.

Os passos do algoritmo de seleção por substituição são:

1. Ler M registros do arquivo para a memória.
2. Selecionar, na memória, o registro com menor chave.
3. Gravar na partição de saída o registro com menor chave.
4. Substituir (em memória) o registro gravado pelo próximo registro do arquivo de entrada.
5. Caso a chave deste último seja menor do que a chave recém-gravada.
 - (a) considerá-lo “congelado” e ignorá-lo no restante do processamento.
6. Caso existam em memória registros não “congelados”
 - (a) volta ao passo (2).
7. Caso contrário
 - (a) fechar a partição de saída.
 - (b) “descongelar” os registros “congelados”.
 - (c) abrir nova partição de saída.
 - (d) voltar ao passo (2).

A seleção por substituição tem a desvantagem que no final da partição grande, parte do espaço em memória principal está ocupado com registros “congelados”.

5.1.3 Seleção natural

A classificação por **seleção natural** reserva um espaço na memória secundário (“*o reservatório*”), que abriga os registros “congelados” em um processo de substituição.

A formação de uma partição se encerra quando o reservatório estiver cheio, ou quando terminarem os registros de entrada. Quando um reservatório encher, a memória é descarregada na partição atual e os dados são copiados do repositório à memória.

A classificação por seleção natural usa como memória interna M e um reservatório que comporta R registros. Para $M = R$, o tamanho médio das partições é $M * e$ onde $e = 2,7182\dots$ (número de Euler).

Os passos do algoritmo de seleção natural são:

1. Ler M registros do arquivo para a memória.
2. Selecionar, na memória, o registro com menor chave.
3. Gravar na partição de saída o registro com menor chave.
4. Substituir (em memória) o registro gravado pelo próximo registro do arquivo de entrada.
5. Caso a chave deste último seja menor do que a chave recém-gravada.
 - (a) gravá-lo no reservatório.
 - (b) substituir (em memória) o registro gravado no reservatório pelo próximo registro do arquivo de entrada.
6. Caso ainda exista espaço livre no reservatório.
 - (a) volta ao passo (2).
7. Caso contrário
 - (a) fechar a partição de saída.
 - (b) copiar os registros do reservatório para a memória.
 - (c) esvaziar o reservatório.
 - (d) abrir nova partição de saída.
 - (e) voltar ao passo (2).

5.1.4 Comparaçāo de processos

A classificação interna gera as menores partições, mas simplifica o estágio de intercalação por usar partições de mesmo tamanho. Os processos de seleção exigem intercalação mais elaborada, porém geram partições maiores, e reduzem o tempo de processamento. A seleção natural gera as maiores partições, porém utiliza mais operações de entrada e saída. Além de usar memória externa adicional.

Os processos de seleção podem utilizar Heapsort para a ordenação interna.

5.2 Intercalação

O objetivo da intercalação é **transformar um conjunto de partições classificadas em uma única partição classificada**. Considera-se que a existência de R partições geradas no particionamento. O ideal seria intercalar todas as partições de uma só vez e obter o arquivo classificado. Porém, todo sistema impõe um limite no número de arquivos abertos simultaneamente. Esse número pode ser menos que o número de partições geradas.

A intercalação exige uma série de passos, e em cada passo:

1. registros são lidos de um conjunto de partições dos arquivos de entrada.
2. e gravados em outro conjunto de partições de arquivos de saída.

As operações de entrada e saída (E/S) tem um alto custo na ordenação externa. Uma medida de eficiência do estágio de intercalação é dada pelo número de passos sobre os dados dado pela equação:

$$N_{passos} = \frac{No. \ total \ de \ registros \ lidos}{No. \ total \ de \ registros \ no \ arquivo \ classificado}$$

O menor número possível é 2 (1 leitura e 1 escrita).

5.2.1 Intercalação balanceada de n caminhos

A **intercalação balanceada de n caminhos** usa n arquivos para a entrada de registros e outros n arquivos para a saída de registros. Sendo F o número máximo de arquivos que podem estar abertos ao mesmo tempo, $F = 2n$. Dessa forma, a intercalação balanceada de n -caminhos utiliza no máximo $F/2$ caminhos.

O particionamento inicial pode usar qualquer estratégia de particionamento. **Cada partição é escrita em um dos arquivos de entrada de forma intercalada.**

Os passos envolvidos são:

1. primeiro passo: determinar o número de arquivos (F) que o algoritmo irá manipular, sendo a primeira metade ($F/2$) será usado para entrada enquanto a outra metade ($F/2$) para saída.
2. passo dois: distribuir todas as partições, de forma **intercalada**, nos arquivos de entrada.
3. **passo de intercalação**: intercalar as primeiras partições, gravando o resultado em um dos $F/2$ arquivos de saída, em ordem.
4. no final de cada fase, o conjunto de partições de saída torna-se o conjunto de entrada.
5. a intercalação termina quando, em uma fase, grava-se apenas uma partição (ordenada).

5.2.2 Intercalação polifásica

A **intercalação polifásica** permite restringir o número de arquivos usados. Ela também usa $F - 1$ arquivos de entrada, mas permite múltiplas partições por arquivo.

Os passos envolvidos são:

1. primeiro passo: determinar o número de arquivos (F) que o algoritmo irá manipular, sendo $F - 1$ para entrada enquanto 1 será de saída.

2. passo dois: distribuir todas as partições, de forma **intercalada**, nos arquivos de entrada.
3. **passo de intercalação:** intercalar as partições dos arquivos de entrada, gravando o resultado em ordem no arquivo de saída.
4. Se todas as partições selecionadas de entrada chegarem ao fim, mas nenhuma dos arquivos estiver no fim, uma nova participação no arquivo de saída é iniciada. As próximas partições de entrada são selecionadas.
5. Se um dos arquivos chegar ao fim, e todas as partições de entrada selecionadas chegarem ao fim, o arquivo de saída se transforma em um arquivo de entrada. O arquivo de entrada (que chegou ao fim) vira o arquivo de saída e uma nova fase é iniciada.
6. a intercalação termina quando restar apenas 1 arquivo de entrada e 1 participação.

5.2.3 Análise dos métodos de intercalação

A tabela ?? mostra uma comparação dos métodos de intercalação conforme exemplos.

Tabela 5.1: Comparação dos métodos de intercalação.

Estratégia	n. de operações	n. de passos	n. de arquivos usados
Intercalação balanceada	72	$72/12 = 6$	4
Intercalação polifásica	60	$60/12 = 4$	4

A intercalação ótima pode ser usada quando **não houver restrição** quanto ao número de arquivos que podem ser criados, ou quando sabe-se que o número de partições é menor que o limite.

A intercalação polifásica pode ser usada quando **haver restrição** quanto ao número de arquivos que podem ser criados.

5.3 Exercícios

1. Dados os números 10, 8, 7, 11, 9, 13, 16, 12, 15, 14, mostre as partições que são criadas por classificação para os métodos: classificação interna, seleção por substituição, seleção natural. Assuma $M = 3$ e $R = 3$.
2. Descreva os passos dos algoritmos de classificação para os métodos: classificação interna, seleção por substituição, seleção natural.
3. Fazer a intercalação das seguintes partições: 20, 24 – 5, 15 – 8, 9 – 12, 18 – 4, 10 – 7, 11 – 2, 13 – 14, 19. Utilize os métodos de: intercalação balanceada de n caminhos, intercalação ótima e intercalação polifásica. Considere $F = 4$.

Parte II

Pesquisa de dados

6 Pesquisa em memória primária

A pesquisa ocorre em uma estrutura de dados armazenada em memória principal e a informação é dividida em registros com **chaves de busca**. O objetivo da pesquisa é encontrar uma ou mais ocorrências de registros com chave de busca igual a chave usada na pesquisa.

O armazenamento dos dados é em geral feita por **vetores** e/ou **listas encadeadas** (pesquisa sequencial e pesquisa binária), ou **árvores de pesquisa binária** (árvores com ou sem平衡amento).

6.1 Pesquisa sequencial

A **pesquisa sequencial** (ou força-bruta) é o método de pesquisa mais simples e lê todos os dados a partir do início do vetor e compara com a chave de busca até encontrar um registro ou até o final do vetor. A lógica de busca depende da ordenação dos dados e da possibilidade de existirem chaves repetidas.

Alguns exemplos de busca são:

- dados **não ordenados** e chaves **repetidas** - deve-ser varrer todos os registros.
- dados **não ordenados** e chaves **não repetidas** - deve-se varrer todos os registros até encontrar a chaves buscada.
- dados **ordenados** e chaves **repetidas** - deve-se varrer todos os registros até encontrar alguma chave maior do que a buscada. No caminho, vários registros podem ser encontrados.
- dados **ordenados** e chaves **não repetidas** - deve-se varrer todos os registros até encontrar alguma chave maior do que a buscada. No caminho, um registro pode ser encontrado.

O custo médio da pesquisa sequencial é medido em função do número de acessos feitos em cada busca. No caso da busca sem sucesso (não ordenado e não repetido) o custo é $C(n) = n + 1$. Quando tiver sucesso (não ordenado e não repetido) o melhor caso é $C(n) = 1$, pior caso $C(n) = n$ e caso médio $C(n) = (n + 1)/2$.

6.2 Pesquisa binária

A **busca binária** pode ser feito em vetores ou listas em que os elementos devem estar **em ordem**.

Os passos envolvidos na busca binária são:

1. Compare a chave com o registro que está na posição do meio do vetor.
2. Se a chave é menor, o registro procurado está na primeira metade do vetor.
3. Se a chave é maior, o registro procurado está na segunda metade do vetor.
4. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é \neq da procurada.

```

1 int Binaria( int* A, int n, int chave ) {
2     int esq, dir, i;
3     esq = 1;
4     dir = n;
5     do{
6         i = (esq+dir)/2;
7         if( A[i] > chave )   esq = i+1;
8         else                  dir = i-1;
9     } while( (A[i] != chave) && (esq <= dir) );
10    if(chave == A[i]) return i;
11    return -1; /* sem sucesso */
12 }

```

Figura 6.1: Algoritmo de busca binária.

O algoritmo de busca binária é ilustrado na figura ??.

Na análise de custo, deve-se considerar que a cada iteração do algoritmo, o tamanho da tabela é dividido ao meio. Então, o número de divisões é $\log n$. Porém, uma ressalva importante é o custo de manter o vetor ordenado. A pesquisa binária não é recomendado para aplicações muito dinâmicas.

6.3 Árvore binária de pesquisa

Uma **árvore de pesquisa** (binária) (*binary-search-tree*) é uma estrutura de dados eficiente para armazenar informação. Ela é particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:

- acesso direto e sequencial eficientes.
- facilidade de inserção e retirada de registros.
- boa taxa de utilização de memória.

A **propriedade das árvores de busca binária** é:

Seja x um nó em uma árvore de busca binária. Se y é um nó na sub-árvore da esquerda de x , então $y.key \leq x.key$. Se y é um nó na sub-árvore da direita de x , então $y.key \geq x.key$.

A **altura** de um nó é o comprimento do caminho mais longo deste nó até um nó folha. A altura de uma árvore é a altura do nó raiz.

A figura ?? ilustra um exemplo de uma árvore de busca binária.

Os passos na operação de **busca** para uma chave x são:

1. Compare com a chave que está na raiz.
2. Se x é menor, vá para a sub-árvore esquerda.

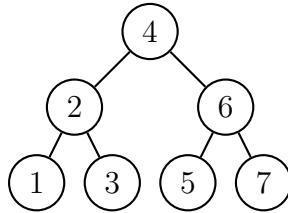


Figura 6.2: Exemplo de árvore de busca binária.

3. Se x é maior, vá para a sub-árvore direita.
4. Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido.

O algoritmo de busca em árvore binária é ilustrado na figura ??.

```

1 Arv* ArvBusca( Arv* a, int chave ) {
2     if( a == NULL )           return NULL;
3     if( a.chave == chave)    return a;
4     if( chave < a.chave )   return ArvBusca( a->esq, chave );
5     else                      return ArvBusca( a->dir, chave );
6     return NULL;
7 }
```

Figura 6.3: Algoritmo de busca em árvore binária.

O procedimento de **inserção** é:

1. Executar o procedimento de busca da chave a ser inserida.
2. O ponto onde deveria estar a chave é o ponto de inserção.

O procedimento de **remoção** não é tão simples:

1. Executar o procedimento de busca da chave a ser removida.
2. Se o nó contém o registro a ser removido possui no máximo um descendente, trocar o nó a ser removido pelo seu descendente.
3. Se possuir dois descendentes o registro a ser removido deve ser primeiro substituído:
 - pelo registro mais à direita na sub-árvore esquerda.
 - ou pelo registro mais à esquerda na sub-árvore direita.

Análise – o número de comparações em uma busca com sucesso no melhor caso é $C(n) = 1$, pior caso $C(n) = n$ e caso médio $C(n) = \log n$. O pior caso ocorre quando as chaves são inseridas em ordem crescente ou decrescente.

6.4 Propriedades em árvores

6.4.1 Percurso em árvores

Há três tipos de percurso em árvores:

Pré-ordem - o nó é visitado antes das sub-árvores descendentes.

Pós-ordem - o nó é visitado depois das sub-árvores descendentes.

Em-ordem - em árvores binárias, o nó é visitado depois da sub-árvore da esquerda e antes da sub-árvore da direita.

6.4.2 Propriedades em árvores binárias

Dada a notação:

- n - número de nós.
- e - número de nós externos (folhas).
- i - número de nós internos (não folhas).
- h - altura.

Segue as seguintes propriedades sobre árvores binárias:

- $e = i + 1$.
- $n = 2e - 1$.
- $h \leq i$.
- $h \leq (n - 1)/2$.
- $e \leq 2^h$.
- $h \geq \log e$.
- $\geq \log(n + 1) - 1$.

6.5 Árvores de busca balanceadas

Uma limitação das árvores binárias de pesquisa é a ordem em que os elementos são inseridos. Por exemplo, as ordens de inserção abaixo:

- ordem 1, 2, 3, 4, 5, 6, 7 gera uma **árvore degenerada** (todo nó tem apenas um filho).
- ordem 4, 6, 2, 5, 1, 7, 3 gera uma árvore binária completa.

O que afeta diretamente o desempenho na busca de elementos.

Idealmente deseja-se que a árvore esteja **completamente balanceada**. A distância média para qualquer nó da árvore é mínima. No entanto, manter uma árvore completamente balanceada tem alto custo. Uma solução é procurar uma solução intermediária que possa manter a árvore balanceada.

Uma **árvore de busca balanceada** garante uma altura de $O(\log n)$ quando implementa um conjunto dinâmico de n itens. Alguns exemplos de árvores平衡adas são AVL, árvores 2-3, árvores 2-3-4, B-trees e árvore rubro-negra.

6.5.1 Árvore AVL

Uma árvore binária é denominada **AVL** (dos seus criadores Georgy Adelson-Velsky e Landis) se para todos os nós, **as alturas de suas duas sub-árvores diferem no máximo em uma unidade**, sendo assim balanceada. Operações de consulta, inserção e remoção de nós tem custo $O(\log n)$.

O **fator de balanceamento** (FB) de um nó é a diferença entre a altura da sub-árvore da esquerda e a altura da sub-árvore da direita. Se não existir uma sub-árvore, a altura é zero. Os resultados de um FB são:

- > 0 - sub-árvore da direita é menor.
- < 0 - sub-árvore da esquerda é menor.
- $= 0$ - sub-árvores tem a mesma altura.

Os valores válidos de FB são -1 (sub-árvore da esquerda é menor), 0 , 1 (sub-árvore da direita é menor).

6.5.1.1 Inserção

O nó é inserido como em uma árvore binária comum. Se a inserção não degenerar a árvore, o processo termina. Caso contrário, é necessário:

1. encontrar o nó cujo FB esteja fora do intervalo (pivô).
2. realizar uma rotação na árvore a partir do nó “pivô” (rotação simples ou rotação dupla).

Apenas uma rotação (simples ou dupla) é necessária. Após essa rotação, a árvore já estará balanceada. Há quatro possibilidades de rotação:

- 2 casos externos (direita e esquerda) com rotação simples.
- 2 casos internos (direita e esquerda) com rotação dupla.

6.5.1.2 Rotação

Suponha que α seja um pivô, nos casos externos, uma rotação simples é necessária caso a inserção ocorra na **sub-árvore à esquerda do filho à esquerda de α** ou na **sub-árvore à direita do filho à direita de α** . A figura ?? demonstra dois exemplos de rotação simples a direta.

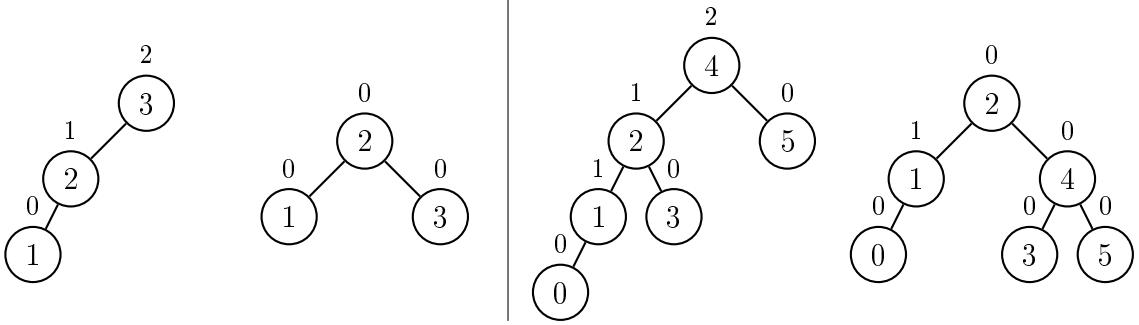


Figura 6.4: Dois exemplos de AVL com rotação simples para a direita e cada nó com seu FB.

Nos casos internos, suponha que o pivô seja α , uma rotação dupla é necessária caso a inserção ocorra na **sub-árvore à esquerda do filho à direita de α** ou na **sub-árvore à direita do filho à esquerda de α** . Uma rotação dupla equivale a duas rotações simples em sequência. A figura ?? demonstra um exemplo de rotação dupla a direta.

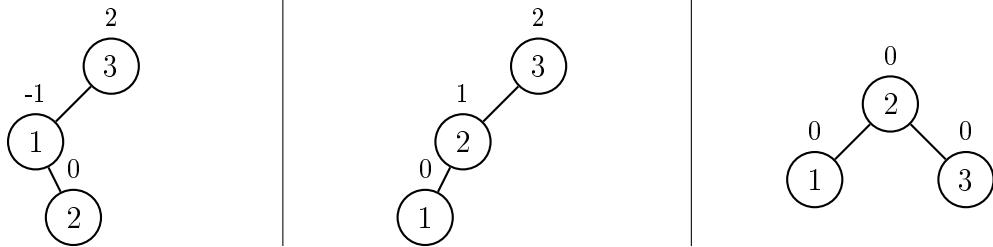


Figura 6.5: Exemplo de AVL com rotação dupla, e cada nó com seu FB.

6.5.1.3 Remoção

A remoção em AVL é similar a inserções, mas mais complexas. Inicialmente, usa-se a estratégia de remoção das árvores não平衡adas. Porém, pode gerar desequilíbrio e exigir balanceamento. Rotações simples e duplas são necessárias durante esse balanceamento.

Após o balanceamento, pode haver desequilíbrio em níveis superiores. Assim, deve-ser analisar também esses níveis superiores. A figura ?? demonstra um exemplo de remoção com rotação simples a esquerda.

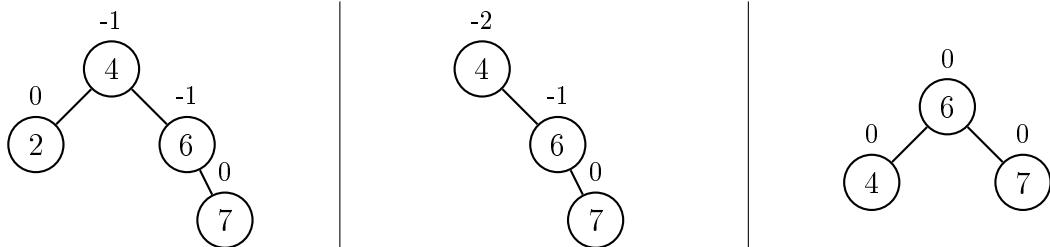


Figura 6.6: Exemplo de remoção em AVL (nó 2) com rotação simples a esquerda.

6.5.1.4 Análise

Entre as possíveis operações temos:

- **rotação** única custo $O(1)$ pois é constante.
- **busca** é $O(\log n)$ pois a altura de árvore é $O(\log n)$ e não necessita balanceamento.
- **inserção** é $O(\log n)$ com o custo da busca inicial de $O(\log n)$ e mais balanceamento para manter FP tem custo $O(\log n)$.
- **remoção** é $O(\log n)$ com a busca inicial de $O(\log n)$ mais balanceamento para manter FP tem custo $O(\log n)$.

6.5.2 Árvore rubro-negra

Uma árvore rubro-negra (*red-black tree*) é um tipo de árvore de pesquisa binária com uma propriedade adicional (1 bit) que representa a **cor** de um nó. Além disso, as folhas não armazenam dados sendo denotados por NIL (nulo). A figura ?? demonstra um exemplo de árvore rubro-negra.

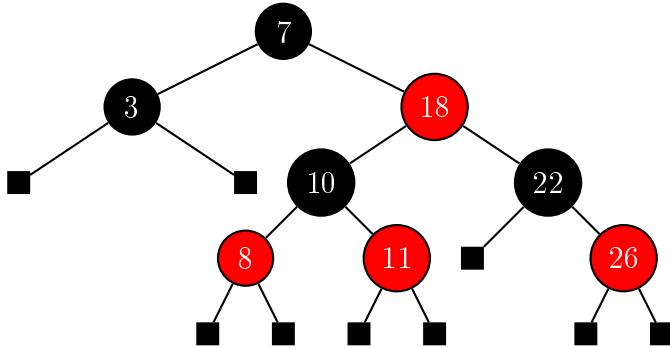


Figura 6.7: Exemplo de árvore rubro-negra.

As propriedades de uma rubro-negra são:

1. Cada nó é vermelho ou preto.
2. A raiz e todas as folhas (NIL) são pretos.
3. Se um nó é vermelho, então seu antecessor (nó pai) é preto.
4. Todos os caminhos simples de qualquer nó x para uma folha descendente tem o mesmo número de nós pretos.

Teorema 1 Uma árvore rubro-negra com n elementos tem altura

$$h \leq 2 \log(n + 1)$$

6.5.2.1 Rotações

Inserções e remoções custam $O(\log n)$ e, como podem modificar a árvore, o resultado pode violar as propriedades de uma rubro-negra. A rotação muda cores de nós e a estrutura da árvore. Há dois tipos de rotação em árvores rubro-negras: rotações a esquerda e rotações a direita. As rotações são explicadas na inserção.

6.5.2.2 Inserção

A inserção adiciona um nó x de cor vermelha na árvore da mesma forma que em uma árvore binária de busca. Nesse caso, apenas a propriedade 3 de rubro-negras pode ser violada. Caso a propriedade 3 foi violada, mover para o nó pai por recoloração até que as propriedades sejam respeitadas através de rotações e recolorações.

Os passos da inserção envolvem 3 casos. Nos exemplos abaixo, cada quadrado azul representa uma sub-árvore com uma raiz preta, e todas tem a mesma altura em nós pretos.

1. insere na árvore o nó x com cor vermelha.

2. se propriedade 3 foi violada:

- (a) **Caso 1** - ocorre quando o nó pai de x e z são vermelhos (figura ??). Consiste em recolorir o nó pai de x e tios (nós vizinhos do pai de x , ou filhos do avô de x). O novo x será o nó avô do antigo x .
- (b) **Caso 2** - quando o pai de x é vermelho e z é preto (figura ??). É feita uma rotação para esquerda, e automaticamente cai no **Caso 3**.
- (c) **Caso 3** - novamente quando o pai de x é vermelho e z é preto (figura ??). É feita uma rotação para direita, ainda recolorir pai de x e avô de x .

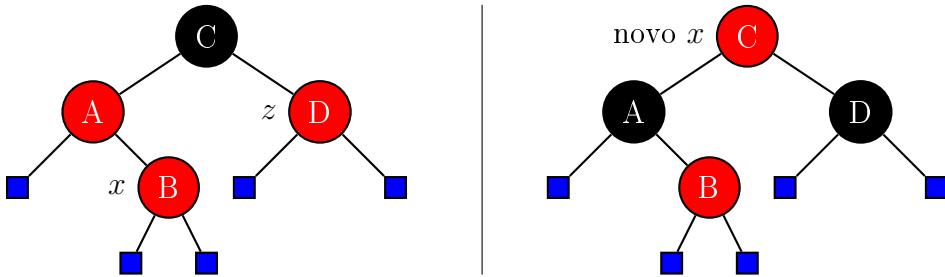


Figura 6.8: Caso 1 da inserção em árvore rubro-negra: recoloração

6.5.2.3 Análise

Como a altura de uma rubro-negra de n nós é $O(\log n)$, o custo da inserção é $O(\log n)$. Esse custo inclui o custo de inserção $O(\log n)$, como em uma árvore binária de busca, e rotações para manter a árvore de no máximo $O(\log n)$. Não obstante, ele não executa mais do que 2 rotações porque sempre executa o caso 1 e em seguida caso 2 ou caso 3.

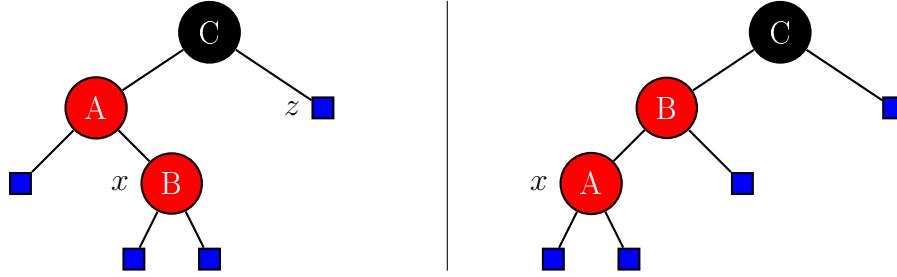


Figura 6.9: Caso 2 da inserção em árvore rubro-negra: rotação para esquerda.

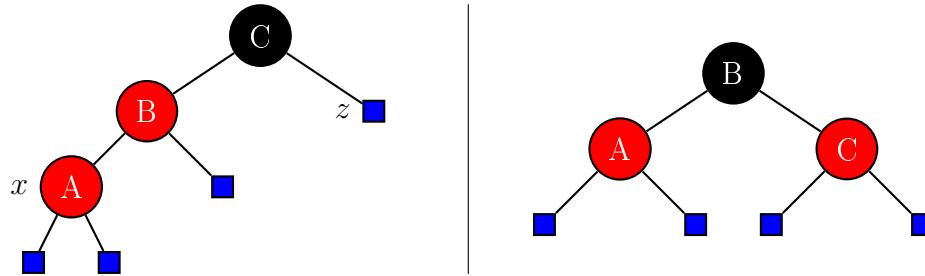


Figura 6.10: Caso 3 da inserção em árvore rubro-negra: rotação para direita.

6.6 Tabelas HASH

Em uma tabela HASH os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa. A operação é chamada transformação de chave ou *hashing*.

A pesquisa com transformação de chave é composta de duas etapas:

1. Computar o valor a **função de transformação**, que transforma a chave de pesquisa em um endereço da tabela.
2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método lidar com **colisões**.

Algumas **colisões** irão ocorrer qualquer que seja a função de transformação, e tais colisões têm de ser resolvidas. Mesmo com alguma função de transformação que distribua registros de forma uniforme, existe uma alta probabilidade de haver colisões.

6.6.1 Funções de hash

Uma função de hash h mapeia um universo U de todas as chaves em *slots* de uma tabela hash $T[0..m - 1]$ para $S = \{0, 1, \dots, m - 1\}$ onde m é muito menor que o universo U . O valor da chave $h(k)$ é o **valor hash** da chave k . A função hash reduz o intervalo de índices e tamanho de um vetor. Ao invés de $|U|$, o vetor (tabela T) tem tamanho m sendo $|U| > m$.

Um método simples de *hashing* é o resto da divisão por m

$$h(k) = k \bmod m$$

onde k é a chave como um inteiro.

Cuidado na escolha do valor de m . m deve ser um número primo não muito próximo do exponencial de 2 ou 10 e não usado com frequência.

Existem diversos métodos para armazenar n registros em uma tabela de tamanho $m > n$ em que utilizam lugares vazios na própria tabela para resolver colisões.

6.6.2 Encadeamento

No caso de colisões, uma solução simples é construir uma lista encadeada para cada endereço da tabela. Dessa forma, todas as chaves com o mesmo endereço são encadeadas em uma lista. O **fator de balanceamento** de T é $\alpha = n/m$ onde n é o número de chaves na tabela e m o tamanho da tabela. A figura ?? ilustra um exemplo de tabela hash com alguns slots com colisões.

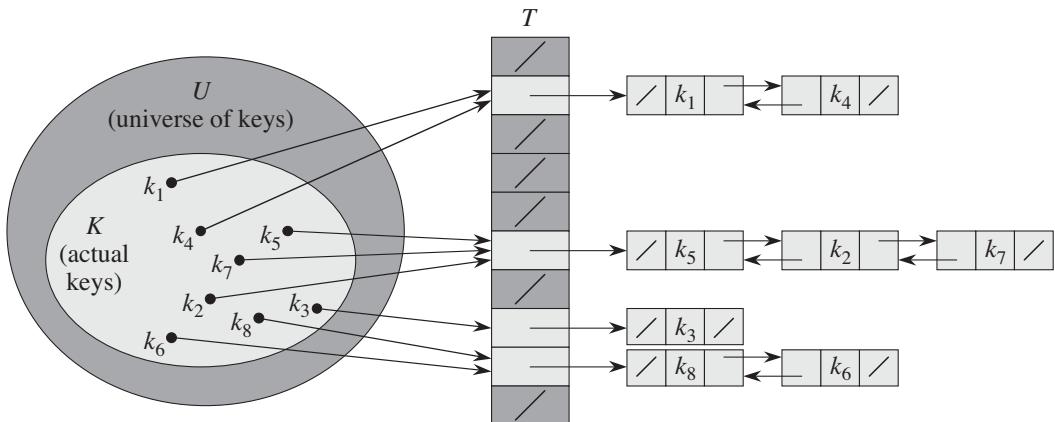


Figura 6.11: Exemplo de uma tabela hash com colisões.

O custo de busca para uma chave é $O(1 + \alpha)$ onde custo 1 é aplicar a função hash e acessar o slot, e α a busca na lista. Então, o custo é $O(1)$ se $\alpha = O(1)$.

6.6.3 Endereçamento aberto

No **endereçamento aberto** todas as chaves são armazenadas na própria tabela, sem uso de memória adicional. Se o endereço e estiver ocupado tenta no endereço $e+1, e+2, \dots, n, 1, 2, \dots, e-1$.

Entre as várias propostas, a mais simples é *hashing linear* dada por

$$k(k, i) = (h'(k) + i) \bmod m.$$

O hashing linear sofre do **agrupamento** (*clustering*) na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas. Mesmo que o hashing linear ter esse problema, os resultados são satisfatórios. O melhor caso, assim como o caso médio é $O(1)$. O pior caso é $O(n)$.

6.6.4 Chaves não numéricas

As chaves não numéricas devem ser transformadas em números dado por

$$k = \sum_{i=1}^n chave[i] \times p[i]$$

onde n é o número de caracteres da chave, $chave[i]$ é o código ASCII do i -ésimo caractere da chave, e $p[i]$ é um inteiro de um conjunto de pesos gerados aleatoriamente para $1 \leq i \leq n$.

O uso de pesos tem a vantagem de gerar duas funções $h_1(k)$ e $h_2(k)$ diferentes para dois conjuntos diferentes $p_1[i]$ e $p_2[i]$ $1 \leq i \leq n$.

6.6.5 Hashing perfeito

Se $h(x_i) = h(x_j)$ se e somente se $i = j$, então não há colisões e a função de transformação é chamada de **função de transformação perfeita**. Se o número de chaves n e o tamanho da tabela m são iguais, então temos uma **função de transformação perfeita mínima (hp)**.

Se $x_i \leq x_j$ e $hp(x_i) \leq hp(x_j)$ então a ordem lexicográfica é preservada. Nesse caso, temos uma **função de transformação perfeita mínima com ordem preservada**.

6.7 Busca digital

A busca digital, ou pesquisa digital, é baseada na representação das chaves como uma sequência de caracteres ou dígitos. Em vez de comparar uma chave de busca com os elementos, os caracteres da chave de busca são comparados. O método de busca digital é análogo à pesquisa manual em dicionários.

Os métodos de busca digital são vantajosos quando as chaves são grandes e de **tamanho variável**. Um aspecto interessante quanto aos métodos de pesquisa digital é a possibilidade de localizar todas as ocorrências de uma determinada cadeia em um texto com tempo de resposta logarítmico em relação ao tamanho do texto.

Os métodos estudados são Trie e Patricia.

6.7.1 Árvores Trie

O nome **Trie** vem de *Retrieval* (recuperação). Uma trie é uma árvore de busca n -ária, o grau equivale ao tamanho do alfabeto, e ordenada, cujos nós irmãos são ordenados. Usada para indexar chaves de busca, em geral caracteres, os arcos os arcos são caracteres da chave.

Considerando as chaves como sequência de bits (quando $n = 2$), o algoritmo de pesquisa digital é semelhante ao de pesquisa em árvore, exceto que em vez de caminhar na árvore de acordo com o resultado da comparação entre chaves, caminha-se de acordo com os bits da chave.

A figura ?? ilustra um exemplo de árvore Trie com palavras indexadas Alma, Almir, Alto, Be, Cal.

6.7.1.1 Chaves

Uma chave é uma palavra sobre um alfabeto. Um símbolo é usado para indicar o término de uma chave. Esse símbolo deve ser um caractere que não faça parte do alfabeto, e garante que uma chave não seja um prefixo de outra. Dessa forma, os registros estarão nos nós folha da árvore.

Exemplo 1 tendo o alfabeto {A, B, C, ..., Z} e o símbolo de término \$. Possíveis chaves: BOTA\$, CARRO\$, LEI\$.

Exemplo 2 tendo o alfabeto {0, 1, ..., 9} e o símbolo de término \$. Possíveis chaves: 134\$, 12978\$, 99777\$.

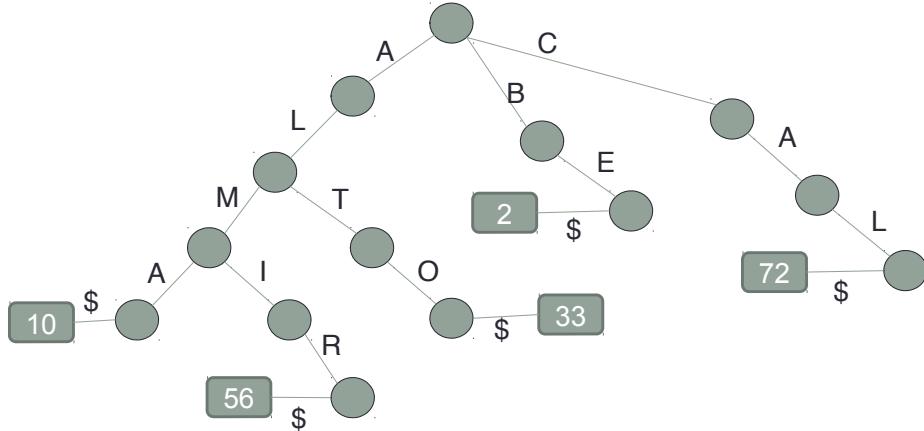


Figura 6.12: Exemplo de uma árvore Trie.

O caminho da raiz até um nó qualquer forma de um prefixo de uma chave indexada.

O formato das tries não depende da ordem em que as chaves são inseridas, depende apenas dos valores das chaves.

6.7.1.2 Estrutura de uma Trie

A estrutura é composta de:

Nó raiz vazio e indexa nós que representem prefixos das chaves indexadas.

Nós intermediários prefixos de chaves de busca indexadas. Indexam outros nós que aumentam o prefixo.

Nós folha chave completa e indexam as regiões de interesse. Ex.: offset da chave de busca dentro de um arquivo de texto.

6.7.1.3 Busca

A busca começa pela raiz e pelo primeiro caractere. Há dois cenários:

Cenário 1 o nó atual não é folha.

1. Se existe um filho que corresponda ao próximo caractere da chave.
 - Avançar um caractere (passar a ignorá-lo a partir deste momento).
 - Visitar o filho.
2. Se não existe esse filho.
 - A chave não está indexada.

Cenário 2 o nó atual é folha.

1. Esse nó de dados representa a resposta.

6.7.1.4 Inserção

Os passos da inserção são:

1. A palavra é buscada.
2. Se encontrar a palavra.
 - (a) Adiciona-se um ponteiro no nó correspondente. O nó vira uma lista invertida contendo os offsets.
3. Caso contrário
 - (a) Localizar o último nó visitado.
 - (b) a partir desse nó, inserir o restante dos caracteres em novos nós.

6.7.1.5 Implementação

Uma Trie pode ser implementada de duas formas diferentes:

- **R-Way** com nós de desvio e nós de informação. Os nós de desvio contêm um filho para cada símbolo do alfabeto mais um filho para o símbolo especial.
- **árvore TST (Ternary Search Tree)** ocupam menos espaço. Cada nó possui um caractere e três ponteiros: próximo caractere (centro), caractere alternativo menor que o atual (esquerda), e caractere alternativo maior que o atual (direita).

6.7.1.6 Análise

O custo em execução é $O(km)$ para k o tamanho do alfabeto e m o tamanho da chave de busca. O tempo de execução é independente do número de chaves, e a altura é igual ao comprimento da chave mais longa.

O custo em espaço é $O(s)$ para s a soma do tamanho de todas as chaves.

6.7.1.7 Trie compacta

O espaço de uma Trie é reduzido por meio de compressão de nós redundantes, ou seja, mesclando nós que possuem apenas um caminho possível. A figura ?? ilustra a versão compacta da árvore Trie da figura ??.

O custo em espaço de uma Trie compacta é $O(n)$ para n o número de chaves, bem menor que $O(s)$ da árvore não compacta. Todavia, o tempo de execução é o mesmo sendo $O(km)$ para k o tamanho do alfabeto e m o tamanho da chave de busca.

6.7.1.8 Trie binária

Uma Trie binária é um tipo especial de Trie em que o alfabeto possui apenas dois símbolos: $\{0, 1\}$. Para compensar a ausência de símbolo especial, todas as cadeias de bits devem ter o mesmo tamanho. Caso alguma chave não tenha tamanho suficiente, basta preencher com 0s à esquerda.

A figura ?? ilustra um exemplo de árvore de árvore Trie binária normal (esquerda) e compacta (direita) para os seguintes elementos:

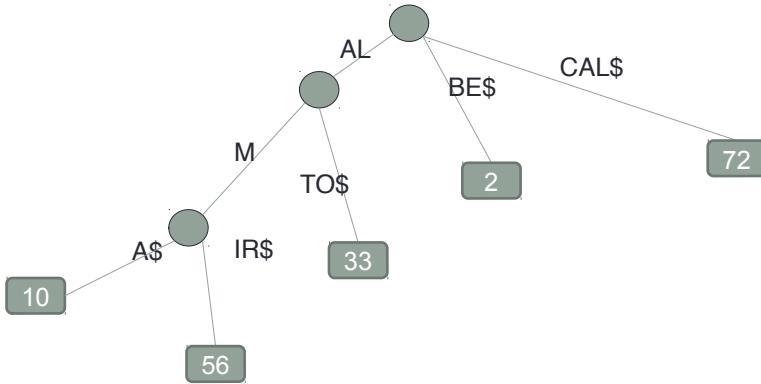


Figura 6.13: Exemplo de uma árvore Trie compacta.

- 0 - chave 000 valor 4.
- 1 - chave 001 valor 22.
- 4 - chave 100 valor 14.
- 5 - chave 101 valor 39.

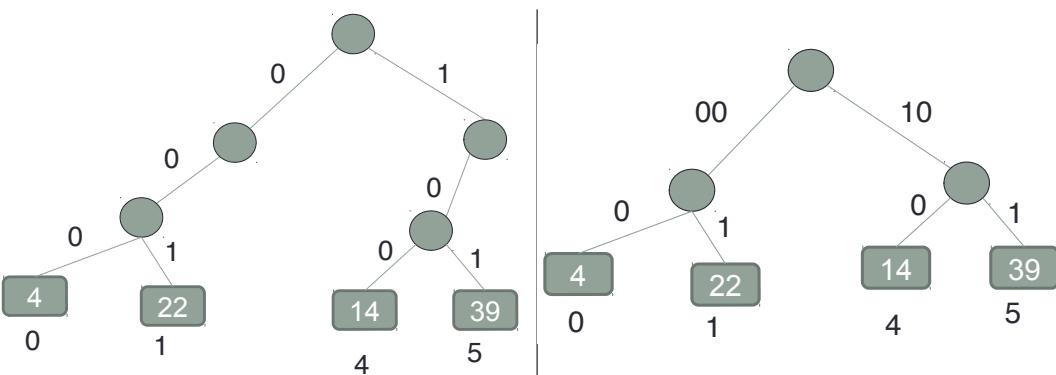


Figura 6.14: Exemplo de uma árvore Trie binária (esquerda) e binária compacta (direita).

No pior caso, há b comparações de bits, o que pode ser ineficiente quando as chaves são grandes. A versão compacta pode reduzir o custo, mas não o suficiente.

6.7.2 Árvores Patricia

Uma árvore Patricia, ou *radix tree*, ou *patricia trie*, vem do acrônimo “*Practical Algorithm To Retrieve Information Coded In Alphanumeric*” e é uma árvore de busca compacta semelhante a uma Trie binária, com alfabeto $\{0, 1\}$. Entretanto, ela não usa um caractere especial para término ($\$$). Todas as cadeias devem ter o mesmo tamanho.

Cada nó possui no máximo 2 filhos, e cada nó tem dois tipos: nós de desvio e nós de conteúdo. Os nós de desvio possuem uma informação numérica que indica o bit da chave a testar. A partir desse bit é decidido qual caminho da árvore a seguir.

A figura ?? mostra um exemplo de árvore Patricia e as respectivas chaves. Os valores dentro de cada nó indicam qual bit a ser testado no próximo desvio.

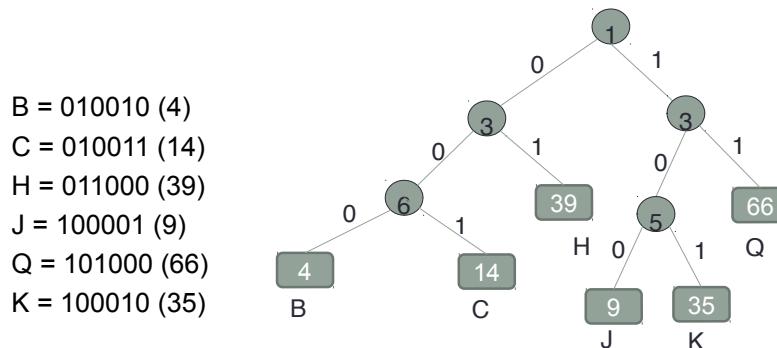


Figura 6.15: Exemplo de uma árvore Patricia.

6.7.2.1 Busca

A busca começa pela raiz e pelo primeiro caractere (bit 0). Há dois cenários:

Cenário 1 o nó atual não é folha.

1. Verificar o dígito da chave de busca equivalente ao valor do nó.
2. Se for 0, seguir a subárvore da esquerda.
3. Caso contrário, seguir a subárvore da direita.
4. Se não houver subárvore a seguir, a chave de busca não está indexada.

Cenário 2 o nó atual é folha.

1. Deve-se verificar se o valor indexado realmente corresponde a chave de busca.

6.7.2.2 Inserção

Os passos da inserção em árvores Patricia são os seguintes:

1. A palavra (cadeia de bits) é buscada.
2. Se a árvore estiver vazia:
 - (a) Cria-se uma estrutura inicial para indexar a palavra.
 - (b) Um nó raiz, com valor igual a 1, e um nó filho (a esquerda ou direita), indexando a palavra.
3. Se a busca parar em um nó anterior a um nó folha.
 - (a) Significa que o nó só possui um único filho.

- (b) Cria-se o outro filho, que indexará a palavra inserida.
4. Se a busca parar em um nó folha.
- o conteúdo indexado é comparado com a palavra a ser inserida.
 - se for igual, é adicionada uma entrada no nó folha para a palavra indexada.
 - Se for diferente.
 - é preciso descobrir o nó da árvore a ser atualizado.
 - para isso, é verificada a posição P do primeiro bit onde ocorre a diferença.
 - Se o bit for **maior** que a última posição utilizada, deve-se adicionar um nó aqui.
 - Se o bit for **menor** que a última posição utilizada, deve-ser subir na árvore para encontrar o local adequado.
5. Para o caso de subida na árvore.
- Sobe-se na árvore até encontrar o nó X , o nó mais baixo cujo valor seja maior que a posição P .
 - Nesse ponto é inserido um novo nó, tendo como filhos:
 - O nó X .
 - Um novo nó para indexar a nova palavra.
 - Os ponteiros e valores dos nós criados/remanejados são atualizados.

6.7.3 Aplicações de Tries e Patricia

As Tries de caracteres podem ser aplicadas em:

- Verificadores de ortografia.
- Motores de busca - nós folhas guardam páginas que possuem a palavra indexada.

As Tries binárias são usadas na compressão de dados, como na Codificação de Huffman.
As árvores Patricia podem ser usadas na indexação de chaves longas.

6.8 Exercícios

- Descreva os passos da busca binária e os custos para o melhor caso, caso médio e pior caso.
- Qual é a principal característica de uma árvore binária de pesquisa ?
- Descreva a árvore binária de busca criada a partir dos seguintes elementos: 5, 8, 3, 6, 7, 1, 9, 4.
- A partir da árvore do exercício anterior, descreva a árvore binária de busca resultante da remoção dos seguintes elementos: 4, 5.
- Desenhe a árvore AVL resultante da inserção dos números: 9, 8, 7, 6, 5, 4, 3, 2, 1.
- Escreva o algoritmo para imprimir o MENOR elemento de uma árvore de busca binária.

CAPÍTULO 6. PESQUISA EM MEMÓRIA PRIMÁRIA

7. Escreva o algoritmo para imprimir o MAIOR elemento de uma árvore de busca binária.
 8. Escreva dois algoritmos que recebem uma árvore AVL como entrada: para imprimir todos os elementos em ordem crescente e decrescente.
 9. Considere a seguinte lista de valores: [0, 9, 10, 3, 8, 4, 5, 1].
 - (a) Construa uma árvore binária de busca não balanceada.
 - (b) Construa uma árvore AVL.
 10. Para uma árvore binária de busca:
 - (a) Descreva um algoritmo **não recursivo** que faz um percurso em-ordem.
 - (b) Descreva algoritmos recursivos para percursos pré-ordem e pós-ordem.
 - (c) Descreva algoritmos recursivos para buscar o elemento mínimo (`ArvoreMinimo`) e elemento máximo (`ArvoreMaximo`).
 11. Supondo que temos números de 1 a 1000 em uma árvore binária de busca, e queremos buscar o número 363. Quais das sequências abaixo **não** poderiam ser sequências de nós percorridos nessa busca ?
 - (a) 2, 252, 401, 398, 330, 344, 397, 363.
 - (b) 924, 220, 911, 244, 898, 258, 362, 363.
 - (c) 925, 202, 911, 240, 912, 245, 363.
 - (d) 2, 399, 387, 219, 266, 382, 381, 278, 363.
 - (e) 935, 278, 347, 621, 299, 392, 358, 363.
 12. Desenhe uma árvore rubro-negra com chaves 1, 2, ..., 15. Adicione os nós folhas com NIL e faça coloração dos nós de forma que o $bh(x)$ (*black-height* ou altura-preto) da árvore seja 2, 3 e 4.
 13. Mostre a árvore rubro-negra resultante depois de inserir sucessivamente os nós 41, 38, 31, 12, 19, 8 em uma árvore vazia.
 14. Agora, baseado no exercício anterior, mostre a árvore rubro-negra resultante da remoção dos nós 8, 12, 19, 31, 38, 41 sucessivamente.
 15. Para as palavras chave da seguinte frase “Marcos Lima marcou o marco”, e considerando o alfabeto {a, b, ..., z}, crie:
 - (a) uma Trie compacta em formato de árvore.
 - (b) uma Trie normal, em formato R-Way.
 - (c) uma Trie normal, em formato TST.
- Obs: artigos não precisam ser indexados.
16. Dadas as seguintes cadeias de bits: 00101 valor 5, 11000 valor 15, 01010 valor 34, 10101 valor 23, 11111 valor 12, 01110 valor 77. Crie:

- (a) uma Trie binária compacta.
- (b) uma árvore Patricia.

7 Pesquisa em memória secundária

Este capítulo aborda questões relacionadas a técnicas de armazenamento e recuperação de dados em memória secundária.

7.1 Memória secundária

A classificação de dispositivos físicos pode ser feita ao considerar:

1. Desempenho no acesso aos dados.
2. Custo por unidade de dados.
3. Confiabilidade para casos de perda de dados em falhas de energia ou falha do sistema, ou mesmo falha física do dispositivo.

Também pode-se diferenciar armazenamento em:

Volátil perde dados quando é desligado.

Não-volátil persistente mesmo quando é desligado. Inclui armazenamento secundário e terciário.

7.1.1 Mídias de armazenamento

As médias de armazenamento disponíveis em geral são:

Cache a mais rápida e custosa, volátil, e gerenciada pelo hardware do processador.

Memória principal acesso rápido da ordem de 10 a 100 nanosegundos ($1\text{ nanosegundo} = 10^{-9}$).

Em geral muito pequena (ou muito cara) para armazenar todos os dados de um banco de dados. Hoje em dia, em alguns casos, armazena todo o banco de dados (*memcached*). Ela é volátil, sendo que os dados são perdidos na falta de energia ou falha do sistema.

Flash dados sobrevivem em falhas de energia. Dados podem ser escritos em um local uma única vez, porém podem ser escritos novamente. Pode ser lida um número ilimitado de vezes, mas há um limite no número de gravações que varia entre 10K e 1M. As leituras podem ser mais rápidas que a memória principal, mas as escritas são lentas. Muito usada em dispositivos embarcados como câmeras digitais, telefones, e memórias flash drive (pendrive).

Magnética dados são armazenados em uma mídia magnética com discos de movimento giratório.

Deve-se ler os dados do disco para a memória principal, e depois escrever de volta ao disco. O acesso direto é possível em qualquer ordem, ao contrário de mídias de fita.

Ótica não volátil e os dados são lidos óticamente de um disco giratório através de um laser. Os mais populares são CD-ROM (640 MB), DVD (4.7GB), e Blu-ray (27 GB a 54 GB). As leituras e escritas são mais lentas comparadas a discos magnéticos.

Fita magnética não volátil, usado primariamente para backup (recuperar falhas de disco), e arquivamento de dados. O acesso é sequencial, muito mais lento que discos. Porém, a capacidade de armazenamento é alta (40 a 300GB).

A figura ?? ilustra a hierarquia de armazenamento de acordo com o desempenho e custo. Os níveis mais altos são mais custosos, porém mais rápidos. No sentido contrário, o custo por bit reduz, mas o tempo de acesso aumenta.

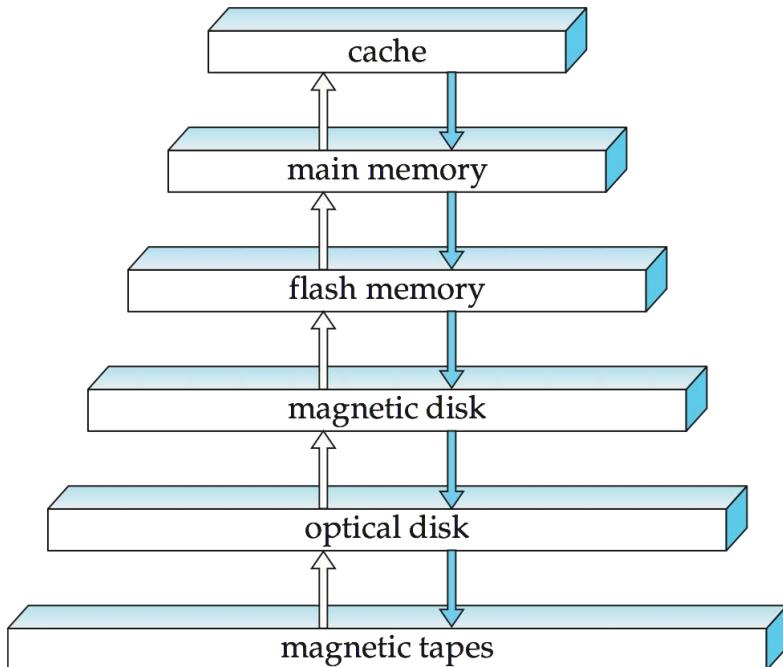


Figura 7.1: Hierarquia de mídias de armazenamento.

Nessa hierarquia, podemos agrupar as mídias em:

Armazenamento primário mais rápido mas volátil. Ex: cache, memória principal.

Armazenamento secundário próximo nível na hierarquia, não volátil, acesso moderadamente rápido. Ex: memória flash, discos magnéticos.

Armazenamento terciário nível mais baixo, não volátil, tempo de acesso lento. Também chamado armazenamento *off-line*. Ex: fita magnética, armazenamento ótico.

7.1.2 Disco magnético

7.1.2.1 Estrutura física

Fisicamente, discos magnéticos são simples. A figura ?? mostra o funcionamento interno de um disco.

Cada prato do disco tem duas superfícies com material magnético, onde a informação é gravada. O cabeçote de escrita-leitura (*read-write head*) fica posicionada bem próxima a superfície do prato, e lê e escreve dados codificados magneticamente. A superfície é dividida logicamente em **trilhas**, que são subdivididas em **setores**. Um setor é a menor unidade que pode ser lida ou escrita do

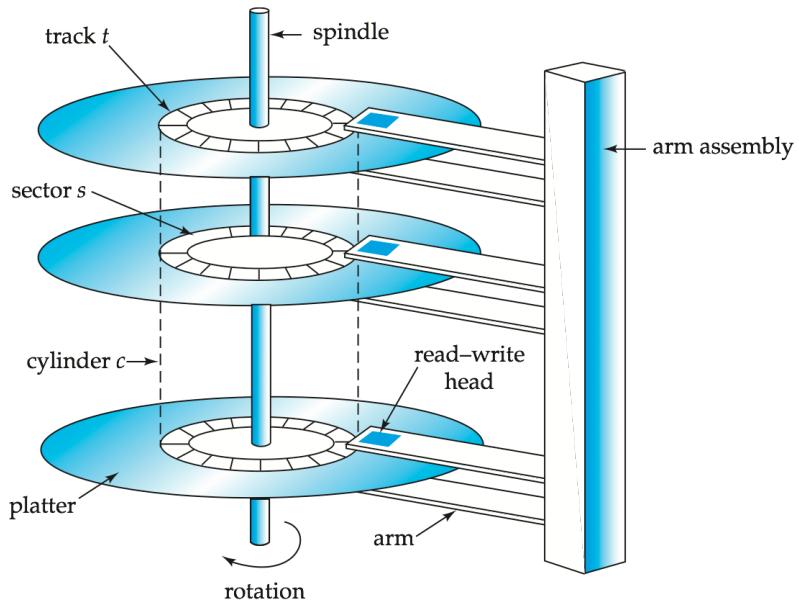


Figura 7.2: Mecanismo de cabeças de leitura em um disco magnético.

disco. Em geral, um setor tem 512 bytes; há perto de 50K a 100K trilhas por prato, e 1 a 5 pratos por disco.

Os pratos são montados em braços de disco. Para ler e escrever um setor, o braço do disco se move para posicionar o cabeçote na trilha correta. O prato gira, e os dados são lidos e/ou escritos a medida que o setor passa pelo cabeçote.

Outro componente de discos é o **controlador de disco** que faz a interface entre o computador e o hardware do disco. Ele aceita comandos de alto nível para ler e escrever setores. Entre outras funções, garante checagem de erros.

7.1.2.2 Medidas desempenho

O **tempo de acesso** é o tempo que leva desde a solicitação de leitura/escrita até o momento em que a transferência começa de fato. O tempo de acesso consiste em:

Tempo de busca tempo que leva para reposicionar o braço na trilha correta. O tempo médio de busca é 1/2 do pior tempo de busca, em geral 4 a 10 milisegundos.

Latência rotacional tempo que leva para o setor a ser acessado aparecer debaixo do cabeçote.

Latência média é 1/2 do pior caso, em geral 4 a 11 milisegundos em discos (5400 a 15000 rpm).

A **taxa de transferência** é a taxa com que os dados são recuperados ou armazenados no disco. A taxa máxima é de 25 a 100 MB por segundo, e essa taxa é menor para trilhas internas.

O *Mean time to failure* (MTTF) é o tempo médio de funcionamento de um disco sem falhas. Esse tempo varia entre 3 a 5 anos. A probabilidade de falhas em disco novos é baixa, que corresponde a um “MTTF teórico” de 500K a 1,2M horas. Ou seja, um MTTF de 1,2M horas para um disco novo significa que dados 1000 novos discos, na média um irá falhar a cada 1200 horas.

7.1.2.3 Otimização de acessos

A transferência de dados entre disco e memória ocorre em unidades de **blocos**, que são uma sequência continua de setores em uma única trilha. O tamanho varia de 512 a vários kilobytes. Blocos pequenos podem implicar em várias transferencias do disco, enquanto que grandes blocos podem desperdiçar espaços devido a blocos parcialmente ocupados. Um tamanho típico utilizado é de 4 a 16 kilobytes.

A otimização de acessos envolve **algoritmos de escalonamento** que são usados para ordenar o acesso às trilhas para minimizar o movimento do braço. O **algoritmo do elevador** funciona da seguinte forma:

1. move o braço do disco em uma direção (de dentro para fora ou de fora para dentro).
2. processa todas as requisições pendentes nessa direção.
3. reverte o movimento e repete a varredura.

Outra forma de otimização é a **organização de arquivos** que otimiza o acesso ao organizar os blocos, de modo a corresponder à forma com que os dados são acessados. Exemplo é armazenar informações relacionadas em regiões próximas. Arquivos podem se **fragmentar** com o passar do tempo. Fragmentação acontece quando o arquivo é modificado, ou quando os blocos livres estão espalhados, e novos arquivos tiverem que ocupar esses blocos. O acesso sequencial a um arquivo fragmentado leva a maior movimentação do braço do disco. Alguns sistemas possuem utilitários para **desfragmentar** o sistema de arquivos, de modo a acelerar o acesso.

7.2 Organização de arquivos

A pesquisa em memória secundária normalmente está associada a **pesquisa em bancos de dados**. O banco de dados é armazenado como uma coleção de arquivos. Cada arquivo é uma sequência de registros, e cada registro é uma sequência de campos.

7.2.1 Registros de tamanho fixo

A estratégia mais simples é assumir **registros de tamanho fixo**. Cada tabela possui o seu próprio arquivo, e cada arquivo possui registros de um tipo específico de dado. O cabeçalho possui informações de controle. Cada registro tem um ponteiro especial, que aponta para o seu próximo registro. O problema com essa abordagem é a ocupação parcial do bloco. Por exemplo, cada registro ocupa 300 bytes e cada bloco ocupa 1024 bytes. O cabeçalho ocupa 20 bytes. Qual o mínimo de espaço que será desperdiçado ?

Uma solução é permitir que registros cruzem a fronteira de um bloco. As alternativas para remoção do registro i são:

1. mover registros $i + 1, \dots, n$ para $i, \dots, n - 1$.
2. mover registro n para i .
3. não mover nada, ligar os registros vazios por uma *free list*.

Com *free lists*, pode-se usar um ponteiro especial para guardar os registros livres. Ela é uma representação mais eficiente que reusa o espaço destinado aos atributos dos registros para guardar os ponteiros. O cabeçalho possui informações de controle como ponteiro para o primeiro registro com dados do bloco, e ponteiro para o primeiro registro livre do bloco.

O principal problema dessa abordagem é o desperdício de espaço dentro do bloco.

7.2.2 Registros de tamanho variável

A estrutura de **slotted-page** é comumente utilizada para organizar registros de tamanho variável dentro de um bloco. A figura ?? ilustra a estrutura. O cabeçalho contém:

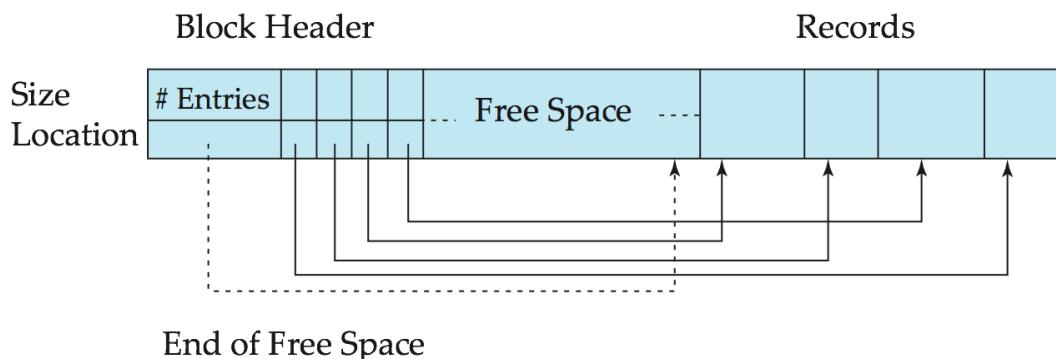


Figura 7.3: Estrutura do *slotted-page*.

- número de registros.
- localização e tamanho de cada registro.
- fim do espaço livre no bloco.

Os registros podem ser movidos dentro de uma página para evitar espaços vazios entre eles; as entradas no cabeçalho precisam ser atualizadas. Ponteiros não apontam diretamente para o registro, mas apontam para o ponteiro de entrada do registro no cabeçalho.

Registros de tamanho variável aparecem em bancos de dados de diversas formas: armazenamento de múltiplos tipos de conteúdo em um mesmo arquivo; armazenamento de atributos que possuem tamanho variável.

Se as slotted-pages são do tamanho de um bloco, o problema dos registros que cruzam é eliminado. Isso limita o tamanho dos registros em um banco de dados, o que é o caso padrão.

7.3 Organização de registros em arquivos

Até agora, vimos como representar registros em uma estrutura de arquivo. Há várias formas de organizar registros em arquivos:

Heap um registro pode ser posicionado em qualquer lugar onde haja espaço.

Sequencial registros armazenados em ordem sequencial, com base no valor da chave de busca de cada registro.

Hashing é uma função hash calculada com base em alguns atributos do registro. O resultado especifica em que bloco do arquivo do registro deve ser posicionado.

7.3.1 Organização de arquivos sequencial

Adequada para aplicações que executam processamento sequencial sobre todo o arquivo. Os registros do arquivo são ordenados por uma **chave de busca**.

A operação de remoção usa *free lists* para registros de tamanho fixo.

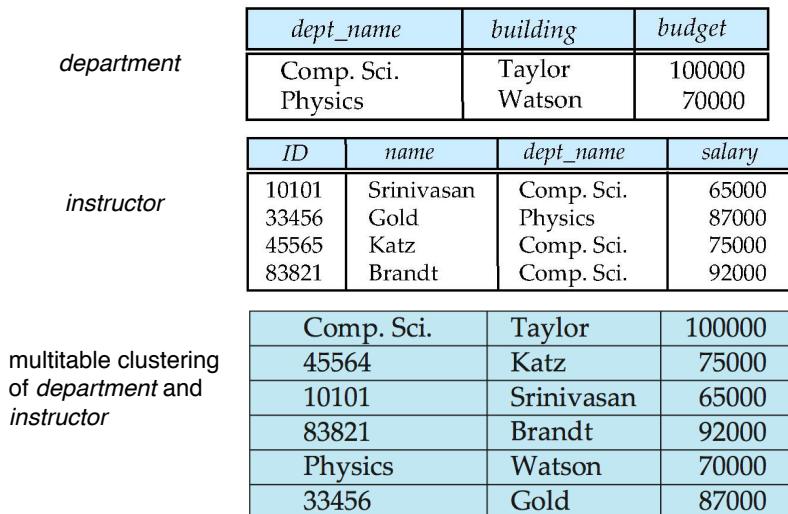
A inserção localiza a posição onde o registro deve ser inserido e:

1. se houver espaço no bloco, insere no bloco.
2. se for o último registro, insere em um novo bloco.
3. se houver espaço em um bloco vizinho, distribuir. Caso contrário, insere em um novo **bloco de estouro** (*overflow block*).
4. de qualquer forma, as cadeias de ponteiros precisam ser atualizadas.

O uso de blocos de estouro demanda a reorganização do arquivo de tempo em tempo para restaurar a ordem sequencial.

7.3.2 Clusterização multitable

Os registros de cada tabela podem ser armazenados em arquivos separados. Em uma **clusterização multitable**, registros de tabelas diferentes são armazenados no mesmo arquivo. A principal motivação para armazenar registros relacionados no mesmo bloco é minimizar a entrada e saída. A figura ?? ilustra a organização de um arquivo com clusterização. Esse arquivo armazena registros de uma ou mais relações em cada bloco. Tal organização permite a leitura de registros que podem satisfazer uma união por meio de uma leitura de bloco.



The diagram shows the organization of data from two tables, *department* and *instructor*, into a single clustered file. The *multitable clustering of department and instructor* is shown on the left, with arrows pointing to each table's data structure. The *department* table has columns *dept_name*, *building*, and *budget*. The *instructor* table has columns *ID*, *name*, *dept_name*, and *salary*. The resulting clustered file contains six blocks. The first block contains the first two rows of the *department* table. The second block contains the last two rows of the *department* table and the first two rows of the *instructor* table. The third block contains the remaining four rows of the *instructor* table. The fourth block contains the last row of the *instructor* table and the first row of the *department* table.

	<i>dept_name</i>	<i>building</i>	<i>budget</i>
<i>department</i>	Comp. Sci.	Taylor	100000
	Physics	Watson	70000

<i>instructor</i>	<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	

multitable clustering of <i>department</i> and <i>instructor</i>	<i>Comp. Sci.</i>	<i>Taylor</i>	<i>100000</i>
	45564	Katz	75000
	10101	Srinivasan	65000
	83821	Brandt	92000
	Physics	Watson	70000
	33456	Gold	87000

Figura 7.4: Estrutura de arquivo de uma clusterização multitable.

Essa organização é boa para consultas com relação, e consultas envolvendo um único departamento e seus professores. Porém, ela é ruim para consultas somente de departamentos.

7.4 Sistemas de arquivos

Em organizações de arquivos sequencias, cada tabela é armazenada em um arquivo. O armazenamento poderia ser feito por meio do sistema de arquivos do sistema operacional (SO). A clusterização multitable pode trazer ganhos significativos em eficiência, mas essa organização não é compatível com o sistema de arquivos de sistemas operacionais.

Muitos dos sistemas de bancos de dados (SGBDs) de grande escala não utilizam diretamente o SO. As tabelas são armazenadas em um único arquivo, e o SGBD gerencia os arquivos por conta própria. Isso exige a implementação de um sistema de arquivos dentro do SGBD.

7.4.1 Buffer manager

O número de transferências de blocos entre o disco e a memória pode ser reduzido mantendo o maior número possível de blocos na memória principal. Um **buffer** é a porção da memória principal disponível para armazenar cópias de blocos de disco. O **buffer manager** é responsável pela alocação de espaço do buffer na memória principal.

Programas utilizam o buffer manager quando precisam ler um bloco do disco:

1. se o bloco já estiver no buffer, o gerenciador retorna o endereço do bloco na memória principal.
2. se o bloco não estiver no buffer, o manager:
 - (a) aloca espaço no buffer para o bloco.
 - i. substituindo algum outro bloco, se necessário, para liberar espaço para o novo bloco.
 - ii. o bloco substituído é escrito de volta no disco somente se ele foi modificado desde a última vez que ele foi escrito/recuperado do disco.
 - (b) Lê o bloco do disco para o buffer e retorna o endereço do bloco na memória principal para o programa.

Sua função é similar ao gerenciamento de memória virtual dos sistemas operacionais. Porém, bancos de dados têm requisitos diferentes.

7.4.2 Políticas de substituição de blocos

A maioria dos sistemas operacionais usam a política de substituição do bloco menos recentemente usado ou *least recently used* (LRU). A ideia por trás do LRU é usar um padrão passado de referência a blocos como uma previsão de futuras referências, já que usualmente nada mais se conhece. Consultas possuem um padrão de acesso bem definido, com varreduras sequenciais. O sistema do BD pode usar informação contida na consulta para prever referências futuras.

O LRU pode ser uma estratégia ruim para certos tipos de acesso que envolvem varreduras repetidas. Por exemplo, na junção de 2 tabelas r e s por um laço aninhado. Uma estratégia mista é preferível, onde o otimizador de consultas pode oferecer dicas para substituição de blocos.

Outras técnicas do buffer manager são:

Pinned block bloco em memória que não tem permissão de ser escrito de volta no disco.

Estratégia Toss-immediate libera o espaço ocupado por um bloco assim que a última tupla do bloco é processada.

Estratégia MRU MRU significa mais recentemente usado (*most recently used*), onde o sistema marca o bloco sendo processado. Depois que a última tupla do bloco é processada, o bloco é desmarcado, e se torna o bloco mais recentemente usado.

Saída forçada o manager também suporta saída forçada de blocos para fins de recuperação.

7.5 Mecanismos de indexação

Muitas buscas referenciam apenas uma pequena porção dos registros de um arquivo. Seria ineficiente ler todos os registros para encontrar apenas uma tupla. Usa-se estruturas adicionais a fim de permitir esse tipo de acesso.

7.5.1 Introdução

Um índice de um arquivo funciona como o índice de um livro. Mecanismos de indexação são utilizados para acelerar o acesso aos dados. Uma **chave de busca** (*search key*) é/são o(s) atributo(s) usado(s) para localizar registros em um arquivo. Um **arquivo de índice** consiste em registros na forma *chave - ponteiro*. Arquivos de índice são muito menores que o arquivo original.

Há dois tipos básicos de índices:

Índices ordenados chaves de busca são armazenadas (encadeadas) em ordem.

Índices hash chaves de busca são distribuídas uniformemente em buckets usando uma função de hash.

As duas técnicas são estudadas. Nenhuma delas é considerada a melhor. Cada técnica é avaliada de acordo com os seguintes fatores:

Tipo de acesso eficiente como registros de um valor específico em um atributo. Esse fator exerce bastante influência na escolha do tipo de índice.

Tempo de acesso

Tempo de inserção

Tempo de remoção

Sobrecarga de espaço sendo o espaço adicional para o índice.

7.5.2 Índices ordenados

Em um **índice ordenado**, as entradas são ordenadas de acordo com o valor da chave de busca, como por exemplo pelo nome do autor em um catálogo. Podem ser classificados quanto:

- A chave de busca indexada.

- índices primários.

- índices secundários.
- Aos registros indexados.
 - índices densos.
 - índices esparsos.

O **índice primário**, em um arquivo sequencialmente ordenado, é o índice cuja chave de busca é usada para ordenar o arquivo. É também chamado de **índice clusterizado** (*clustering index*). A chave de busca de um índice primário é quase sempre a chave primária.

O **índice secundário** é um índice cuja chave de busca não está armazenada de acordo com a ordem sequencial do arquivo. Também chamado de **índice não clusterizado** (*non-clustering index*).

7.5.3 Índices densos e esparsos

Uma entrada no índice consiste em uma chave de busca e ponteiros para um ou mais registros. O ponteiro para um registro consiste no identificador de um bloco do disco e um deslocamento dentro do bloco para identificar o registro. Há dois tipos de índices: denso e esparso.

Índices esparsos contém entradas no índice para apenas alguns valores de chave de busca. Para localizar um registro com uma chave de busca k , deve-se:

1. encontrar a entrada com o maior valor de chave de busca $< k$.
2. varrer o arquivo sequencialmente a partir do resgistro apontado por essa entrada.

Índices esparsos são somente aplicáveis quando os registros estão ordenados pela chave de busca do índice.

Em **índices densos** registros do índice aparecem para todos os valores de chave de busca do arquivo. Índices primários podem ser densos ou esparsos, mas índices secundários precisam ser densos.

A figura ?? ilustra um exemplo entre índices denso e esparso.

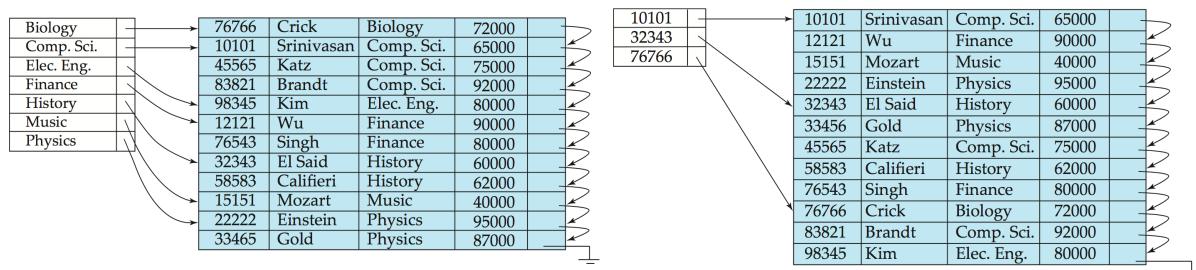


Figura 7.5: Exemplo de um índice denso (esquerda) e esparso (direita).

7.5.4 Índices multinível

Para casos em que o índice primário não cabe em memória, o acesso torna-se caro. Uma solução é tratar o índice primário em disco como um arquivo sequencial e construir um índice esparso sobre

ele. O **índice externo** é um índice esparso do índice primário, e o **índice interno** é o arquivo do índice primário.

Se mesmo o índice externo for muito grande para caber na memória, outro nível pode ser criado, e assim por diante. Note que índices em todos os níveis devem ser atualizados quando ocorrer atualizações no arquivo. A figura ?? mostra um simples exemplo de índice esparso construído de um índice primário.

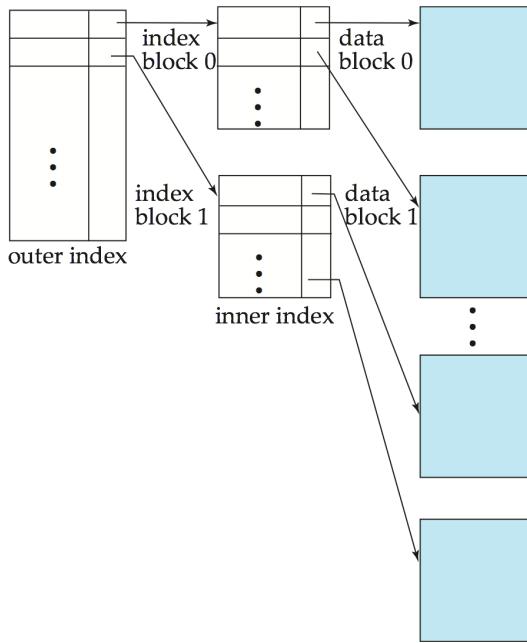


Figura 7.6: Índice esparso externo construído de um índice primário.

7.5.5 Atualização de índice

Em qualquer das formas de índice, cada índice precisa ser atualizado quando um registro é inserido ou removido.

Remoção - se o registro removido é o único registro no arquivo com uma chave de busca, a chave de busca também é removida do índice. Em índices densos, remoção de uma chave de busca é similar a remoção de um registro de arquivo. Em índices esparsos, se uma chave de busca existe no índice, ela é removida e substituída pela próxima chave de busca do arquivo. Se a próxima chave de busca já for indexada, a entrada no índice é simplesmente removida.

Inserção - em índices densos, realiza uma busca no índice usando a chave de busca do registro a ser inserido. Se a chave não existe, ela é inserida no índice. Em índices esparsos, se o índice mantém uma entrada para cada bloco do arquivo, nenhuma mudança é necessária, a não ser que um novo bloco seja criado (ou sofra distribuição). Se um novo bloco for criado (ou sofrer distribuição), a primeira chave de busca nova do bloco é inserida no índice.

Em índices multinível, as inserções e remoções são simples extensões dos algoritmos usados em índices de um nível só. As inserções e remoções são propagadas dos níveis internos para os externos.

As atualizações dos índices significam atualizações nos blocos físicos onde os índices estão armazenados. A organização física e lógica desses blocos é feita de forma similar aos blocos de um arquivo de dados.

7.5.6 Índices secundários

Em índices secundários, registros do índice apontam para o bucket que contem ponteiros para todos os registros com uma chave de busca. A figura ?? mostra um exemplo de índice secundário. Note que índices secundários tem de ser densos.

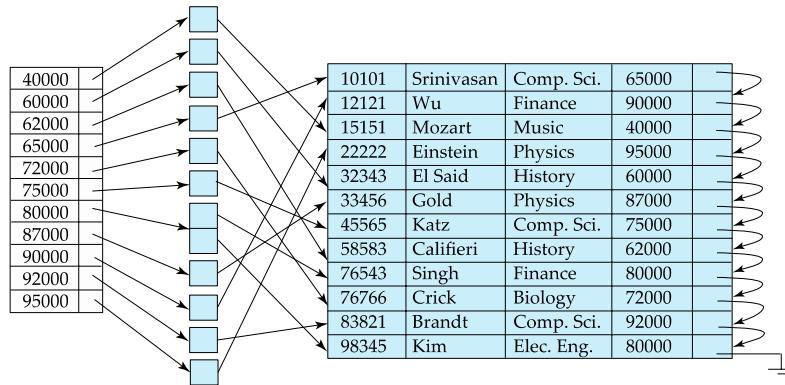


Figura 7.7: Exemplo de índice secundário sobre o campo *salary*.

Índices oferecem benefícios substanciais na busca por registros. Porém, atualizações de índices impõem sobrecustos em atualizações.

A varredura sequencial de um índice primário é eficiente, mas a varredura sequencial em índices secundários é cara pois é preciso uma transferência de bloco do disco para cada acesso ao registro. Uma transferência de bloco requer cerca de 5 a 10 milisegundos, contra 100 nanosegundos de um acesso à memória.

7.6 Índices em árvores B+

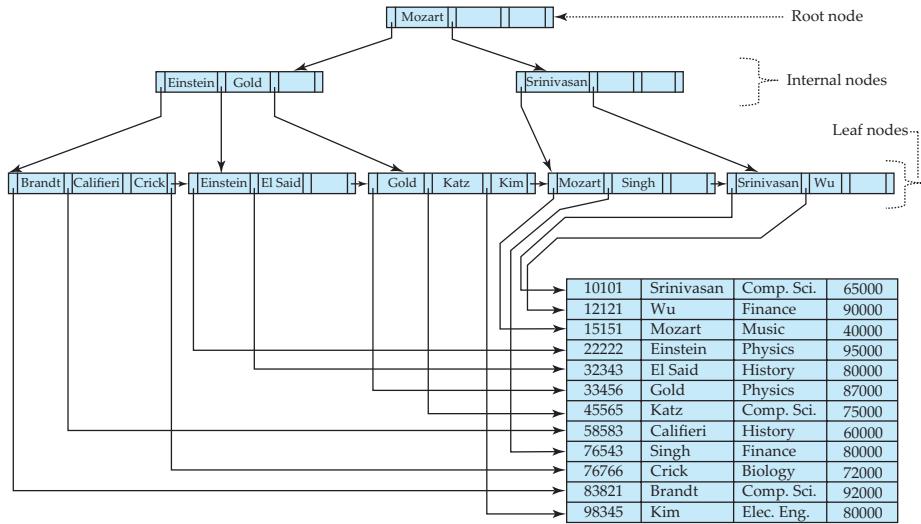
A principal desvantagem de índices sequenciais é a perda de desempenho a medida que o arquivo de índice cresce, tanto buscas e varreduras sequenciais. Reorganizações periódicas são necessárias, mas reorganizações frequentes são indesejadas. **Índices em árvores B+** é um tipo de árvore balanceada que mantém a eficiência mesmo com inserções e remoções. Cada nó não-folha de uma árvore B+ tem entre $\lceil n/2 \rceil$ e n filhos, sendo n fixo para uma árvore particular e N o número de registros.

Árvores B+ tem a vantagem de se auto-organizar com modificações pequenas e locais em inserções e remoções, além de não requerer a reorganização de todo o arquivo para manter o desempenho. Uma desvantagem menor é o sobrecusto de inserção e remoção para manter o balanceamento, além do sobrecusto em espaço.

A figura ?? mostra um exemplo de árvore B+.

Uma árvore B+ é uma árvore com raiz única que segue as seguintes propriedades:

- Todos os caminhos da raiz até as folhas tem o mesmo tamanho.
- Cada nó que não seja raiz nem folha possui de $\lceil n/2 \rceil$ até n filhos.
- Um nó folha tem de $\lceil (n - 1)/2 \rceil$ até $n - 1$ valores.
- Casos especiais:

Figura 7.8: Exemplo de uma árvore B+ com $n = 4$.

- Se a raiz não é uma folha, ela possui pelo menos dois filhos.
- Se a raiz é uma folha (ou seja, é o único nó da árvore), ela pode ter de 0 a $(n - 1)$ valores.

7.6.1 Estrutura

Um nó típico de árvore B+, como na figura ??, possui:

- K_i são valores de chaves de busca.
- P_i são ponteiros para filhos (nós não folha), ou para registros ou buckets de registros (para nós folha).

As chaves de busca de um nó são ordenadas com $K_1 < K_2 < K_3 < \dots < K_{n-1}$. Normalmente o nó tem o tamanho de um bloco.

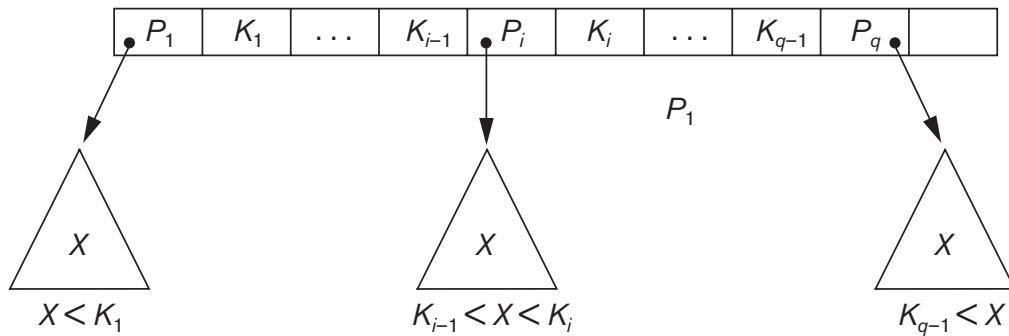


Figura 7.9: Um nó em uma árvore B+.

As propriedades de um nó folha, como na figura ??, são:

- Para $i = 1, 2, \dots, n - 1$, o ponteiro P_i aponta para o registro com chave de busca K_i , ou para um bucket de apontadores para registros. Buckets só são necessários se a chave de busca não for chave primária.
- Se L_i, L_j são nós folha e $i < j$, as chaves de busca de L_i são menores que as chaves de busca de L_j .
- P_n aponta para o próximo nó folha na ordem da chave de busca.

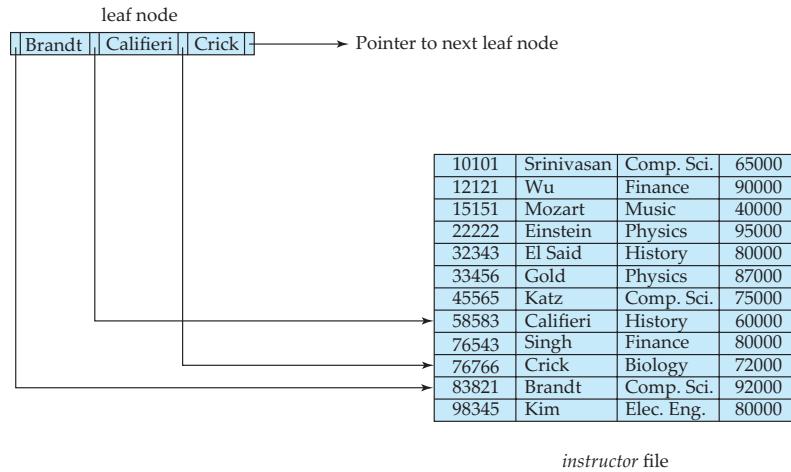


Figura 7.10: Exemplo de um nó folha ($n = 4$) de uma árvore B+.

Ao passo que nós não folha formam um índice esparsão multinível sobre os nós folha. Para um nó não folha, como na figura ??, com m ponteiros:

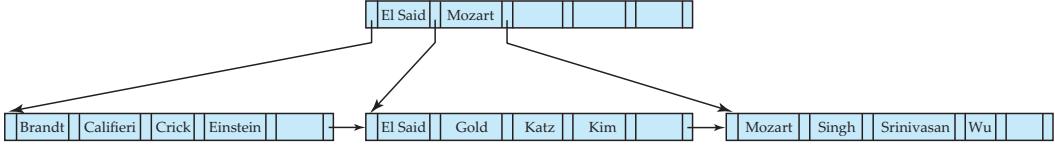
- Todas as chaves de busca da subárvore apontada por P_1 são menores do que K_1 .
- Para $2 \leq i \leq n - 1$, todas as chaves de busca da sub-árvore apontada por P_i são maiores ou iguais a K_{i-1} e menores que K_i .
- Todas as chaves de busca da sub-árvore apontada por P_n são maiores ou iguais a K_{n-1} .

A figura ?? ilustra um exemplo de árvore B+ com $n = 6$. Baseado nessa figura, podemos afirmar que:

- Nós folha tem de 3 a 5 valores ($\lceil (n-1)/2 \rceil$ e $n-1$, com $n = 6$).
- Nós não folha (que não a raiz) tem de 3 a 6 filhos ($\lceil n/2 \rceil$ e n com $n = 6$).
- A raiz precisa ter pelo menos 2 filhos.

Algumas observações sobre árvores B+:

- Visto que as conexões entre nós são feitas por ponteiros, blocos “logicamente” próximos não precisam estar “fisicamente” próximos.
- Os níveis não folha de uma árvore B+ formam uma hierarquia multinível de índices esparsos.
- A árvore possui um número relativamente pequeno de níveis:

Figura 7.11: Árvore B+ com $n = 6$.

- O nível abaixo da raiz possui pelo menos $2 * \lceil n/2 \rceil$ valores.
 - O próximo nível possui pelo menos $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ valores.
 - ... etc.
 - Se existem K chaves de busca, a altura não é maior do que $\lceil \log_{n/2} K \rceil$.
 - Por isso as buscas são eficientes.
- Inserções e remoções são eficientes, já que o índice é reestruturado em tempo logarítmico, como será visto adiante.

7.6.2 Busca

Os passos na busca em árvores B+ para uma chave de busca de valor V onde retorna C e o índice i tal que $C.P_i$ aponta para o primeiro registro com chave de busca V :

1. $C = \text{raiz}$.
2. **Repete** enquanto C não é um nó folha:
 - (a) Seja $i =$ o menor número tal que $V \leq C.K_i$.
 - (b) **Se** não existir um i , então $C = C.P_m$ onde P_m é o último ponteiro não-nulo do nó.
 - (c) **Senão** (existe um i):
 - i. **Se** $V = C.K_i$, então $C = C.P_{i+1}$.
 - ii. **Senão** $C = C.P_i$ ($V < C.K_i$).
3. Seja i o valor mínimo tal que $K_i = V$.
4. **Se** existe um i , retorna (C, i) , **senão** retorna *null*.

Se o arquivo de índice possui K chaves de busca, a altura da árvore não ultrapassa $\lceil \log_{\lceil n/2 \rceil} K \rceil$. Um nó é geralmente do mesmo tamanho de um bloco, tipicamente 4 Kbytes, e n é tipicamente cerca de 100 (40 bytes por entrada no índice). Com 1 milhão de chaves de busca e $n = 100$, no máximo $\log_{50}(1.000.000) = 4$ nós são acessados em uma busca, ou seja, no máximo 4 acessos a blocos. Compare isso com uma árvore binária balanceada com 1 milhão de chaves de busca, que acessa cerca de 20 nós em uma busca. A diferença é significativa já que cada acesso a nó requer um acesso a disco.

7.6.3 Atualização

Quando um registro é inserido ou removido, os índices relacionados precisam ser atualizados. Inserções e remoções são mais complicadas porque pode ser necessário *dividir* um nó muito largo depois da inserção, ou *combinar* nós pequenos (com menos de $\lceil n/2 \rceil$ ponteiros).

- **Inserção** - usando a mesma técnica de busca, primeiro procura o nó folha em que a chave de busca pode ser encontrada. Em seguida insere a entrada (o par chave de busca e ponteiro ao registro) de forma a manter a ordem das chaves de busca.
- **Remoção** - usando a mesma técnica de busca, procura o nó folha contendo a entrada a ser removida. Então, remove a entrada do nó folha.

7.6.3.1 Inserção

Os passos para inserção em uma árvore B+ são:

1. Encontre o nó folha onde a chave de busca deveria aparecer.
2. Se a chave existir no nó folha:
 - (a) Adicionar registro no arquivo.
 - (b) Adicionar o ponteiro no bucket.
3. Se a chave de busca não estiver presente:
 - (a) Adicionar o registro no arquivo e criar o bucket.
 - (b) Se tiver espaço no nó folha, inserir par (chave, ponteiro).
 - (c) Senão, dividir o nó folha, junto com o novo par (chave-valor).

A divisão do nó folha envolve os seguintes passos:

1. Pegar os n pares (chave, apontador) (incluindo o novo) em ordem. Colocar os primeiros $\lceil n/2 \rceil$ no nó original e o restante no novo nó.
2. Considere que o novo nó é p , e k é a menor chave em p . Inserir (k, p) no pai do nó sendo dividido.
3. Se o pai estiver cheio, divida-o e propague a divisão para cima. A divisão prossegue até que se encontre um nó não cheio.

Um exemplo de inserção é mostrado na figura ??.

No pior caso, a raiz será dividida, aumentando a altura da árvore em uma unidade.

Os passos na divisão de um nó não folha, quando ao inserir (k, p) em um nó interno N já cheio, são:

1. Copiar N para uma área temporária M com espaço para $n + 1$ ponteiros e n chaves.
2. Inserir (k, p) em M .
3. Copiar $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ de M para o nó N .

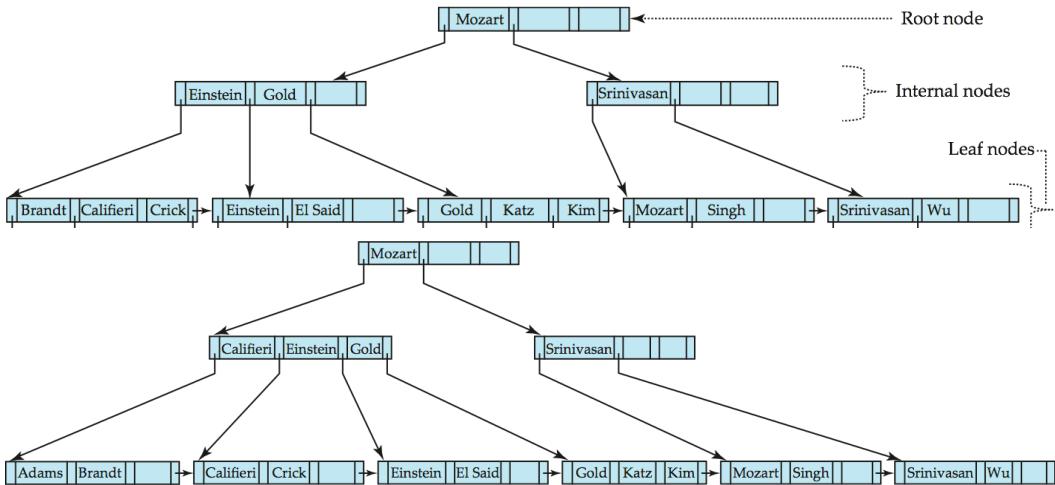


Figura 7.12: Árvore B+ antes e depois da inserção de “Adams”.

4. Copiar $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ de M para o novo nó N' .
5. Inserir $(K_{\lceil n/2 \rceil}, N')$ no pai N .

A figura ?? mostra um exemplo da divisão de um nó não-folha cheio e inserção do no pai N .

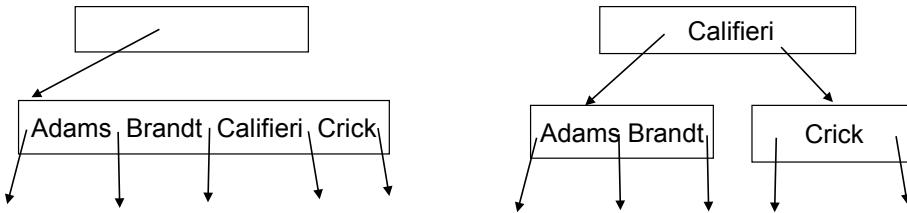


Figura 7.13: Exemplo de uma divisão de nó não-folha cheio.

7.6.3.2 Remoção

Os passos da remoção são:

1. Encontrar os registros a serem removidos, removê-los do arquivo e do bucket (se necessário).
2. Remover (chave, ponteiro) do nó folha se não tiver bucket ou se o bucket estiver vazio.
3. Se o nó possuir poucas entradas devido a remoção, e as entradas do nó e seu vizinho couberem no mesmo nó, então **mesclar vizinhos**:
 - (a) Inserir todas as chaves dos dois nós em um só (o da esquerda) e remover o outro.
 - (b) Remover o pai do par (K_{i-1}, P_i) , onde P_i é o ponteiro para o nó removido.
 - (c) Repetir o procedimento se o pai ficar com poucas entradas.
4. Senão, se o nó possuir poucas entradas, mas as entradas no nó e no vizinho não couberem em um só, então **redistribuir ponteiros**:

- (a) Redistribuir os ponteiros entre o nó e o vizinho de modo que ambos tenham mais do que o número mínimo de entradas.
- (b) Atualizar as chaves correspondentes do nó pai.

As remoções podem se propagar para cima até que se encontre um nó com pelo menos $\lceil n/2 \rceil$ ponteiros. Se o nó raiz possuir apenas um ponteiro após a remoção, ele é removido e seu filho se torna a raiz. A figura ?? mostram o exemplo de remoção.

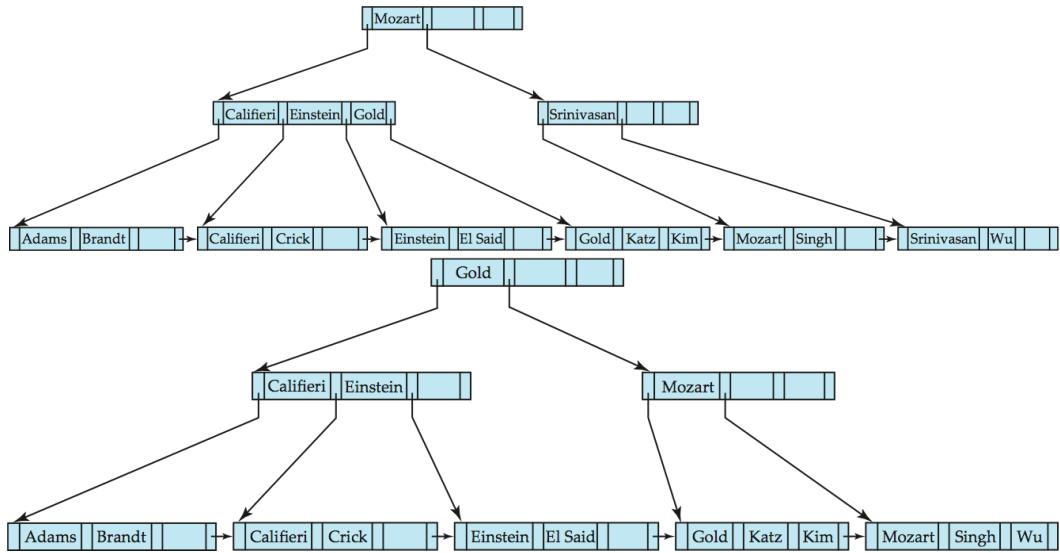


Figura 7.14: Árvore B+ antes e depois da remoção de “Srinivasan”.

7.6.4 Organização de arquivos

Árvores B+ podem ser usadas diretamente para organizar arquivos em vez de simplesmente para indexação. Nesse caso, os nós folhas armazenam registros em vez de ponteiros. O problema da degradação de arquivo é resolvida usando essa solução. A inserção e remoção são tratadas da mesma forma.

As folhas ainda precisam estar preenchidas até a metade visto registros são mais longos do que ponteiros, o máximo de registros em um nó folha é menor do que o número de ponteiros dos nós não-folha.

Para melhorar a utilização de espaço, pode-ser envolver mais vizinhos na redistribuição durante as divisões e mesclas. Por exemplo, usar 2 vizinhos na redistribuição (para evitar divisões/mesclas) resulta em cada nó tendo pelo menos $\lfloor 2n/3 \rfloor$ entradas.

7.7 Índices Hash

7.7.1 Hashing estático

Por definição, um **bucket** é uma unidade de armazenamento com um ou mais registros. Tipicamente, um bucket pode ser um bloco de disco. Em uma organização de arquivo com hash, o bucket de um registro é obtido diretamente de sua chave de busca por meio da função hash.

Uma **função hash** mapeia todas as chaves de busca para buckets, e é usada para localizar registros para acesso, inserção e remoção. Registros com valores de chave de busca diferentes podem ser mapeados para o mesmo bucket. Para localizar um registro, todo o bucket deve ser varrido sequencialmente.

A figura ?? mostra um exemplo de arquivo hash com hashing estático.

bucket 0			

bucket 1			
15151	Mozart	Music	40000

bucket 2			
32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3			
22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4			
12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5			

bucket 6			
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7			

Figura 7.15: Organização de arquivo hash para *instructor*, com *dept_name* como chave de busca.

7.7.1.1 Funções hash

A pior função mapeia todas as chaves de busca para o mesmo bucket, o que o tempo de acesso proporcional ao número de registros do arquivo.

Uma função hash ideal é:

- **Uniforme** - cada bucket recebe o mesmo número de chaves de busca, a partir do conjunto de todos os valores possíveis.
- **Aleatória** - cada bucket recebe o mesmo número de chaves de busca, qualquer que seja a distribuição de chaves de busca real no arquivo.

Funções típicas realizam o cálculo a partir da representação binária das chaves de busca. Por exemplo, somando a representação binária dos caracteres e dividindo pelo número de buckets. O resto da divisão seria o bucket designado.

7.7.1.2 Tratamento dos estouros de buckets

Em caso de colisão, os registros que colidem ocupam o mesmo bucket e pode ocorrer estouro do bucket (*bucket overflow*). O estouro pode ocorrer devido:

- Buckets insuficientes.

- Desequilíbrio na distribuição dos registros. Os motivos são:

- muitos registros com a mesma chave de busca.
- a função de hash não é uniforme.

Embora a probabilidade de estouro possa ser reduzida, ela não pode ser eliminada. Ela pode ser tratada através de **buckets de estouro**. No **encadeamento de estouro**, os buckets de estouro são ligados por uma lista encadeada (ver figura ??).

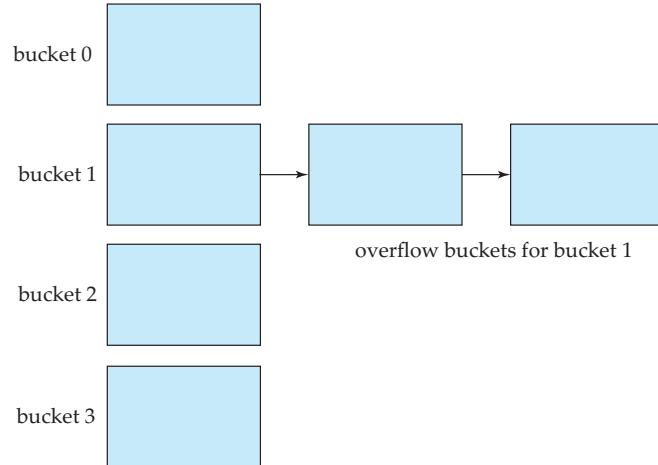


Figura 7.16: Exemplo de encadeamento de estouro.

O esquema acima é chamado **endereçamento fechado** (*closed hashing*). Uma outra alternativa, chamada **endereçamento aberto** (*open hashing*), não usa buckets de estouro. Porém, não é adequada para aplicações de banco de dados.

7.7.1.3 Índices hash

Hashing também pode ser usado como índices. Um **índice hash** organiza as chaves de busca com seus ponteiros para registros.

7.7.1.4 Deficiências do hashing estático

No hashing estático, a função h mapeia chaves de busca para um número fixo de buckets. Com o crescimento ou encolhimento do banco de dados:

- Se o número de buckets for baixo, e o arquivo crescer, o desempenho irá cair devido aos estouros.
- Se o número de buckets for alto, uma grande quantidade de espaço será desperdiçada inicialmente.
- Se o arquivo encolher, novamente o espaço será desperdiçado.

Uma solução seria reorganização periódica do arquivo com uma nova função de hash. Porém, essa solução custa caro e interrompe operações sobre o arquivo. Uma solução melhor é permitir que o número de buckets seja modificado dinamicamente.

7.7.2 Hashing dinâmico

Usado para banco de dados cujo tamanho costuma mudar, e permite que a função de hash mude dinamicamente. O **Hashing extensível** é uma das formas de hashing dinâmico. A função de hash gera valores a partir de um intervalo elevado, tipicamente inteiros com b -bits, onde $b = 32$.

Apenas um prefixo da função de hash é usado para indexar uma **tabela de buckets**. Considere que a largura do prefixo seja de i bits, $0 \leq i \leq 32$. O tamanho da tabela de buckets é 2^i , sendo inicialmente $i = 0$. O valor i cresce ou diminui junto com o banco de dados.

Múltiplas entradas da tabela de buckets podem apontar para o mesmo bucket. Assim, o número real de buckets é $< 2^i$. O número de buckets também muda dinamicamente pela sua mescla ou divisão.

Em uma hash extensível, cada bucket j guarda o valor i_j . Todas as entradas que apontam para o mesmo bucket tem o mesmo valor para os primeiros i_j bits. Para localizar o bucket contendo a chave de busca k_j :

1. Computar $h(K_j) = X$.
2. Usar os bits de maior ordem i de X como um deslocador na tabela de buckets, e seguir o ponteiro para o bucket apropriado.

7.7.2.1 Inserção em estruturas de hash extensível

Para inserir um registro com chave de busca K_j :

1. localizar o bucket apropriado j .
2. se tiver espaço no bucket j , inserir o registro no bucket.
3. senão, o bucket deve ser dividido e a inserção tentada de novo. Em alguns casos, são necessários buckets de estouro.

Para dividir o bucket j quando inserindo registro com a chave de busca K_j :

1. Se $i > i_j$ (mais de um apontador para o bucket j):
 - (a) alocar um novo bucket z e atribuir $i_z = i_j + 1$.
 - (b) atualizar a segunda metade das entradas da tabela de buckets que apontavam para j , fazendo-as apontar para z .
 - (c) remover os registros do bucket j e reinseri-los (em j ou z).
 - (d) localizar o bucket para K_j e inserir registro no bucket apropriado (realizar divisões sucessivas enquanto o bucket continuar cheio).
2. Se $i = i_j$ (apenas um ponteiro para o bucket j):
 - (a) se i atingir o limite b , ou se muitas divisões ocorrerem, criar um bucket de estouro.
 - (b) senão
 - i. incrementar i e dobrar o tamanho da tabela de buckets.
 - ii. substituir cada entrada da tabela por duas entradas que apontam para o mesmo bucket.

- iii. localizar a entrada na tabela de endereçamento para K_j . Agora $i > i_j$ estão no primeiro caso acima.

A figura ?? demostra a representação binária gerada pela função hash h em uma chave de busca exemplo, enquanto que na figura ?? ilustra a estrutura do arquivo hash depois de 11 inserções.

<i>dept_name</i>	$h(dept_name)$
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Figura 7.17: Função de hash extensível para o exemplo a seguir de inserção.

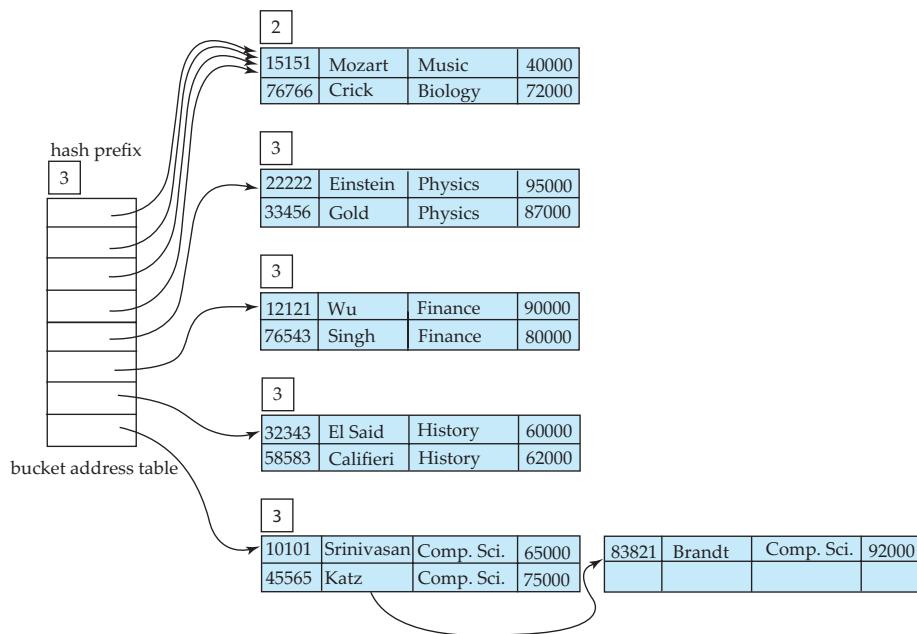


Figura 7.18: Estrutura do arquivo hash depois de 11 inserções.

7.7.2.2 Remoção em estruturas de hash extensível

Para remover uma chave:

1. localizá-la no bucket e removê-la.
2. o bucket pode ser removido se ficar vazio (com as devidas atualizações na tabela de buckets).
3. a mescla de buckets pode ser feita (pode mesclar apenas com um bucket que tenha o mesmo valor i_j e o mesmo prefixo $i_j - 1$, se existir).

A redução da tabela de buckets também é possível. Porém, a redução é cara e deve ser feita apenas se o número de buckets for muito menor do que o tamanho da tabela.

7.7.2.3 Comparação com outros esquemas

As vantagens do hash extensível são:

- O desempenho não degrada com o aumento do arquivo, sendo $C(n) = 1$ se a função for uniforme e aleatória.
- Sobrecarga de espaço é mínima.

As desvantagens são:

- Nível de indireção extra para localizar registros.
- A tabela de buckets pode se tornar muito grande (maior que o espaço disponível na memória).
 - Não se pode alocar espaços contíguos em disco.
 - Solução: usar uma árvore B+ para representar os registros em uma tabela de buckets.
- Mudar o tamanho da tabela de endereço também é uma operação custosa.
- Custo para recuperar os registros em ordem é alto.
- Pior caso é $C(n) = n$.

7.8 Índices multidimensionais

Os atributos usados em uma busca podem ser considerados como dimensões. Consultas que envolvem buscas em mais de um atributo ou dimensão podem usar dois tipos de índices.

- Índices unidimensionais.
- Índices multidimensionais.

Em índices unidimensionais a chave de pesquisa é única. Também é possível recuperar registros que correspondem a um dado valor (ou intervalo) da chave de busca. Os tipos de índices que vimos anteriormente são por exemplo árvores B+ e tabelas hash.

Índices multidimensionais são usados para consultas em mais do que um atributo (ou dimensão). Como exemplo, um sistema de informações geográficas armazena objetos em um espaço bidimensional. As consultas que poderiam ser feitas são:

- Consultas de correspondência parcial.
- Consultas de intervalo.
- Consultas de vizinho mais próximo.
- Consultas do tipo onde estou.

Outro exemplo utilizado em sistemas de decisão são cubos de dados que, em geral, utiliza a dimensão tempo nas consultas. Por exemplo, pode-se fazer consultas de vendas de um produto para cada loja em um determinado período de tempo.

Nesses exemplos, índices unidimensionais apresentam limitações:

- Em consultas por intervalos, o número de blocos acessados pode ser excessivo. Imagine que os passos envolvidos caso a consulta fosse realizada em uma árvore B+ são:

1. Usar um índice para cada uma das dimensões x, y.
2. Obter os ponteiros da árvore B+ para os registros no intervalo de x.
3. Obter os ponteiros da árvore B+ para os registros no intervalo de y.
4. Fazer a interseção dos ponteiros.
5. Examinar blocos da interseção.

- Em consultas de vizinho mais próximo, não se sabe qual intervalo buscar.

A estruturas de índices multidimensionais podem ser baseadas em:

Índices bitmap código sucintos (bits) para responder com eficiência consultas sobre múltiplas chaves.

Estruturas baseadas em hash a resposta da consulta pode não estar em um único bucket.

Contudo, pode estar em um número baixo de buckets. As estruturas são: arquivos de grade e funções de hash particionado.

Estruturas baseadas em árvores não há correspondência entre os nós da árvore e blocos do disco. Todavia, árvores podem ficar desbalanceadas e a eficiência nas atualizações pode ser menor. As estruturas são: índices de várias chaves, árvore kd, árvore de quadrante, e árvores R.

7.8.1 Índices de bitmap

Chaves de busca compostas são chaves de busca que possuem mais de um atributo como em (*branch_name, balance*).

Índices de mapa de bits (bitmap) foram projetados para responder com eficiência consultas sobre múltiplas chaves. Registros em uma relação devem possuir uma numeração sequencial. Dado um número n deve ser fácil recuperar o registro n .

Um índice de bitmap é aplicável para atributos que assumem uma quantidade pequena de valores. Exemplos são: sexo, país, estado, nível salarial (salário categorizado em níveis como 0-9999, 10000-19999, 20000, 50000, 50000-infinito). Um mapa de bits é simplesmente um vetor de bits.

Um índice de bitmap sobre um atributo tem um bitmap para cada valor possível para esse atributo. O bitmap tem tantos bits quantos forem os registros. Em um bitmap para o valor v , o bit para um registro é 1 se o registro possui o valor v , e 0 caso contrário.

Índices de bitmap são úteis para consultas por múltiplos atributos, e não muito úteis para consultas por um único atributo. Consultas são respondidas usando os seguintes operadores de bitmap:

- Interseção (**AND**).
- União (**OR**).
- Complementação (**NOT**).

Cada operação usa dois bitmaps de mesmo tamanho e aplica a operação nos bits correspondentes para gerar o bitmap de resultado. Exemplos dessas operações são:

- Interseção – 100110 **AND** 110011 = 100010
- União – 100110 **OR** 110011 = 110111
- Complementação – **NOT** 100110 = 011001

Índices de bitmap geralmente são muito pequenos se comparados com o tamanho da tabela.

A figura ?? mostra um exemplo de índice bitmap com a relação de dois campos *gender* e *income_level*. Por exemplo, em uma busca *gender* = *f*(01101) e *income_level* = *L2*(01000) aplica-se a operação de interseção (AND) e resulta no bitmap 01000.

record number	<i>ID</i>	Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>		
		m	f	L1	L2	L3
0	76766	m		L1		
1	22222		f	L2		
2	12121		f	L1		
3	15151	m		L4		
4	58583		f	L3		

m	10010	Bitmaps for <i>income_level</i>										
f	01101	<table border="1"> <tr><td>L1</td><td>10100</td></tr> <tr><td>L2</td><td>01000</td></tr> <tr><td>L3</td><td>00001</td></tr> <tr><td>L4</td><td>00010</td></tr> <tr><td>L5</td><td>00000</td></tr> </table>	L1	10100	L2	01000	L3	00001	L4	00010	L5	00000
L1	10100											
L2	01000											
L3	00001											
L4	00010											
L5	00000											

Figura 7.19: Índices bitmap para uma relação.

Quanto a implementação, bitmaps são empacotados em palavras, uma única palavra em uma instrução de CPU computa AND de 32 ou 64 bits de cada vez. Por exemplo, um mapa com milhão de bits pode ser computado com apenas 31.250 instruções.

A contagem do número de 1s pode ser otimizada ao usar cada byte para indexar um array de 256 posições. Cada posição guarda a contagem de 1s na sua respectiva representação binária. Pode-se usar pares de bytes para agilizar o cálculo, com um custo de memória adicional. Em seguida, basta somar as contagens recuperadas. Bitmaps podem ser usados nos nós folhas das árvores B+ em vez de listas de ponteiros de registros. É vantajoso se $> 1/64$ dos registros tiver esse valor, supondo que o ponteiro ocupe 64 bits. Essa técnica combina vantagens dos bitmaps e das árvores B+.

7.8.2 Arquivos de grade

Em arquivos de grade, há um espaço de pontos dispostos em uma grade. Imagine que cada dimensão possue linhas de grade que particionam o espaço entre faixas. Os pontos que caírem em uma linha de grade serão considerados pertencentes à faixa da qual essa linha de grade é o limite

inferior. O número de linhas de grade em cada dimensão, assim como o espaçamento entre elas, pode variar.

Cada região em que um espaço é partitionado pode ser imaginada como um bucket de uma tabela hash, e os pontos dentro dessa região são inseridos em um bucket que pertence a essa região. Quando necessário, pode-se usar buckets de estouro. A figura ?? demonstra um exemplo de arquivo de grade com buckets para cada região.

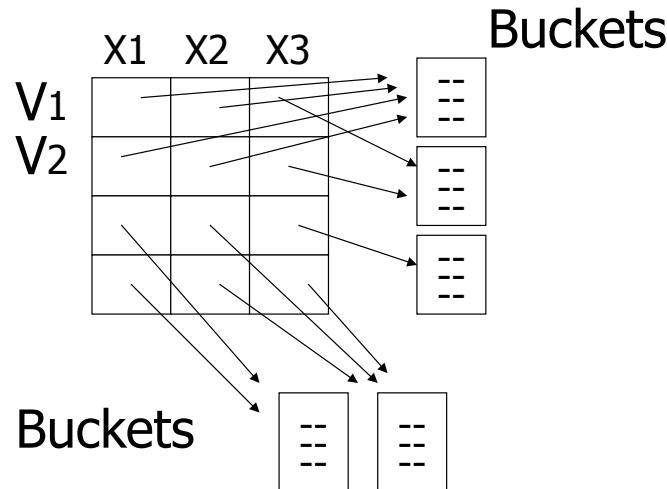


Figura 7.20: Arquivo de grade com buckets.

7.8.2.1 Pesquisa

Ao invés de usar uma tabela unidimensional de buckets, como em tabelas hash, o arquivo de grade usa tabelas auxiliares, ou **array de buckets**, cujo número de dimensões é igual ao do arquivo de dados. Para realizar uma pesquisa, é preciso saber quais faixas delimitam os buckets em cada dimensão. A partir do array de buckets podemos determinar a região em que um ponto está.

A figura ?? mostra um exemplo de arquivo de grade com duas dimensões: salário (y) e departamento (x). As tabelas auxiliares indicam o valor de cada linha da grade.

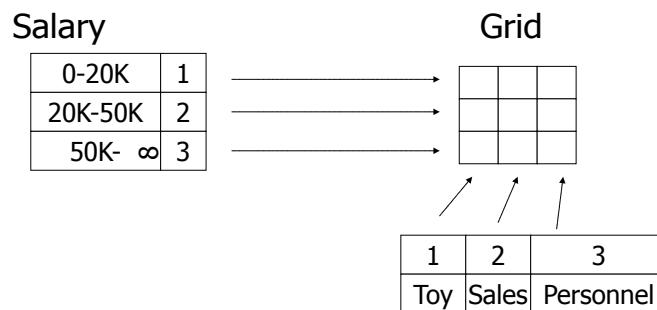


Figura 7.21: Arquivo de grade com duas dimensões.

7.8.2.2 Inserção

Segue a mesma lógica da pesquisa. Se houver espaço no bucket para o registro novo, não haverá nada mais a fazer. Se não houver espaço, existem duas abordagens:

1. Adiciona-se buckets de estouro.
2. Reorganiza-se a estrutura, adicionando-se ou movendo as linhas da grade.

7.8.2.3 Desempenho

Um problema importante no caso de altos números de dimensões é que o número de buckets cresce exponencialmente com a dimensão. Por exemplo, 3 dimensões com 10 faixas em cada dimensão resulta em $10^3 = 1000$ blocos.

O ideal é usar poucos buckets, e distribuir bem os pontos para evitar estouro de buckets. Importante notar que esse ideal é difícil alcançar se os pontos não tem uma distribuição uniforme.

7.8.3 Hash particionado

Funções de **hash particionado** aplicam uma função hash sobre uma lista de atributos. Cada valor de atributo contribui para formar parte da chave hash gerada. A figura ?? mostra o uso de 4 bits para um atributo A e 6 bits para um atributo B, formando assim um identificador de 10 bits. Essa tabela hash contém 1024 buckets (2^{10}).

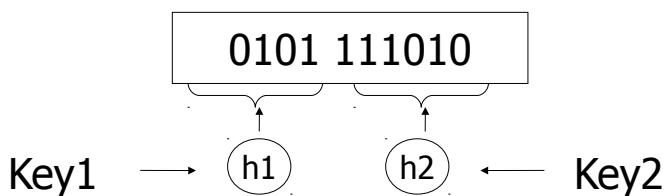


Figura 7.22: Chave hash para uma tabela com 1024 buckets e 10 bits.

Comparado com arquivos de grade, o hash particionado tem pouca utilidade em consultas por intervalos e consultas de vizinho mais próximo. Nessa caso a distância física entre os pontos não reflete na proximidade dos números de buckets.

Os arquivos de grade são mais adequados para consultas de vizinho mais próximo e consultas em intervalos. Porém, o número alto de dimensões pode levar a muitos buckets vazios ou quase vazios.

7.8.4 Índices de várias chaves

Índices de várias chaves nada mais são do que um índice de índices, ou seja, uma árvore que o índice sobre um atributo A conduz a índices sobre outro atributo B para cada valor possível de A. Os nós de cada nível dessa árvore são índices para um único atributo.

O índice sobre primeiro atributo é a raiz da árvore. Além disso, cada índice ou nível pode estar em qualquer formato como em tabelas hash ou árvores B+. A figura ?? demonstra um índice de várias chaves onde o primeiro nível é o atributo “idade”, enquanto que o segundo nível é do atributo “salário”. Note que os índices internos são menores do que os externos.

Sobre o desempenho, o que deve ser considerado:

- Em consultas de correspondência parcial, se o 1º atributo é especificado, então o acesso será eficiente. Caso contrário, todo o sub-índice (nível 2) terá de ser pesquisado.

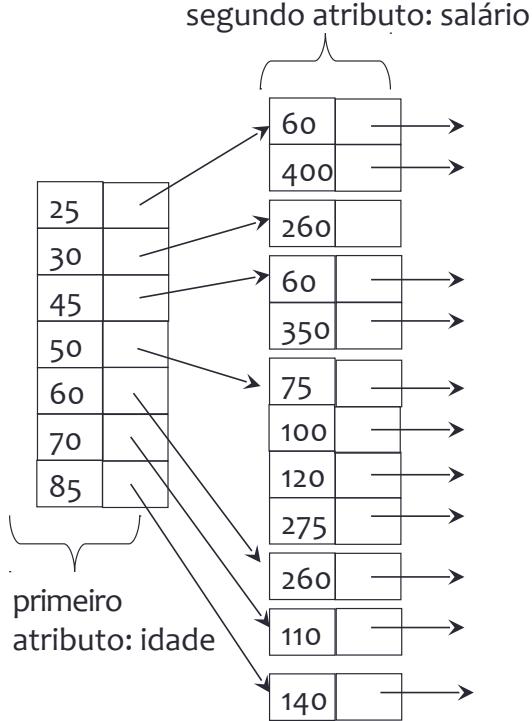


Figura 7.23: Índice de várias chaves com atributo idade (nível 1) e salário (nível 2).

- Para consultas de intervalo, pode ser eficiente quando os índices individuais forem implementados com estruturas que permitem consultas de intervalos, como por exemplo árvores B+. Além disso, será mais eficiente se todos os atributos do índice são especificados (como no caso anterior).
- Uma consulta de vizinho mais próximo é semelhante a consulta de intervalo onde deve-se informar o intervalo da pesquisa. No caso de pontos, deve-se encontrar pontos entre $-d \leq x \leq x_0$ e $-d \leq y \leq y_0$. Todavia, o desafio é encontrar o valor d .

7.8.5 Árvores *kd*

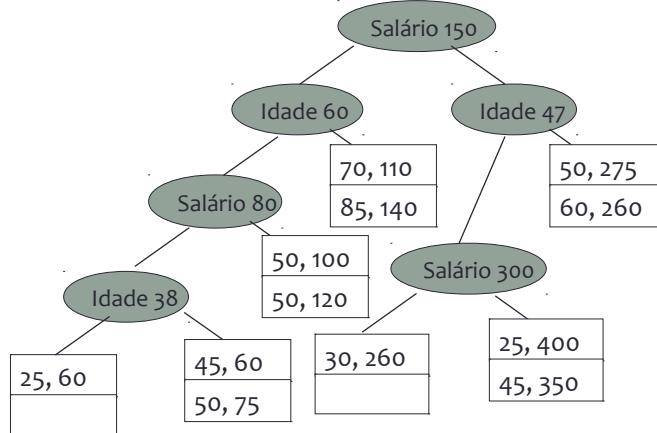
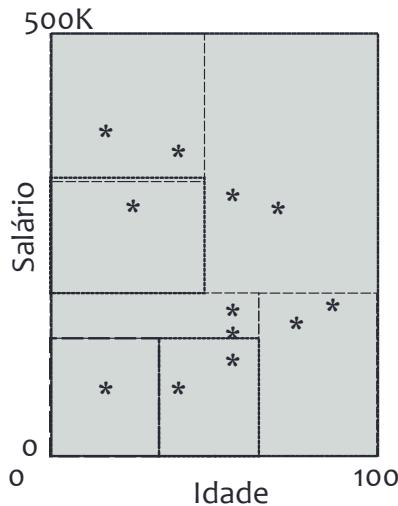
Uma árvore *kd* (árvore de pesquisa *k*-dimensional) é uma estrutura de dados da memória principal que generaliza a árvore de pesquisa binária para dados multidimensionais. Os nós tem um atributo que divide os filhos da esquerda e direita, e as folhas são blocos (buckets) com espaço para tantos registros quantos um bloco pode conter. Os atributos em diferentes níveis da árvore são diferentes, com os níveis se revezando entre os atributos de todas as dimensões.

A figura ?? mostra uma árvore *kd* de salário e idade, onde as folhas possuem buckets de até dois registros.

A figura ?? sugere como os nós da figura ?? dividem o espaço de pontos em blocos de folhas.

7.8.5.1 Operações em árvores *kd*

A **inserção** ocorre de uma forma semelhante a uma árvore binária. Procura-se atingir um único nó folha. Se houver espaço, adiciona-se o registro. Caso contrário, o nó existente terá que ser dividido.

Figura 7.24: Uma árvore *kd* de salário e idade.Figura 7.25: As partições resultantes da árvore *kd* anterior.

A **pesquisa** é feita como em uma árvore binária até atingir um nó folha. Os tipos de consultas podem ser:

- Correspondência parcial que pode precisar seguir as duas sub-árvores de um nó.
- Intervalos em que a divisão pode não satisfazer inteiramente o intervalo da consulta.
- Vizinhos mais próximos que novamente demanda um parâmetro *d*.

7.8.5.2 Adaptação para armazenamento secundário

Caso a árvore *kd* não couber na memória, uma solução trivial seria armazenar os nós em memória secundária onde cada um ocupa um bloco. Todavia, em árvores *kd* pode haver muitos espaços vazios no bloco e caminhos longos até as folhas.

Duas abordagens que melhoraram o desempenho são:

Solução 1 - Ramificações de várias vias em nós interiores, como nas árvores B+ - Nessa solução a altura da árvore diminui e os blocos se tornam cheios. Porém, aumenta dificuldade de mesclar e distribuir nós.

Solução 2 - Nós interiores agrupados em blocos - Nós de um mesmo bloco tendem a ser acessados juntos. Mas aumenta a dificuldade de gerenciar a organização dos blocos.

7.8.5.3 Análise

Árvores *kd* possui várias semelhanças com arquivos de grade:

- Responde ao mesmo tipo de consulta.
- Se os dados não estão bem distribuídos, a árvore não ficará balanceada, e consultas e atualizações ficarão custosas.
- Desempenho cai com muitas dimensões. Em consultas por intervalo ou correspondência parcial, muitos nós teriam que ser acessados, o que pode ser pior do que uma varredura completa.

7.8.6 Árvores de quadrante

7.8.7 Árvores R

7.9 Exercícios