Módulo 3

Estruturas de Dados

Lição 10

Hash Table e Técnicas de *Hashing*

Versão 1.0 - Mai/2007

Autor
Joyce Avestro

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Reginald Hutcherson
Rebecca Ong
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Anna Carolina Ferreira da Rocha
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bárbara Angélica de Jesus Barbosa
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alexandre de Sene
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Everaldo de Souza Santos
Fabrício Ribeiro Brigagão
Fernando Antonio Mota Trinta
Frederico Dubiel
Givailson de Souza Neves
Jacqueline Susann Barbosa
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Kefreen Ryenz Batista Lacerda
Leonardo Leopoldo do Nascimento
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luís Carlos André
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Marcus Borges de S. Ramos de Pádua
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Renato Alves Félix
Renê César Pereira
Reyderson Magela dos Reis
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Fernandes Suguiura
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Silvio Sznifer
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

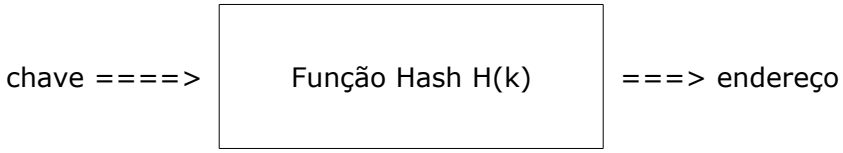
- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Hashing é a aplicação de uma função matemática (chamada **função *hash***) em valores de chave que resultam no mapeamento dos possíveis valores de chave para uma faixa menor de endereços relativos. A função ***hash*** é algo como uma caixa trancada que tem necessidade de uma **chave** para se obter a saída que, neste caso, é o **endereço** onde a chave está armazenada:



No *hashing* não existe conexão óbvia entre a chave e o endereço gerado, pois a função "seleciona randomicamente" um endereço para um valor específico de chave, sem se preocupar com a seqüência física dos registros no arquivo. Por essa razão, *hashing* também é conhecido como **esquema de randomização**.

Ao fim da lição, o estudante deve ser capaz de:

- Definir *hashing* e explicar como o *hashing* funciona
- Implementar **técnicas simples de *hashing***
- Discutir como colisões são evitadas/minimizadas através da utilização de **técnicas de resolução de colisão**
- Explicar os conceitos por trás de **arquivos dinâmicos** e *hashing*

2. Técnicas Simples de *Hash*

Duas ou mais chaves de entrada, sejam k_1 e k_2, quando aplicadas em uma função *hash*, podem resultar no mesmo endereço, um acidente conhecido como **colisão**. A colisão pode ser reduzida alocando-se mais espaço de arquivo que o mínimo necessário para armazenar a quantidade de chaves. Entretanto, essa abordagem leva a desperdício de espaço. Existem diversas formas de lidar com colisões, que serão discutidas mais adiante.

Em *hashing*, existe a necessidade de se escolher uma boa função *hash* e, conseqüentemente, selecionar um método para resolver, se não eliminar, as colisões. Uma boa função *hash* executa cálculos eficientes, com complexidade de tempo O(1), e produz pouca (ou nenhuma) colisão.

Existem muitas técnicas de *hash* disponíveis, mas discutiremos apenas duas – *divisão por número primo* e *desdobramento.*

2.1. Método de Divisão por Números Primos

Este é um dos métodos mais comuns de randomização. Se o valor chave é dividido por um número **n**, a faixa de endereços gerados vai variar entre **0** e **n-1**.

A fórmula é:

h(k) = k mod n

onde **k** é a chave, um número inteiro, e **n** é um número primo.

Se **n** é o número total de locações relativas no arquivo, este método pode ser usado para mapear as chaves em **n** locações de registro. **n** deve ser escolhido de forma a reduzir o número de colisões. Se **n** é par, o resultado da função *hash* é um número par, se **n** é impar, o resultado é ímpar. A divisão por número primo não resultará em muitas colisões. Por isso é a melhor escolha para o divisor nesse método. Poderíamos escolher um número primo que seja próximo ao número de registros no arquivo. Entretanto, este método pode ser usado mesmo se **n** não for primo, mas prepare-se para lidar com mais colisões.

Por exemplo, considere n = 13

Valor da Chave k	Valor *Hash* h(k)
125	8
845	0
444	2
256	9
345	7
745	4
902	5
569	10
254	7
382	5

Valor da Chave k	Valor *Hash* h(k)
234	0
431	2
947	11
981	6
792	12
459	4
725	10
652	2
421	5
458	3

Para implementar este *hashing* em Java, basta usar:

```
public int hash(int k, int n) {
  return (k % n);
}
```

2.2. Desdobramento

Outra técnica simples de *hashing* é o desdobramento. Nessa técnica, o valor chave é dividido em duas ou mais partes e então passam por uma operação de *adição*, *AND* ou *XOR* para se obter o endereço *hash*. Se o resultado obtido tiver mais dígitos que o maior endereço no arquivo, os dígitos excedentes de maior ordem são eliminados.

Existem diferentes formas de desdobramento. O valor chave pode ser ***desdobrado ao meio***. Isso é ideal para valores de chave relativamente pequenos já que eles poderiam facilmente caber nos endereços disponíveis. Se, por qualquer motivo, a chave for desdobrada de forma desigual, a parte da esquerda deve ser maior que a parte da direita. A chave também pode ser ***desdobrada em terços***. Isso é ideal para valores de chave um pouco maiores. Também podemos ter ***desdobramento por dígitos alternados***. Os dígitos das posições ímpares formam uma parte e os dígitos das posições pares formam outra. Desdobrar ao meio e em terços pode ser feito ainda de duas formas. Uma é o ***desdobramento pelos extremos*** onde algumas partes da chave desdobrada são invertidas (imitando o jeito em que dobramos papel) e então somadas. Por último, há o ***desdobramento por substituição*** onde nenhuma parte desdobrada das chaves é invertida.

A seguir, alguns exemplos de ***desdobramento por substituição***:

1. Dígitos pares, desdobrando ao meio
   125758 => 125+758 => 883

2. Desdobrando em terços
   125758 => 12+57+58 => 127

3. Dígitos ímpares, desdobrando ao meio
   7453212 => 7453+212 => 7665

4. Dígitos desiguais, desdobrando em terços
   74532123 => 745+32+123 => 900

5. Usando XOR, desdobrando ao meio
   100101110 => 10010 XOR 1110 => 11100

6. Alternando dígitos
   125758 => 155+278 => 433

A seguir, alguns exemplos de **desdobramento pelos extremos**:

1. Dígitos pares, desdobrando ao meio
   125758 => 125+857 => 982

2. Desdobrando em terços
   125758 => 21+57+85 => 163

3. Dígitos ímpares, desdobrando ao meio
   7453212 => 7453+212 => 7665

4. Dígitos desiguais, desdobrando em terços
   74532123 => 547+32+321 => 900

5. Usando XOR, desdobrando ao meio
   100100110 => 10010 XOR 0110 => 10100

6. Alternando dígitos
   125758 => 155+872 => 1027

Este método é útil para converter chaves com grande número de dígitos em chaves com menos dígitos de forma que o endereço caiba em uma palavra de memória. É também mais fácil de armazenar, pois as chaves não precisam de muito espaço para serem guardadas.

3. Técnicas de Resolução de Colisões

Escolher um bom algoritmo de *hashing* baseado em quão poucas colisões espera-se que ocorram é a primeira etapa para se evitar colisões. Entretanto, isso irá apenas minimizar, e não erradicar o problema. Para evitar colisões, poderíamos:

- **espalhar os registros**: por exemplo, encontrar um algoritmo de *hashing* que distribua os registros de maneira uniforme entre os endereços disponíveis. Entretanto é difícil achar um algoritmo de *hashing* que distribua os registros dessa forma.
- **usar mais memória**: se temos muitos endereços de memória para distribuir registros, é mais fácil de se encontrar um algoritmo de *hashing* do que se tivermos aproximadamente o mesmo número de endereços e registros. Uma vantagem é que os registros ficarão distribuídos uniformemente, conseqüentemente diminuindo as colisões. Entretanto, esse método desperdiça espaço.
- **utilizar *buckets***: por exemplo, coloque mais de um registro no mesmo endereço.

Existem várias técnicas de resolução de colisões e nessa seção iremos cobrir *encadeamento*, *utilização de buckets* e *endereçamento aberto*.

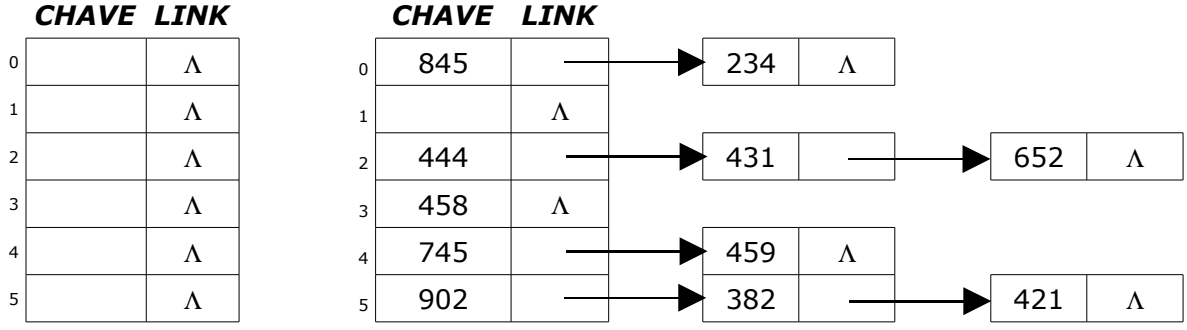
3.1. Encadeamento

No encadeamento, **m** listas ligadas são mantidas, uma para cada possível endereço na tabela *hash*. Utilizando encadeamento para resolver colisão no armazenamento do exemplo de *hashing* do Método de Divisão por Número Primo:

Valor Chave k	Valor *Hash* h(k)
125	8
845	0
444	2
256	9
345	7
745	4
902	5
569	10
254	7
382	5

Valor Chave k	Valor *Hash* h(k)
234	0
431	2
947	11
981	6
792	12
459	4
725	10
652	2
421	5
458	3

Temos a seguinte tabela *hash*:



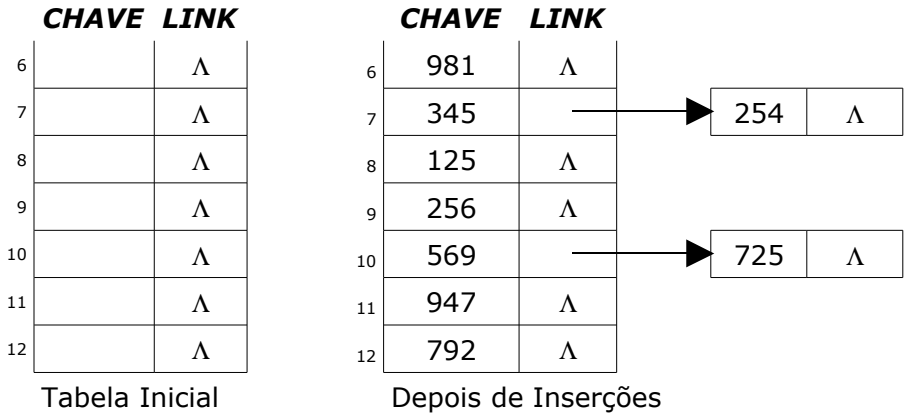


Tabela Inicial Depois de Inserções

As chaves 845 e 234 têm *hash* para o endereço 0, então estão conectadas ao endereço. É o mesmo caso para os endereços 2, 4, 5, 7 e 10, enquanto que os demais endereços não têm colisão. O método de encadeamento resolve a colisão fornecendo *nodes* de conexão adicionais para cada um dos valores.

3.2. Utilização de Buckets

Assim como no encadeamento, este método divide a tabela *hash* em **m** grupos de registros onde cada grupo contém exatamente **b** registros, sendo cada endereço considerado um ***bucket***.

Por exemplo:

	CHAVE1	CHAVE2	CHAVE3
0	845	234	
1			
2	444	431	652
3	458		
4	745	459	
5	902	382	421
6	981		
7	345	254	
8	125		
9	256		
10	569	725	
11	947		
12	792		

A colisão é redefinida nessa abordagem. Ela acontece quando um *bucket* estoura – ou seja, quando se tenta uma inserção em um *bucket* cheio. Por esse motivo, existe uma redução significativa no número de colisões. Entretanto, este método desperdiça espaço e não está livre de ficar cheio futuramente, caso em que uma regra para estouro deve ser criada. No exemplo acima, existem três vagas em cada endereço. Por ter tamanho estático, surgirão problemas quando mais de três valores tiverem *hash* para um mesmo endereço.

3.3. Endereçamento Aberto (Por Verificação)

No endereçamento aberto, quando o endereço produzido por uma função *hash* h(k) não tem mais espaço para inserções, um *slot* diferente de h(k) é alocado na tabela *hash*. Este processo é chamado de **verificação**. Neste *slot* vazio, o novo registro que colidiu com o anterior, conhecido como **chave de colisão**, pode ser seguramente alocado. Neste método, introduzimos a permutação da tabela de endereços, digamos β_0, β_1, ... β_{m-1}, esta permutação é chamada **seqüência de verificação**.

Nesta lição, iremos cobrir duas técnicas: *verificação linear* e *hashing duplo*.

3.3.1. Verificação Linear

Verificação linear é uma das técnicas mais simples para lidar com colisões em que o arquivo é lido ou verificado seqüencialmente, como um arquivo circular, e a chave de colisão é armazenada no espaço disponível mais próximo ao endereço. Isso é usado nos sistemas em que o esquema "primeiro a chegar, primeiro a sair" é utilizado. Um exemplo é o sistema de reservas de uma empresa de linhas aéreas em que os lugares para os passageiros na lista de espera são oferecidos quando os passageiros aos quais os lugares estavam reservados não aparecem. Sempre que uma colisão ocorre em um certo endereço, os endereços seguintes são testados, ou seja, procurados seqüencialmente até que um vago seja encontrado. A chave utiliza então esse endereço. Um *array* deve ser considerado circular, de forma que quando a última localização é alcançada, a pesquisa continua na primeira posição do *array*.

Neste método, é possível que o arquivo inteiro seja pesquisado, a partir da posição *i+1*, e as chaves de colisão sejam distribuídas por todo o arquivo. Se uma chave tem *hash* para a posição **i**, que está ocupada, as posições *i+1, …,n* são pesquisadas procurando-se por uma que esteja vaga.

Os *slots* em uma tabela *hash* podem conter somente uma chave ou também podem conter um *bucket*. Nesse exemplo, *buckets* de capacidade 2 são usados para armazenar as seguintes chaves:

Valor Chave k	Valor *Hash* h(k)
125	8
845	0
444	2
256	9
345	7
745	4
902	5
569	10
254	7
382	5

Valor Chave k	Valor *Hash* h(k)
234	0
431	2
947	11
981	6
792	12
459	4
725	10
652	2
421	5
458	3

resultando na seguinte tabela *hash*:

	CHAVE1	CHAVE2
0	845	234
1		
2	444	431
3	**652**	458

	CHAVE1	CHAVE2
4	745	459
5	902	382
6	981	**421**
7	345	254
8	125	
9	256	
10	569	725
11	947	
12	792	

Nessa técnica, a chave 652 indicou endereço de *hash* 2, mas já está cheio. Verificar o próximo endereço disponível nos leva ao endereço 3, onde a chave é armazenada. Prosseguindo no processo de inserção, a chave 458 tem endereço de *hash* 3 e é armazenada no segundo *slot* do endereço. Com a chave 421 que tem *hash* para o endereço cheio 5, o espaço disponível seguinte é o endereço 6, onde a chave é armazenada.

Esta abordagem resolve o problema de estouro no endereçamento do *bucket*. Além disso, procurar por espaço disponível faz com que as chaves excedentes sejam armazenadas próximas de seus endereços originais, na maioria dos casos. Entretanto, este método sofre com o problema de deslocamento onde as chaves que por direito detém um endereço podem ser deslocadas por outras chaves que simplesmente encontraram aquele endereço vago. Além disso, verificar uma tabela *hash* cheia levará um tempo de complexidade de O(n).

3.3.2. *Hashing* Duplo

O *hashing* duplo faz uso de uma segunda função *hash*, digamos $h_2(k)$, sempre que houver colisão. O registro é inicialmente indicado para um endereço de *hash* utilizando-se a primeira função. Se o endereço de *hash* não estiver disponível, é aplicada uma segunda função *hash* e acrescentada ao primeiro valor *hash*, e a chave de colisão é levada ao novo endereço *hash* se houver espaço disponível. Se não houver, o processo é repetido. A seguir o algoritmo:

1. Utilize a função *hash* primária $h_1(k)$ para determinar a posição i onde colocar o valor.

2. Se houver colisão, utilize a função de *rehash* $r_h(i, k)$ sucessivamente até que um *slot* vago seja encontrado:

   $r_h(i, k) = (i + h_2(k)) \bmod m$

   onde **m** é a quantidade de endereços

Utilizando a segunda função hash $h_2(k)= k \bmod 11$ no armazenamento das seguintes chaves:

Valor Chave k	**Valor *Hash* h(k)**
125	8
845	0
444	2
256	9

Valor Chave k	**Valor *Hash* h(k)**
234	0
431	2
947	11
981	6

Valor Chave k	Valor *Hash* h(k)
345	7
745	4
902	5
569	10
254	7
382	5

Valor Chave k	Valor *Hash* h(k)
792	12
459	4
725	10
652	2
421	5
458	3

Para as chaves 125, 845, 444, 256, 345, 745, 902, 569, 254, 382, 234, 431, 947, 981, 792, 459 e 725, o armazenamento é direto – não houve estouro.

	CHAVE1	CHAVE2
0	845	234
1		
2	444	431
3		
4	745	459
5	902	382
6	981	
7	345	254
8	125	
9	256	
10	569	725
11	947	
12	792	

Inserir 652 na tabela *hash* resulta em estouro no endereço 2, então fazemos *rehash*:

$h_2(652) = 652 \bmod 11 = 3$
$r_h(2, 652) = (2 + 3) \bmod 13 = 5,$

mas o endereço 5 já está cheio, então aplicamos *rehash* novamente:

$r_h(5, 652) = (5 + 3) \bmod 13 = 8$, tem espaço – então armazena aqui.

	CHAVE1	CHAVE2
0	845	234
1		
2	444	431
3		
4	745	459

	CHAVE1	CHAVE2
5	902	382
6	981	
7	345	254
8	125	**652**
9	256	
10	569	725
11	947	
12	792	

Fazendo *hash* para 421 também resulta em colisão, então fazemos o *rehash*:

$h_2(421) = 421 \bmod 11 = 3$

$r_h(5, 421) = (5 + 3) \bmod 13 = 8$,

mas o endereço 8 já está cheio, então aplicamos *rehash* novamente:

$r_h(8, 421) = (8 + 3) \bmod 13 = 11$, tem espaço – então armazenamos aqui.

	CHAVE1	CHAVE2
0	845	234
1		
2	444	431
3	458	
4	745	459
5	902	382
6	981	
7	345	254
8	125	**652**
9	256	
10	569	725
11	947	**421**
12	792	

Por último, a chave 458 é armazenada no endereço 3. Este método é uma evolução da verificação linear em termos de performance, ou seja, o novo endereço é computado utilizando-se outra função *hash* ao invés de percorrer a tabela *hash* seqüencialmente por um espaço vago. Entretanto, assim como a pesquisa linear, este método também sofre com o problema de deslocamento.

4. Arquivos Dinâmicos & *Hashing*

As técnicas de *hashing* discutidas até agora fazem uso de espaço de endereçamento fixo (tabela *hash* com **n** endereços). Com espaço de endereçamento estático, o estouro de armazenamento é possível. Se os dados a serem armazenados tiverem natureza dinâmica, ou seja, muitas exclusões e inclusões possíveis, não é recomendado usar tabelas *hash* estáticas. É aqui que tabelas *hash* dinâmicas tornam-se úteis. Nessa seção, discutiremos dois métodos: *hashing extensível e hashing dinâmico*.

4.1. Hashing Extensível

Hashing extensível utiliza uma estrutura auto-ajustável com tamanho de *bucket* ilimitado. Este método de *hashing* é construído sobre o conceito de **árvore.**

4.1.1. Árvore

A idéia básica é construir um índice baseado na representação numérica binária do valor *hash*. Utilizamos um conjunto mínimo de dígitos binários e acrescentamos mais dígitos se necessário. Por exemplo, considere que cada chave em *hash* é uma seqüência de três dígitos, e no início, precisamos de apenas três *buckets*:

BUCKET	ENDEREÇO
A	00
B	10
C	11

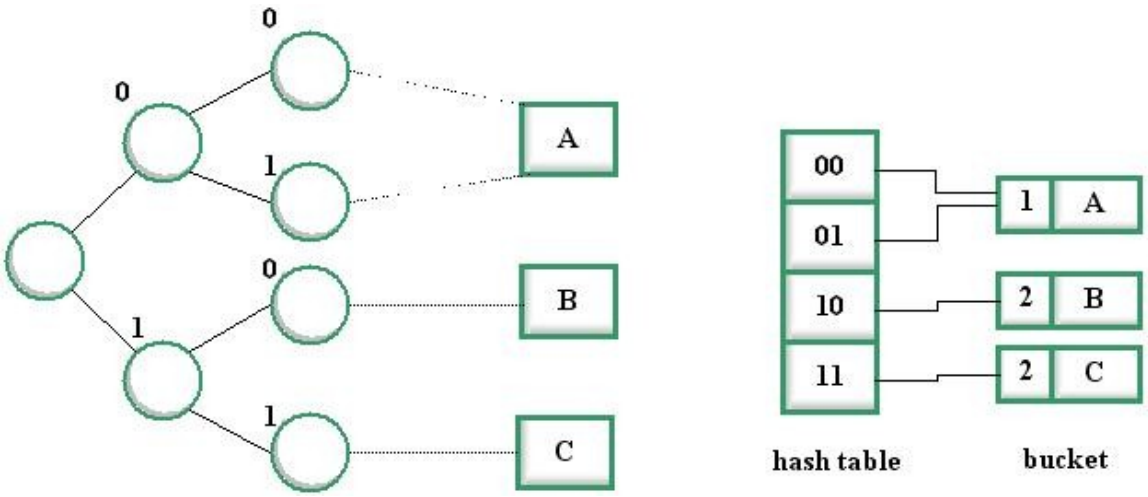


Figura 1. Árvore e Hash Table

A figura na esquerda mostra a árvore. A figura na direita mostra a tabela *hash* e os ponteiros para os *buckets* reais. Para o *bucket* **A**, já que somente um dígito é considerado, ambos endereços 00 e 01 apontam para ele. O *bucket* tem a estrutura (PROFUNDIDADE, DADO) onde **PROFUNDIDADE** é a quantidade de dígitos considerados no endereçamento ou a quantidade de dígitos considerados na árvore. No exemplo, a profundidade do *bucket* **A** é 1, enquanto que para os *buckets* **B** e **C**, é 2.

Quando **A** estoura, um novo *bucket* é criado, digamos **D**. Isso irá criar mais espaço para inserção.

Por esse motivo o endereço **A** é expandido para dois endereços:

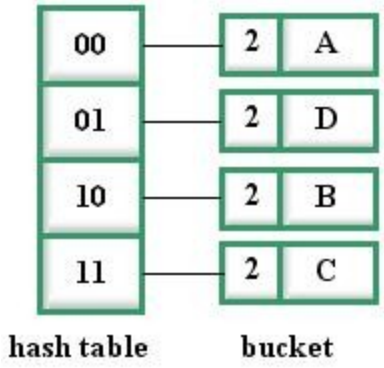


Figura 2. Hash Table

Quando **B** estoura, um novo *bucket* **E** é criado e o endereço de **B** é expandido para três dígitos:

Normalmente oito *buckets* são necessários para o conjunto de todas as chaves com três dígitos, mas, nesta técnica, utilizamos o conjunto mínimo de *buckets* necessário. Note que quando o espaço de endereçamento é aumentado, seu tamanho original é dobrado.

A exclusão de chaves leva a *buckets* vazios e resulta na necessidade de colapsar ***buckets* vizinhos**. Dois *buckets* são vizinhos se eles estão na mesma profundidade e seus *bits* iniciais são os mesmos. Por exemplo, na figura anterior, os *buckets* **A** e **D** têm a mesma profundidade e seus *bits* iniciais são os mesmos. O caso é similar com os *buckets* **B** e **E**, pois ambos têm a mesma profundidade e seus dois *bits* iniciais são iguais.

4.2. Hashing Dinâmico

O *hashing* dinâmico é muito semelhante ao hashing extensível, pois também aumenta em tamanho à medida que novos registros são acrescentados. Eles diferem somente na forma em que o tamanho cresce dinamicamente. Se no *hashing* extensível a tabela hash tem seu tamanho duplicado cada vez que é expandida, no hashing dinâmico, o crescimento é lento e incremental. O *hashing* dinâmico inicia com um tamanho de endereçamento fixo semelhante ao hashing estático, e então cresce conforme necessário. Normalmente, duas funções hash são usadas. A primeira é para verificar se o *bucket* está no espaço de endereçamento original e a segunda é usada caso não esteja. A segunda função *hash* é usada para guiar a pesquisa através da árvore.

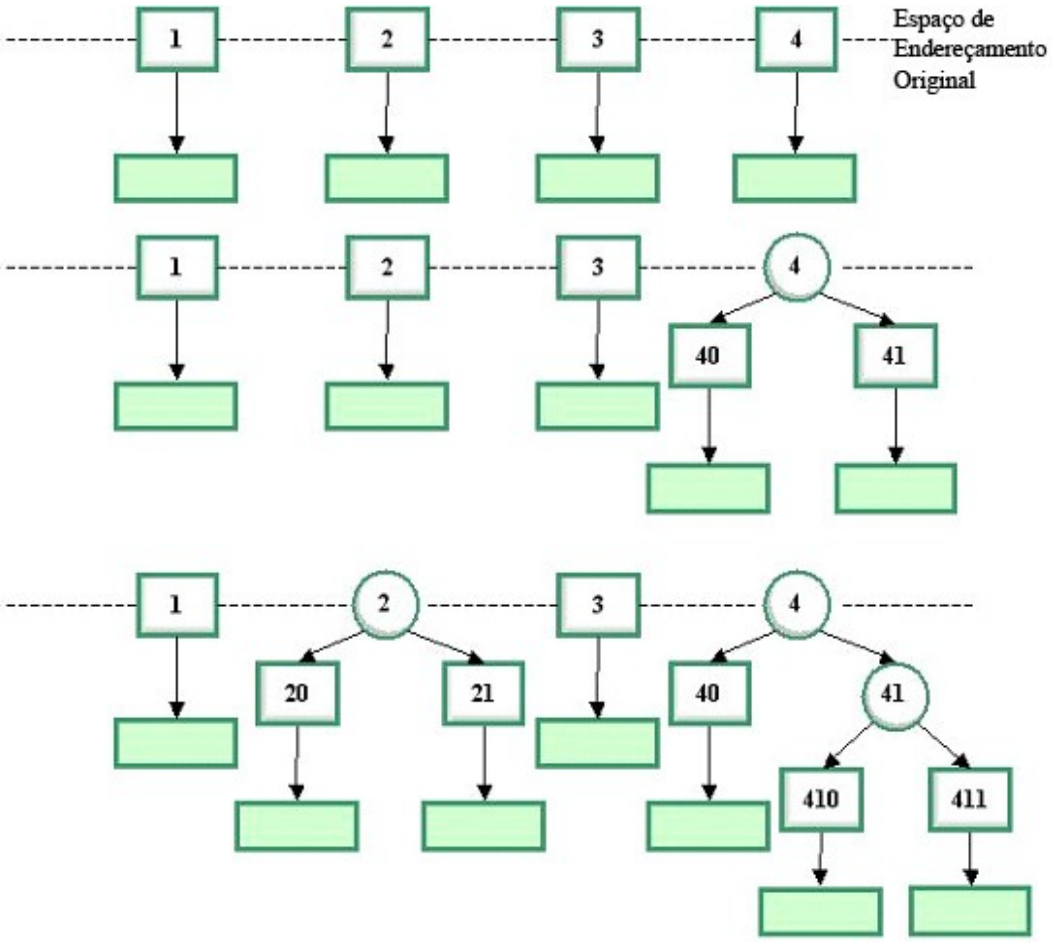


Figura 3. Exemplo de Hashing Dinâmico

O espaço de endereçamento original consiste de quatro endereços. Um estouro no endereço 4 resulta na sua expansão para usar dois dígitos 40 e 41. Existe também um estouro no endereço 2, então ele é expandido para 20 e 21. Um estouro no endereço 41 resultou na utilização de três dígitos 410 e 411.

5. Exercícios de Fixação

1. Considere as chaves a seguir:

12345	21453	22414	25411	45324	13541	21534	54231	41254	25411

   a) Qual deve ser o valor de **n** se o método de *hashing* usado for Divisão por Número Primo?

   b) Com o **n** encontrado em (a), faça o *hash* das chaves em uma tabela *hash* com tamanho **n** e endereçamento de 0 a n-1. No caso de colisão, utilize verificação linear.

   c) Utilizando desdobramento ao meio pelos extremos que resulte em endereços de três dígitos, quais são os valores *hash*?

2. Usando *Hashing* Extensível, armazene as chaves abaixo em uma tabela *hash* na ordem apresentada. Utilize primeiro os dígitos mais à esquerda. Utilize mais dígitos quando necessário. Inicie a tabela *hash* com tamanho 2. Apresente a tabela a cada nova extensão.

Chave	Valor *Hash*	Equivalente Binário
Banana	2	010
Melon	5	101
Raspberry	1	001
Kiwi	6	110
Orange	7	111
Apple	0	000

5.1. Exercícios para programar

1. Crie um programa Java que implemente o desdobramento ao meio por substituição. Este programa recebe os parâmetros **k**, **dk** e **da**, onde **k** é a chave para fazer *hash*, **dk** é o número de dígitos na chave e **da** é o número de dígitos no endereço. O programa deve retornar o endereço com **da** dígitos.
2. Escreva um programa Java completo que usa o método da divisão como método de *hash* e verificação linear como técnica de resolução de colisão.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.