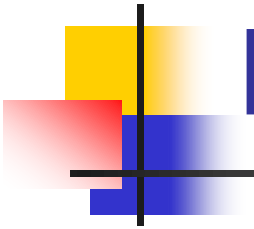
Hashing (Espalhamento)

Estrutura de Dados II

Parte 2

Hashing Extensível

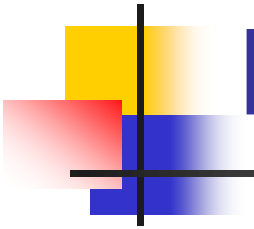
- O hashing estático não se adapta bem a arquivos dinâmicos, que crescem e diminuem com o tempo.
- O problema é que, a medida que o arquivo aumenta, torna-se mais frequente a necessidade de procurar por registros fora do seu endereço base, o que deteriora o desempenho.
- Nesse caso, ao invés de 1 acesso (o ideal que pode ser atingido com o espalhamento), pode-se precisar de mais acessos do que seriam necessários se fosse mantido um índice em árvore-B adequado para o arquivo.



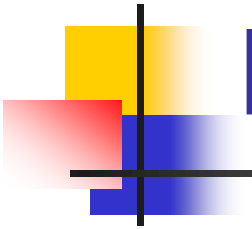
Hashing Extensível

- O hashing extensível é uma das várias técnicas que permitem um auto ajuste do espaço de endereçamento do espalhamento, para que este aumente e diminua de acordo com as variações no tamanho do arquivo.
- Uma vez que o tamanho do espaço de endereçamento pode crescer junto com o arquivo, é possível usar espalhamento sem área de overflow (para acomodar chaves com mesmo endereço base) mesmo que o arquivo aumente muito em tamanho.
- Outros métodos de espalhamento, além do extensível, são o **dinâmico** e o **linear**, apresentados no livro.

Hashing Extensível



- A ideia chave do espalhamento extensível é combinar o espalhamento convencional com uma técnica de recuperação de informações (information re**trie**val) denominada trie.
- Uma trie (também conhecida como radix searching tree) é uma árvore de busca na qual o fator de sub-divisão (branching factor), ou número máximo de filhos por nó, é igual ao número de símbolos do alfabeto.



Hashing Extensível

- Trie: onde são utilizadas? Problema de busca: conjunto de chaves K e chave x a localizar em K
- Assumido até agora que: (i) chaves são elementos indivisíveis; (ii) chaves têm mesmo tamanho
- E se a busca consistir em frases ou palavras em um texto?
  - Busca digital, por meio de uma estrutura de árvore digital (Trie, Patricia)

Hashing Extensível

- Suponha que desejamos construir uma trie para armazenar as chaves able, abrahms, adams, anderson, andrews, baird.

A idéia é que a busca prossiga letra por letra ao longo da chave.

Como existem 26 letras no alfabeto, tem-se uma árvore com radix 26 (o número potencial de sub-divisões a partir de cada nó é 26).

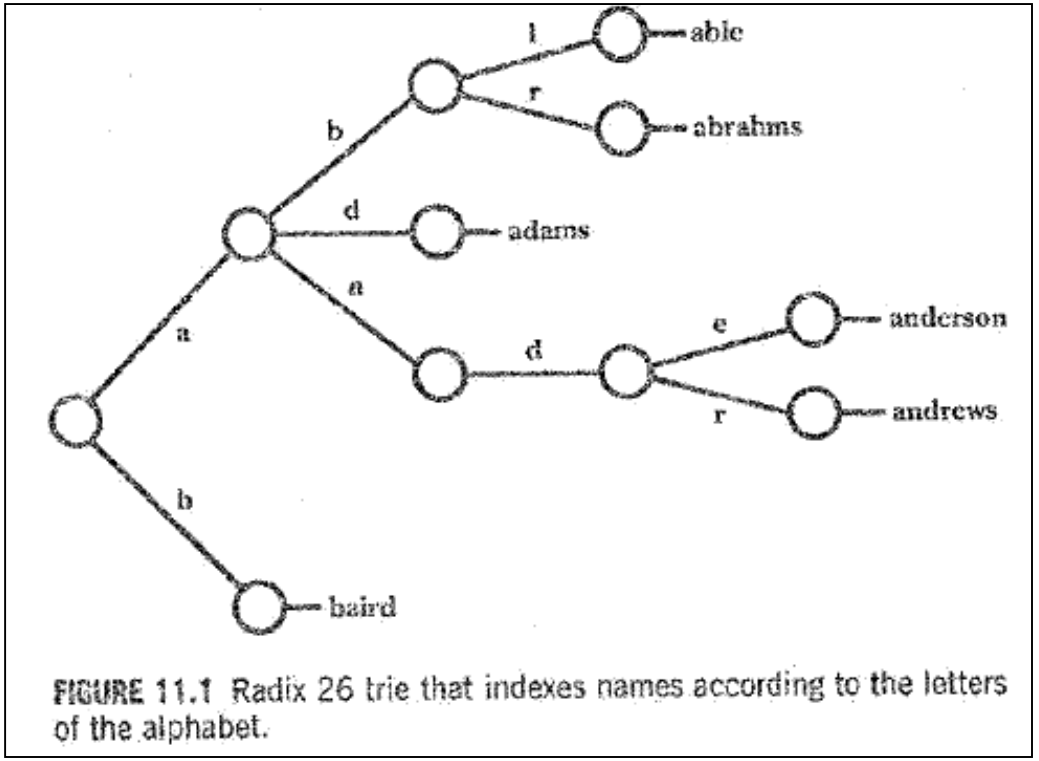
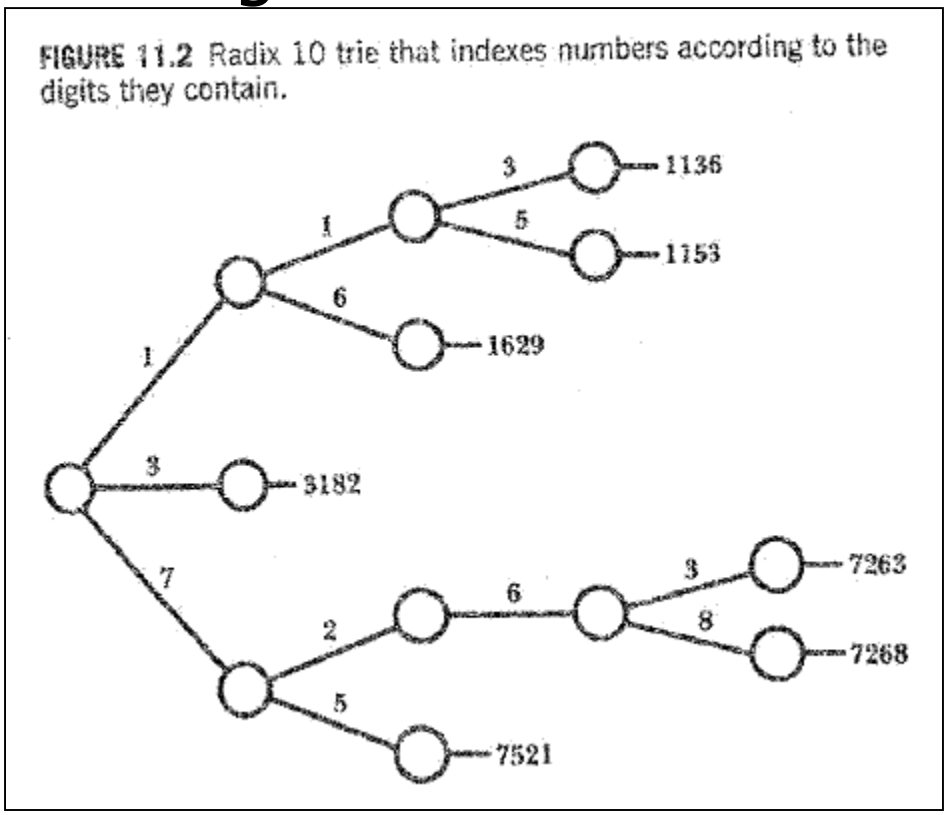


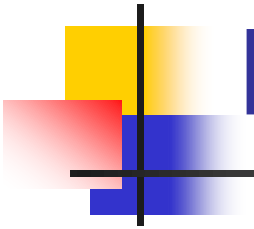
FIGURE 11.1 Radix 26 trie that indexes names according to the letters of the alphabet.

Hashing Extensível

- Se estivéssemos trabalhando com números, seria 10, pois temos 10 dígitos entre 0 e 9.

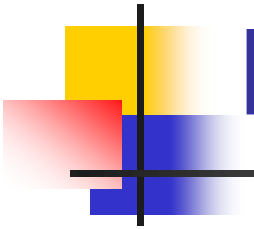
Observe que, na busca em uma trie, é possível que precisemos usar apenas parte da chave. Na verdade, usamos apenas a informação necessária para identificá-la.





Hashing Extensível

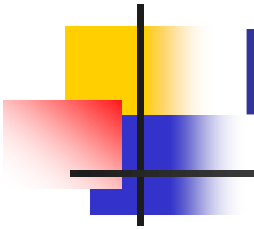
- Aplicações
  - Dicionários (telefone celular)
  - Corretores Ortográficos
  - Auto-preenchimento
    - browsers, e-mail, linguagens de programação



Hashing Extensível

- Maior utilização de árvores digitais é caso binário
  - Chaves/códigos binários são mais empregados em computação

Transformando a Trie em um Diretório



- O espalhamento extensível usa tries de radix 2 como índices: as decisões durante a busca por uma certa chave são baseadas em uma análise bit-a-bit da chave.
- Além disso, como estamos recuperando informações da memória secundária, trabalha-se com cestos contendo várias chaves, e não com chaves individuais.

Transformando a Trie em um Diretório

- A figura ilustra uma trie que endereça 3 cestos A, B e C.

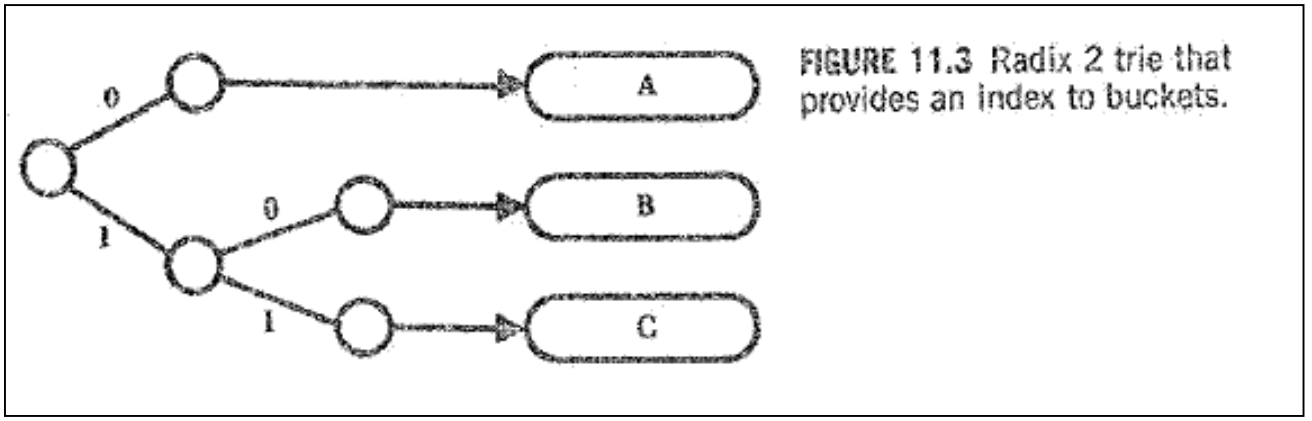
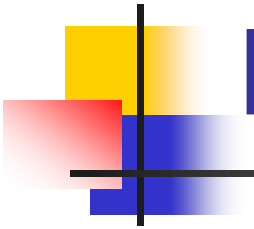


FIGURE 11.3 Radix 2 trie that provides an index to buckets.

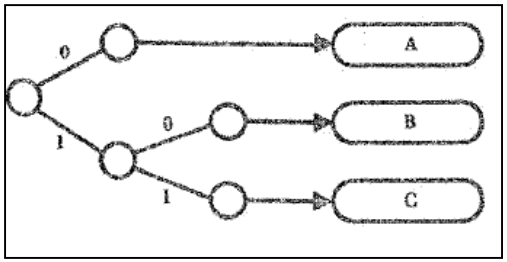
Transformando a Trie em um Diretório



- Como representar a trie?
- Se for mantida como uma árvore, será necessário fazer várias comparações para descer ao longo de sua estrutura.
  - Como vimos anteriormente, inviável para armazenamento secundário.
- Assim, o que se faz é transformar a trie em um vetor de registros consecutivos, formando um diretório de endereços de espalhamento e ponteiros para os cestos associados.

Transformando a Trie em um Diretório

- O primeiro passo para transformar a trie em um vetor é estendê-la de modo que se torne uma árvore binária completa (todas as folhas estão no mesmo nível).
- Uma vez estando "completa", a trie pode ser representada pelo vetor.



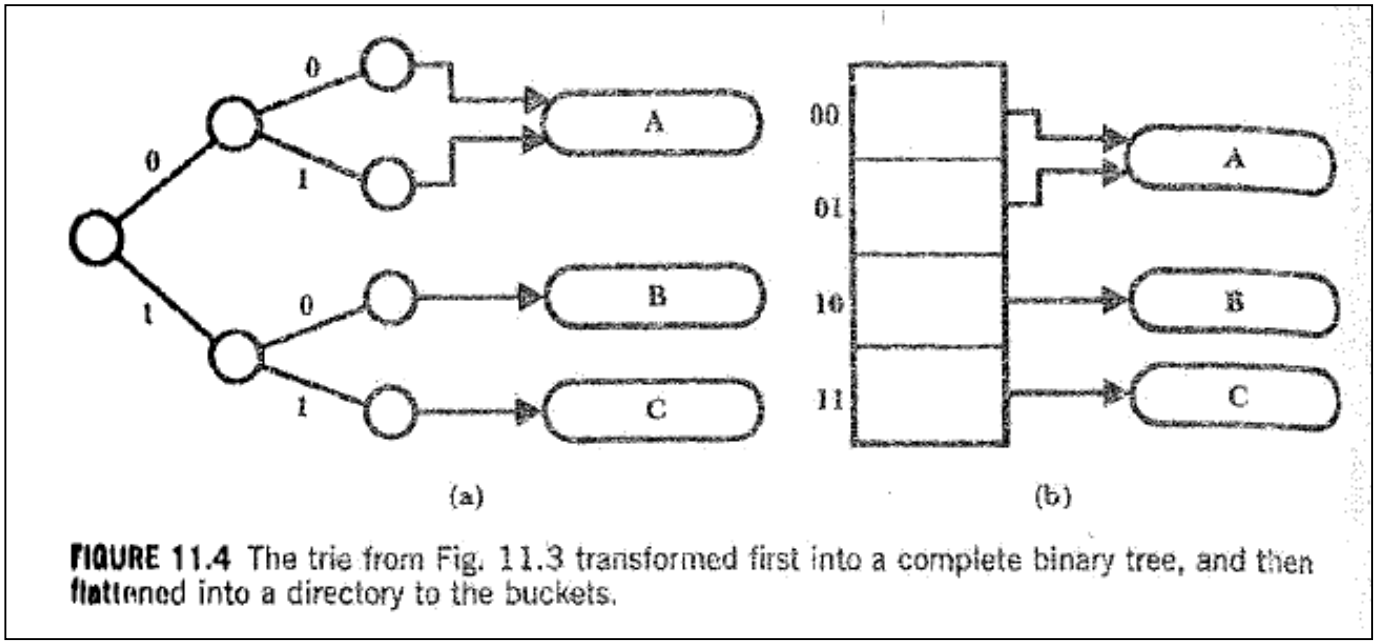


FIGURE 11.4 The trie from Fig. 11.3 transformed first into a complete binary tree, and then flattened into a directory to the buckets.

Transformando a Trie em um Diretório

- Agora temos uma estrutura que fornece o acesso direto associado ao processo de espalhamento: dado um endereço começando com os bits 10, o diretório na posição 10 (base 2) do vetor nos dá um ponteiro para o cesto associado.

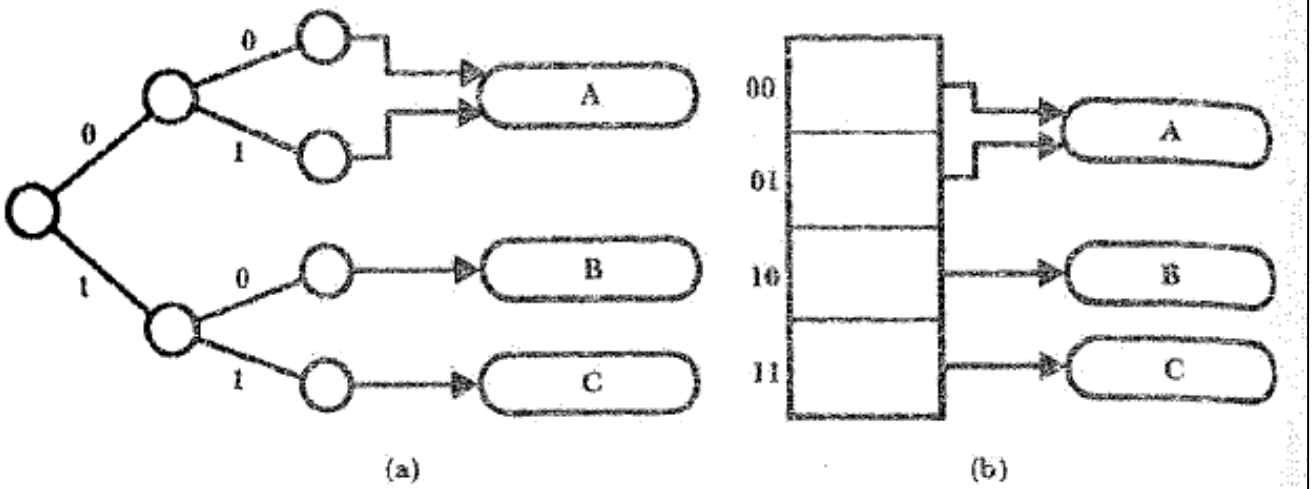
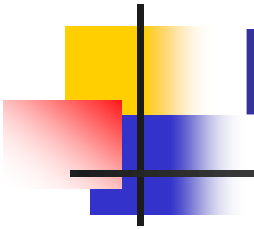


FIGURE 11.4 The trie from Fig. 11.3 transformed first into a complete binary tree, and then flattened into a directory to the buckets.

Divisão para Tratar Overflow



- Se um registro precisa ser inserido e não existe espaço no cesto que é o seu endereço base, o cesto é dividido (splitting), aumentando o espaço de endereçamento.
  - Diferente dos anteriores que respondem ao overflow criando longas sequências de registros que devem ser pesquisadas linearmente.
- Utiliza-se um bit adicional dos valores de espalhamento das chaves para dividir os registros entre dois cestos.
  - Se o novo espaço de endereçamento resultante da consideração desse novo bit já estava previsto no diretório, nenhuma alteração adicional se faz necessária.
  - Caso contrário, é necessário dobrar o espaço de endereçamento do diretório para acomodar o novo bit.

Divisão para Tratar Overflow

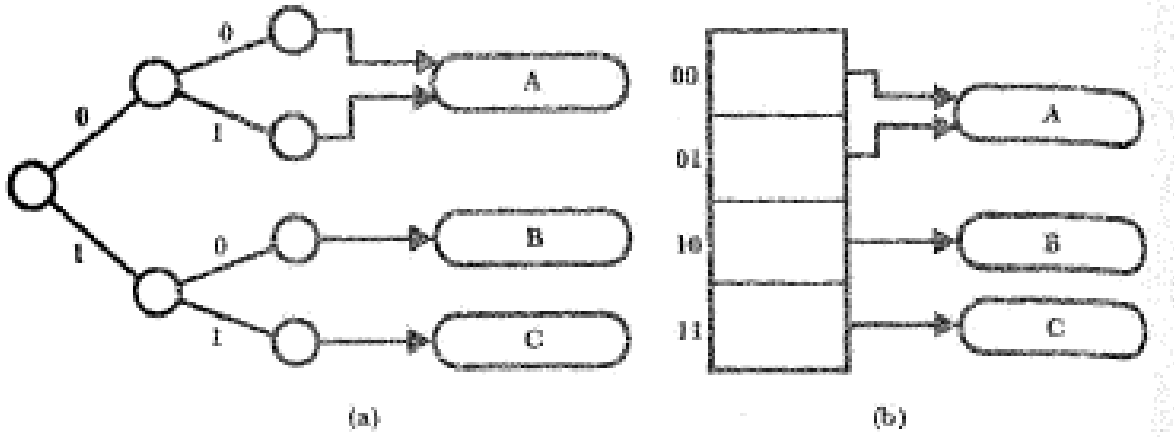


FIGURE 11.4 The trie from Fig. 11.3 transformed first into a complete binery tree, and then flattened into a directory to the buckets.

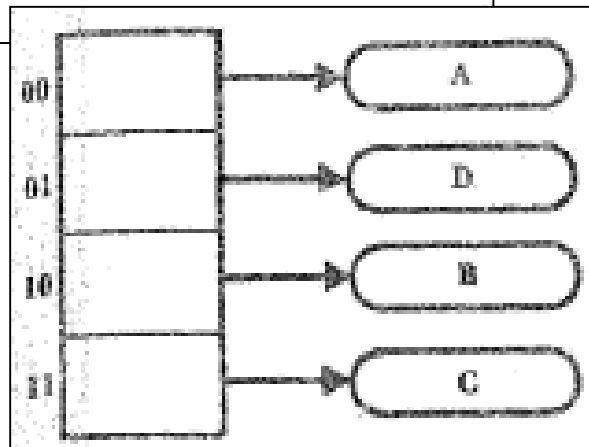


FIGURE 11.5 The directory from Fig. 11.4(b) after bucket A overflows.

Divisão para Tratar Overflow

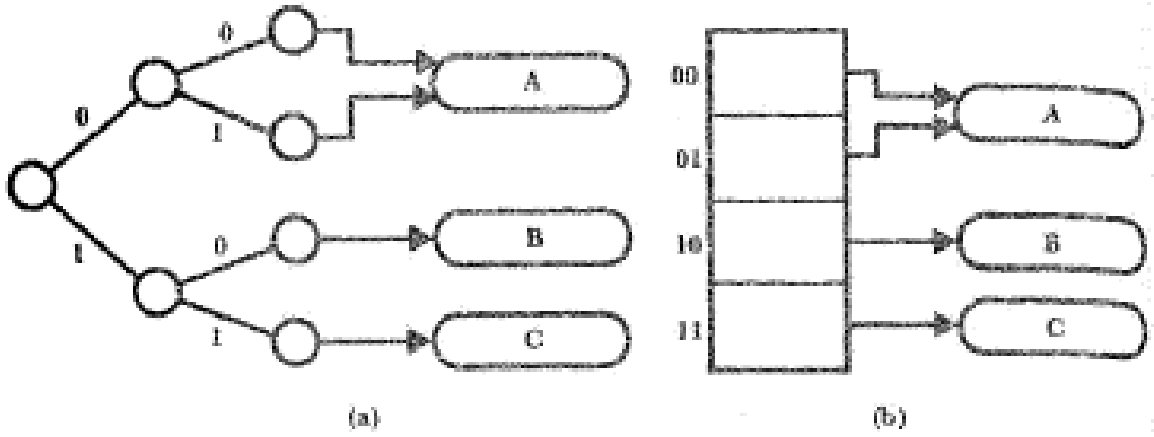


FIGURE 11.4 The trie from Fig. 11.3 transformed first into a complete binary tree, and then flattened into a directory to the buckets.

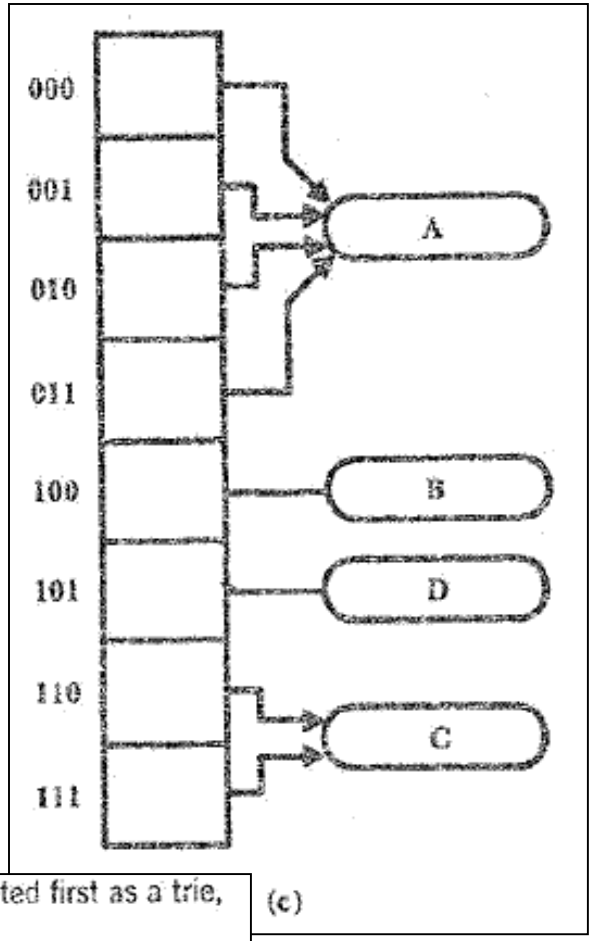
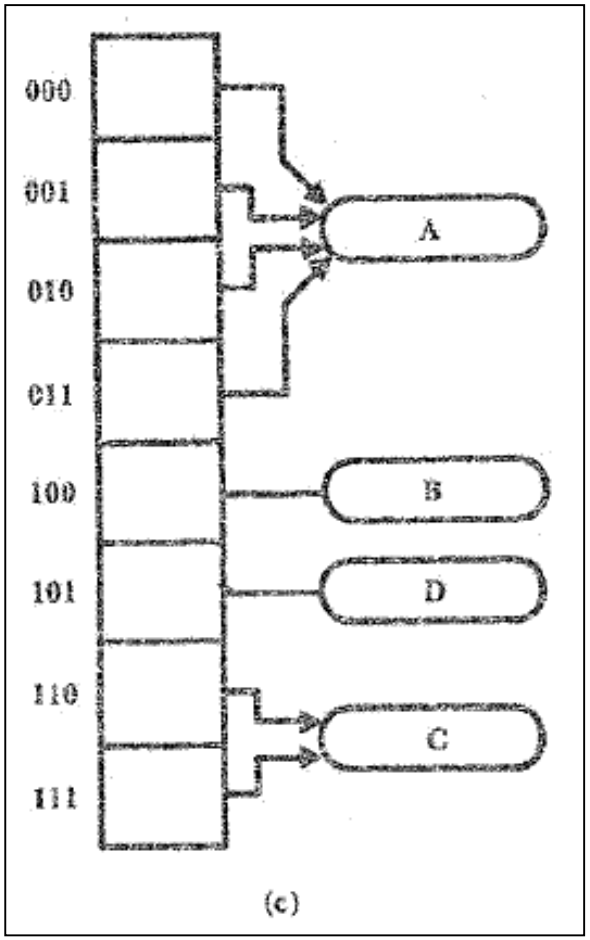


FIGURE 11.6 The results of an overflow of bucket *B* in Fig. 11.4(b), represented first as a trie, then as a complete binary tree, and finally as a directory.

Divisão para Tratar Overflow



bill	0000	0011	0110	1100
lee	0000	0100	0010	1000
pauline	0000	1111	0110	0101
alan	0100	1100	1010	0010
julie	0010	1110	0000	1001
mike	0000	0111	0100	1101
elizabeth	0010	1100	0110	1010
mark	0000	1010	0000	0111

FIGURE 11.8 Output from the *hash* function for a number of keys.

Implementação
Função Hashing

```
FUNCTION hash(KEY)

    set SUM to 0
    set J to 0
    set LEN to the length of the key

    if LEN is odd, concatenate a blank to the key
         to make the length even

    while (J < LEN)
         SUM := (SUM + 100*KEY[J] + KEY[J+1]) mod 19937
         increment J by 2
    endwhile

    return SUM

end FUNCTION
```

FIGURE 11.7 Function *hash(KEY)* returns an integer hash value for KEY for a 15-bit address space.

Interpretação de trás para frente (menos significativo para o mais significativo).

A cada passo retorna uma profundidade.

b i l l
98 105 108 108
(9800+105) + (10800+108)
9905 + 10908 = 20813
20813 mod 19937 = 876
876 = 0000 0011 0110 1100

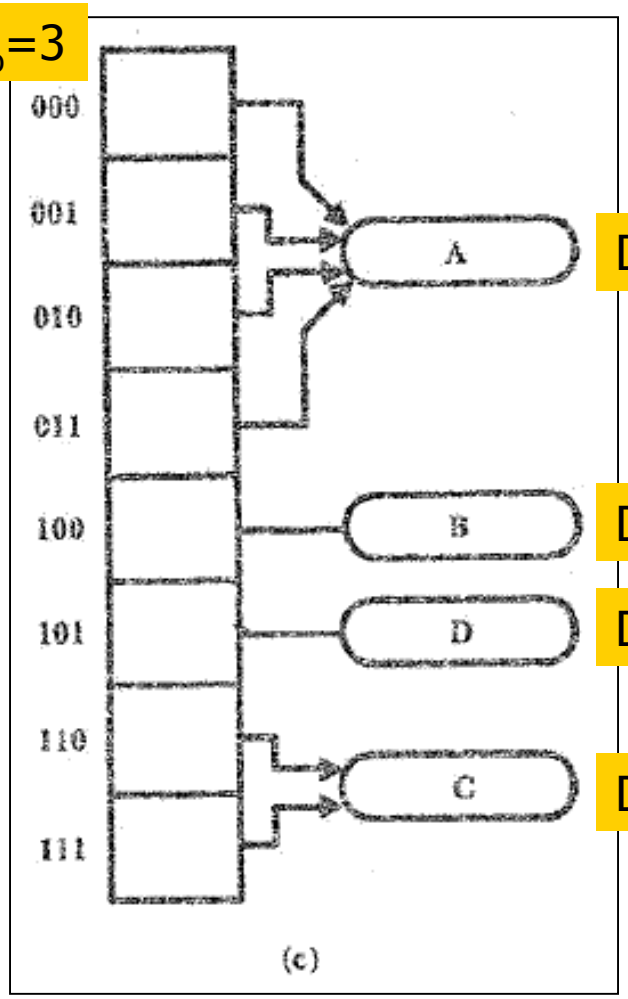
bill	0000 0011 0110 1100
lee	0000 0100 0010 1000
pauline	0000 1111 0110 0101
alan	0100 1100 1010 0010
julie	0010 1110 0000 1001
mike	0000 0111 0100 1101
elizabeth	0010 1100 0110 1010
mark	0000 1010 0000 0111

FIGURE 11.8 Output from the *hash* function for a number of keys.

Implementação
Estrutura Diretório/Cesto

$D_{Diretório}=3$

RRN do Cesto



$D_{Cesto:A}=1$

$D_{Cesto:B}=3$

$D_{Cesto:D}=3$

$D_{Cesto:C}=2$

Profundidade	Contador Chaves	Chaves

Tamanho da tabela sempre cresce como potência de 2!!!

Implementação
Estrutura Diretório/Cesto

Tamanho da tabela sempre cresce como potência de 2!!!

$D_{Diretório}=3$

RRN do Cesto

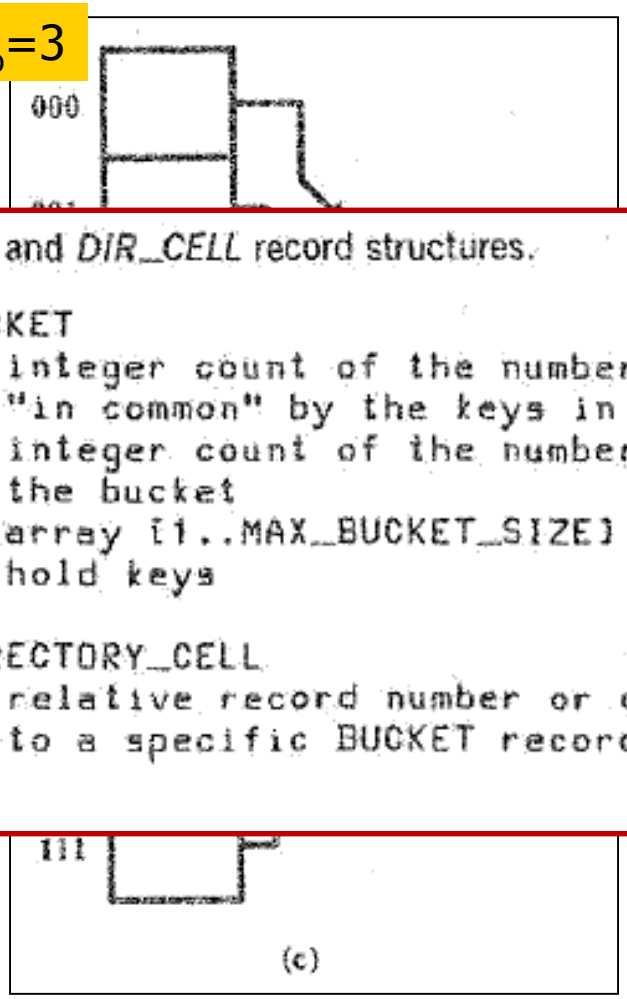


FIGURE 11.10 *BUCKET* and *DIR_CELL* record structures.

```
Record Type: BUCKET
    DEPTH        integer count of the number of bits used
                 "in common" by the keys in this bucket
    COUNT        integer count of the number of keys in
                 the bucket
    KEY[]        array [1..MAX_BUCKET_SIZE] of strings to
                 hold keys

Record Type: DIRECTORY_CELL
    BUCKET_REF   relative record number or other reference
                 to a specific BUCKET record on disk
```

Profundidade	Contador Chaves	Chaves

Inserção

Function: Find

address = make_address(Key,DIR_DEPTH)
Bucket = Bucket RRN in address
Search Key in Bucket
if (found) return (1) else return (0)

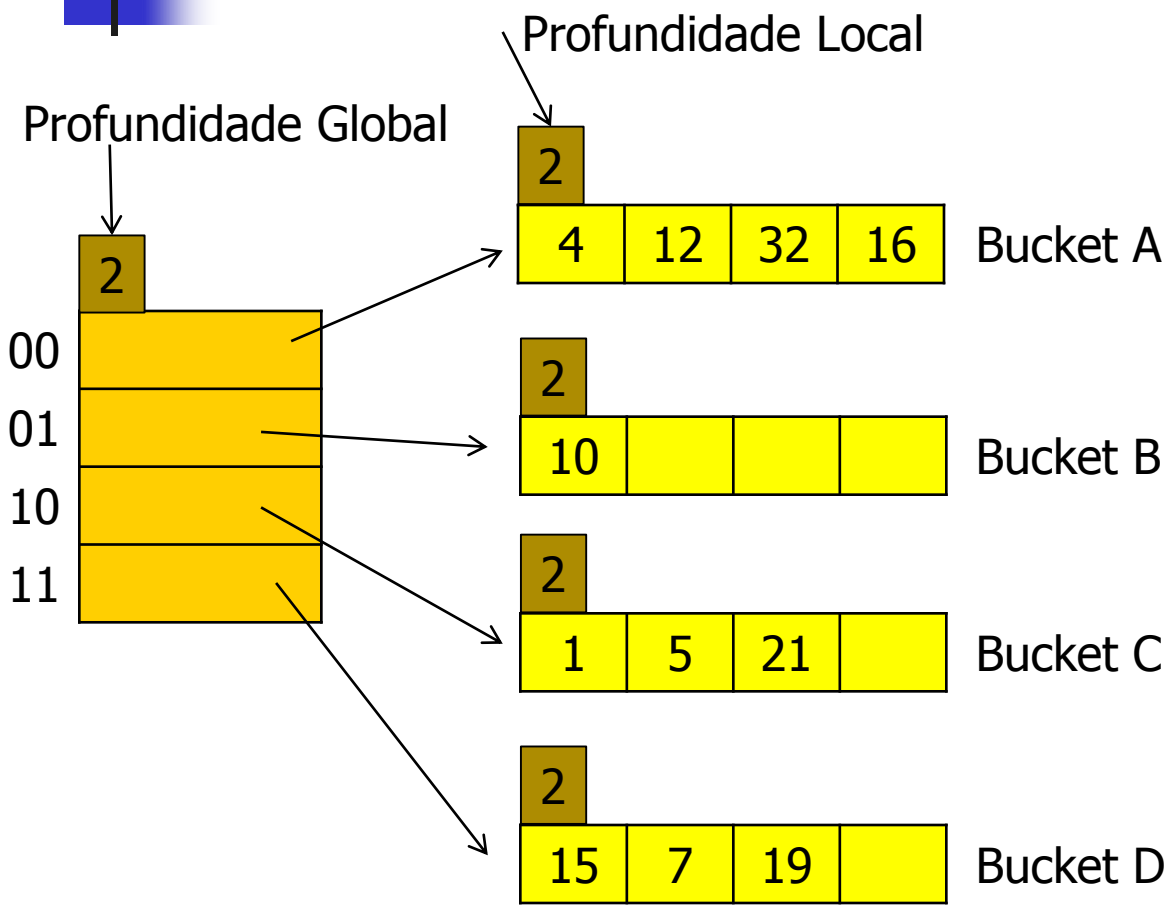
Function: Insert

Find(Bucket,Key)
if (Bucket.Count < Max_Bucket_Size)
    add Key
else
    {
    Split(Bucket)
    Insert(Key) //Recursivo
    }

Function: Split

if (Bucket.Depth == Dir_Depth)
    Dir_double()
allocate NewBucket
Find_range_NewB(Bucket,Start,End)
Update(NewBucket, Start,End)
Increment Bucket.Depth
NewBucket.Depth = Bucket.Depth
Redistribute the Keys between the two bucktes

Exemplo



04 = 0000 01**00**
12 = 0000 11**00**
32 = 0010 00**00**
16 = 0001 00**00**

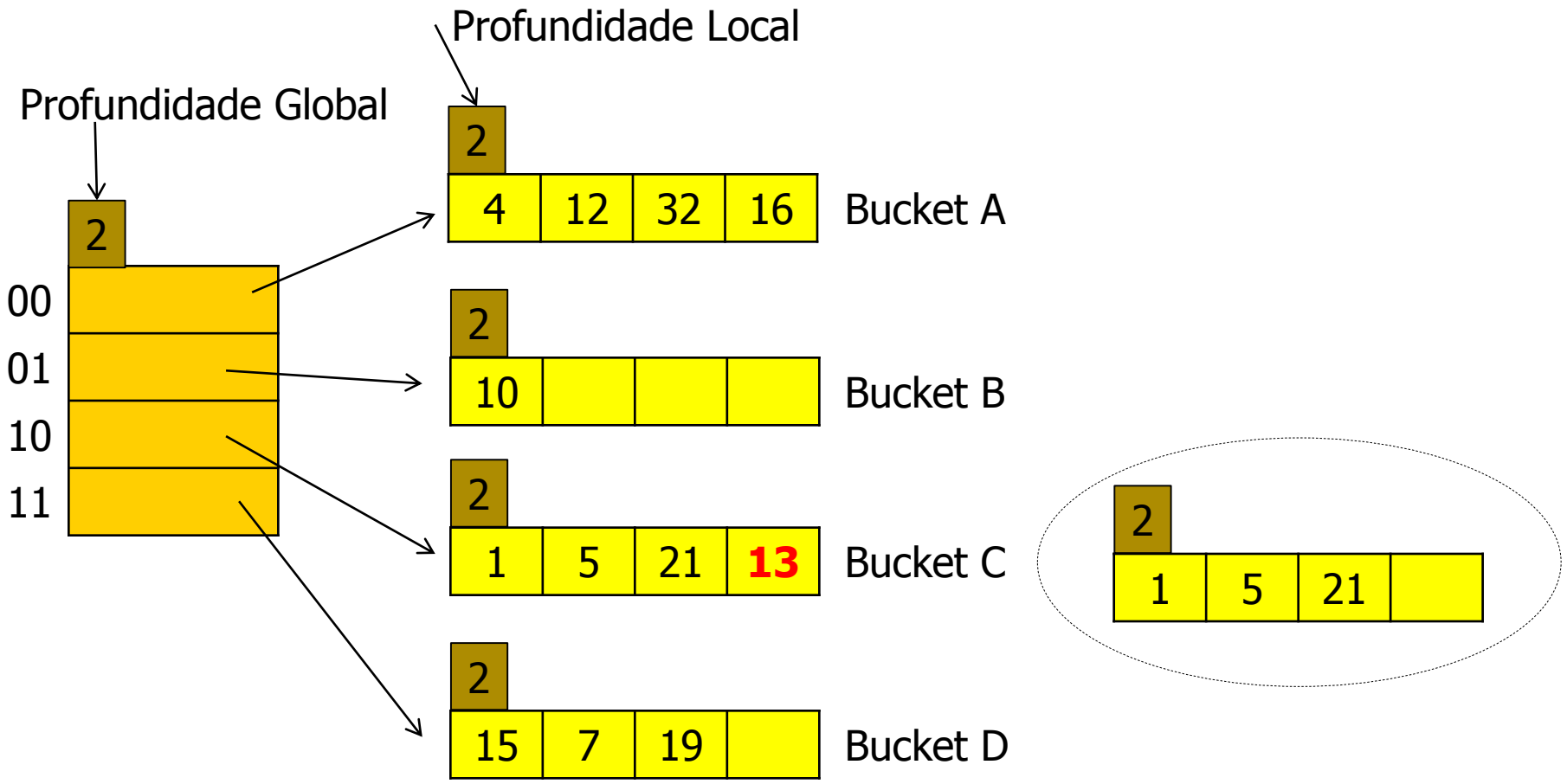
01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**

10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**

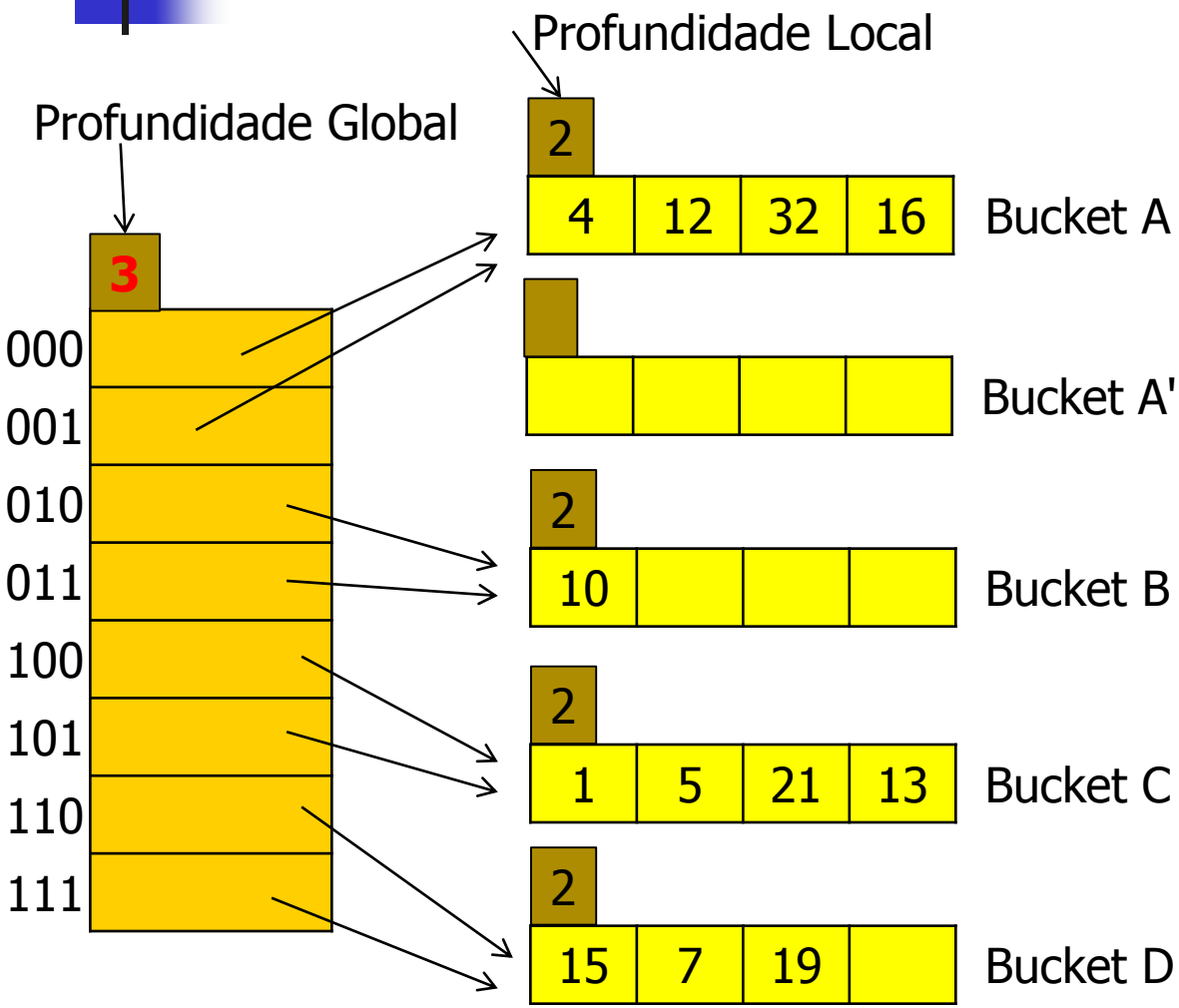
Exemplo

Inserir 13 = 0000 11**01**



Exemplo

Duplica e Aloca



Inserir 20 = 0001 01**00**

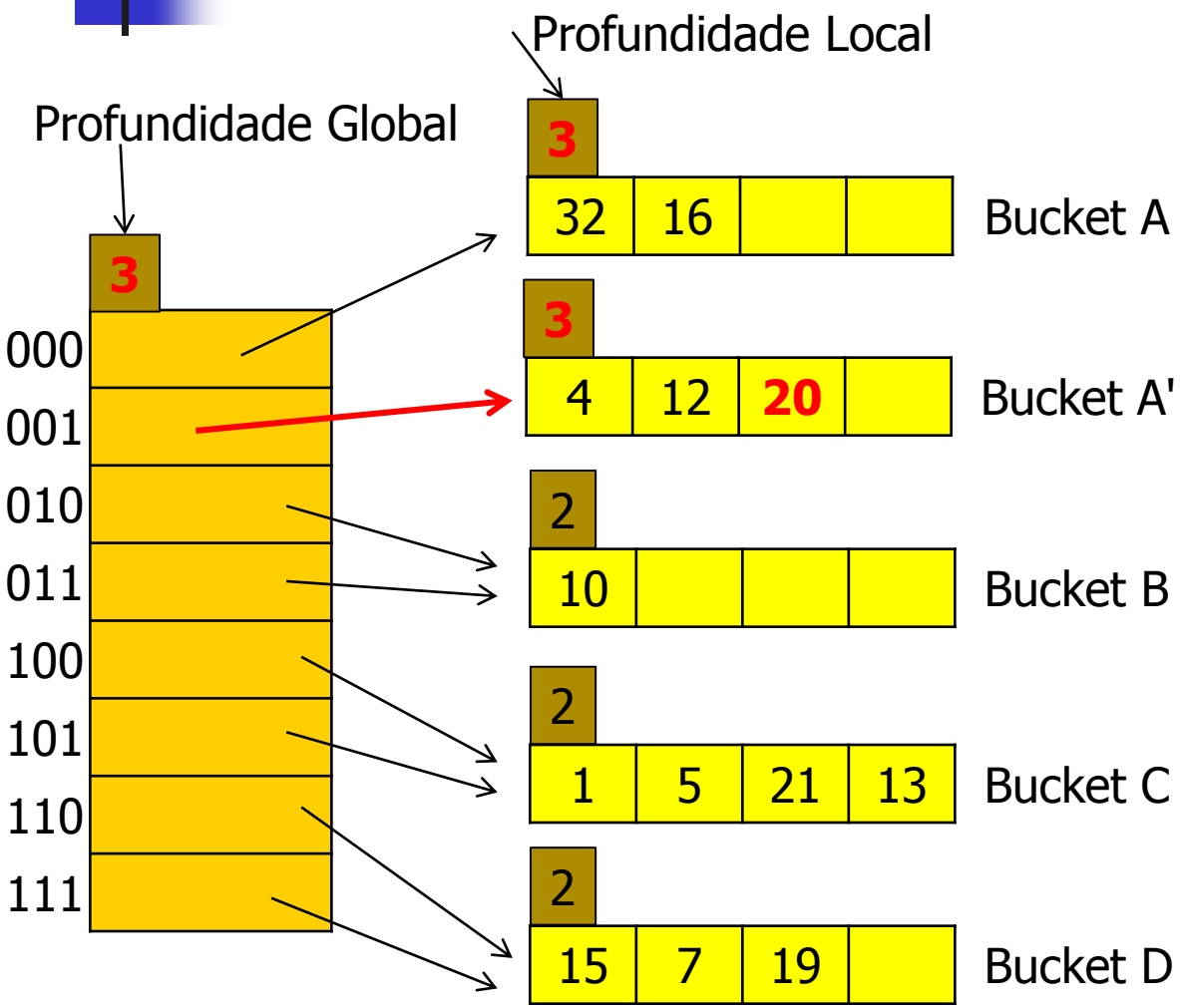
04 = 0000 01**00**
12 = 0000 11**00**
32 = 0010 00**00**
16 = 0001 00**00**

01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**

10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**

Exemplo



Atualiza, Incrementa, Distribui e Insere

04 = 0000 0**100**
12 = 0000 1**100**
20 = 0001 0**100**

32 = 0010 0**000**
16 = 0001 0**000**

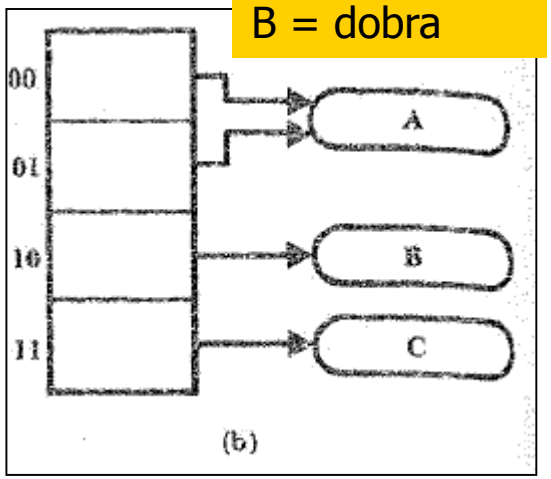
01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**
13 = 0000 11**01**

10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**

Inserção

A = não dobra
B = dobra



Function: Split

if (Bucket.Depth == Dir_Depth)

Dir_double()

allocate NewBucket

Find_range_NewB(Bucket,Start,End)

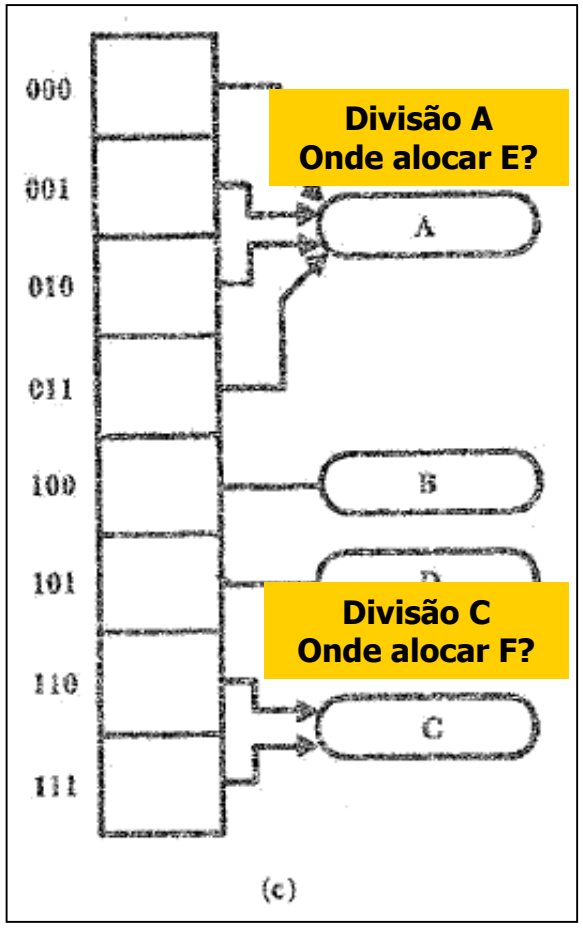
Update(NewBucket, Start,End)

Increment Bucket.Depth

NewBucket.Depth = Bucket.Depth

Redistribute the Keys between the two bucktes

Inserção



Function: Split

if (Bucket.Depth == Dir_Depth)

Dir_double()

allocate NewBucket

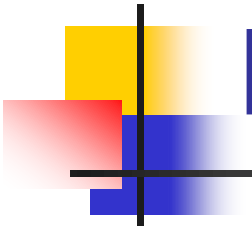
Find_range_NewB(Bucket,Start,End)

Update(NewBucket, Start,End)

Increment Bucket.Depth

NewBucket.Depth = Bucket.Depth

Redistribute the Keys between the two bucktes



Remoção

Function: Delete

Find(Bucket,Key)

Remove and Decrement Bucket.Count

BCombine(Bucket)

Function: BCombine

Find_Buddy_Bucket(Bucket)

if (Buddy found)

if (Buddy.Count + Bucket.Count <= MAX_Bucket_Size)

Combine(Bucket,Buddy) and Decrement Bucket.Depth

Free Buddy Bucket

Reassign the Directory for Buddy to point Bucket

if (*Dir_Try_Colapse*()) //$\text{Bucket.Depth}_{\text{TODOS}}$ < Dir_Depth

BCombine(Bucket) //Recursivo

Remoção (cont.)

Function: Find_Buddy_Bucket

```
if (Dir_Depth == 0)
        return NO
if (Bucket.Depth < Dir_Depth) //Se não for "folha", i.e., não estiver no
   return NO                          //último nível, não tem vizinho
return Buddy_Bucket //Endereço do atual com o último bit trocado
```

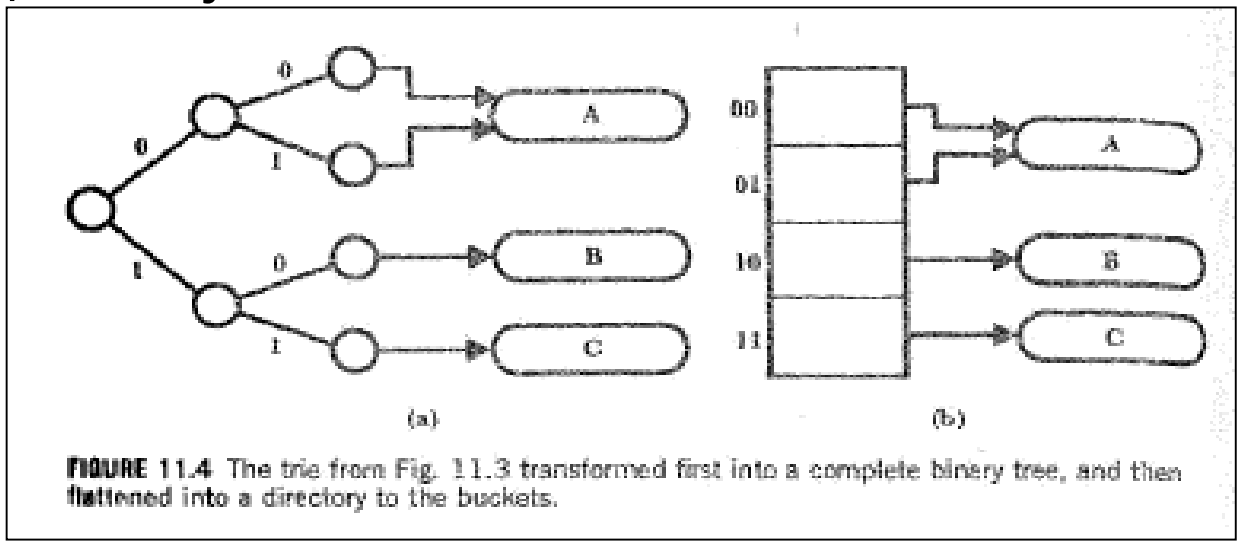
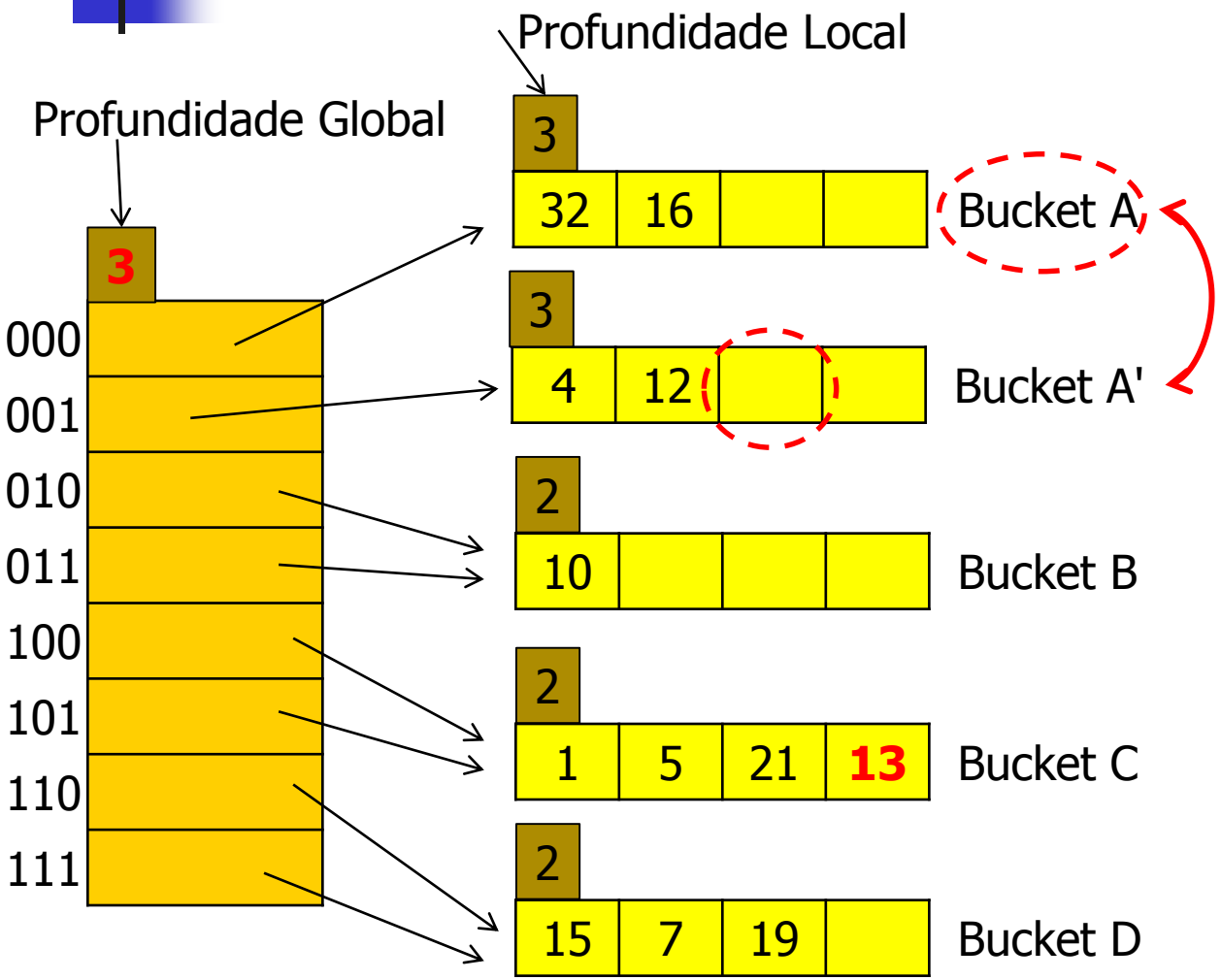


FIGURE 11.4 The trie from Fig. 11.3 transformed first into a complete binary tree, and then flattened into a directory to the buckets.

Exemplo



Deleta e Encontra Buddy

Deletar 20 = 0001 0**100**

04 = 0000 0**100**
12 = 0000 1**100**
20 = 0001 0**100**

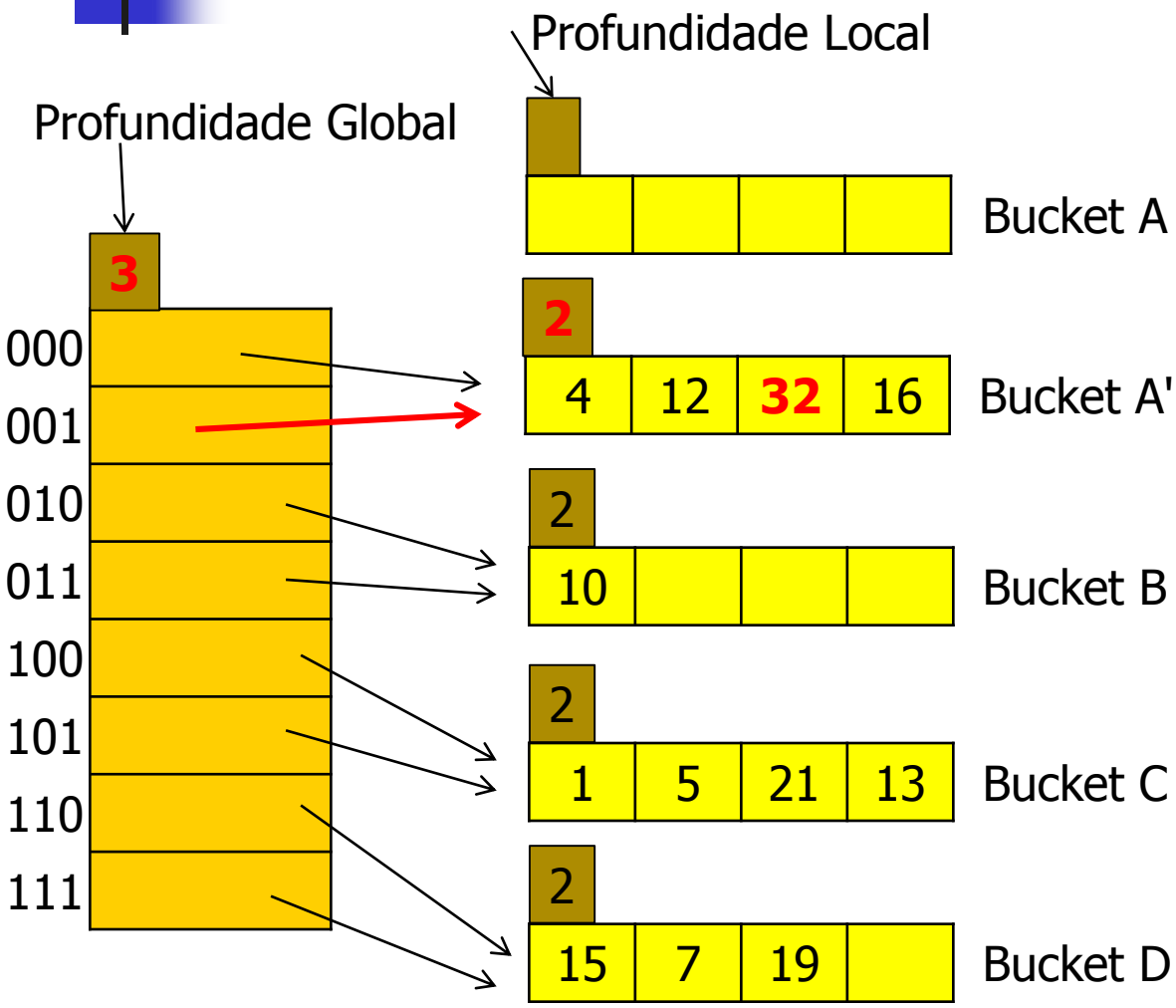
32 = 0010 0**000**
16 = 0001 0**000**

01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**
13 = 0000 11**01**

10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**

Exemplo



Combina, Decrementa, Libera e Atualiza Ponteiro

Deletar 20 = 0001 0**100**

04 = 0000 01**00**
12 = 0000 11**00**
32 = 0010 00**00**
16 = 0001 00**00**

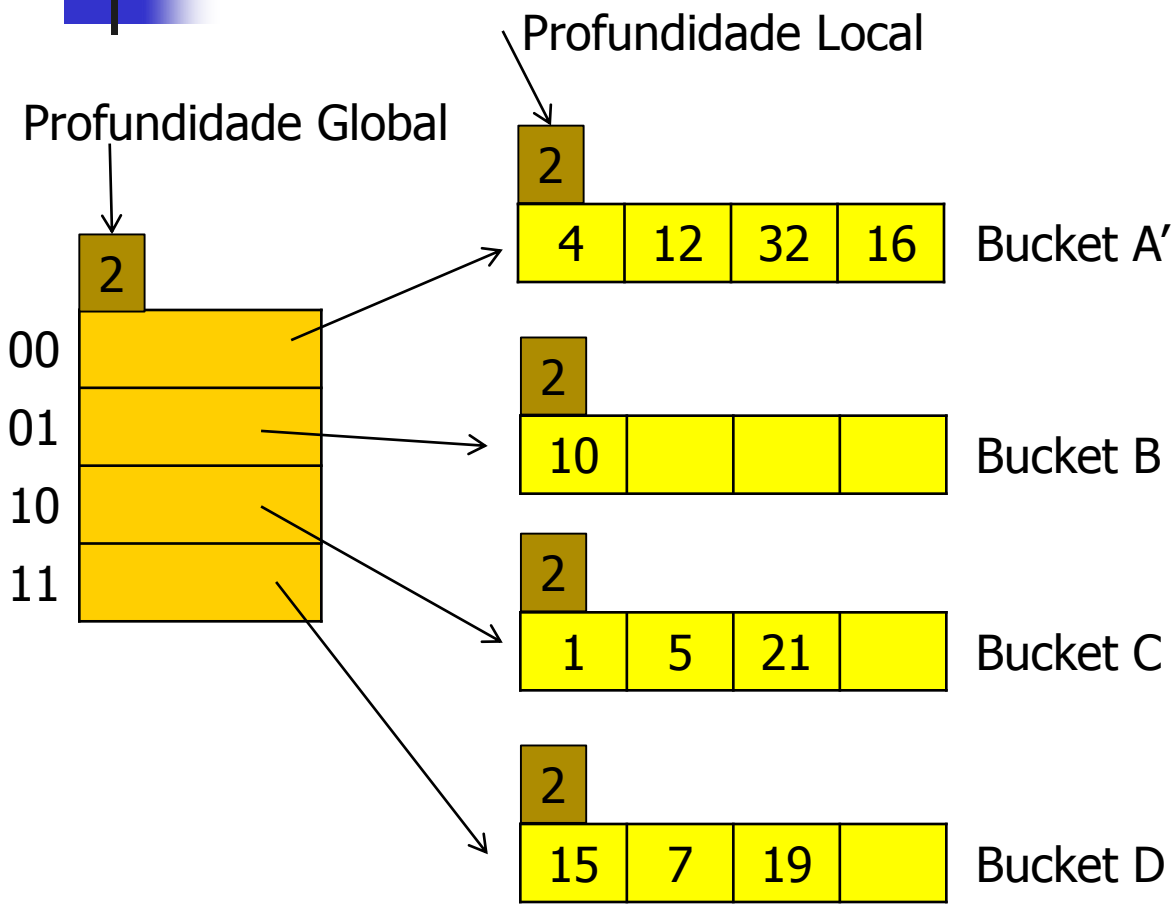
01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**

10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**

Exemplo

Une Diretório



Deletar 20 = 0001 0**100**

04 = 0000 01**00**
12 = 0000 11**00**
32 = 0010 00**00**
16 = 0001 00**00**

01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**

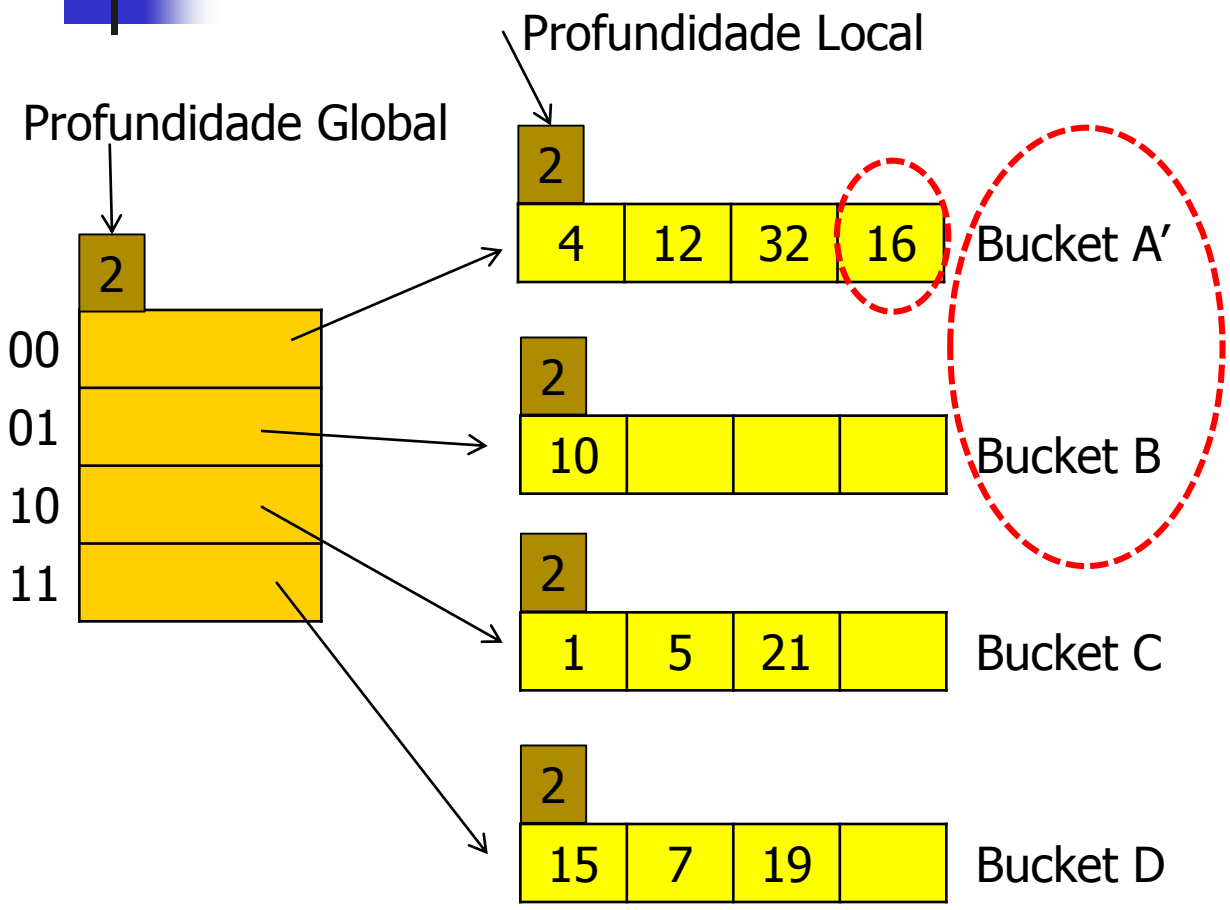
10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**

Exemplo

Se foi possível realizar o colapso, pode ter surgido um novo "amigo": assim é necessário uma chamada recursiva a *BCombine*(Bucket).

Une Diretório



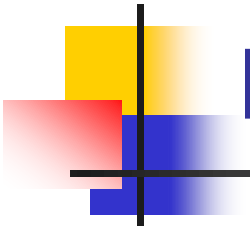
Deletar 20 = 0001 0**100**

04 = 0000 01**00**
12 = 0000 11**00**
32 = 0010 00**00**
16 = 0001 00**00**

01 = 0000 00**01**
05 = 0000 01**01**
21 = 0001 01**01**

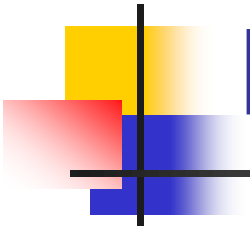
10 = 0000 10**10**

15 = 0000 11**11**
07 = 0000 01**11**
19 = 0001 00**11**



Desempenho do Hashing Extensível

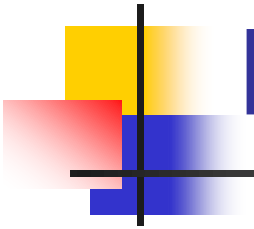
- Se o diretório puder ser mantido em RAM, é necessário apenas um seek.
- Se o diretório precisa ser alocado em disco, tem-se 2 seeks no pior caso.
- A utilização do espaço (densidade de ocupação) alocado aos cestos é de aproximadamente 69%.
- Desvantagens:
  - Complexidade para gerenciar o aumento do diretório e a divisão dos cestos.
  - Podem existir sequências de inserções que façam a tabela crescer rapidamente tendo, contudo, um número pequeno de registros.



Exercício

- Insira as seguintes chaves na sequência dada: 0001, 1001, 1100, 0000, 0111 e 1000
- Em seguida elimine as chaves 1100 e 0111

Considere Buckets de tamanho 2



Bibliografia

- FOLK, M.; ZOELLICK, B. File Structures, Second Edition. Addison-Wesley, 1992.
- FOLK, M.; ZOELLICK, B.; RICCARDI, G. File Structures: An Object-Oriented Approach Using C++. Third Edition. Addison-Wesley, 1998.
- Pereira, S. L. Estruturas de dados fundamentais : conceitos e aplicações. 2000.
- Ziviani, N. Projeto de algoritmos: com implementações em Pascal e C. 2004.
- Tenenbaum, A. M.; Langsan, Y.; Augenstein, M. Estruturas de dados usando C. 1995.
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. Introduction to Algorithms. The MIT Press. Mc Graw-Hill, 1990.
- SZWARCFITER, J. L.; MARKENZON, L. Estruturas de Dados e seus Algoritmos. LTC, 1994.