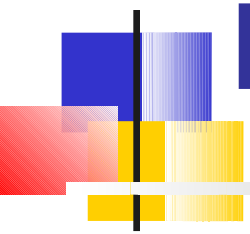


Hashing Externo

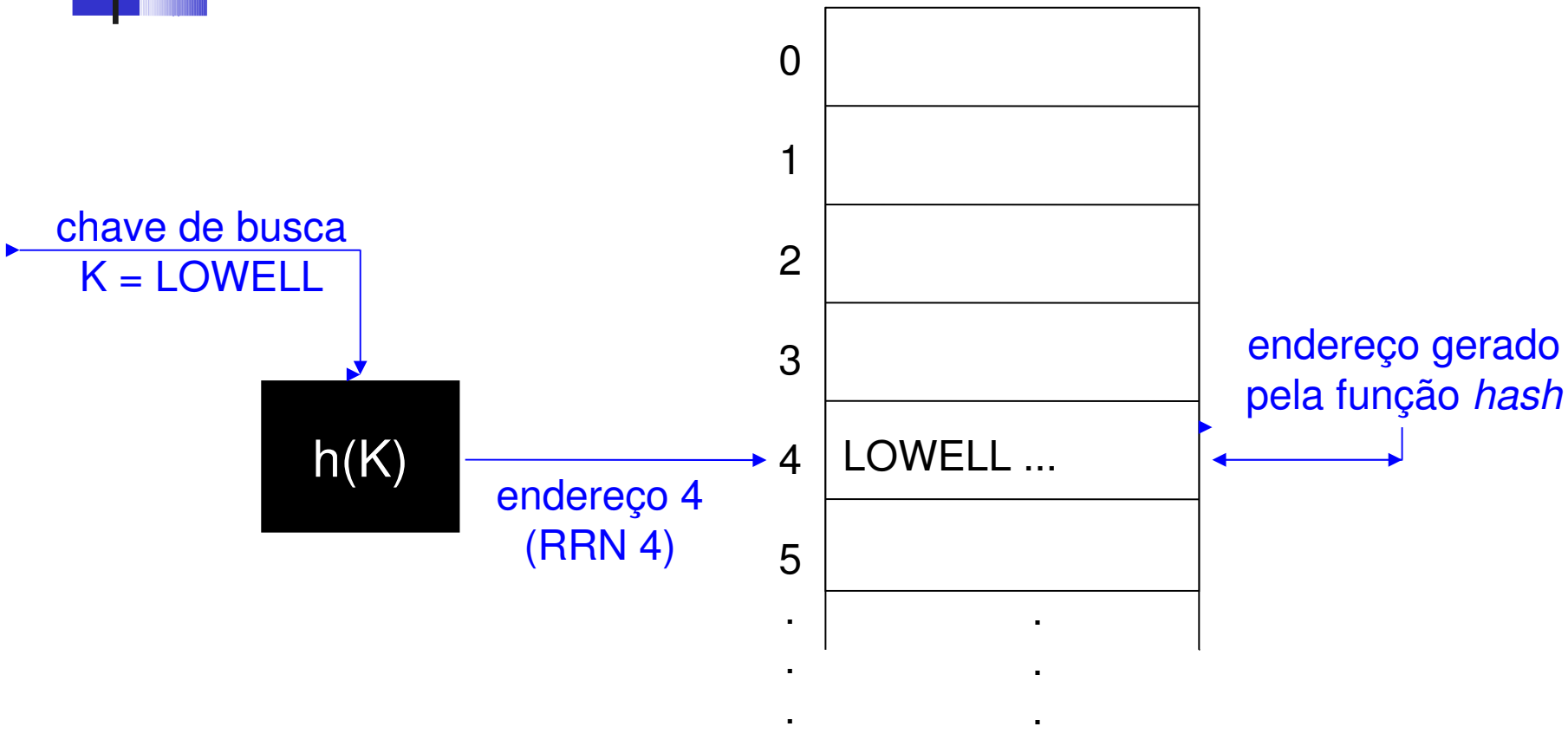
SCC-503 Algoritmos e Estruturas de Dados II

Thiago A. S. Pardo

M.C.F. de Oliveira

Cristina Ciferri

Hashing



espaço de endereçamento: registros de tamanho fixo
→ endereçamento de RRNs

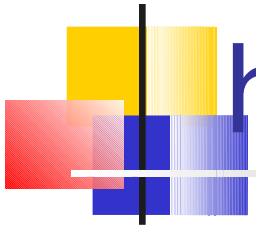
Hashing

- Função *hash*
  - caixa preta que produz um endereço toda vez que uma chave de busca é passada como parâmetro

- Endereço resultante
  - usado para armazenamento e recuperação de registros no arquivo de dados

  - Nomenclatura
    - h(K) $\rightarrow$ endereço
      - K: chave de busca

Organização de índices hashing



- único arquivo
  - Os dados e o índice *hashing* ficam no mesmo arquivo
- dois arquivos
  - Os dados ficam em um arquivo e o índice hashing das chaves fica em outro

Hashing

- Hashing & indexação

  - Semelhança
    - associação de uma chave de busca a um endereço de registro

  - Diferença (*hashing*)
    - endereço gerado é (teoricamente) aleatório
      - não existe relação óbvia entre a chave e a localização do registro no arquivo de dados
    - duas chaves diferentes podem ser transformadas para o mesmo endereço

colisão

Exemplo

nome	código ASC II 1ª e 2ª letras		produto	endereço gerado
BALL	66	65	66 x 65 = 4.290	290
LOWELL	76	79	76 x 79 = 6.004	004
TREE	84	82	84 x 82 = 6.888	888
OLIVER				

Exemplo

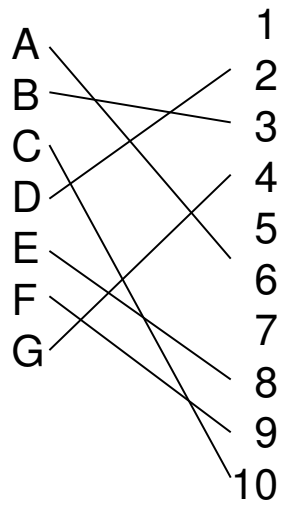
nome	código ASC II 1ª e 2ª letras		produto	endereço gerado
BALL	66	65	66 x 65 = 4.290	290
LOWELL	76	79	76 x 79 = 6.004	004
TREE	84	82	84 x 82 = 6.888	888
OLIVER	79	76	79 x 76 = 6.004	004

chaves sinônimas: LOWELL e OLIVER

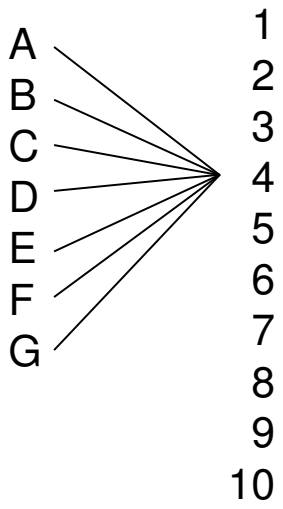
Distribuição de Registros

- Como uma função *hash* distribui (espalha) os registros no espaço de endereços?

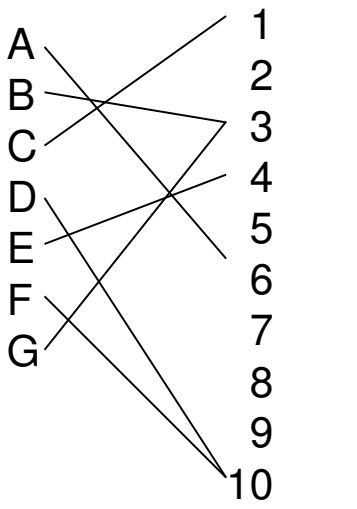
(a) melhor caso



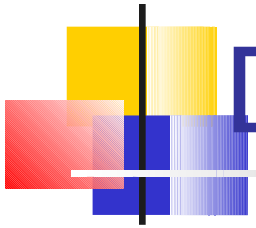
(b) pior caso



(c) caso aceitável



7 registros x 10 endereços



Distribuição Uniforme

- Registros espalhados uniformemente entre os endereços

- Características
  - pouca ou nenhuma colisão
  - muito difícil de ser obtida

Distribuição Aleatória

- Os registros espalhados no espaço de endereços com algumas colisões

- Propriedades (função randômica)
  - para uma certa chave, todos os endereços possuem a mesma probabilidade de serem escolhidos
  - a probabilidade de um endereço ser escolhido por uma outra chave não varia em função deste endereço já ter sido escolhido
  - na geração de um grande número de endereços, alguns endereços são gerados mais freqüentemente que outros

Alguns Métodos de *Hash*

- Pegar o resto da divisão da chave pelo tamanho do espaço disponível
- Examinar as chaves em busca de um padrão
- Segmentar a chave em diversos pedaços e depois fundir os pedaços
- Dividir a chave por um número
- Elevar a chave ao quadrado e pegar o meio
- Transformar a base

Colisão: Solução 1

- Encontrar um algoritmo de *hashing* perfeito que não produza colisões
  - Muito difícil encontrar um esquema desse tipo

- Cenário de uso
  - conjunto de dados pequenos e estáveis

Colisão: Solução 2

- Encontrar um algoritmo de *hashing* que produza poucas colisões
- Objetivo
  - evitar o agrupamento de registros em certos endereços
- Funcionalidade
  - espalhar os registros aleatoriamente no espaço disponível para armazenamento
  - distribuir o mais uniformemente possível

Colisão: Solução 3

- Ajustar a forma de armazenamento dos registros

- Possibilidade 1: usar memória extra
  - aumentar o espaço de endereçamento, para um mesmo conjunto de registros
  - cenário de uso
    - poucos registros para serem distribuídos entre muitos endereços

Colisão: Solução 3

- Possibilidade 1: uso de memória extra
  - complexidade de espaço
    - perda de espaço de armazenamento

- Exemplo
  - registros: 75
  - espaço de endereçamento: 1.000
  - alocado= 7,5%
  - não usado = 92,5%

Colisão: Solução 3

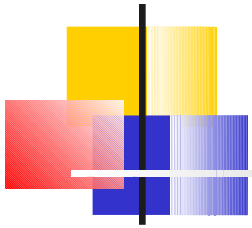
- Possibilidade 2: armazenar mais de um registro em um único endereço
  - uso de *buckets* (cestos) → técnica de *blocagem*
    - cada endereço é suficientemente grande para armazenar diversos registros → mais registros de uma vez, menos seeks
  - exemplo
    - registros de 80 bytes
    - *bucket* de 512 bytes

cada endereço pode armazenar até 6 registros!

  - complexidade de espaço
    - perda de espaço para registros sem sinônimos

Colisão: Solução 3

- Para as duas possibilidades, soluções convencionais
  - Overflow progressivo
  - *Hashing* duplo
  - Encadeamento
    - Como funcionavam?
      - Como era inserção e remoção?
      - Qual o efeito no arquivo a longo prazo?



Categorias de Hashing

- Hashing estático: garante acesso O(1), para arquivos estáticos
  - Organização do arquivo pode deteriorar se houver muitas inserções e remoções

- Hashing "dinâmico"/não estático: extensão do *hashing* estático para tratar arquivos dinâmicos, ou seja, que sofrem muitas inserções e remoções de registros
  - Muitos tipos semelhantes
    - Extensível, dinâmico, linear, etc.

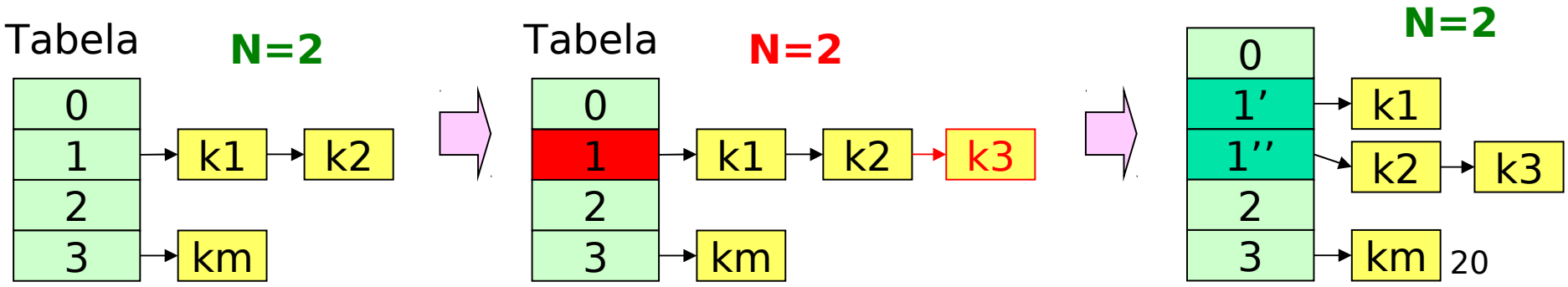
Hashing dinâmico

- O tamanho do espaço de endereçamento (número de *buckets*) pode aumentar

- Exemplo
  - *Hashing* extensível

Hashing dinâmico

- *Hashing* extensível
  - Conforme os elementos são inseridos na tabela, o tamanho aumenta, se necessário
    - Supondo que o número máximo de elementos por bucket é N, sempre que o elemento N+1 surgir, o *bucket* é dividido juntamente com os elementos



Hashing dinâmico

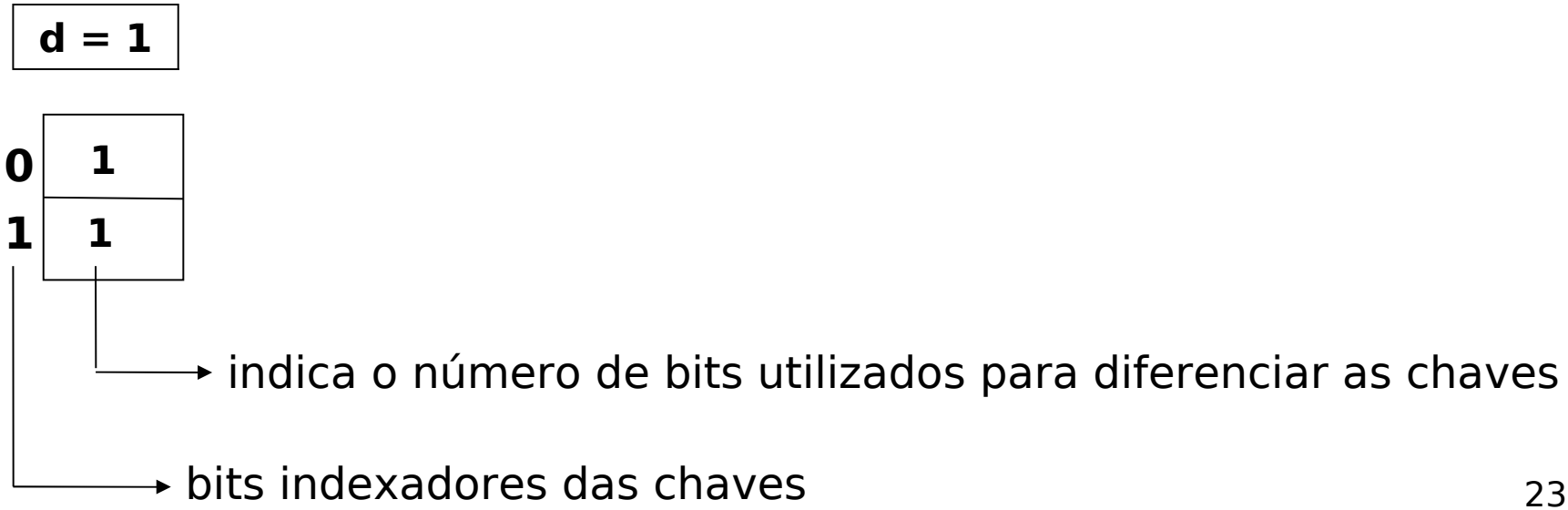
- *Hashing* extensível
  - Em geral, trabalha-se com índice binário
  - Após h(k) ser computada, uma segunda função f transforma o índice h(k) em uma seqüência de bits
    - Os bits são utilizados para indexar de fato a chave
  - Alternativamente, h e f podem ser unificadas como uma única função hash final

Hashing dinâmico

- *Hashing* extensível
  - Função *hash* computa seqüência de **m** bits para uma chave **k**, mas apenas os **i** primeiro bits (**i** <= **m**) do início da seqüência são usados como endereço
    - Se i é o número de bits usados, a tabela de *buckets* terá 2^i entradas
      - Portanto, tamanho da tabela de *buckets* cresce sempre como potência de 2
  - N é o número de nós permitidos por *bucket*

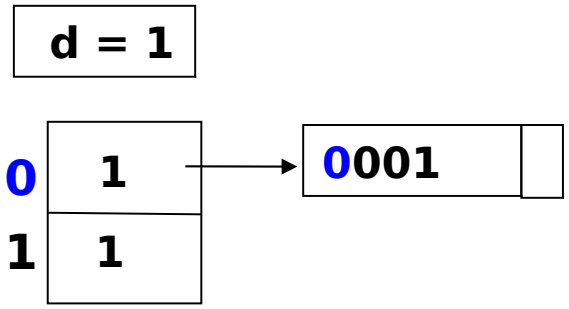
Hashing dinâmico

- *Hashing* extensível: inicialmente, tabela vazia
  - m = 4 (bits), N = 2



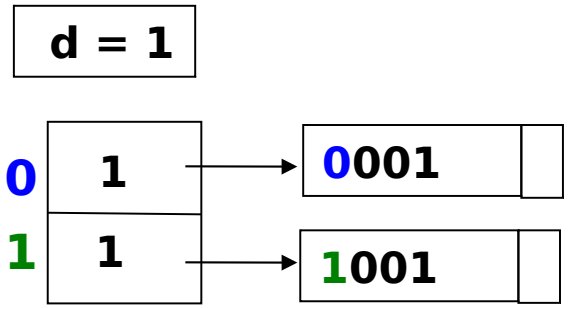
Hashing dinâmico

- *Hashing* extensível: inserção do elemento 0001
  - m = 4 (bits), N = 2



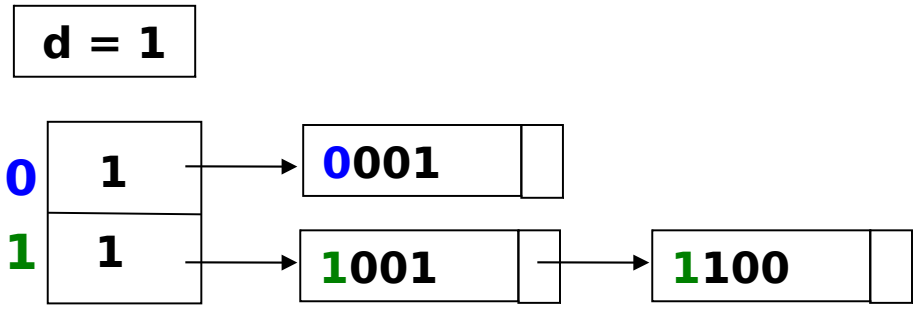
Hashing dinâmico

- *Hashing* extensível: inserção do elemento 1001
  - m = 4 (bits), N = 2



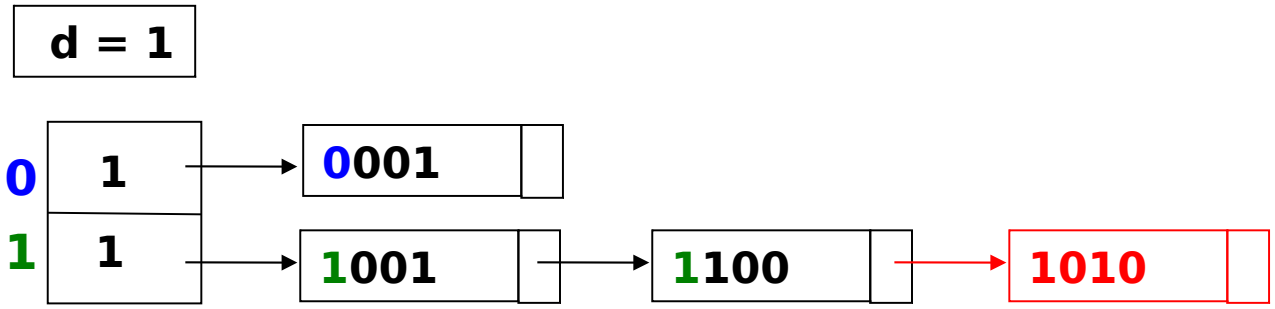
Hashing dinâmico

- *Hashing* extensível: inserção do elemento 1100
  - m = 4 (bits), N = 2



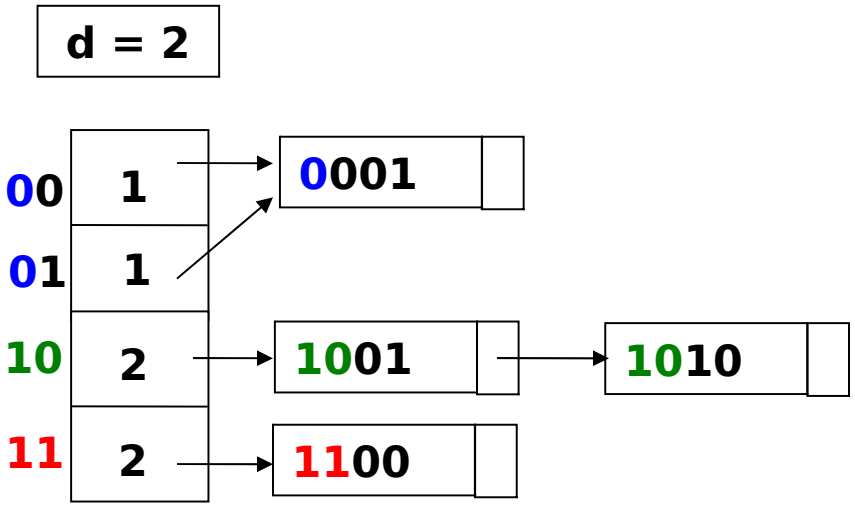
Hashing dinâmico

- *Hashing* extensível: inserção do elemento 1010
  - m = 4 (bits), N = 2
    - N é ultrapassado e a tabela precisa ser **rearranjada**, pois um único bit não é suficiente para diferenciar os elementos, sendo que o índice em que houve problema tem seu bit incrementado



Hashing dinâmico

- *Hashing* extensível: rearranjando tabela
  - m = 4 (bits), N = 2
    - Número de posições aumenta para observar a restrição de N e chaves são rearranjadas

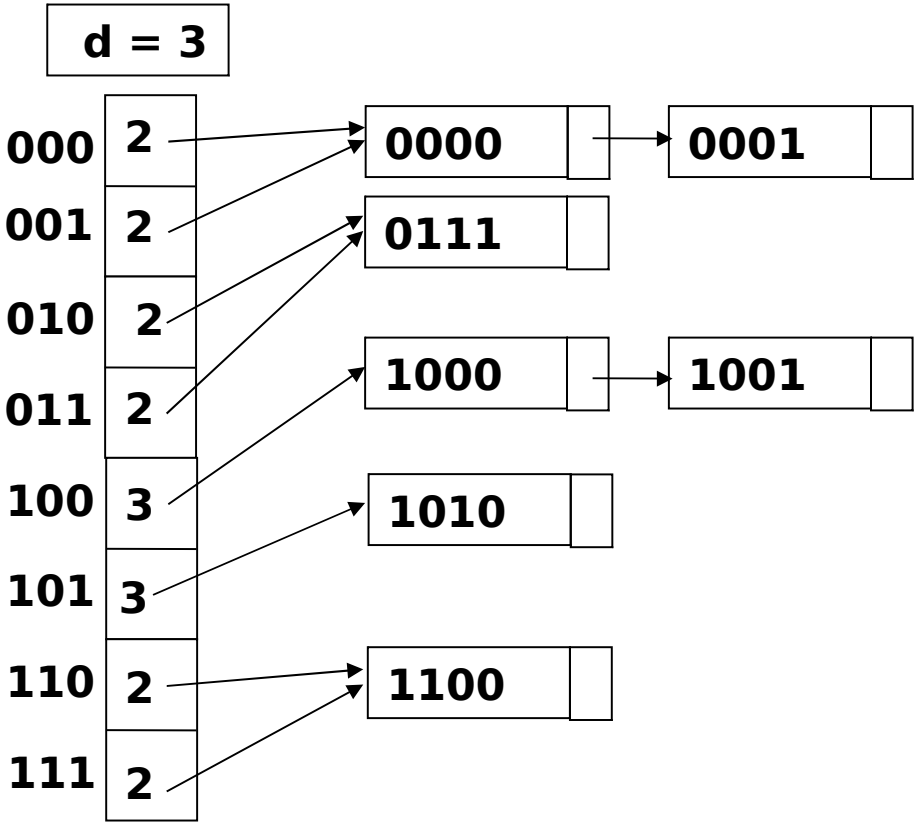


Hashing dinâmico

- Exercício
  - Insira os elementos 0000, 0111 e 1000, nesta ordem

Hashing dinâmico

- *Hashing* extensível: resultado das inserções



Hashing dinâmico

- Exercício
  - Elimine da organização anterior a chave 1000

Hashing dinâmico

Idealmente

- Tabela de bits deve caber na memória
  - Incremento e decremento do tamanho é fácil de se lidar
- Buckets no arquivo
  - Novos buckets no fim do arquivo
    - Organização lógica diferente da física

Hashing dinâmico

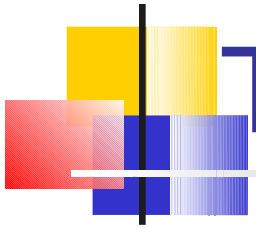
- Vantagens
  - Evita deterioração do arquivo com inserções e remoções
    - O registro sempre estará na posição indicada no arquivo, não sendo necessário buscar em outras posições

Hashing Extensível – **embasamento teórico**

- Espalhamento convencional: pouco adequado a arquivos dinâmicos, que crescem e diminuem com o tempo

- **Espalhamento Extensível** (*Extendible Hashing*): permite um auto-ajuste do espaço de endereçamento do espalhamento

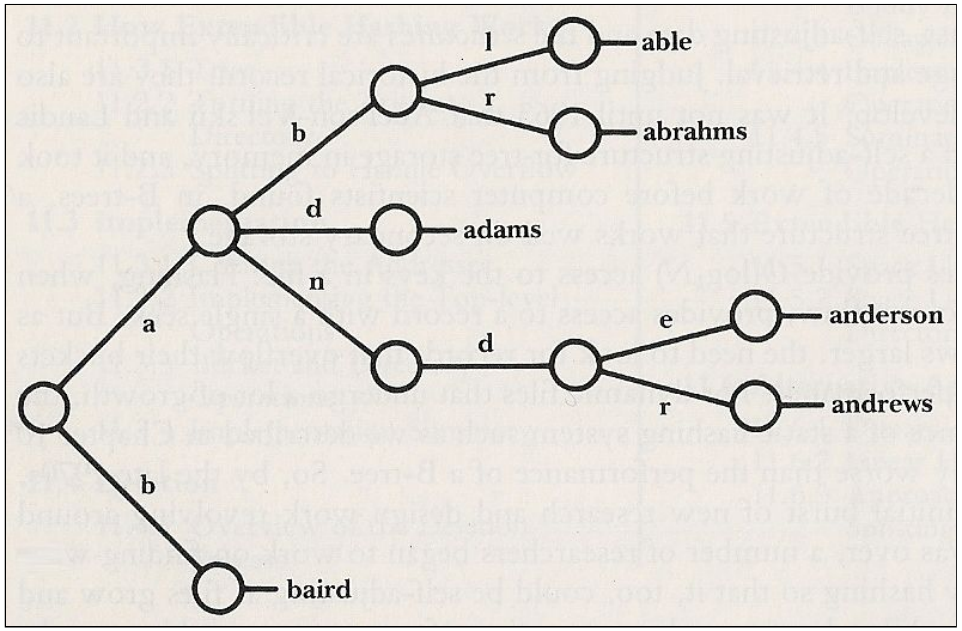
  - Idéia chave é combinar o espalhamento convencional com uma técnica de recuperação de informações denominada **trie**

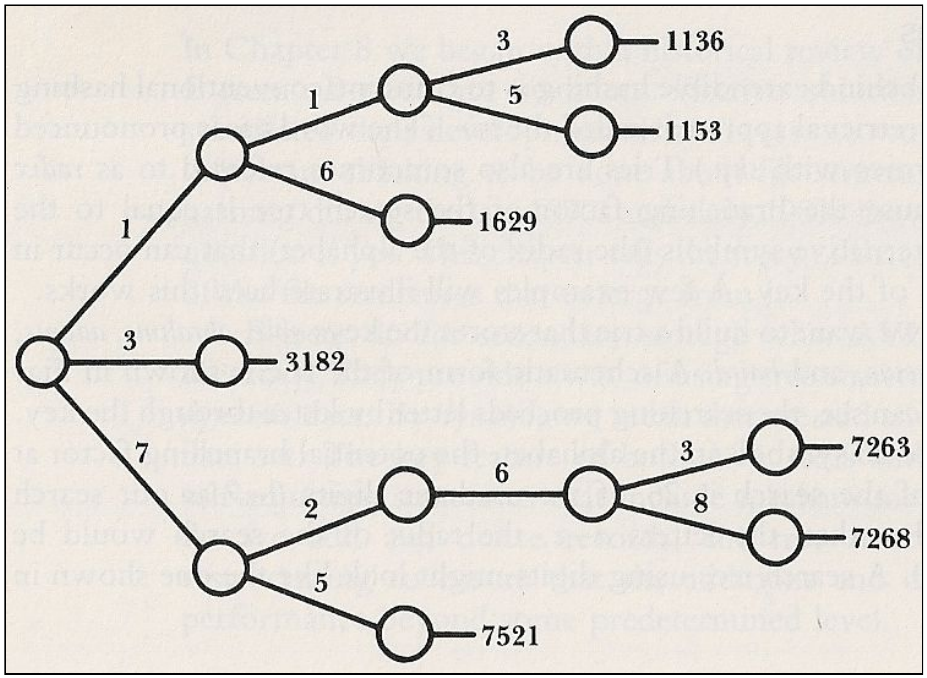


Trie

- origem do nome: palavra reTRIEval
- também conhecida como *radix searching tree*, ou árvore de busca digital
- árvore de busca na qual o fator de sub-divisão, ou número máximo de filhos por nó, é igual ao número de símbolos do alfabeto que compõe as chaves
- boa opção para manter chaves grandes e de tamanho variável

Exemplos de tries



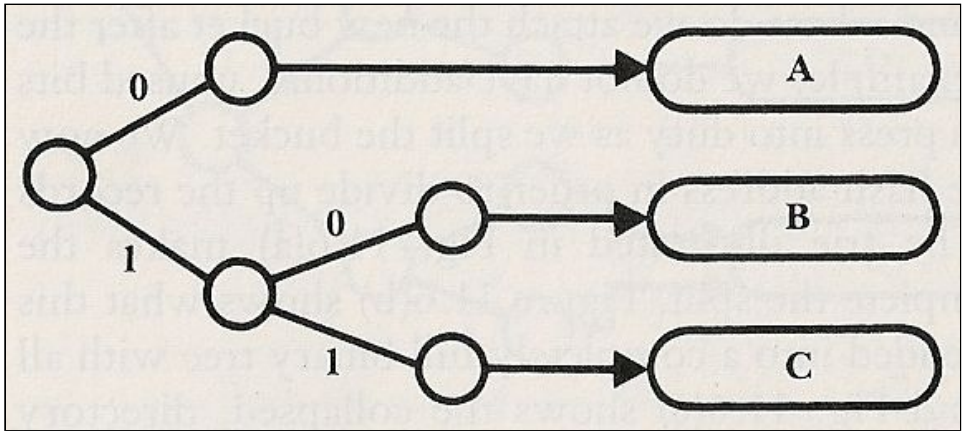


Tries e espalhamento extensível

- Tabela de espalhamento indexa um conjunto de *buckets*
  - Conjunto de chaves (ou registros) armazenadas em buckets
  - Busca por chave: análise bit-a-bit do valor de *key* ou do valor de *h(key)* permite localizar o seu cesto

Tries e espalhamento extensível

- Nível dos nós folha: buckets contendo várias chaves (ou registros)



Bucket

- Segmento físico útil de armazenamento externo
  - página, trilha ou segmento de trilha

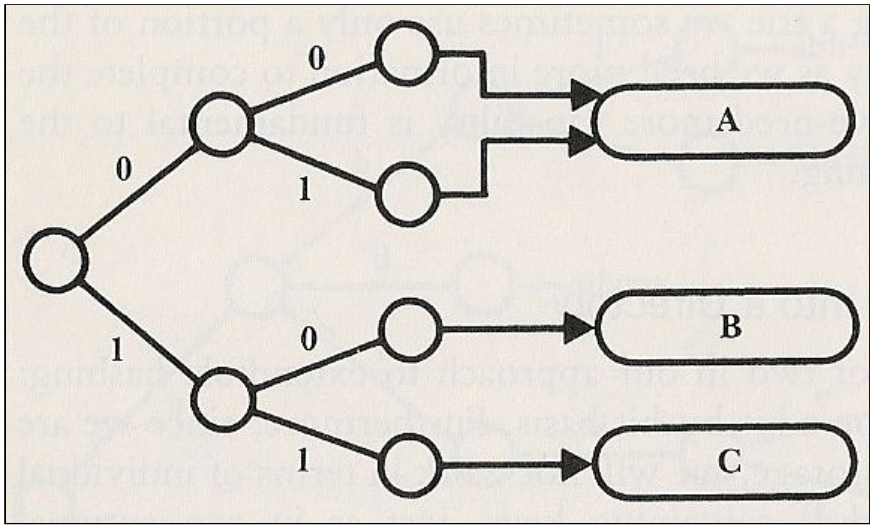
Como representar a *trie*?

- Se for mantida como uma árvore, são necessárias várias comparações para descer ao longo de sua estrutura

- Solução mais eficiente: representá-la como um diretório de endereços

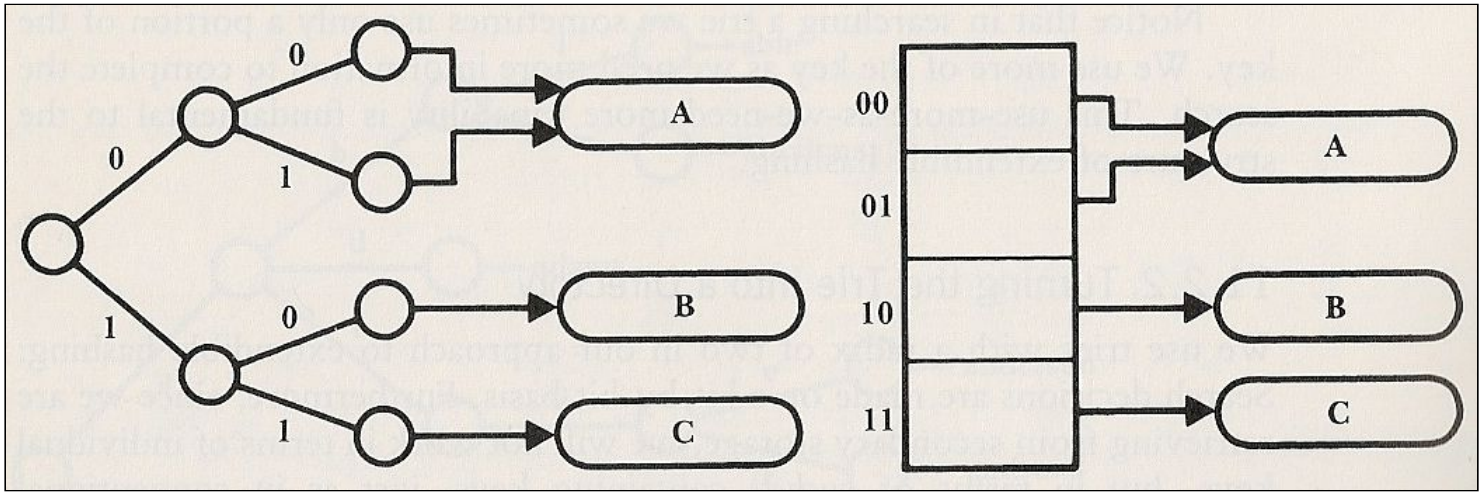
Transformando uma trie em um diretório

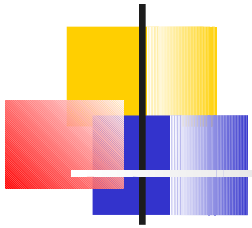
- Passo 1: a trie deve ser uma árvore binária completa
  - Se não for, pode ser estendida



Transformando uma trie em um diretório

- Passo 2: uma vez “completa”, trie pode ser representada por um vetor
  - O vetor fornece acesso direto aos endereços
  - Exemplo: endereço começando com os bits 10 pode ser encontrado a partir de um ponteiro na posição 10_2 do diretório

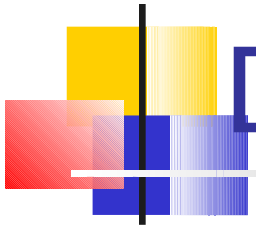




Diretório

- Profundidade do bucket: indicação do número de bits da chave necessários para determinar quais registros ele contém

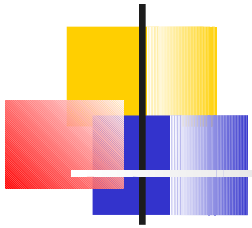
- Informação mantida junto ao bucket
  - Exemplo:
    - Cesto A: profundidade 1
    - Cestos B e C: profundidade 2



Diretório

- Inicialmente, a profundidade é a mesma para todos os cestos, e define a profundidade do diretório

Subdivisão para tratar overflow

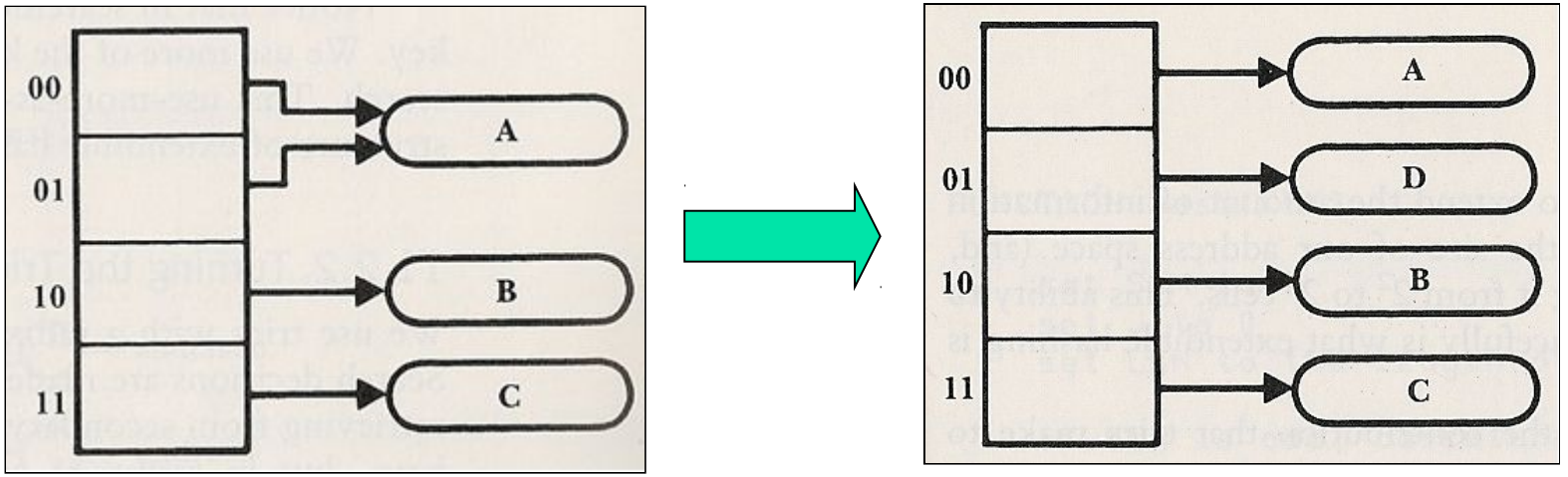


- Se um registro precisa ser inserido
  - se não há espaço, bucket é sub-dividido (*spliting*)
    - adiciona-se mais um bit aos endereços
      - se o novo espaço de endereçamento já estava previsto no diretório, nenhuma alteração é necessária
      - senão, é necessário dobrar o espaço de endereçamento do diretório para acomodar o novo bit

Subdivisão para tratar overflow

Exemplo: o bucket A sobre overflow

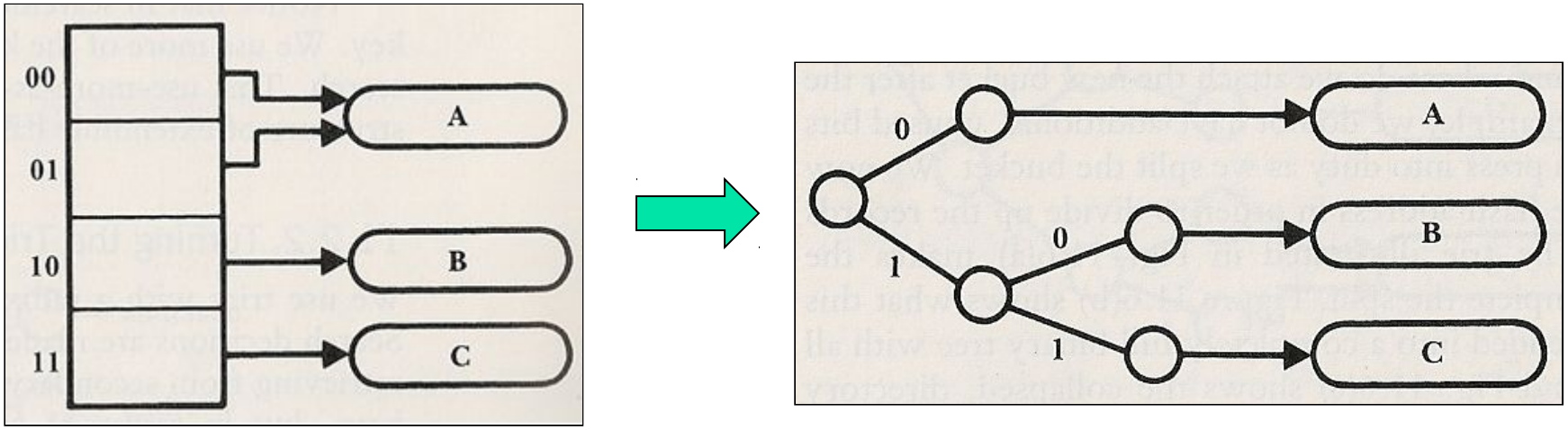
- adiciona-se um bit, sendo que já havia espaço



Subdivisão para tratar overflow

Exemplo: overflow do bucket B

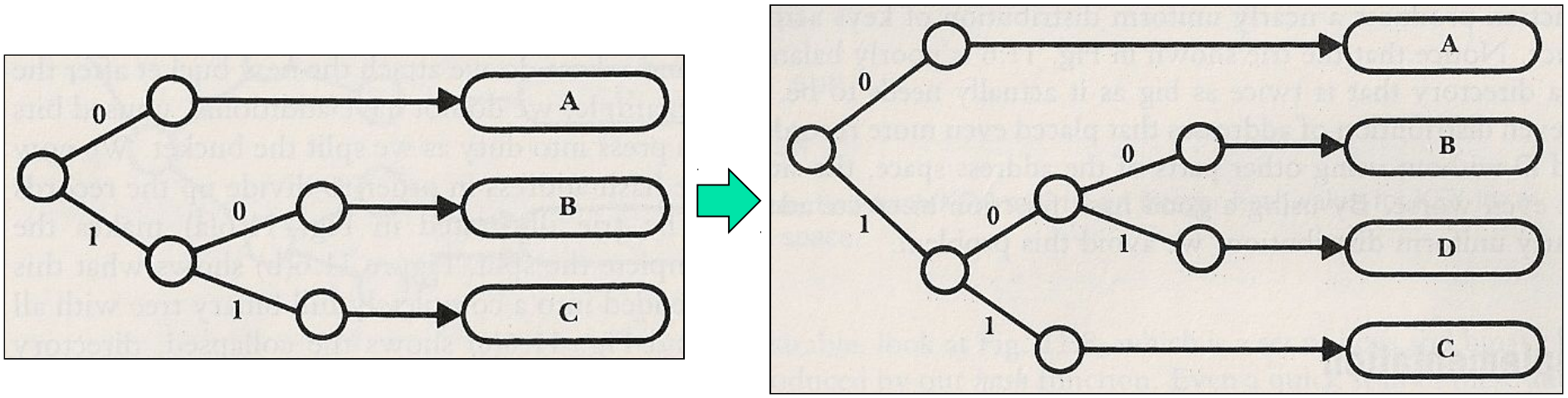
- não há espaço
- dobra-se vetor



Subdivisão para tratar overflow

Exemplo: overflow do bucket B

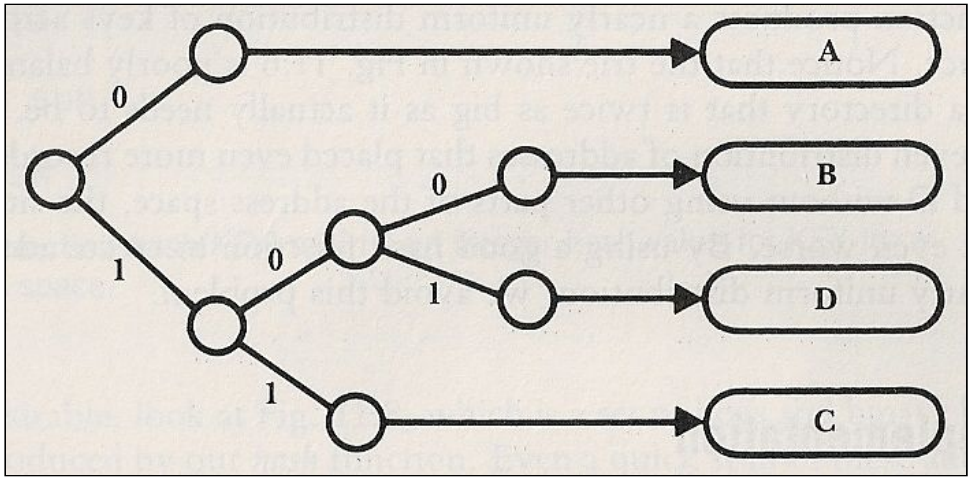
- não há espaço
- dobra-se vetor

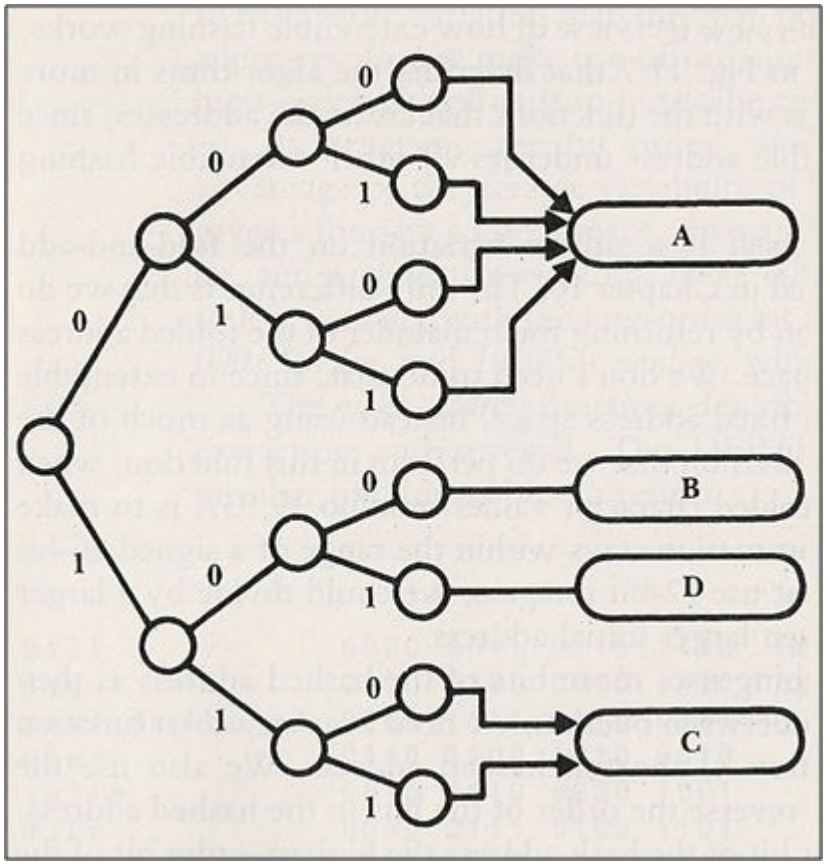


Subdivisão para tratar overflow

Exemplo: overflow do bucket B

- não há espaço
- dobra-se vetor





Subdivisão para tratar overflow

Exemplo: overflow do bucket B