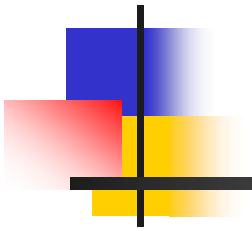


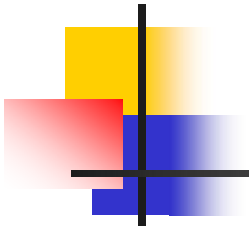
Métodos de Busca

Parte 2

ICC2

Prof. Thiago A. S. Pardo

Baseado no material do Prof. Rudinei Goularte

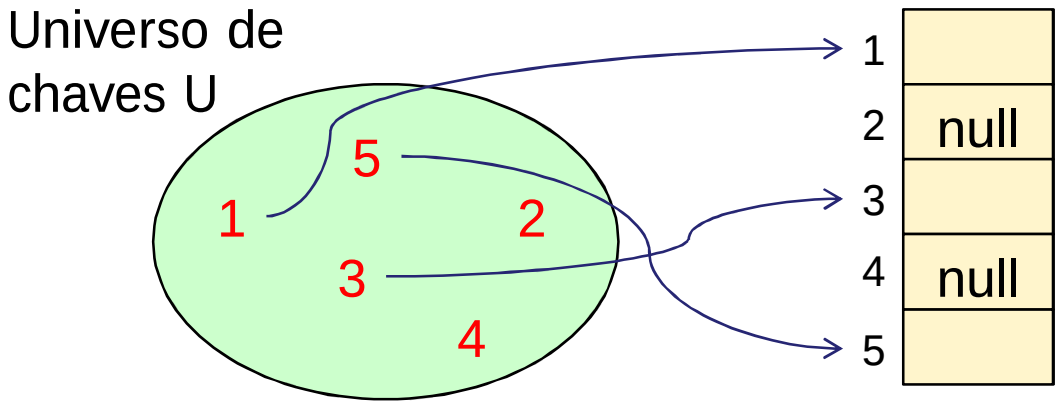


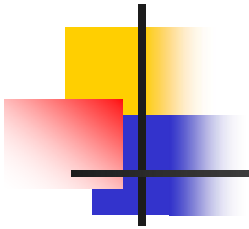
Introdução

- Acesso seqüencial = O(n)
  - Quanto mais as estruturas (tabelas, arquivos, etc.) crescem, mais acessos há
  - Quando armazenamento é em disco, reduzir acessos é essencial
- Busca binária = O(log(n))
  - Restrita à arranjos
- Árvores AVL (no melhor caso) = O(log(n))
  - Não importa tamanho da tabela

Introdução

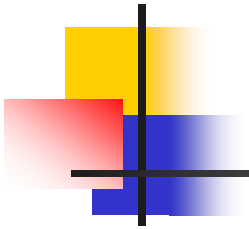
- Acesso em tempo constante
  - Tradicionalmente, endereçamento direto em um arranjo
    - Cada chave k é mapeada na posição k do arranjo
      - Função de mapeamento f(k)=k





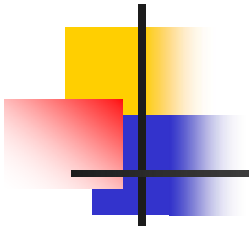
Introdução

- Endereçamento direto
  - Vantagens
    - Acesso direto e, portanto, rápido
      - Via indexação do arranjo
  - Desvantagens
    - Uso ineficiente do espaço de armazenamento
      - Declara-se um arranjo do tamanho da maior chave?
      - E se as chaves não forem contínuas? Por exemplo, {1 e 100}
      - Pode sobrar espaço? Pode faltar?



Introdução

- *Hashing*
  - Acesso direto, mas endereçamento indireto
    - Função de mapeamento h(k) ≠ k, em geral
    - Resolve uso ineficiente do espaço de armazenamento
  - Ideal: O(1), em média, independente do tamanho do arranjo

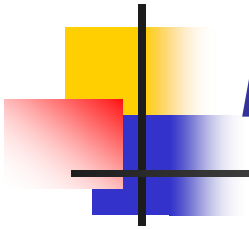


Introdução

- Hash significa (*Webster's New World Dictionary*):

1. Fazer picadinho de carne e vegetais para cozinhar

2. Fazer uma bagunça

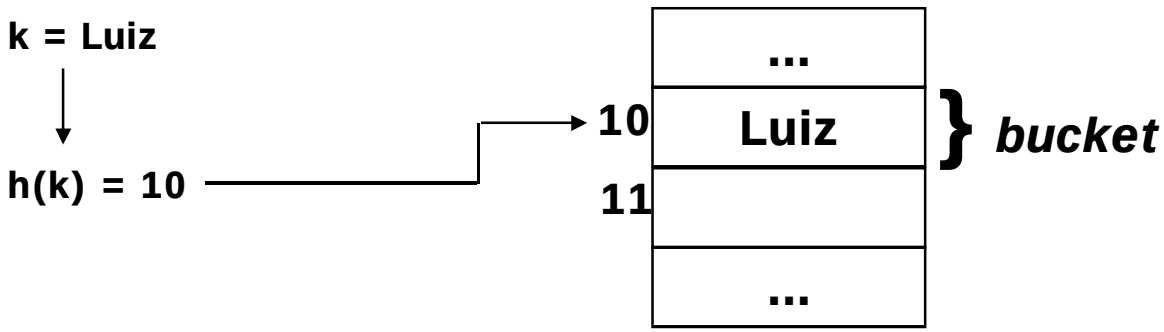


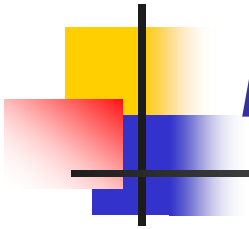
Hashing: conceitos e definições

- Também conhecido como tabela de espalhamento ou de dispersão
- *Hashing* é uma técnica que utiliza uma função h para transformar uma chave k em um endereço
  - O endereço é usado para armazenar e recuperar registros
- Idéia: particionar um conjunto de elementos (possivelmente infinito) em um número finito de classes
  - B classes, de 0 a B – 1
  - Classes são chamadas de *buckets*

Hashing: conceitos e definições

- Conceitos relacionados
  - A função h é chamada de função *hash*
  - h(k) retorna o valor *hash* de k
    - Usado como endereço para armazenar a informação cuja chave é k
  - k pertence ao *bucket* h(k)



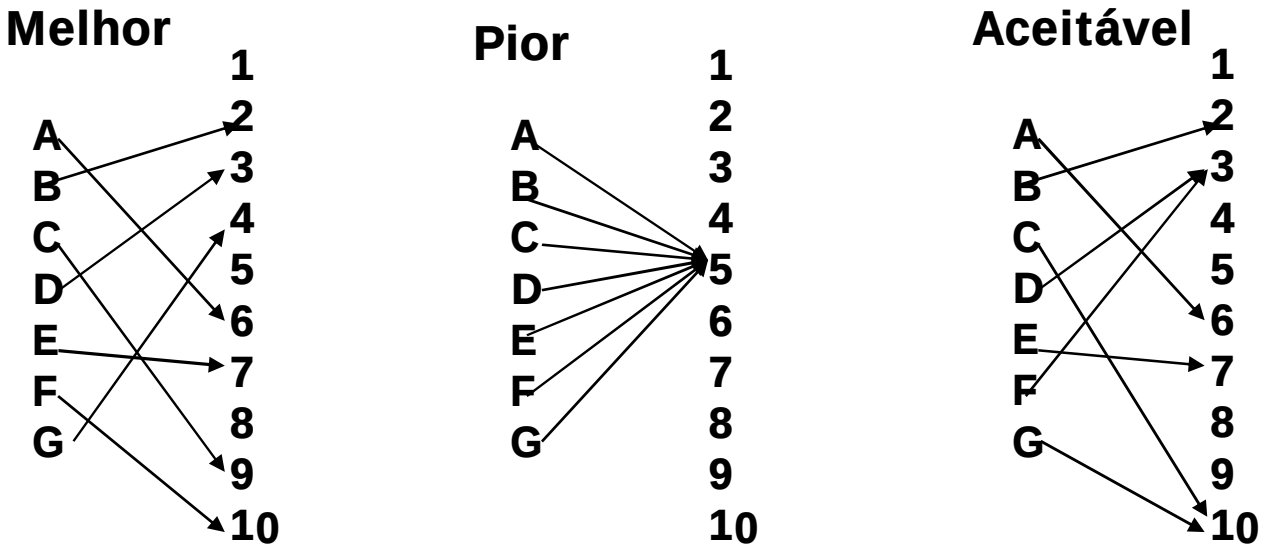


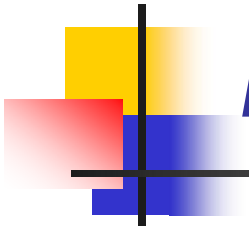
Hashing: conceitos e definições

- A função hash é utilizada para inserir, remover ou buscar um elemento
  - Deve ser determinística, ou seja, resultar sempre no mesmo valor para uma determinada chave

Hashing: conceitos e definições

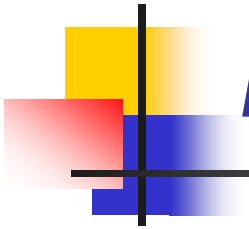
- Colisão: ocorre quando a função *hash* produz o mesmo endereço para chaves diferentes
  - As chaves com mesmo endereço são ditas “sinônimos”





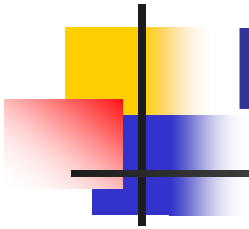
Hashing: conceitos e definições

- Distribuição uniforme é muito difícil
  - Dependente de cálculos matemáticos e estatísticos complexos
- Função que aparente gerar endereços aleatórios
  - Existe chance de alguns endereços serem gerados mais de uma vez e de alguns nunca serem gerados
- Existem alternativas melhores que a puramente aleatória



Hashing: conceitos e definições

- Segredos para um bom *hashing*
  - Escolher uma boa função hash (em função dos dados)
    - Distribui uniformemente os dados, na medida do possível
      - Hash uniforme
    - Evita colisões
    - É fácil/rápida de computar
  - Estabelecer uma boa estratégia para tratamento de colisões

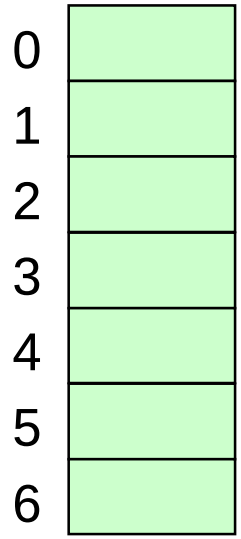


Exemplo de função *hash*

- Técnica simples e muito utilizada que produz bons resultados
  - Para chaves inteiras, calcular o resto da divisão k/B (k%B), sendo que o resto indica a posição de armazenamento
    - k = valor da chave, B = tamanho do espaço de endereçamento
  - Para chaves tipo *string*, tratar cada caracter como um valor inteiro (ASCII), somá-los e pegar o resto da divisão por B
  - B deve ser primo, preferencialmente

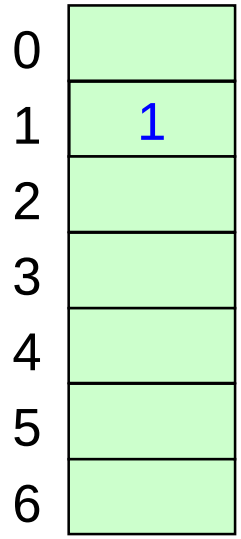
Exemplo de função *hash*

- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24



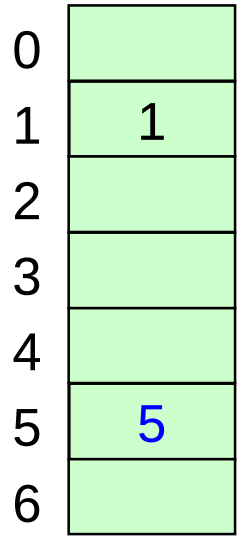
Exemplo de função *hash*

- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 1 % 7 = 1



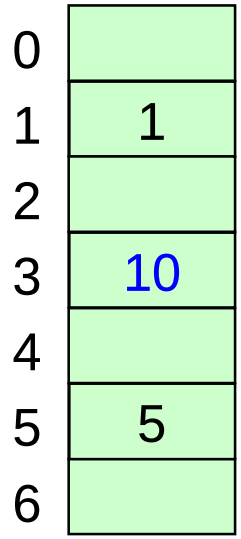
Exemplo de função *hash*

- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 5 % 7 = 5



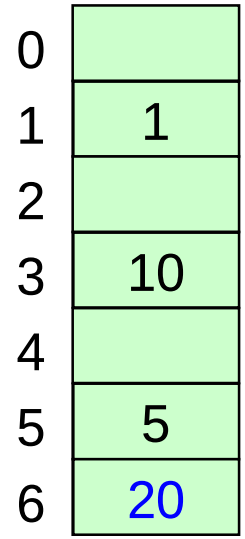
Exemplo de função *hash*

- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 10 % 7 = 3



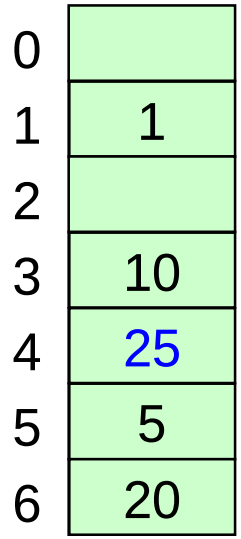
Exemplo de função *hash*

- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 20 % 7 = 6



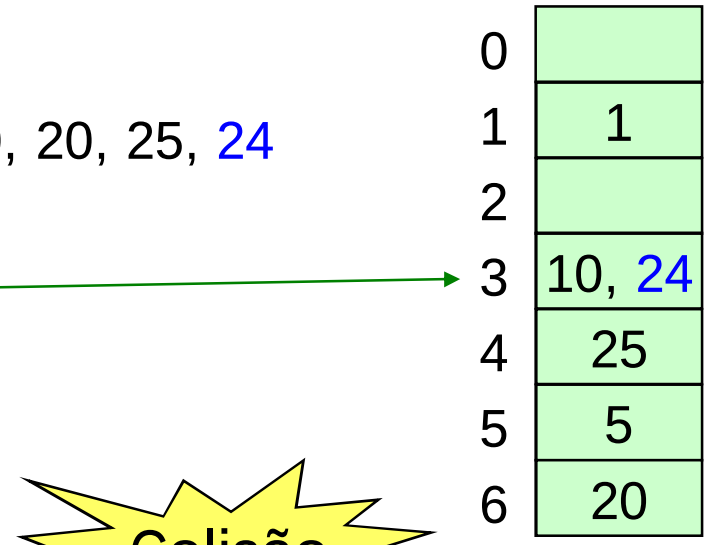
Exemplo de função *hash*

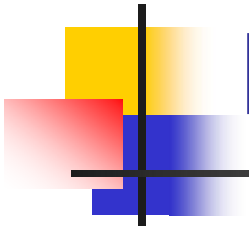
- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 25 % 7 = 4



Exemplo de função *hash*

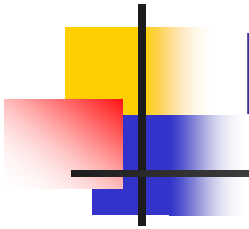
- Exemplo
  - Seja B um arranjo de 7 elementos
    - Inserção dos números 1, 5, 10, 20, 25, 24
      - 24 % 7 = 3





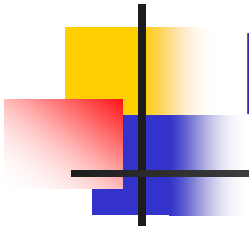
Exemplo de função *hash*

- Exemplo com string: mesmo raciocínio
  - Seja B um arranjo de 13 elementos
    - LOWEL = 76 79 87 69 76
      - L+O+W+E+L = 387
    - h(LOWEL) = 387 % 13 = 10



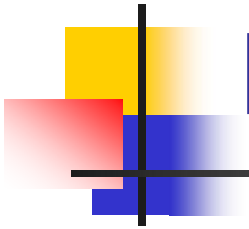
Exemplo de função *hash*

- Qual a idéia por trás da função hash que usa o resto?



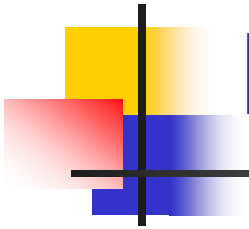
Exemplo de função *hash*

- Qual a idéia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo entre 0 e n-1



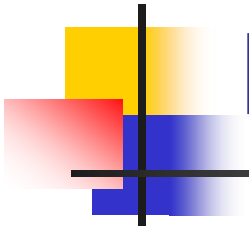
Exemplo de função *hash*

- Qual a idéia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo entre 0 e n-1
- Outras funções hash?



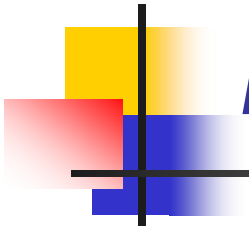
Exemplo de função *hash*

- Qual a idéia por trás da função hash que usa o resto?
  - Os elementos sempre caem no intervalo entre 0 e n-1
- Outras funções hash?
- Como você trataria colisões?



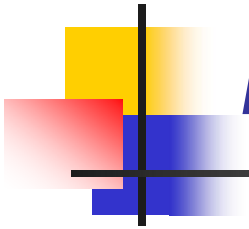
Funções *hash*

- Às vezes, deseja-se que chaves próximas sejam armazenadas em locais próximos
  - Por exemplo, em um compilador, os identificadores de variáveis pt e pts
- Normalmente, não se quer tal propriedade
  - Questão da aleatoriedade aparente
    - Hash uniforme, com menor chance de colisão
- Função hash escolhida deve espelhar o que se deseja



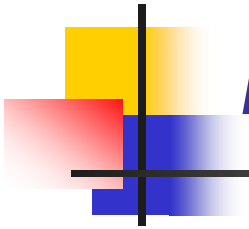
Hashing

- Pergunta: supondo que se deseja armazenar n elementos em uma tabela de m posições, qual o número esperado de elementos por posição na tabela?



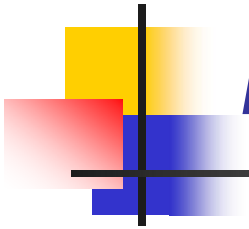
Hashing

- Pergunta: supondo que se deseja armazenar n elementos em uma tabela de m posições, qual o número esperado de elementos por posição na tabela?
  - Fator de carga α=n/m



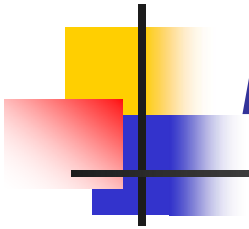
Hashing

- Tipos de *hashing*
  - Estático
    - Fechado
      - Técnicas de *rehash* para tratamento de colisões
        - *Overflow* progressivo
        - 2ª. função hash
    - Aberto
      - Encadeamento de elementos para tratamento de colisões
  - Dinâmico



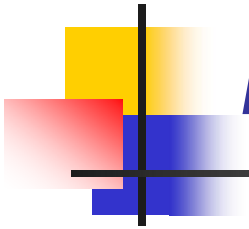
Hashing

- 2 tipos básicos
  - Estático
    - Espaço de endereçamento não muda
  - Dinâmico
    - Espaço de endereçamento pode aumentar



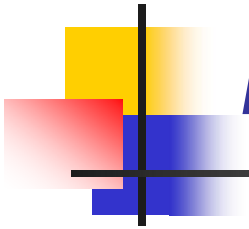
Hashing estático

- 2 tipos básicos
  - Fechado
    - Permite armazenar um conjunto de informações de tamanho limitado
  - Aberto
    - Permite armazenar um conjunto de informações de tamanho potencialmente ilimitado



Hashing estático

- *Hashing* fechado
  - Uma tabela de *buckets* é utilizada para armazenar informações
    - Os elementos são armazenados na própria tabela
      - Normalmente conhecido como endereçamento aberto
  - Colisões: aplicar técnicas de *rehash*
    - *Overflow* progressivo
    - 2ª função *hash*

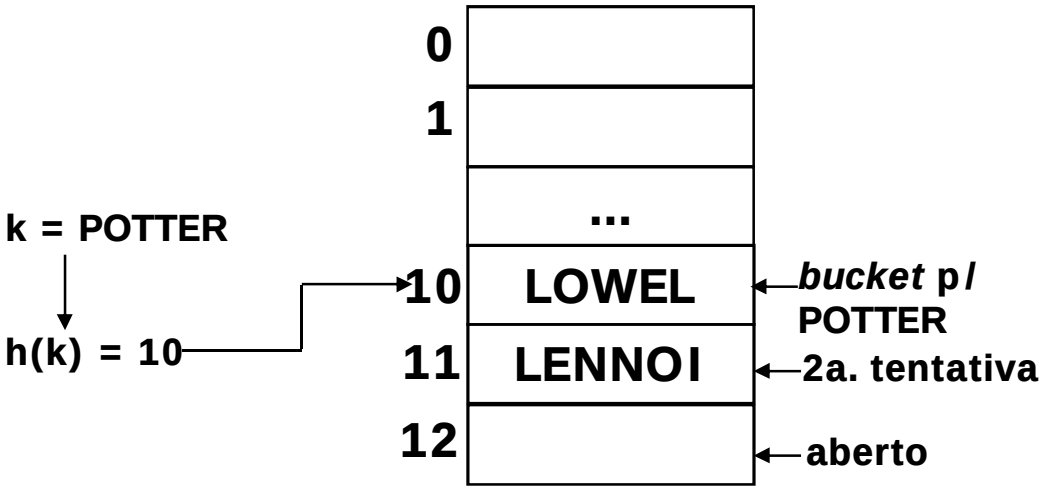


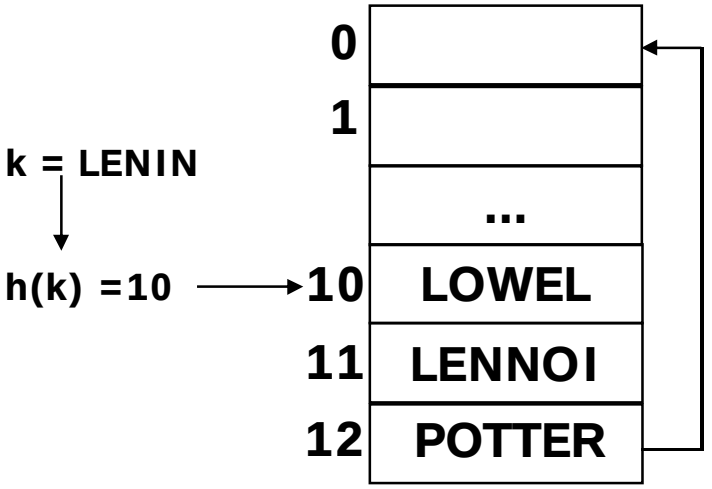
Hashing estático

- Técnicas de *rehash*
  - Se posição h(k) está ocupada (lembre-se de que h(k) é um índice da tabela), aplicar função de rehash sobre h(k), que deve retornar o próximo *bucket* livre
    - rh(h(k))
    - Características de uma boa função de rehash
      - Cobrir o máximo de índices entre 0 e B-1
      - Evitar agrupamentos de dados
  - Além de utilizar o índice resultante de h(k) na função de rehash, pode-se usar a própria chave k e outras funções hash

Hashing estático

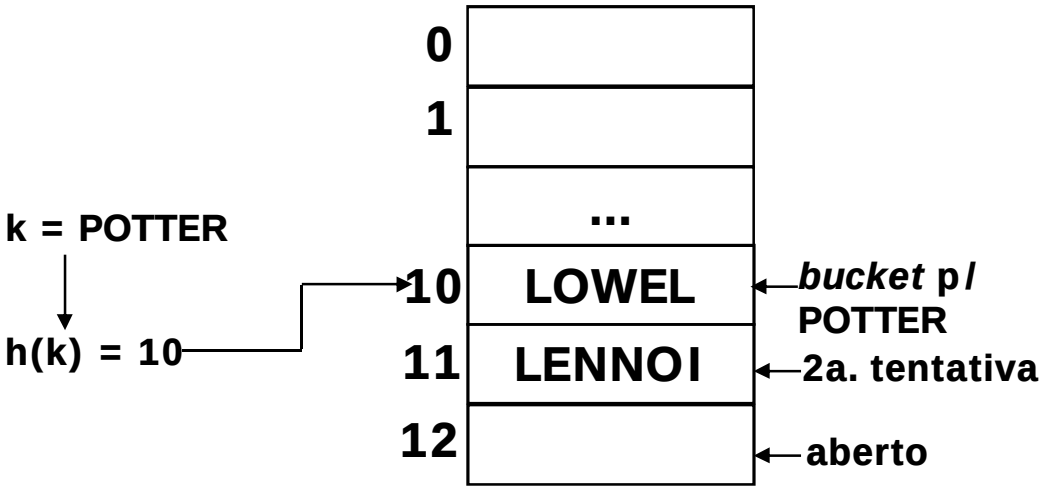
- *Overflow* progressivo
  - rh(h(k)) = (h(k) + i) % B, com i variando de 1 a B-1 (i é incrementado a cada tentativa)

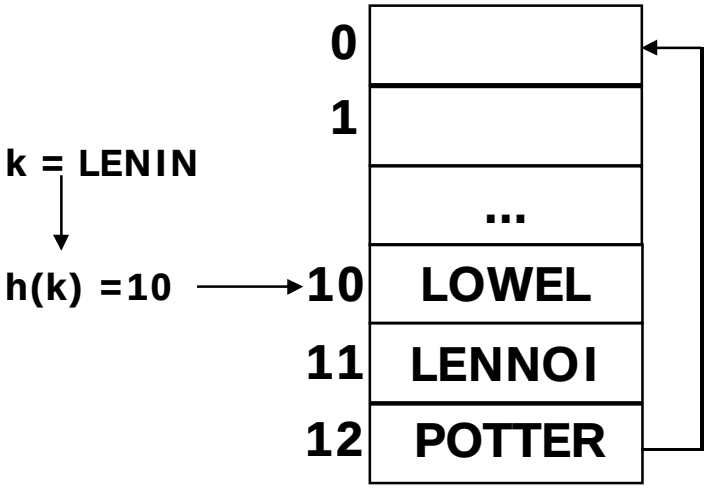




Hashing estático

- *Overflow* progressivo
  - rh(h(k)) = (h(k) + i) % B, com i variando de 1 a B-1 (i é incrementado a cada tentativa)





Como saber que a informação procurada não está armazenada?

Hashing estático

- Exemplo de dificuldade: busca pelo nome “Smith”

h(SMITH) = 7

	...
7	ADMS
8	JONES
9	MORRIS
10	SMITH

Pode ter que percorrer muitos campos

Hashing estático

- Exemplo de dificuldade: busca pelo nome “Smith”

h(SMITH) = 7

	...
7	**ADMS**
8	**JONES**
9	
10	**SMITH**

A remoção do elemento no índice 9 pode causar uma falha na busca

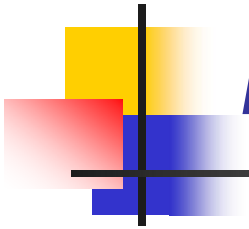
Hashing estático

- Exemplo de dificuldade: busca pelo nome “Smith”

h(SMITH) = 7

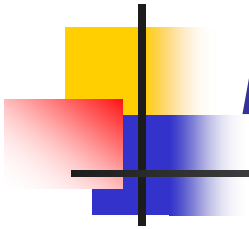
	...
7	**ADMS**
8	**JONES**
9	**####**
10	**SMITH**

Solução para remoção de elementos: não eliminar elemento, mas indicar que a posição foi esvaziada e que a busca deve continuar



Hashing estático

- *Overflow* progressivo
  - Exemplo anterior
    - rh(h(k)) = (h(k) + i) % B, com i=1...B-1
      - Chamada sondagem linear, pois todas as posições da tabela são checadas, no pior caso
  - Outro exemplo
    - rh(h(k)) = (h(k) + c_1*i + c_2*i^2) % B, com i=1...B-1 e constantes c_1 e c_2
      - Chamada sondagem quadrática, considerada melhor do que a linear, pois evita "mais" o agrupamento de elementos

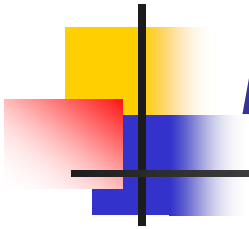


Hashing estático

- *Overflow* progressivo
  - Vantagem
    - Simplicidade
  - Desvantagens
    - Agrupamento de dados (causado por colisões)
    - Com estrutura cheia, a busca fica lenta
    - Dificulta inserções e remoções

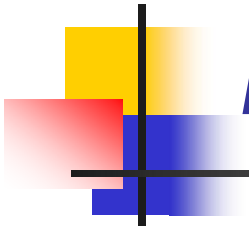
Característica do *overflow progressivo*

Características do *hashing* fechado



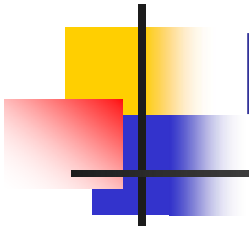
Hashing estático

- 2ª função *hash*, ou *hash* duplo
  - Uso de 2 funções
    - h(k): função *hash* primária
    - h_{aux}(k): função *hash* secundária
  - Exemplo: rh(h(k)) = (h(k) + i*h_{aux}(k)) % B, com i=1...B-1
  - Algumas boas funções
    - h(k) = k%B
    - h_{aux}(k) = 1+k%(B-1)



Hashing estático

- 2ª função *hash*, ou *hash* duplo
  - Vantagem
    - Evita agrupamento de dados, em geral
      - Por quê?
  - Desvantagens
    - Difícil achar funções *hash* que, ao mesmo tempo, satisfaçam os critérios de cobrir o máximo de índices da tabela e evitem agrupamento de dados
    - Operações de buscas, inserções e remoções são mais difíceis

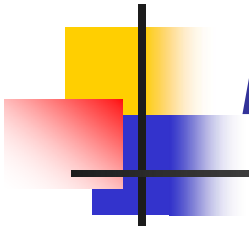


Exercício

- Assumindo que:
  - B=10
  - h(k)=k%B
  - rh(h(k))=(h(k)+i)%B, com i=1...B-1

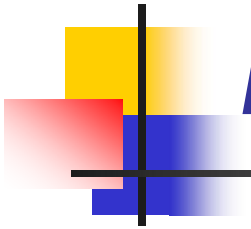
insira os seguintes elementos em uma tabela hash utilizando *hashing* fechado com *overflow* progressivo

41, 10, 8, 7, 13, 52, 1, 89 e 15



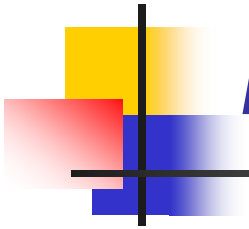
Hashing estático

- Alternativamente, em vez de fazer o *hashing* utilizando uma função *hash* e uma funçao de *rehash*, podemos representar isso em uma única função dependente do número da tentativa (i)
  - Por exemplo: h(k, i) = (k+i)%B, com i=0...B-1
    - A função h depende agora de dois fatores: a chave k e a iteração i
    - Note que i=0 na primeira execução, resultando na função *hash* tradicional de divisão que já conhecíamos
    - Quando i=1...B-1, já estamos aplicando a função de *rehash* de sondagem linear



Hashing estático

- Exercício: implemente uma sub-rotina de inserção utilizando função *hash* anterior

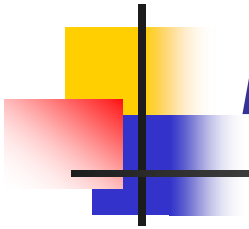


Hashing estático

- Exercício: implemente uma sub-rotina de inserção utilizando função *hash* anterior

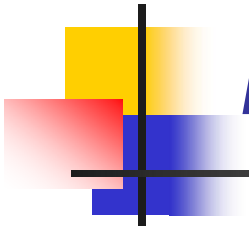
```
#define B 100
#define h(k,i) (k+i)%B

int inserir(int T[], int k) {
    int i, j;
    for (i=0; i<B; i++) {
        j=h(k,i);
        if (T[j]==-1) {
            T[j]=k;
            return(j);
        }
    }
    return(-1)        //tabela já está cheia
}
```



Hashing estático

- Como seria a função de busca?

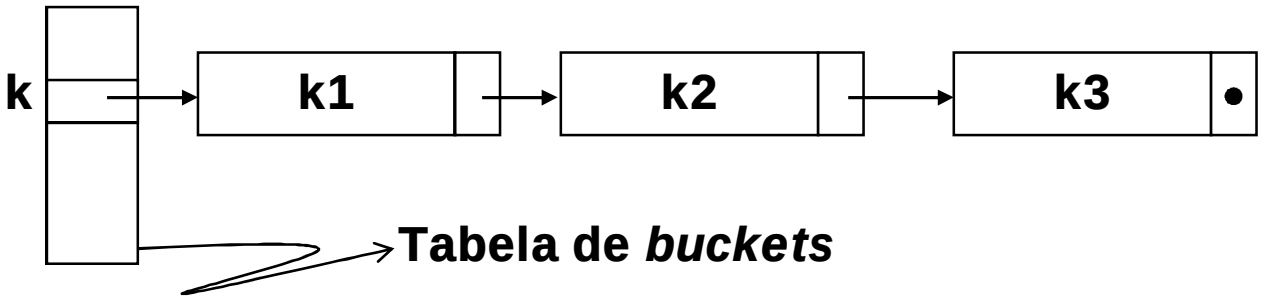


Hashing estático

- Como seria a função de remoção?

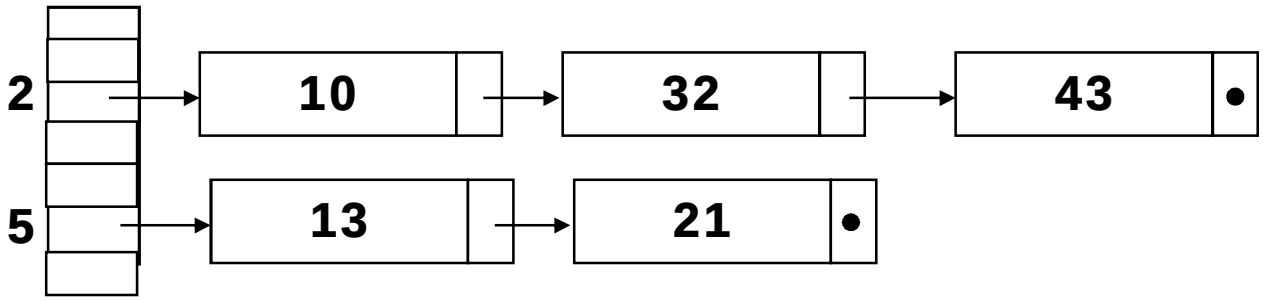
Hashing estático

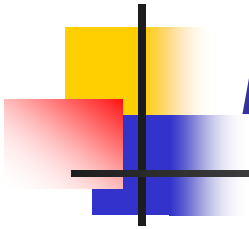
- *Hashing* aberto
  - A tabela de *buckets*, indo de 0 a B – 1, contém apenas ponteiros para uma lista de elementos
  - Quando há colisão, o sinônimo é inserido no *bucket* como um novo nó da lista
  - Busca deve percorrer a lista



Hashing estático

- Se as listas estiverem ordenadas, reduz-se o tempo de busca
  - Dificuldade deste método?

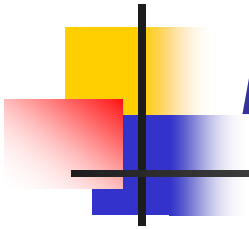




Hashing estático

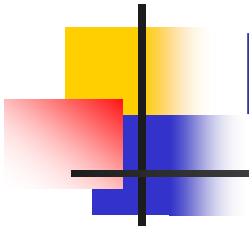
- Vantagens
  - A tabela pode receber mais itens mesmo quando um *bucket* já foi ocupado
  - Permite percorrer a tabela por ordem de valor *hash*

- Desvantagens
  - Espaço extra para as listas
  - Listas longas implicam em muito tempo gasto na busca
    - Se as listas estiverem ordenadas, reduz-se o tempo de busca
    - Custo extra com a ordenação



Hashing estático

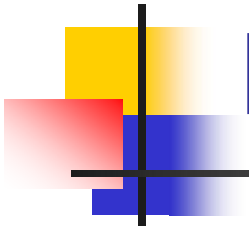
- Eficiência
  - *Hashing* fechado
    - Depende da técnica de *rehash*
      - Com *overflow* progressivo, após várias inserções e remoções, o número de acessos aumenta
    - A tabela pode ficar cheia
    - Pode haver mais espaço para a tabela, pois não são necessários ponteiros e campos extras como no *hashing* aberto
  - *Hashing* aberto
    - Depende do tamanho das listas e da função *hash*
      - Listas longas degradam desempenho
      - Poucas colisões implicam em listas pequenas



Exercício

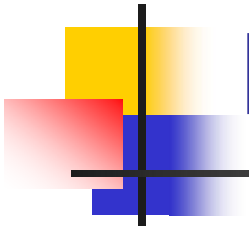
- Em grupos de 4 alunos, implemente uma sub-rotina (em C) de inserção de elementos em uma tabela *hash* de tamanho 100
  - Defina uma função *hash* qualquer
    - Suponha que você está lidando com números inteiros positivos lidos do usuário (até que -1 seja dado como entrada)
  - Utilize *hashing* estático aberto (com listas encadeadas)
  - A cada inserção, imprima a posição da tabela em que o elemento foi inserido
  - Ao fim da execução, imprima todos elementos da tabela *hash* e libere toda a memória utilizada

(valendo nota)



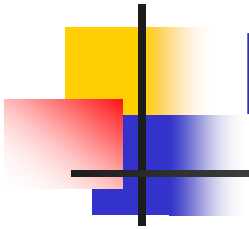
Funções *hash*

- Algumas boas funções
  - Divisão
    - h(k) = k%m, com m tendo um tamanho primo, de preferência



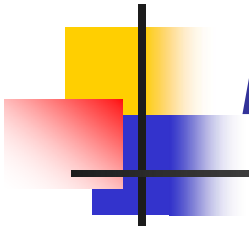
Funções *hash*

- Algumas boas funções
  - Multiplicação
    - h(k) = (k*A % 1)*m, com A sendo uma constante entre 0 e 1
      - (k*A % 1) recupera a parte fracionária de k*A
      - Knuth sugere A = (√5 – 1)/2 = 0,6180...



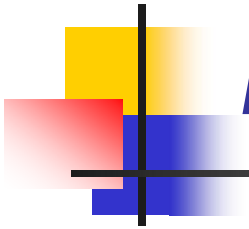
Funções *hash*

- Algumas boas funções
  - *Hash* universal
    - A função hash é escolhida aleatoriamente no início de cada execução, de forma que minimize/evite tendências das chaves
      - Por exemplo, h(k)=((A*k + B) % P) % m
        - P é um número primo maior do que a maior chave k
        - A é uma constante escolhida aleatoriamente de um conjunto de constantes {0, 1, 2, ..., P-1} no início da execução
        - B é uma constante escolhida aleatoriamente de um conjunto de constantes {1, 2, ..., P-1} no início da execução
      - Diz-se que h representa uma coleção de funções universal



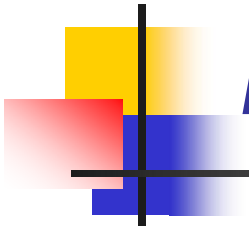
Hashing

- *Hash* perfeito
  - Quando não há colisão
    - Aplicável em um cenário em que o conjunto de chaves é estático
      - Exemplo de cenário deste tipo?
  - Exemplo de *hash* perfeito
    - *Hashing* em 2 níveis
    - Uma primeira função *hash* universal é utilizada para encontrar a posição na tabela, sendo que cada posição da tabela contém uma outra tabela (ou seja, outro arranjo)
    - Uma segunda funçao *hash* universal é utilizada para indicar a posição do elemento na nova tabela



Hashing

- *Hash* perfeito
  - É do tipo fechado ou aberto?



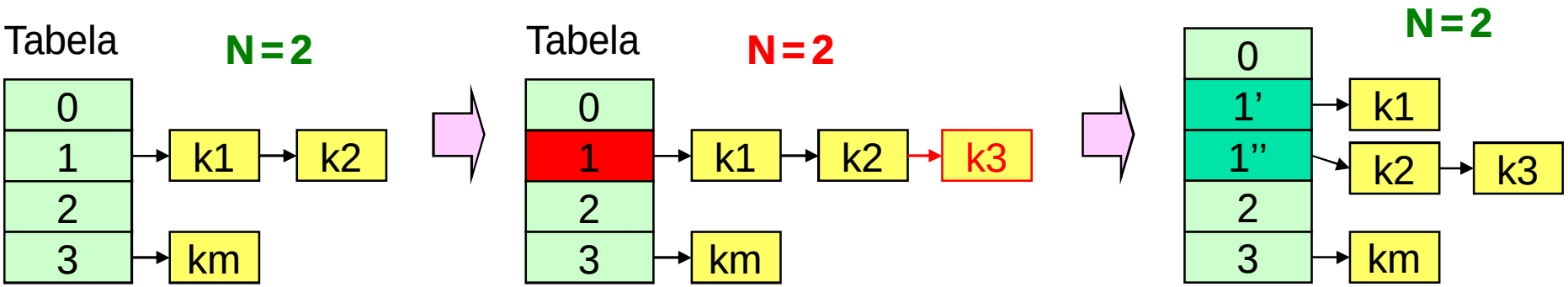
Hashing dinâmico

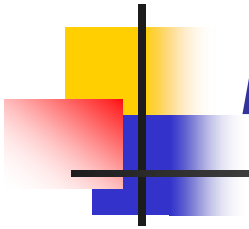
- O tamanho do espaço de endereçamento (número de *buckets*) pode aumentar

- Exemplo de *hashing* dinâmico
  - *Hashing* extensível

Hashing dinâmico

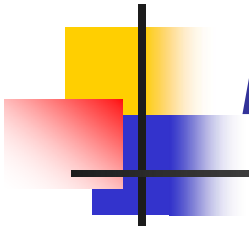
- *Hashing* extensível
  - Conforme os elementos são inseridos na tabela, o tamanho aumenta se necessário
    - Supondo que o número máximo de elementos por bucket é N, sempre que o elemento N+1 surgir, o *bucket* é dividido juntamente com os elementos





Hashing dinâmico

- *Hashing* extensível
  - Em geral, trabalha-se com bits
  - 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

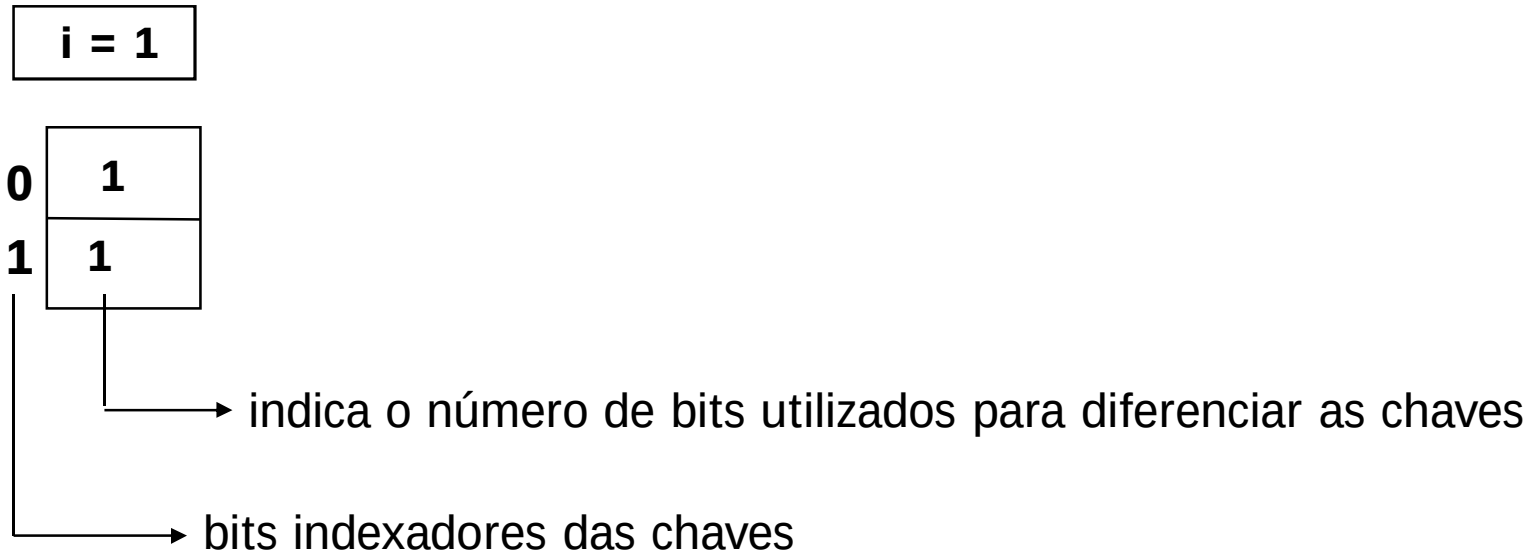


Hashing dinâmico

- *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*
  - Tratamento de colisões: listas encadeadas, em geral

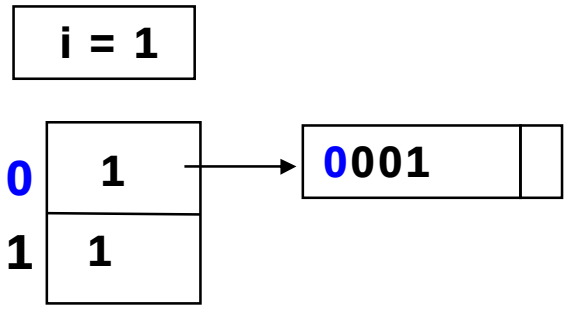
Hashing dinâmico

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



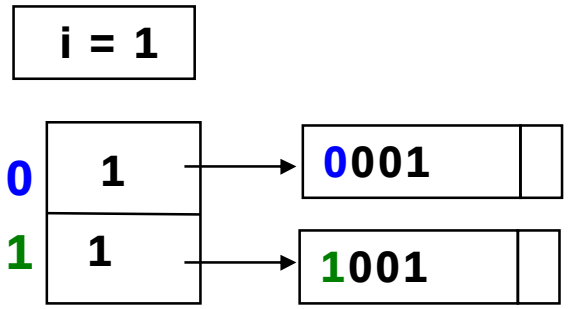
Hashing dinâmico

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



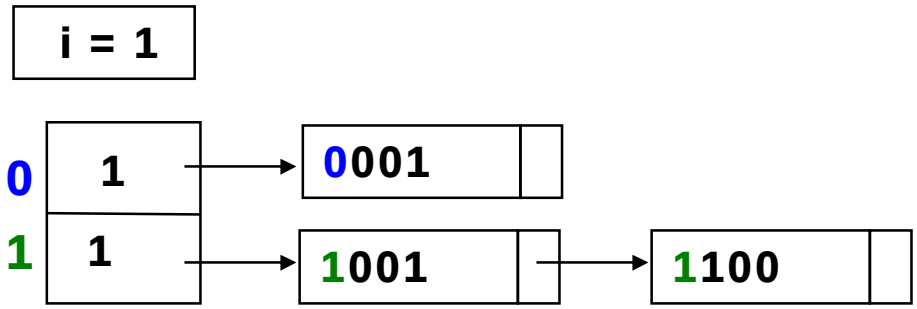
Hashing dinâmico

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



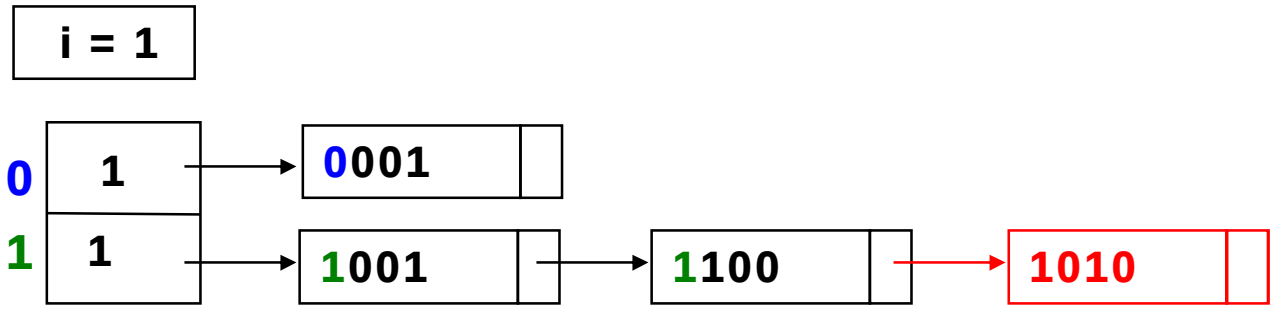
Hashing dinâmico

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



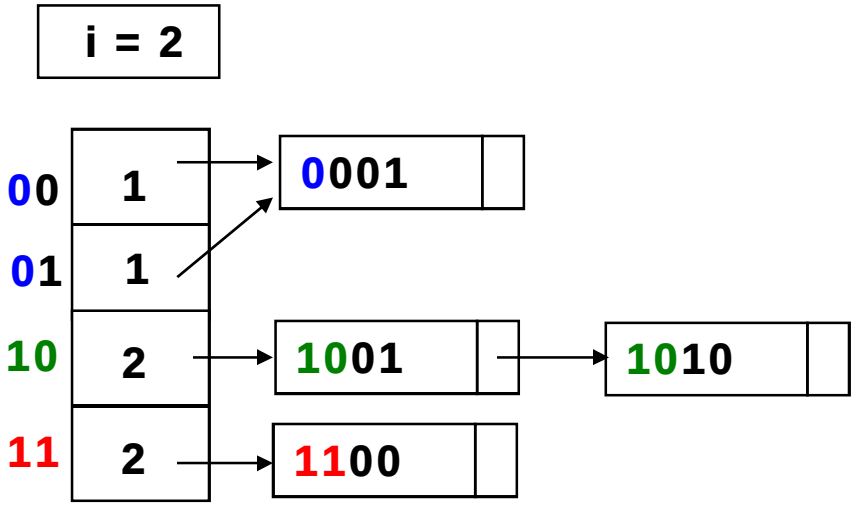
Hashing dinâmico

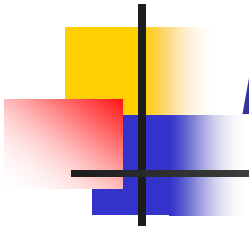
- *Hashing* extensível: inserção do elemento 1010
  - m = 4 (bits), N = 2
    - N é ultrapassado e a tabela precisa ser **rearranjada**, pois um único bit não é suficiente para diferenciar os elementos, sendo que o índice em que houve problema tem seu bit incrementado



Hashing dinâmico

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



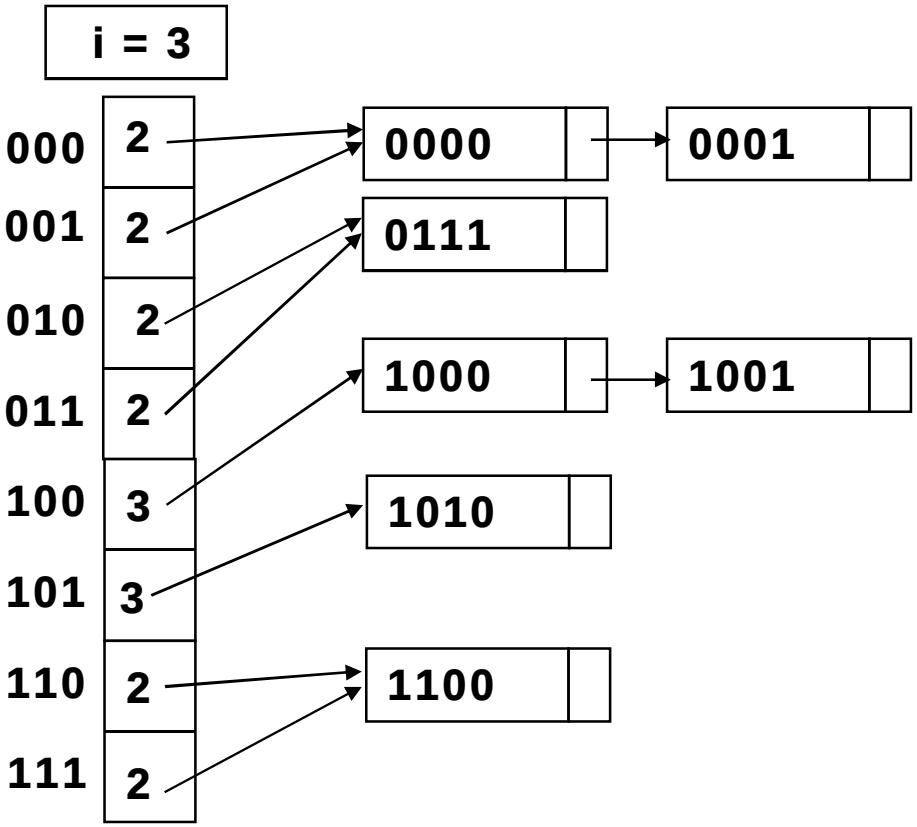


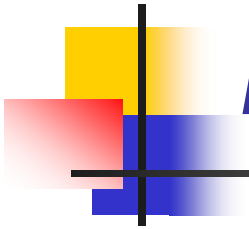
Hashing dinâmico

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

Hashing dinâmico

- *Hashing* extensível: resultado das inserções

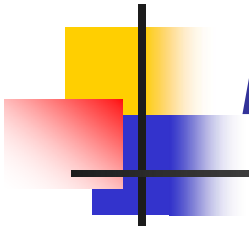




Hashing dinâmico

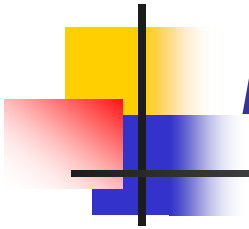
- Vantagens
  - Custo de acesso constante, determinado pelo tamanho de N
  - A tabela pode crescer

- Desvantagens
  - Complexidade extra para gerenciar o aumento do arranjo e a divisão das listas
  - Podem existir seqüências de inserções que façam a tabela crescer rapidamente, tendo, contudo, um número pequeno de registros



Hashing

- Pergunta
  - Quais são as principais desvantagens de *hashing*?



Hashing

- Pergunta
  - Quais são as principais desvantagens de *hashing*?
    - Os elementos da tabela não são armazenados seqüencialmente e nem sequer existe um método prático para percorrê-los em seqüência