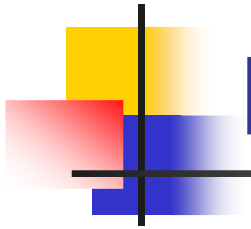
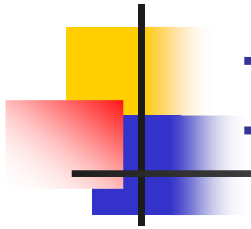
Tabelas de Espalhamento (hash)

Universidade Federal do Amazonas
Departamento de Eletrônica e Computação



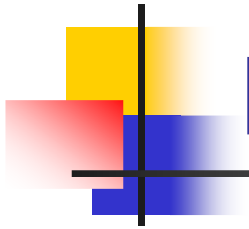
Hashing

- Método eficiente de busca com base em **assinaturas**
- Objetivos
  - Conceitos
  - Tratamento de Colisões
  - Escolha da Função de Hashing
  - Tratamento Avançado de Colisões



Idéia Geral

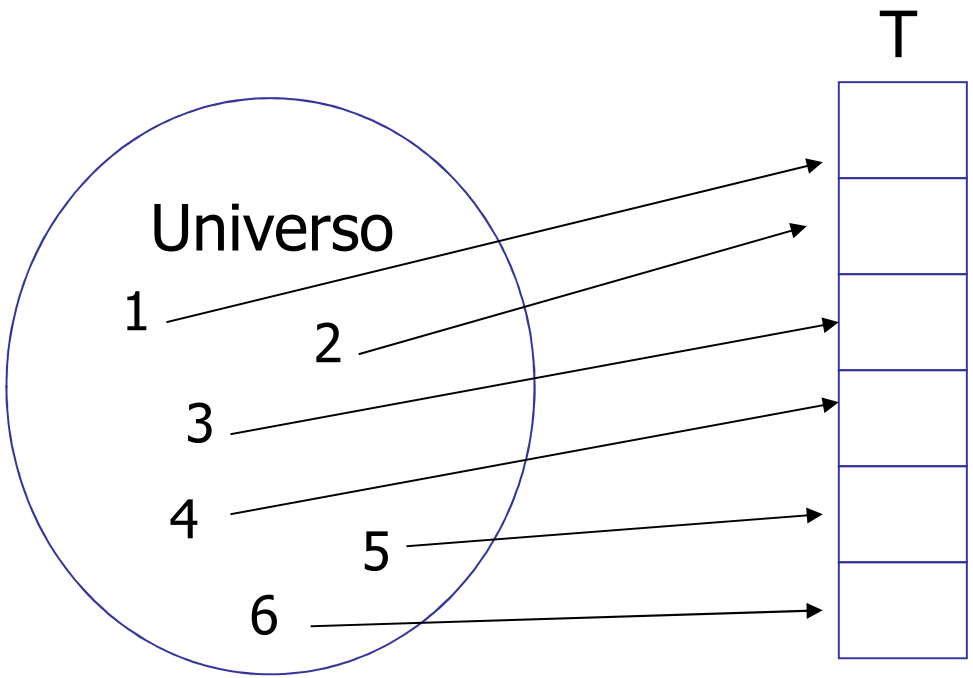
- Estrutura de dados onde as posições de **inserção** e **busca** são calculadas através de uma função que visa **distribuir** os elementos **aleatoriamente** ao longo de um vetor
- O tempo esperado para a inserção, remoção e pesquisa é O(1)
  - Tempo $\Theta(n)$ no pior caso para a busca
- Usada em situações onde precisa-se apenas de operações **inserir**, **buscar** e **remover**
- Não se pode, por exemplo, fazer caminhamento ordenado



Histórico

- A primeira referência a hash foi feita em 1953 por pesquisadores da IBM que estavam construindo um compilador
  - Tabela de símbolos: as chaves de elementos são cadeias de caracteres que correspondem a identificadores na linguagem
- Primeiro artigo sobre *hashing* foi publicado em 1956 numa revista Americana
- **Endereçam. aberto** publicado em 1957 na Rússia
  - todos os elementos são armazenados na própria tabela hash (não existem listas nem elementos armazenados fora da tabela, evitando assim o uso de ponteiros)

Tabelas de Endereçamento Direto (*lookup-tables*)

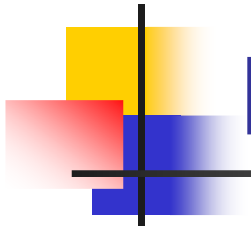



```
search(T,k)
  return T[k]
```

```
insert(T,x)
  T[chave[x]]=x
```

```
delete (T,x)
  T[chave[x]]=NIL
```

Se o conjunto não contém nenhuma elemento com chave k, então T[k]=NIL

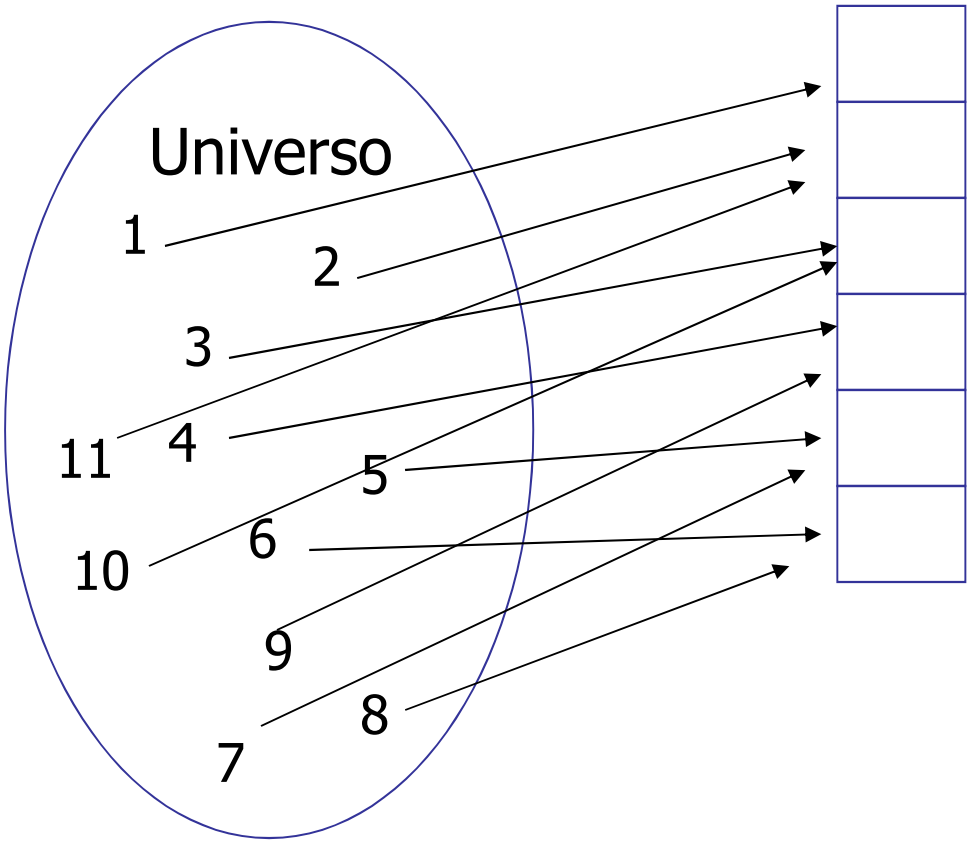


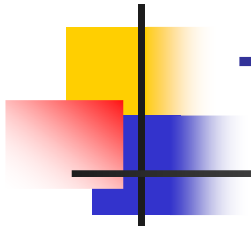
Exercício

- Considere um conjunto dinâmico *S* que é representado por uma tabela de endereço direto *T* de comprimento *m*. Descreva um procedimento que encontre o elemento máximo de *S*. Qual é o desempenho de seu procedimento no pior caso?

Tabela Hash (Universo Grande)

- Universo contém muitos elementos
- Número de chaves a serem inseridas é muito menor que o universo
- Nestes casos não faz sentido ter uma posição para cada elemento do universo

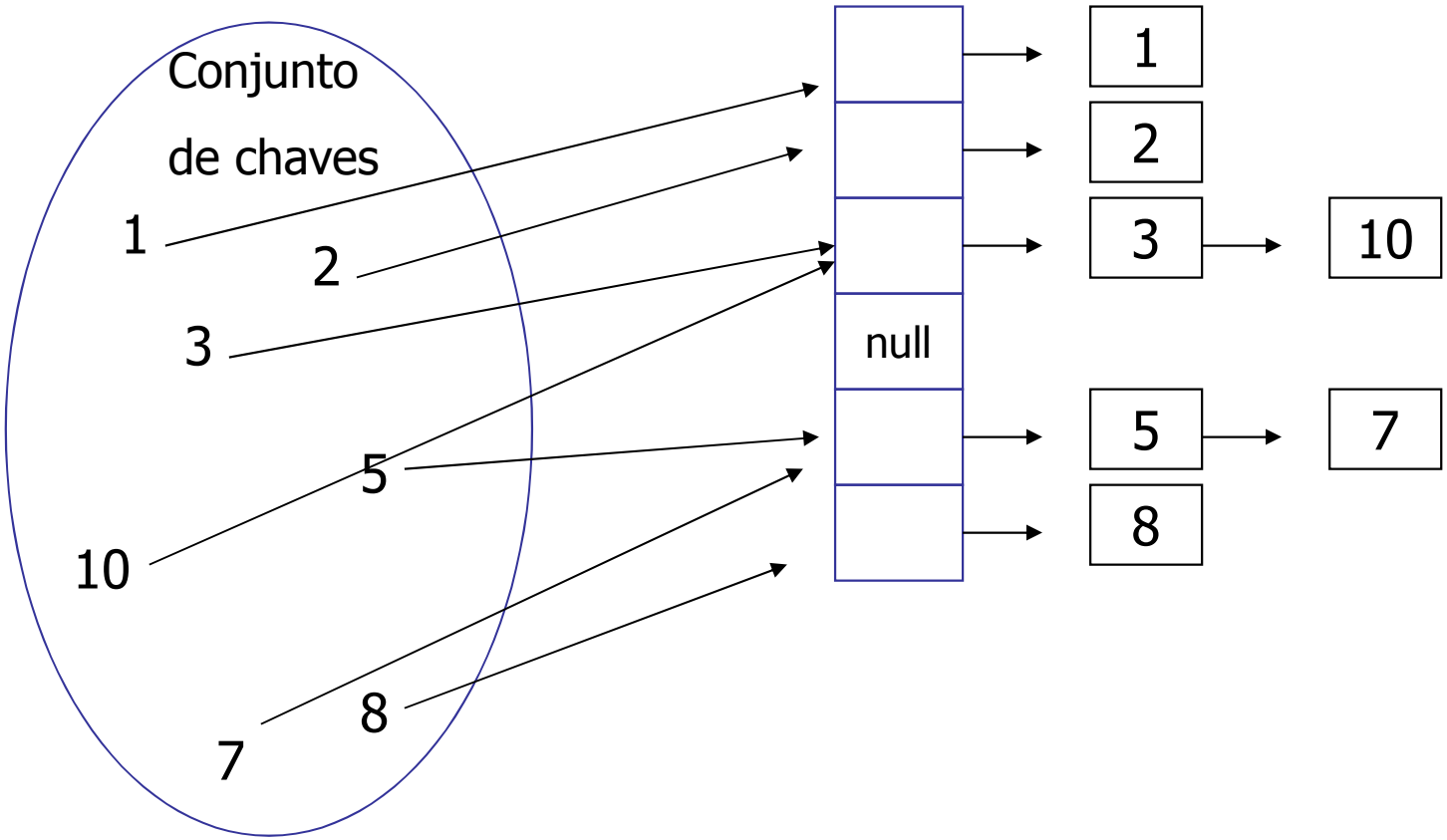




Tabelas Hash

- Elemento com chave *k* é armazenado na posição *h(k)*, onde *h* é a função de espalhamento
- Se a função *h* produzir o mesmo resultado para duas chaves, então temos uma colisão
- O que fazer nestes casos?

Tabelas Hash:encadeamento



- h(3)=h(10) e h(7)=h(5)

Tabelas Hash:encadeamento (2)

- Operações sobre uma tabela *hash T* quando as colisões são resolvidas por listas simpl. ligadas:

insert(T,x)
insere x no início da lista T[h(chave[x])]

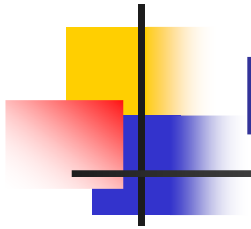
O(1)

search(T,k)
procura por um elemento com a chave k na lista T[h(k)]

O(n)

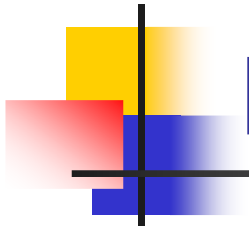
delete (T,x)
elimina x da lista T[h(chave[x])]

O(n)



Escolha da Função Hash

- Transformamos qualquer chave em um número natural
- Depois calculamos a posição a ser inserida, transformando o número natural, em uma das possíveis posições, dentro da tabela hash
  - Método da divisão
  - Método da multiplicação



Função Hash

- Método da divisão

  h(k) = k % M

  Onde M é o tamanho da tabela

- O valor de M deve ser de preferência um número primo. Por quê?
  - O uso de números primos melhora a distribuição dos elementos porque dificulta a formação de padrões em posições da tabela
- O método da divisão é a forma mais comum de função hash

Exemplos (M= 12 e M=13)

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	…							
24	25	26	27	…							
36	37	38	39	…							
48	49	50	51	…							

M=12

M=13

- O 12 tende a agrupar elementos com muitas propriedades matemáticas em comum
- Por exemplo, busca por uma chave que é divisível por 2

0	1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	…								
26	27	28	29	…								
39	40	41	42	…								
52	53	54	55	…								

Função Hash

- Método da multiplicação

$$h(k) = \lfloor m((k \cdot A) \bmod 1) \rfloor$$

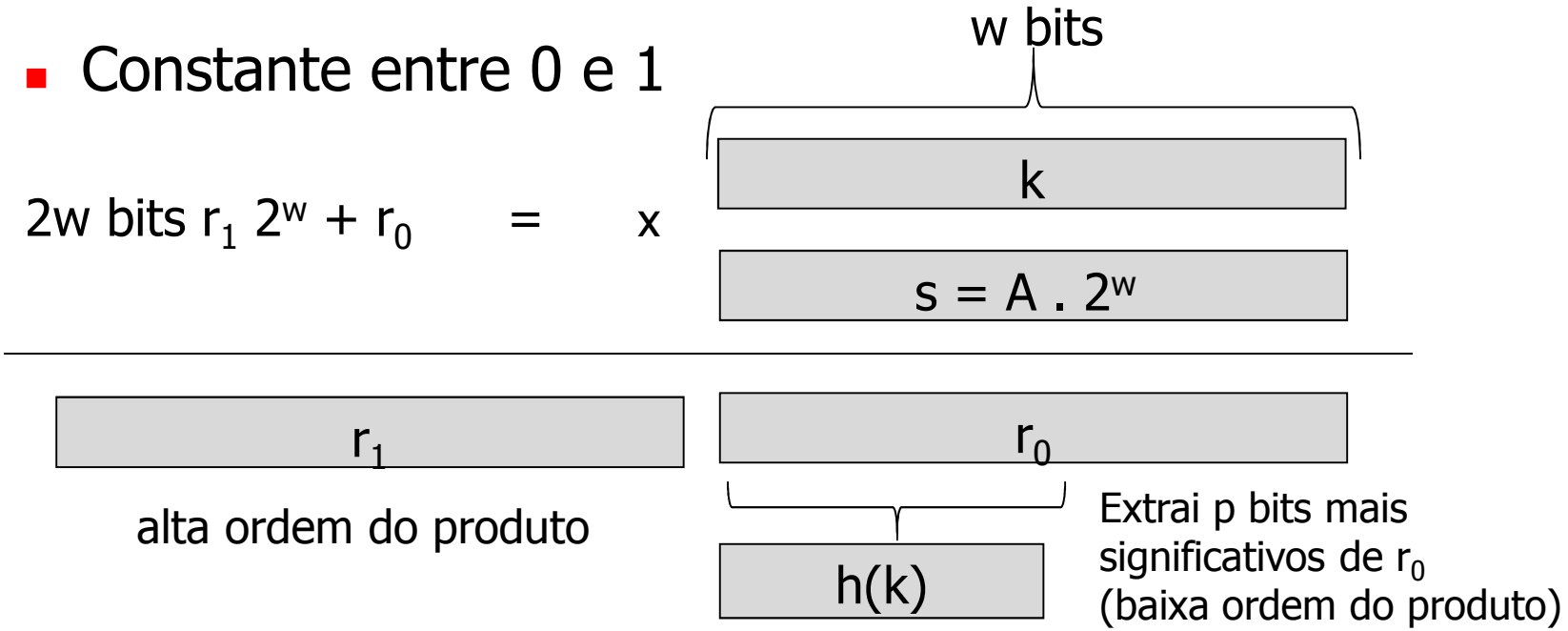
Onde:

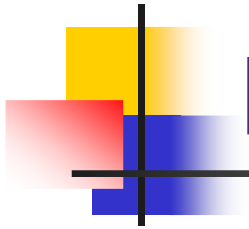
  - *m* é o tamanho da tabela
  - *A* é uma constante entre 0 e 1
  - *(k.A) mod 1* = parte fracionária de *kA (kA-⌊kA⌋)*

- O valor de *m* não é crítico e pode ser 2^p (para algum inteiro *p*) para facilitar operações
- Sugestão de Knuth: $A = (\sqrt{5} - 1)/2$

Método da Multiplicação Hash

- Suponha tamanho da palavra da máquina seja *w* bits e que *k* se encaixe em uma única palavra
- Restringimos A a ser uma fração da forma $s/2^w$, onde s é um inteiro no intervalo $0 < s < 2^w$
  - Constante entre 0 e 1



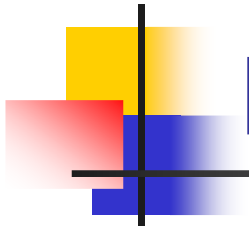


Exercício

- Suponha que temos k=123456, p=14, $m=2^{14}=16384$ e w=32. Qual seria o valor de h(k)=?

Exercício

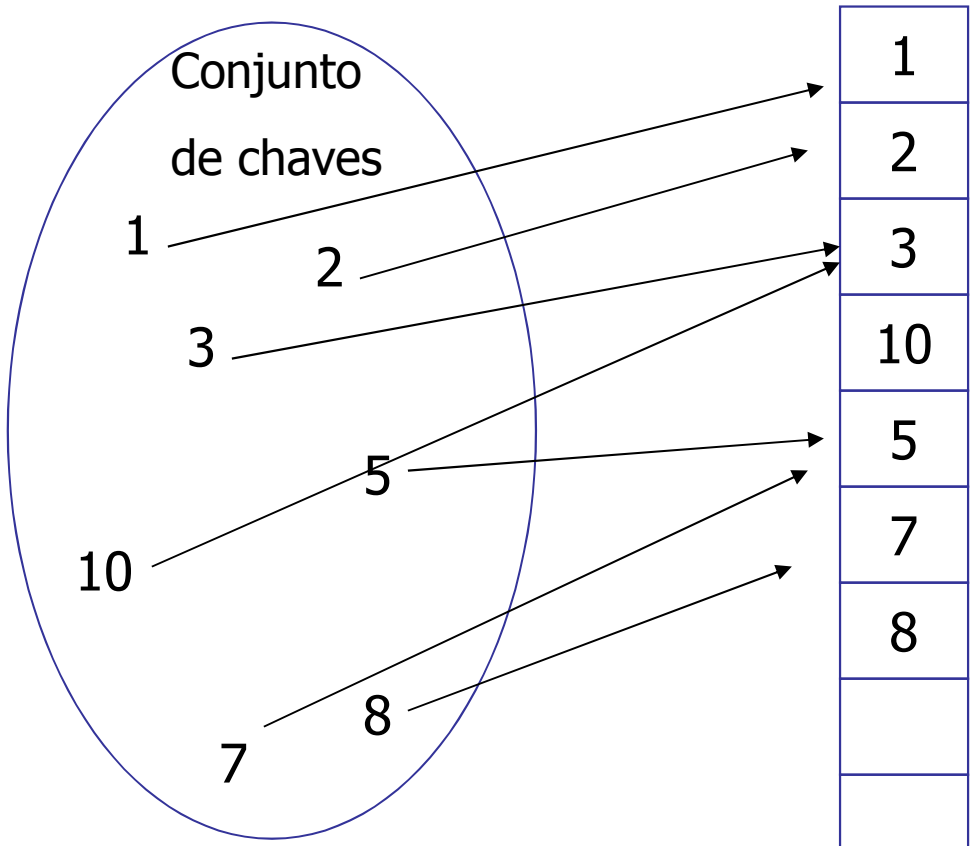
- Suponha que temos k=123456, p=14, $m=2^{14}=16384$ e w=32. Qual seria o valor de h(k)=?
  - Escolhemos $A = \frac{s}{2^{32}}$ mais próximo de $A = \frac{\sqrt{5}-1}{2} = 0{,}618033\ldots$
  - Portanto, $A= 2654435769/2^{32}$ ($s \cong 2654435769$)
  - k.s=327706022297664= 1001010100000110000000010000011001100000001000000=(76300.2^{32})+17612864
  - r_1=76300 e r_0=17612864
  - O 14 bits mais significativos de r_0, h(k)=67



Exercício

- Implemente os métodos de inserção, remoção e busca para uma tabela hash com encandeamento

Hash com Endereçamento Aberto

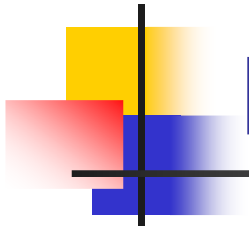


- h(3)=h(10) e h(7)=h(5)

Não há nenhuma lista ou elemento armazenado fora da tabela

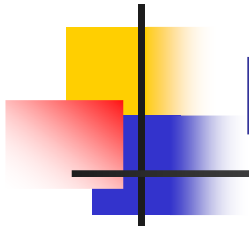
Quantas colisões nós temos?

Colisão entre 3 e 10; 5 e 7; 7 e 8



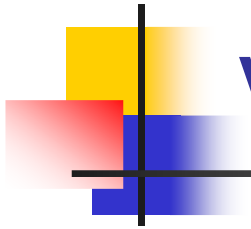
Endereçamento Aberto

- Cada entrada da tabela contém um elemento do conjunto ou NIL
  - Não existe lista e elemento armazenado fora da tabela
- Quando há colisão, nós calculamos uma nova posição de inserção
- A tabela hash pode ficar cheia
  - fator de carga (n/m) < 1
- Como remover elementos no hash de endereçamento aberto?
  - Examinamos sistematicamente as posições da tabela até encontrarmos o elemento desejado



Exercício

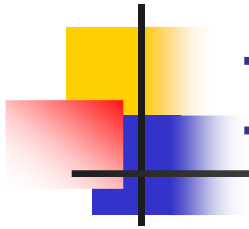
- Quais as vantagens e desvantagens do endereçamento aberto em relação ao encadeamento?



Vantagens do End. Aberto

- Evita por completo o uso de ponteiros
  - Em lugar de ponteiros, calculamos a sequência de posições a serem armazenadas
- Memória extra liberada por não se armazenarem ponteiros
  - Maior número de posições para a mesma quantidade de memória usada no encadeamento
- Gera potencialmente um menor número de colisões e recuperação mais rápida

Inserção usando End. Aberto (1)



- Examinamos sucessivamente ou "sondamos" a tabela hash até encontrarmos uma posição vazia
- Estendemos a função hash com o objetivo de incluir o **número de sondagens** (ou colisões):
  - $h: U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$
- Para toda chave k, a sequência de sondagem $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ é uma permutação de $\langle 0,1,\ldots,m-1 \rangle$
  - Para uma tabela com seis posições, cada ordem possível produz uma lista de posições sem ordenação (por exemplo, $\langle 3,4,5,6,1,2 \rangle$)

Inserção usando End. Aberto (2)

- Cada posição contém uma chave ou NIL (se a posição é vazia)

```
hash-insert(T,k) {
  i=0
  repeat j=h(k,i)
    if T[j]=NIL
      then T[j]=k
        return j
    else i=i+1
  until i=m
  error "hash table overflow"
}
```

A chave k é idêntica ao elemento que contém a chave k (são chaves sem info. satélites)

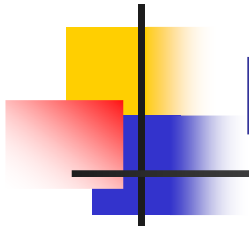
Não podemos armazenar mais elementos que o tamanho máximo da tabela

Busca usando End. Aberto

- A pesquisa pode terminar ao encontrar uma posição vazia (não mais adiante em sua sequência de sondagem)

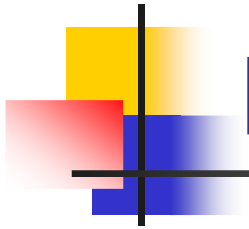
```
hash-search(T,k) {
  i=0
  repeat j=h(k,i)
    if T[j]=k
      return j
    i=i+1
  until T[j]=NIL or i=m
  return NIL
}
```

A chave k é idêntica ao elemento que contém a chave k (são chaves sem info. satélites)



Eliminação usando End. Aberto

- Como eliminar uma chave da posição i?
  - Não podemos simplesmente assinalar essa posição *i* como vazia, pois tornaria impossível recuperar qualquer chave *k*
- Uma solução é, assinalar a posição armazenando nela, com o valor **DELETED**. Quais seriam as modificações no *hash-insert* e *hash-search*?
  - Exige modificar *hash-insert* para tratar tal posição
  - Nenhuma modificação para *hash-search*, pois ele passará sobre valores **DELETED**



End. Aberto em C: Função Hash

- Definição de um tipo de nodo

```
typedef struct node {
  int data;
  int state; /*0->NIL, 1->DELETED, 2->BUSY*/
} node_hash;
```

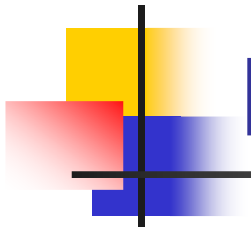
- Calcula a função de espalhamento

```
int hash_func(int k, int m, int i) {
    return ((k+i)%m);
}
```

End. Aberto em C: Inserção

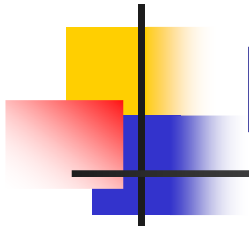
- Insere um elemento *k* na tabela *T* de tamanho *m*

```
int insert_hash(node_hash *T, int m, int k) {
  int j, i = 0;
  do {
    j = hash_func(k, m, i);
    if (T[j].state == NIL || T[j].state == DELETED) {
      T[j].data = k; T[j].state = BUSY;
      return j;
    } else i++;
  } while(i < m);
  return -1;
}
```



End. Aberto em C: Busca

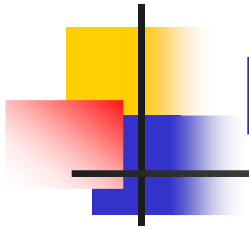
```
int search_hash(node_hash *T, int m, int k, int i) {
  int j;
  do {
    j = hash_func(k, m, i);
    if (T[j].data == k)
      return j;
    i++;
  } while ((T[j].state != NIL) && (i < m) )
  return -1;
}
```



Exercício 1

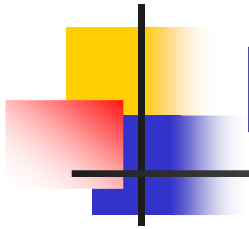
- Implemente o método da remoção em uma tabela hash de endereçamento aberto

```
node_hash *create_hash(int m) {
  node_hash *temp; int i;
  temp = (node_hash*)malloc(m*sizeof(node_hash))
  if (temp != NULL) {
    for (i=0; i<m; i++)
      temp[i].state = 0;
    return temp;
  }
  else exit(0);
}
```



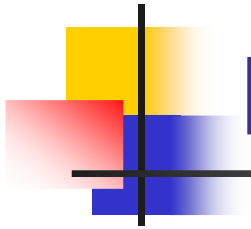
End. Aberto em C: Remoção

```
int remove_hash(no_hash *T, int m, int k) {
  int i;
  i = search_hash(T, m, k, 0);
  if (i == -1)
    return -1;
  else {
    T[i].state = DELETED;
    return 1;
  }
}
```



Exercício 2

- Qual seria o consumo de memória se implementarmos uma tabela hash de 1M, 10M e 100M de posições usando o encadeamento e o endereçamento aberto?

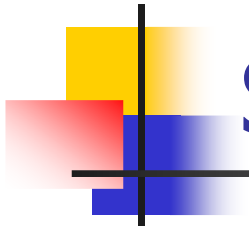


Hash Uniforme

- Supomos que cada chave tem igual probabilidade de ter qualquer das *m!* permutações de *⟨0,1,...,m-1⟩* como sua sequência de sondagem
  - O hash uniforme é difícil de implementar e na prática são usadas aproximações
- Após inserções/eliminações, o desempenho das funções sofre uma queda substancial
  - falta de organização na estrutura
- Três técnicas para calcular as sequências de sondagem: linear, quadrática e hash duplo

Sondagem Linear (1)

- Simples para calcular a sequência de sondagem
- Dada $h': U \rightarrow \{0,1,\ldots,m-1\}$ (função auxiliar), o método usa a função hash
  - $h(k,i)=(h'(k)+i) \bmod m$, para $i=0,1,\ldots,m-1$
- Dada a chave k, a primeira posição sondada é $T[h'(k)]$ (posição dada pela função auxiliar)
  - Se a posição calculada já está ocupada, então a próxima posição disponível na tabela (p.e., T[h'(k)+1]) será ocupada pela chave k (até o limite de $T[m-1]$)
  - Voltamos às posições $T[0]$, $T[1]$,... até finalmente sondarmos a posição $T[h'(k)-1]$

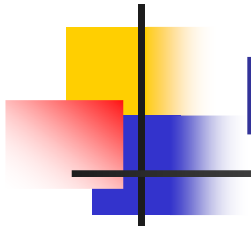


Sondagem Linear (2)

- Sofre de um problema conhecido como agrupamento primário
  - Longas sequências de posições ocupadas são construídas, aumentando o tempo médio de pesquisa
- Surgem agrupamentos, pois uma posição vazia precedida por *i* posições completas é preenchida com probabilidade *(i+1)/m*
  - Se aumentarmos o número de posições completas, aumentamos a probabilidade de termos pos. vazias
  - Sequência de posições ocupadas tendem a ficar mais longas, e o tempo médio de pesquisa aumenta

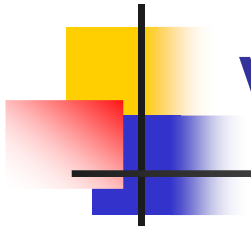
Sondagem Quadrática

- Utiliza uma função hash da forma
  - $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, onde h' é uma função hash auxiliar, $c1$ e $c2 \neq 0$ são constantes auxiliares e $i=0,1,\ldots,m-1$
- Posições sondadas são deslocadas por quantidades que dependem da forma quadrática do número da sondagem i
- Se duas chaves têm a mesma posição de sondagem inicial, então suas sequências de sondagem são iguais (agrupamento secundário)



Hash Duplo

- Um método mais efetivo de fazer sequência de sondagem é por hash duplo
  - $h(k,i)=(h_1(k)+ih_2(k)) \bmod m$, onde h_1 e h_2 são funções hash auxiliares
- A posição inicial sondada é $T[h_1(k)]$
  - Posições de sondagem sucessivas são deslocadas a partir de posições anteriores pela quantidade h2(k), módulo m
- A sondagem inicial e o deslocamento dependem da chave k



Valores de $h_2(k)$ e m

- O valor ***$h_2(k)$*** e o tamanho ***m*** da tabela devem ser **primos** entre si para que a tabela hash inteira possa ser pesquisada
  - Pemitir que m seja uma potência de 2 e projetar h_2 de modo que ele sempre produza um número ímpar
  - Permitir que m seja primo e projetar h_2 de forma que ele sempre retorne um inteiro positivo menor que m
    - $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$

Exercício: Hash Duplo

- Temos uma tabela hash T de tamanho 13. Construa T para armazenar a sequência ⟨13, 79, 69, 72, 98, 14⟩
  - *h(13,0)=(0+0×3) mod 13 = 0*
  - *h(79,0)=(1+0×3) mod 13 = 1*
  - *h(69,0)=(4+0×4) mod 13 = 4*
  - *h(72,0)=(7+0×7) mod 13 = 7*
  - *h(98,0)=(7+0×11) mod 13 = 7*
  - *h(98,1)=(7+1×11) mod 13 = 5*
  - *h(14,0)=(1+0×4) mod 13 = 1*
  - *h(14,1)=(1+1×4) mod 13 = 5*
  - *h(14,2)=(1+2×4) mod 13 = 9*

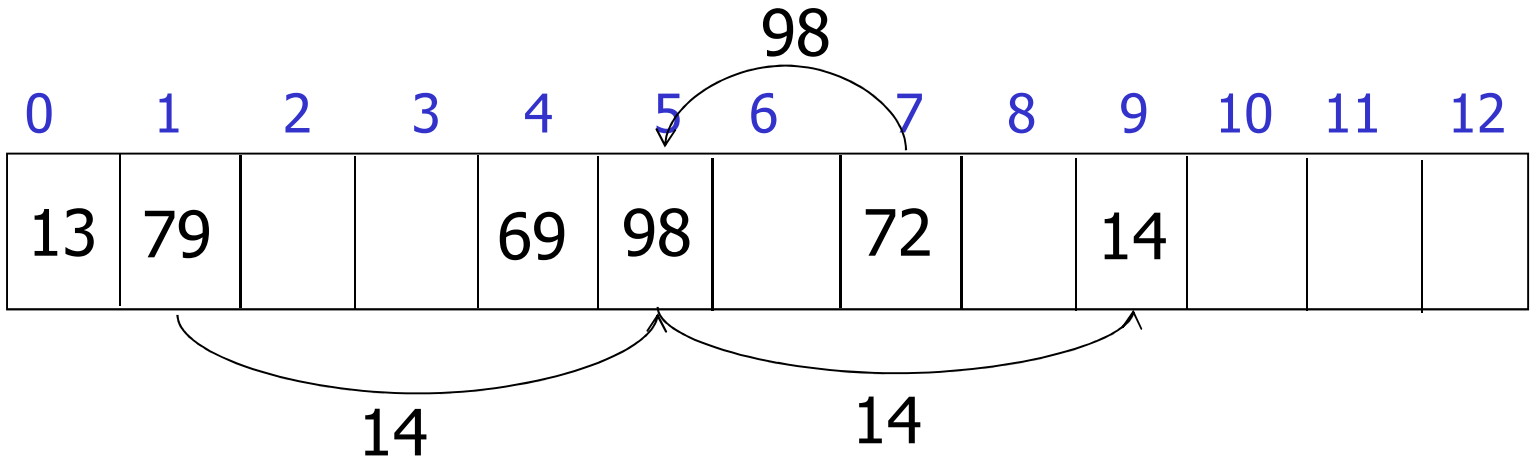
$h_1(k) = k \bmod 13$

$h_2(k) = 1 + k \bmod 11$

$h(k,i) = (h_1(k) + i h_2(k)) \bmod 13$

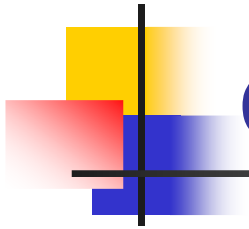
Exercício: Hash Duplo

- Temos uma tabela hash de tamanho 13 com *h1(k)=k mod 13* e *h2(k)=1+(k mod 11)*



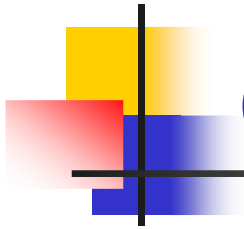
- Como *h(14,0)=1 mod 13* e *h(14,1) = 5 mod 13*, a chave 14 será inserida na posição 9, depois que as posições 1 e 5 tiverem sido examinadas e se descobrir que elas já estão ocupadas

Opções para o armazenamento de dados



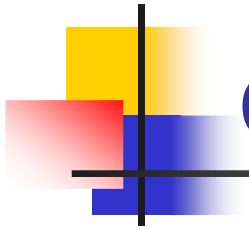
- Vetores ordenados
  - Inserção O(n) *
  - Remoção O(n) *
  - Pesquisa O(log n) *
  - Espaço extra –
  - Número de elementos é fixo
  - * caso esperado

Opções para o armazenamento de dados



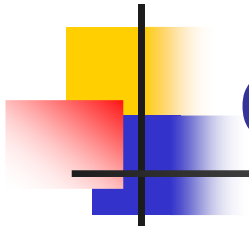
- Listas encadeadas
  - Inserção O(1)
  - Remoção O(n)
  - Pesquisa O(n)
  - Espaço extra – 1 ponteiro por elemento (ou 2)
  - Número de elementos é ajustado de acordo com as necessidades

Opções para o armazenamento de dados

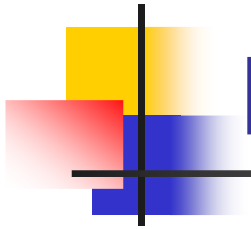


- Árvores
  - Inserção O(log n) *
  - Remoção O(log n) *
  - Pesquisa O(log n) *
  - Espaço extra – 2 ponteiros (ou 3 ponteiros)
  - Número de elementos é ajustado de acordo com as necessidades
  - * caso esperado

Opções para o armazenamento de dados

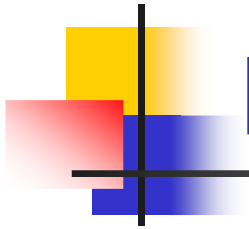


- Hash
  - Inserção O(1) *
  - Remoção O(1) *
  - Pesquisa O(1) * (* caso esperado)
  - Espaço extra – 1 ou 2 ponteiros (listas simplesmente ou duplamente ligadas)
  - Número de elementos pode ser ajustado de acordo com as necessidades, mas há necessidade de estimativa prévia



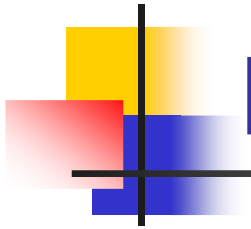
Função Hash

- O primeiro passo para a implementação de um hash é a transformação da chave em número. Exemplos:
  - strings*: $h(s)=h_1(s[0])+h_2(s[1])+\ldots$*, com *hi(x)* sendo funções criadas com geradores aleatórios
  - números inteiros: h(x) = x
  - Números reais: Podemos transformar sua representação binária em número natural



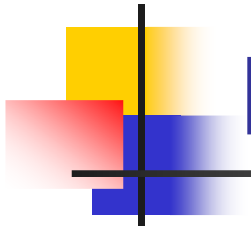
Exercício

- Dê exemplos de aplicações onde, dadas as opções de uso de um hash, uma árvore binária, uma lista encadeada e um vetor ordenado:
  - a) Um hash é a melhor opção
  - b) Uma árvore é a melhor opção
  - c) Um vetor é a melhor opção
  - d) Uma lista encadeada é a melhor opção



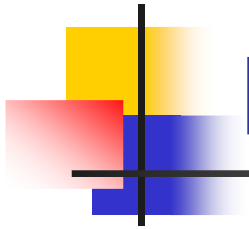
Dicionários (1)

- Dicionários armazenam elementos, que podem ser rapidamente localizados através de **chaves**
- Exemplo: contas bancárias
  - Cada conta é um objeto identificado por um código único
  - Armazena ainda diversas informações adicionais
    - saldo da conta
    - o nome e o endereço do correntista
    - histórico dos depósitos e retiradas
  - Um aplicação que necessite manipular a conta, precisa fornecer o código da conta como uma **chave de busca**



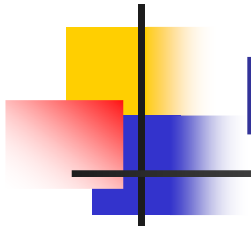
Dicionários (2)

- Um elemento tem duas partes: uma chave e um dado satélite
- Operações:
  - **Busca(S, k)** – *operação de consulta que retorna um elemento **x** tal que x.chave = k*
  - **Insere(S, x)** – *Insere um novo elemento **x** em S*
  - **Remove(S, x)** – *remove o elemento **x** de **S***



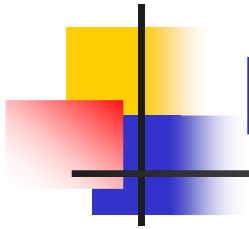
Dicionários (3)

- Suporte à ordenação (*min, max, sucessor, predecessor*) não é necessário
- É suficiente que as chaves possam ser comparadas com relação a **igualdade**



Dicionários (4)

- Várias estruturas de dados podem ser usadas para implementar dicionários
  - arranjos, listas encadeadas (ineficientes)
  - **Tabelas Hash**
  - Árvores Binárias de Pesquisa
  - Árvores AVL
  - Árvores-B



Exercício: Problema Prático (1)

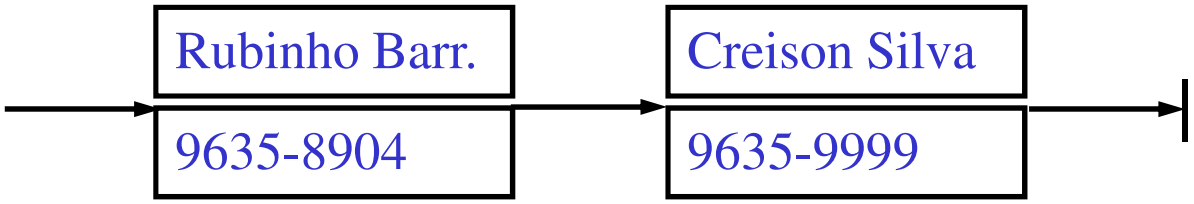
- A Fala! é uma grande companhia telefônica que precisa identificar rapidamente os nomes de quem faz chamadas telefônicas:
  - Dado um número de telefone (de 8 digitos), encontrar o nome de quem está ligando
  - Os números de telefone se encontram na faixa de 0 a $r = 10^8 - 1$
  - A busca deve ser bastante eficiente!!!

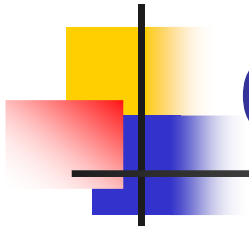
Problema Prático (2)

- Como projetar este dicionário
  - Endereçamento direto – arranjo indexado pela chave:
    - Leva tempo O(1),
    - Ocupa espaço O(*r*) – Muito espaço desperdiçado !!!

(vazio)	(vazio)	Rubinho Barrichelo	(vazio)	(vazio)
9635-9804	9635-8903	9635-8904	9635-8905	9635-8906

  - Lista encadeada: tempo O(*n*), espaço O(*n*)



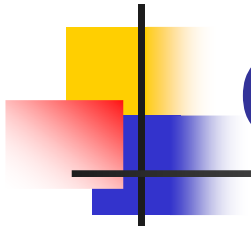


Outra Solução (1)

- **Tabela Hash** -- O(1) tempo esperado, espaço O(*n+m*), onde *m* é o tamanho da tabela

- Como um arranjo, mas utiliza uma função para mapear uma longa faixa de valores em uma faixa menor
  - Ex.: Recebe a chave original e calcula o resto da divisão inteira (módulo) pelo tamanho do arranjo
  - Mantém o resultado como um índice

Outra Solução (2)



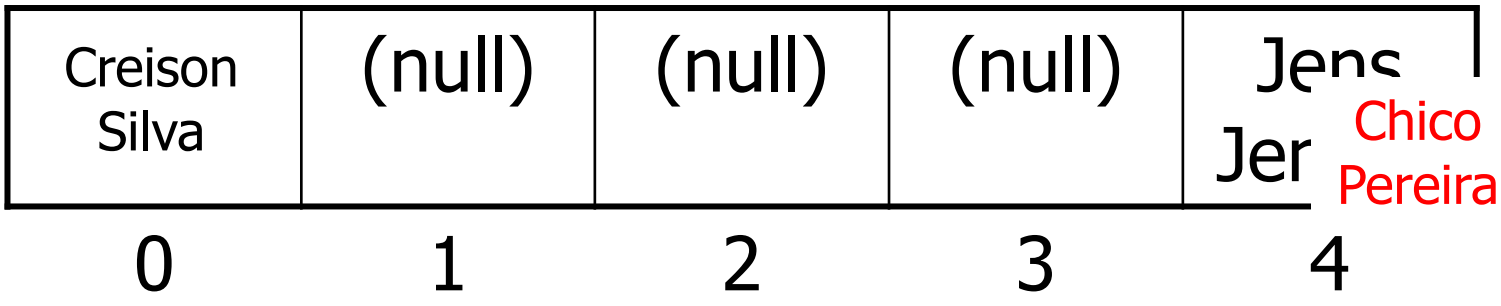
- Inserir (9635-8904, Rubinho Barrichelo) em um tabela hash com 5 entradas
  - 96358904 mod 5 = 4

(null)	(null)	(null)	(null)	Rubinho Barrichelo
0	1	2	3	4

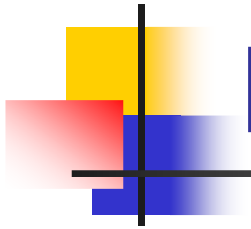
- A busca usa o mesmo processo: calcula-se o valor de hash para a chave e verifica-se a posição do array gerado

Outra Solução (3)

- Inserir (9635-8900, Creison Silva)
  - 96358900 mod 5 = 0
- Inserir (9635-8004, Chico Pereira)
  - 96358004 mod 5 = 4



A solução proposta requer uso de listas encandeadas



Exercício

- Compare a eficiência da tabela hash com encadeamento e com hash duplo. Qual delas você sugeriria para a companhia telefônica fala!