Hashing

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Hashing

- Hashing, ou transformação de chave, ou espalhamento é um algoritmo usado para solucionar de maneira eficiente problemas de armazenamento e recuperação de dados
  - Normalmente muito eficiente para sistemas de arquivos grandes

Hashing

- O método de *hashing* consiste em armazenar registros em uma tabela e endereçá-los diretamente a partir de uma transformação aritmética sobre a chave de pesquisa
- *Hashing*, definição:
  - Fazer picadinho de carne e vegetais para cozinhar
  - Fazer bagunça

Hashing

- ➔ Um método de *hashing* tem 2 etapas
  1. Computar o valor da função de transformação (ou *hashing*)
     - ➔ Transforma a chave de pesquisa em endereço da tabela
  2. Lidar com colisões
     - ➔ Pois duas ou mais chaves podem ser transformadas em um mesmo endereço na tabela

Hashing

- ➔ Caso chaves fossem inteiros de 1 a *n*, seria possível armazená-las de acordo com o índice da tabela, com acesso imediato
- ➔ Porém, nem sempre os dados são assim na vida real

Hashing

- Caso a tabela pudesse armazenar 100 chaves
  - E as chaves fossem decimais de 4 dígitos
  - Existiriam 10.000 chaves possíveis
  - A função de *hashing* não pode ser de 1 pra 1
- Mesmo com poucos dados não dá para garantir que não haja colisão!
- Então precisamos resolver tais colisões de algum jeito

Hashing

- Caso a tabela pudesse armazenar 100 chaves
  - E as chaves fossem decimais de 4 dígitos
  - Existiriam 10.000 chaves possíveis!
  - A função de *hashing* não pode ser de 1 pra 1
- Mesmo com poucos dados não dá para garantir que não haja colisão!
- Então precisamos resolver tais colisões de algum jeito

Função de *Hashing*

Função de *Hashing*

- A função de transformação deve mapear chaves em inteiros que estejam dentro do intervalo do tamanho da tabela
  - [0..M-1]
- Função ideal:
- Simples de ser computada
- Para cada chave, a probabilidade de qualquer saída possível ser escolhida é igual

Função de *Hashing*

- Como a função opera sobre números, primeiro devemos transformar chaves não-numéricas em números (se necessário)
  - Algum tipo de ordenação geralmente é usado

Função de *Hashing*

- Uma função que costuma funcionar bem é a que usa módulo do tamanho total de chaves
  - $h(K) = K \bmod M$
- K é um inteiro correspondente à chave
- Obtido com uma soma usando um conjunto de pesos p
  - $K = \sum_{i=1}^{n} Chave[i] * p[i]$

Função de *Hashing*

- n
  - número de caracteres da chave
- Chave [i]
  - representação ASCII do i-ésimo caractere da chave
- p[i]
  - inteiro de um conjunto de pesos gerados aleatoriamente para $1 \leq i \leq n$

Função de *Hashing*

- ➔ O uso de pesos é vantajoso pois dois conjuntos de pesos diferentes levam a duas funções de transformações diferentes

Função de Hashing

```
TipoIndice h(TipoChave chave, TipoPesos p)
{
    unsigned int soma = 0;
    int comp = strlen(chave);
    for(int i = 0; i < comp; ++i)
        soma += (unsigned int)chave[i]*p[i];
    return (soma%M);
}
```

Função de Hashing

```
void gerarPesos(TipoPesos p){
    struct timeval semente;
    gettimeofday(&semente, NULL);
    srand((int)(semente.tv_sec + 1000000*semente.tv_usec));
    for(int i = 0; i < N; ++i){
        p[i] = 1+(int) (10000.0*rand()/(RAND_MAX+1.0));
    }
}
```

Função de *Hashing*

- A escolha do M é importante!
  - De preferência, um primo
- Mas evitar números primos obtidos com:
  - $b^i \pm j$
- b
  - Base do conjunto de caracteres (128 para ASC II)
- i e j
  - pequenos inteiros

Função de *Zobrist*

Função de *Zobrist*

- Se usarmos a função anterior para espalhar uma palavra, cada caractere dela seria multiplicado pelo peso
- Zobrist propôs uma solução que troca espaço por tempo
- Sua função gera aleatoriamente um peso diferente para cada um dos 256 caracteres ASCII possíveis na i-ésima posição da chave

Função de *Zobrist*

```
void gerarPesosHZ(TipoPesos p){
    struct timeval semente;
    gettimeofday(&semente, NULL);
    srand((int)(semente.tv_sec + 1000000*semente.tv_usec));
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < TAMALFABETO; ++j)
            p[i][j] = 1 + (int)(10000.0*rand()/(RAND_MAX+1.0));
}
```

Função de *Zobrist*

- Para obter a posição da função de espalhamento com estes pesos vai ser necessário o mesmo número de adições da função anterior, mas não serão realizadas multiplicações!
- Porém, o espaço para armazenar o dicionário é O(n*|Σ|)
  - |Σ| é o tamanho do alfabeto
- Antes era preciso apenas O(n)

Função de *Zobrist*

```
TipoIndice hZobrist(TipoChave chave, TipoPesos p)
{
    unsigned int soma = 0;
    int comp = strlen(chave);
    for(int i = 0; i < comp; ++i)
        soma += p[i][(unsigned int)chave[i]];
    return (soma%M);
}
```

Função de Hashing

```
TipoIndice h(TipoChave chave, TipoPesos p)
{
    unsigned int soma = 0;
    int comp = strlen(chave);
    for(int i = 0; i < comp; ++i)
        soma += (unsigned int)chave[i]*p[i];
    return (soma%M);
}
```

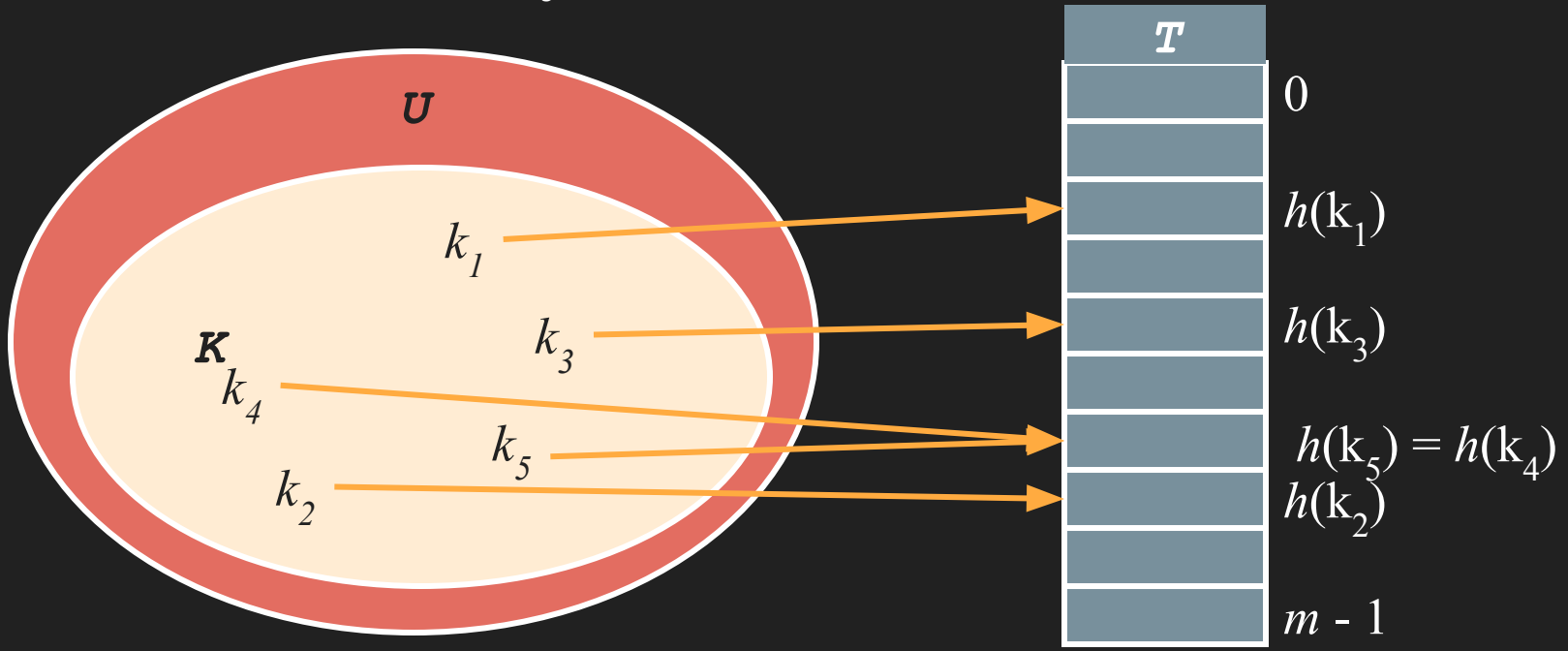
Resolvendo Colisões - Listas Encadeadas

Listas Encadeadas

- ➔ Se criarmos uma lista linear encadeada para cada endereço da tabela, resolvemos o problema das colisões
- ➔ Caso haja colisão de mais de uma chave, basta encadeá-las na lista
- ➔ Também conhecido como *chaining*

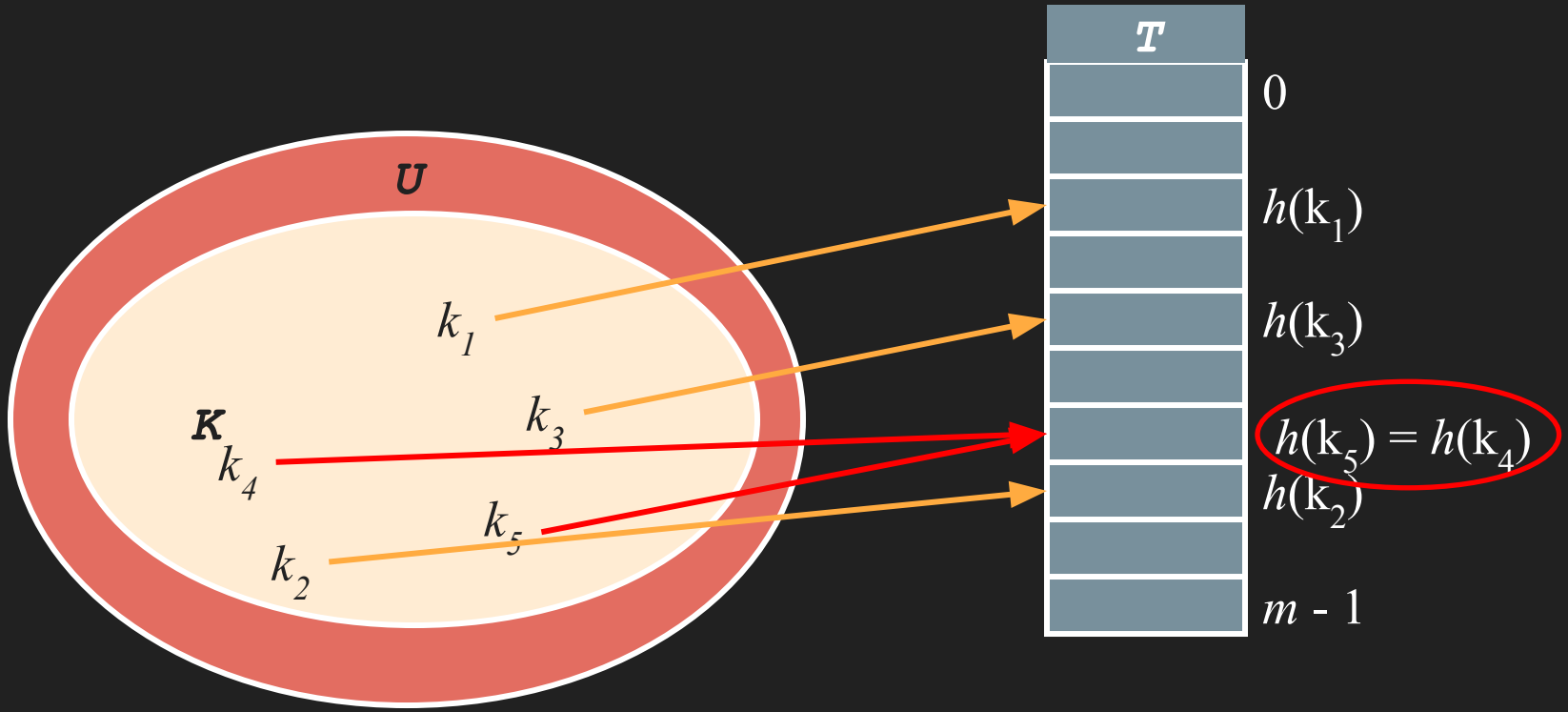
Listas Encadeadas [1]

➔ Mapeia o universo de chaves possíveis U para um conjunto {0,...,m-1} usando **funções hash**.



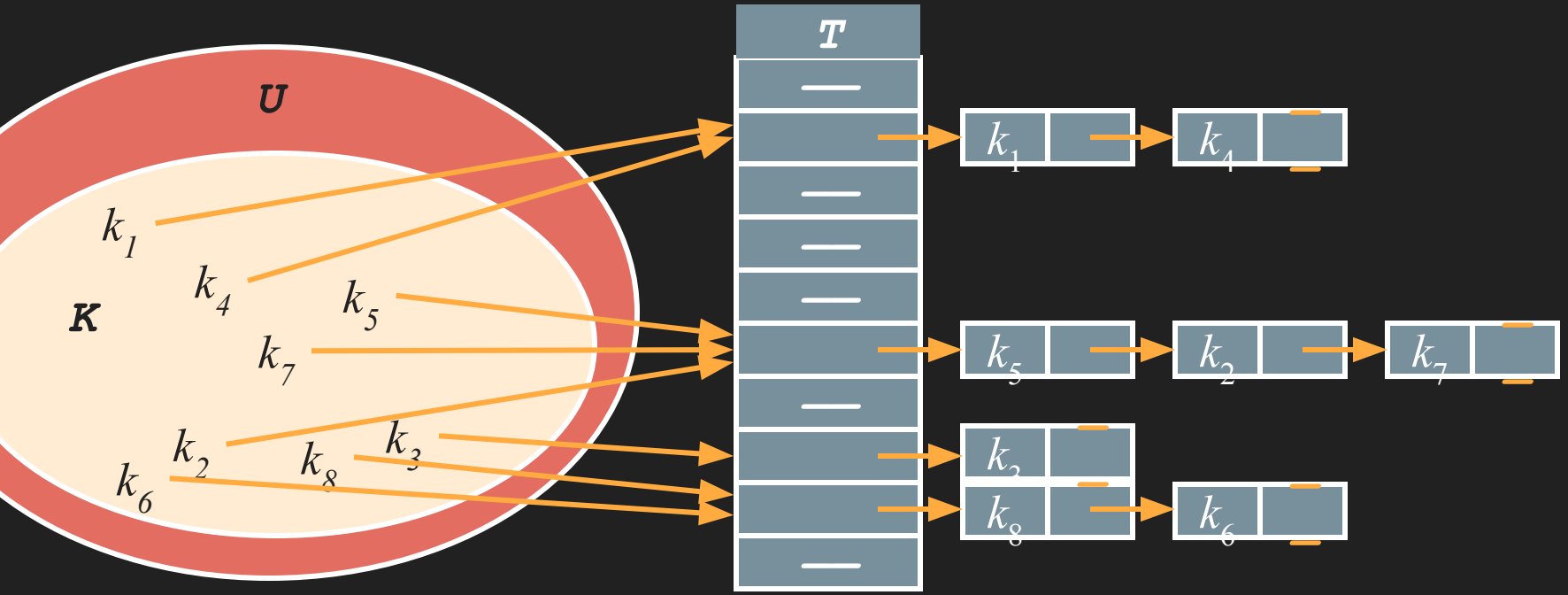
Listas Encadeadas [1]

- ➔ U: Universo de chaves possíveis
- ➔ K: chaves atuais



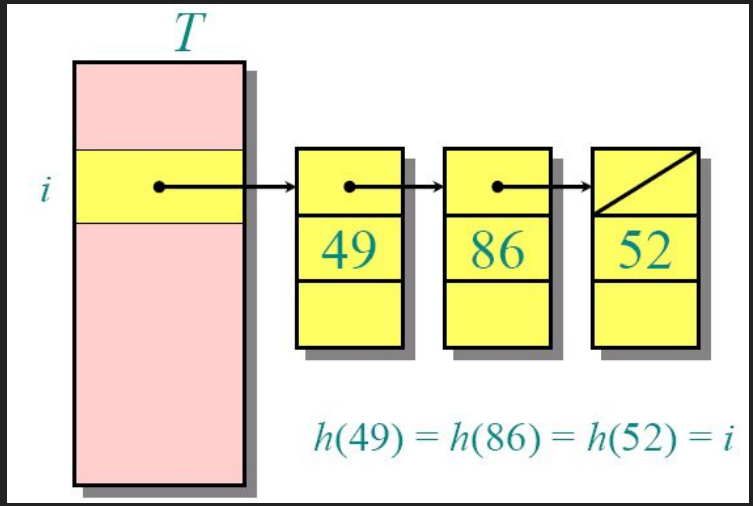
Listas Encadeadas [1]

- ➔ Solução: Chaining
  - ◆ Insere elementos atribuídos ao mesmo slot em uma lista ligada



Listas Encadeadas [1]

- Pior caso ocorre quando toda chave está atribuída ao mesmo slot
- Tempo de acesso Θ(n), se |K| =n



Listas Encadeadas

➔ Vamos ver agora a implementação destes algoritmos :)

Referências

Referências

1. CORMEN, T. H.; RIVEST, R. L.; LEISERSON, C. E.; STEIN, C.. Algoritmos: teoria e prática. Elsevier, 2012.
2. ZIVIANI, N. Projeto de Algoritmos. 2º edição, Thomson, 2004.