ESTRUTURAS DE DADOS

Tratamento de Colisões

Roteiro

- Introdução
- Encadeamento Separado
- Teste Linear
- Detalhes de Implementação

Introdução

A presença de colisões, quando duas chaves k_1 e k_2 geram $h(k_1) = h(k_2)$, impede que se faça imediatamente a inserção de um novo item (k,v) diretamente em $A[h(k)]$ no arranjo A.

Para resolver as colisões, podemos utilizar tanto um espaço de memória adicional quanto um espaço no próprio arranjo.

Encadeamento Separado

Uma ideia simples para tratar colisões é fazer com que cada endereço A[*i*] seja, na verdade, um ponteiro para uma lista encadeada.

Uma boa função de hash fará com que a maior parte dos endereços esteja vazio ou com apenas um elemento.

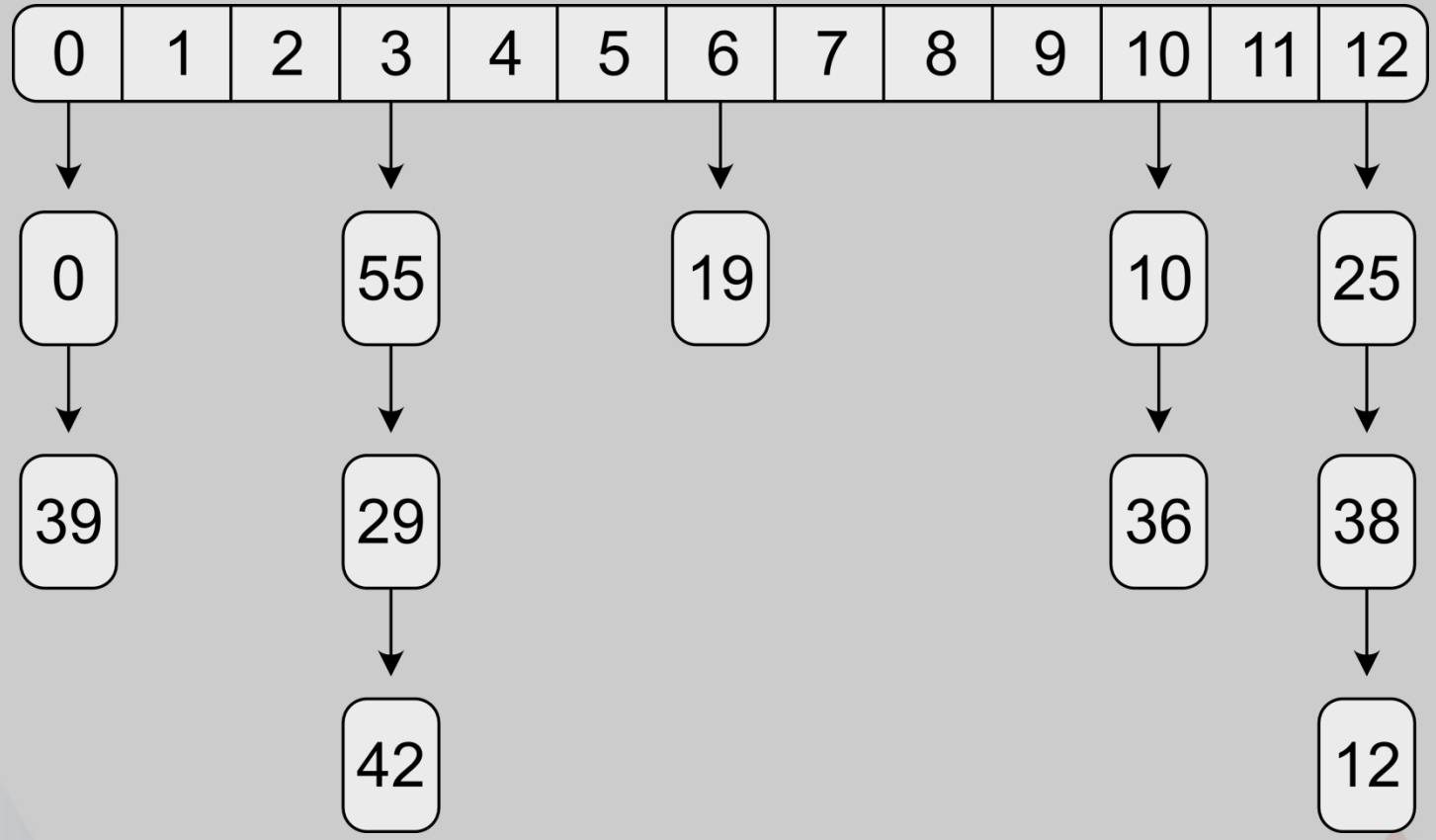
- **Ao colocar n elementos em N endereços, espera-se n/N entradas por endereço.**

- **O valor n/N (também chamado fator de carga) deveria ser limitado por uma constante, idealmente menor que 1.**

- **Implementam-se as operações insertItem, deleteItem e retrieveItem para executarem em um tempo esperado constante.**

Tabela Hash com encadeamento separado

$$h(k) = k \bmod 13$$



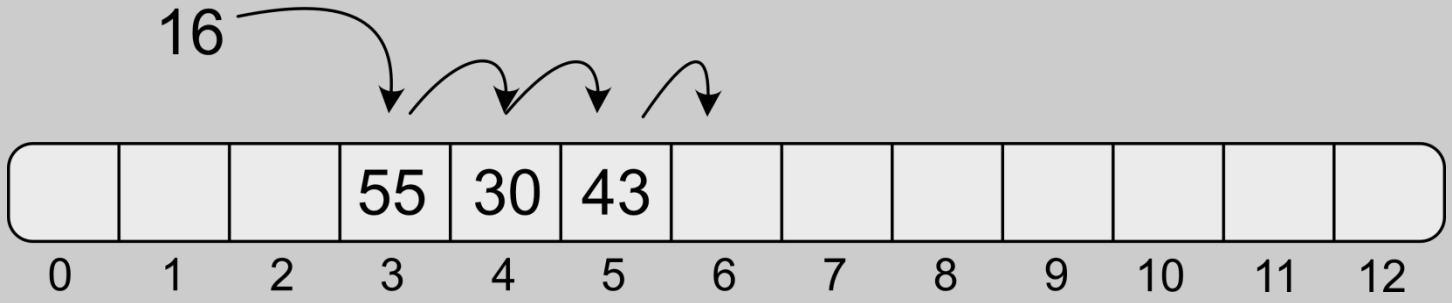
Teste Linear

As colisões serão tratadas sem alocação de memória adicional, usaremos o próprio arranjo.

Se tentarmos inserir um item (k,v) em um endereço $A[i]$ ocupado, com $i = h(k)$, tenta-se de novo no endereço $A[(i+1) \bmod N]$.

As tentativas continuam até se encontrar um endereço que aceite o novo item.

- **Assumindo que queremos colocar um elemento com chave $k = 16$ tendo uma função $h(k) = k \bmod 13$.**



- **Inicialmente, tentamos adicionar a entrada na posição 3. Como não é possível, tentamos na 4 e na 5 até chegarmos na posição 6 livre.**

- As operações **retrieveItem** e **deleteItem** devem também ser atualizadas.

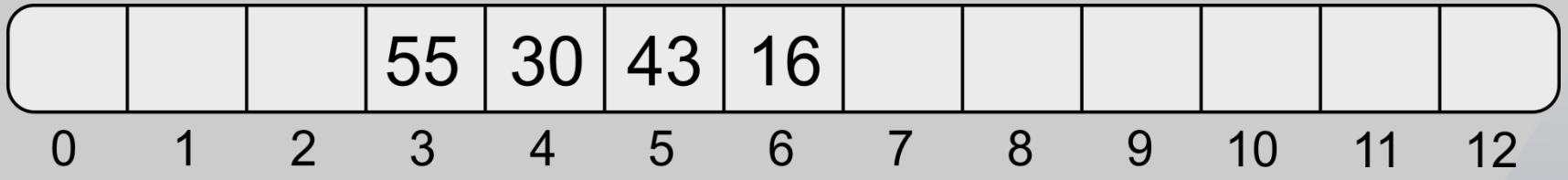
- Por exemplo, **retrieveItem** deverá examinar endereços consecutivos, iniciando em **A[*h*(*k*)]**, até encontrar a chave igual a ***k***.

- Se ***k*** não existir, então a **retrieveItem** finalizará em uma posição vazia.

- O nome "Teste Linear" ocorre porque acessar **A[*h*(*k*)]** implica em testar a chave para verificar se encontramos a entrada desejada.

- As operações **deleteItem** não poderão mais remover os itens, pois isso fariam com que alguma chave não pudesse mais ser achada pelo **retrieveItem**.

- Por exemplo, **retrieveItem** não mais achará a chave **16** se remover **55**, **30** ou **43**.



- **A solução será fazer com que deleteItem não remova o elemento, mas substitua por um marcador "disponível".**

- **Nesse caso, as buscas pela chave *k* podem pular pelos endereços "disponíveis" e continuar até encontrar a chave ou uma célula vazia.**

- **A operação *insertItem* pode usar os endereços "disponíveis" para inserir entradas.**

O teste linear agrupa os elementos em posições contíguas. As pesquisas podem se tornar lentas.

Outra estratégia de agrupamento é o Teste Quadrático, que testa posições $A[(i+f(j)) \bmod N]$.

- $j = 0,1,2,\ldots$
- $f(j) = j^2$

Isso evita o agrupamento sequencial, mas cria novos padrões de agrupamento.

Se N não for primo, então o teste quadrático poderia falhar em encontrar uma posição, mesmo havendo posições livres no arranjo.

O hashing duplo consiste em encontrar uma função h' para tratamento de colisões.

Se A[i] já está ocupado para alguma chave k, tal que $i = h(k)$, então são testados os endereços:

- **$A[(i+f(j)) \bmod N]$, para $j = 1, 2, 3, \ldots$**
- **$f(j) = j \cdot h'(k)$.**

A função de hashing secundária h' não pode resultar em zero. Uma escolha comum é:

- **$h'(k) = q - (k \bmod q)$, para algum número primo $q < N$.**

Detalhes de Implementação

Os únicos métodos que precisam ser mudados são o retrieveItem, insertItem e o deleteItem.

No deleteItem, vamos marcar os elementos removidos com o RA igual a –2.

- **Isso fará com que sejam diferentes dos elementos inicialmente vazios, que possuíam RA igual a –1.**

O insertItem irá tratar qualquer RA negativo da mesma maneira.

Em retrieveItem, o ponto retornado pela função de hash é visto como o início da busca.

```
void Hash::retrieveItem(Aluno& aluno, bool& found) {
  int startLoc = getHash(aluno);
  bool moreToSearch = true;
  int location = startLoc;
  do {
    if (structure[location].getRa() == aluno.getRa() ||
        structure[location].getRa() == -1)
      moreToSearch = false;
    else
      location = (location + 1) % max_items;
  } while (location != startLoc && moreToSearch);

  found = (structure[location].getRa() == aluno.getRa() );
  if (found) {
    aluno = structure[location];
  }
}
```

Paramos a busca com RA igual a –1, mas não com RA igual a -2

Em deleteItem, faremos uma busca semelhante ao retrieveItem antes de marcar o elemento com RA –2.

```
void Hash::deleteItem(Aluno aluno) {
  int startLoc = getHash(aluno);
  bool moreToSearch = true;
  int location = startLoc;
  do {
    if (structure[location].getRa() == aluno.getRa() ||
        structure[location].getRa() == -1)
      moreToSearch = false;
    else
      location = (location + 1) % max_items;
  } while (location != startLoc && moreToSearch);

  if (structure[location].getRa() == aluno.getRa()) {
    structure[location] = Aluno(-2,"");
    length--;
  }
}
```

Observe a adição do aluno com RA -2

Em insertItem, tratamos os elementos com RA –1 e os elementos com RA –2 como iguais.

```
void Hash::insertItem(Aluno aluno) {
  int location;
  location = getHash(aluno);
  while (structure[location].getRa() > 0)
    location = (location + 1) % max_items;
  structure[location] = aluno;
  length++;
}
```

Note que não tratamos o caso em que a estrutura está cheia. Isso levaria a um looping infinito

ESTRUTURAS DE DADOS

Tratamento de Colisões