Notas de Aula - AED1 – Tabela Hash (parte 2)
Prof. Jefferson T. Oliva

Hoje veremos tipos de hashing e formas de tratamento de colisões.

Tipos de Hashing

Estático
- Espaço de endereçamento não muda
- **Fechado**: Permite armazenar um conjunto de informações de tamanho limitado
→ Tratamento de colisões: overflow progressivo ou segunda função hash

- **Aberto**: Permite armazenar um conjunto de informações de tamanho potencialmente ilimitado
→ Tratamento de colisões: encadeamento de elementos

Hashing dinâmico
- Espaço de endereçamento pode aumentar
- Hashing extensível
- Pode aumentar se houver colisões

Tratamento de Colisões

Uma função hashing pode gerar a mesma posição para chaves diferentes. Essa situação é chamada de colisão.

Suponha que utilizamos o resto de divisão para definirmos posições em uma tabela de tamanho 50
- Se inserirmos as chaves 12 e 62, ocorrerá colisão,
→ 12 % 50 = 12
→ 62 % 50 = 12

Qualquer função hashing pode acarretar em colisões.

Em uma tabela hash deve haver uma forma para tratar colisões.

Desse modo, a estrutura da tabela hash é formada em duas partes:
- Função hashing
- Tratamento de colisões

As estratégias para tratamento de colisões são aplicadas de acordo com o tipo de hashing: estático ou dinâmico

Hashing estático

Hashing fechado: aplicação de técnicas de rehash para lidar com colisões
- Overflow progressivo
- Segunda função hash

Técnica rehash
- Se posição h(k) está ocupada, aplicar função de rehash sobre h(k), que deve retornar o próximo bucket livre: rh(h(k)).
- Uma boa função rehash cobre o máximo de índices entre 0 e o tamanho da tabela - 1 e evita agrupamento de dados.
- Além do índice resultante de h(k), na rehash também pode ser utilizada a própria chave k e outras funções hash.

Overflow progressivo
- Também conhecido como sondagem linear.
- Caso a função hash (h(k)) resulte em uma posição ocupada, tentar a próxima posição: rh(h(k)) = (h(k) + i)%B, sendo i variando de 1 a B − 1 e B (buckets) é o tamanho da tabela.
- Na primeira tentativa: i = 1
- A variável i é incrementada até que seja encontrada uma posição vazia ou todas as opções sejam esgotadas

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

ver slides de 11 a 13 (P = 80; O = 79; T = 84; E = 69; R = 82; L = 76; N = 78; I = 73)

POTTER % 12 = (80 + 79 + 84 + 84 + 69 + 82) % 12 = 478 % 12 = 10
LeNIN % 12 = (76 + 101 + 78 + 73 + 78) % 12 = 406 % 12 = 10

Overflow progressivo
- Vantagem: simplicidade
- Desvantagens: agrupamento de dados (causado por colisões); com a tabela cheia, a busca fica lenta, dificulta as inserções e remoções.

Segunda função hash
- Também conhecida como hash duplo
- Utiliza duas funções como auxiliares
→ h1(k): função hash primária
→ h2(k): função hash secundária
- Algumas boas funções auxiliares
→ h1(k) = k%B
→ h2(k) = 1 + k%(B − 1)
- Função rehash: rh(k, i) = (h1(k) + i ∗ h2(k))%B
- Vantagem: geralmente evita agrupamento de dados

- 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.

Hashing aberto
- A tabela de buckets, indo de 0 a B − 1, contém apenas ponteiros para uma lista de elementos
- Quando há colisão, o item é inserido no bucket como um novo nó da lista
- Se as listas estiverem ordenadas, reduz-se o tempo de busca
- **ver slides 17 e 18.**
- Principal vantagem
→ A tabela pode receber mais itens mesmo quando um bucket já foi ocupado
- Desvantagens:
→ Quantidade de endereços não pode ser aumentado
→ Espaço extra para as listas
→ Listas longas pode implicar em muito tempo gasto na busca

Eficiência
- Hashing fechado
→ Depende da técnica de rehash
→ 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

Hashing dinâmico

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

Exemplo de hashing dinâmico:
- Hashing extensível **(slide 21)**: conforme os elementos são inserido na tabela, o tamanho aumenta se necessário

Hashing extensível
- Em geral, trabalha-se com bits
- Após h(k) ser computada, uma segunda função f transforma o índice h(k) em uma sequência de bits
- Alternativamente, h e f podem ser unificadas como uma única função hash final
- Função hash computa sequência de m bits para uma chave k, mas apenas os i primeiro bits (i ≤ m) do início da sequência são usados como endereço
→ Se i é o número de bits usados, a tabela de buckets terá 2i entradas
→ Portanto, tamanho da tabela de buckets cresce sempre com potência de 2 (aumenta a quantidade de bits em uma unidade)
- N é o número de nós permitidos por bucket
- Tratamento de colisões: geralmente por listas encadeadas
- Exemplo **(ver slide 23)**: tabela inicialmente vazia, m = 4 (bits) e (N = 2)

Ver slides 23, 24, 25, 26 e 27.

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 sequências de inserções que façam a tabela crescer rapidamente, tendo, contudo, um número pequeno de registros

Principal desvantagem de hashing: Os elementos da tabela não são armazenados sequencialmente e nem sequer existe um método prático para percorrê-los em sequência.

Implementações:

```
static int overflowProgressivo(int hashCode, int B, tentativa){
   return (hashCode + tentativa) % B;
}


// Antes
int buscar(int key, HashT *t){
   int x = hashingF(key, t->tam);
   if (t->buckets[x] == key)
      return x;
   return -1;
}

// Depois
int buscar(int key, HashT *t){
   int x = hashingF(key, t->tam);
   int i, rh;
   if (t->buckets[x] == key)
      return x;
   else if (t->buckets[x] >= 0){
      i = 1;
      rh = x;
      while ((i < t->tam) && (t->buckets[rh] != key) && (t->buckets[rh] > -1)){
         rh = overflowProgressivo(x, i, t->tam); //(x + i) % t->tam;
         i++;
      }
      if ((i < t->tam) && (t->buckets[rh] == key))
         return rh;
   }
   return -1;
}

// Antes
int inserir(int key, HashT *t){
   int x;
   if (t != NULL){
      x = hashingF(key, t->tam);
      if (t->buckets[x] < 0){
         t->buckets[x] = key;
         return 1;
      }
   }
   return 0;
}
```

```
// Depois
int inserir(int key, HashT *t){
   int x;
   int i, rh;
   if ((t != NULL) && (key > 0)){
      x = hashingF(key, t->tam);
      if (t->buckets[x] < 0){
         t->buckets[x] = key;
         return 1;
      }else{
         i = 1;
         rh = x;
         while ((i < t->tam) && (t->buckets[rh] >= 0)){
            rh = overflowProgressivo(x, i, t->tam); //(x + i) % t->tam;
            i++;
         }
         if ((i < t->tam) && (t->buckets[rh] <= 0)){
            t->buckets[x] = key;
            return 1;
         }
      }
   }
   return 0;
}

//Antes
int remover(int key, HashT *t){
   int x;
   if (t != NULL){
      x = hashingF(key, t->tam);
      if (t->buckets[x] == key){
         t->buckets[x] = -1;
         return 1;
      }
   }
   return 0;
}


//Depois
int remover(int key, HashT *t){
   int x;
   if (t != NULL){
      x = buscar(key, t->tam);
      if (t->buckets[x] == key){
         t->buckets[x] = 0;
         return 1;
      }
   }
   return 0;
}
```

Exercício: reimplemente as funções anteriores, mas em vez de usar overflow progressivo, implemente uma segunda função hash (hash duplo) para tratamento de colisões.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Madalosso, E. Hashing Universal. AE22CP - Algoritmos e Estrutura de Dados I. Notas de Aula. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2019.

Oliva, E. Tratamento de Colisões. AE22CP - Algoritmos e Estrutura de Dados I. Notas de Aula. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2020.

Rosa, J. L. G. Métodos de Busca. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2018.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.