Tabelas de Hash

Algoritmos e Complexidade

LEI-LCC 2010-2011

MBB

Novembro de 2010

Tabelas e Acesso a Informação

- As estruturas de dados apresentadas anteriormente têm como objectivo o armazenamento de informação, para posterior recuperação.
- No entanto, como as pesquisas se baseiam apenas em comparações sobre os valores das entradas, há um limite para a eficiência das operações de acesso à informação: não é possível fazer pesquisas com complexidade temporal inferior a O(log n).
- As estruturas de dados apresentadas nesta secção permitem quebrar este limite.
- Quando temos uma lista de N registos numerados de 1 a N, e pretendemos aceder ao registo no índice *i*, existe uma solução mais eficiente: armazenar os registos num array, e efectuar um acesso aleatório com base no índice do registo. Este tipo de acesso é a base do conceito de **tabela**.

Tabelas e Acesso a Informação

- Definição: Uma tabela com conjunto de índices I e conjunto de entradas T é uma função $I \rightarrow T$, sobre a qual se podem efectuar as seguintes operações:
  - Aceder à tabela – calcular o valor da função para um dado índice em I.
  - Alterar a tabela – alterar a própria função, modificando o valor que é devolvido para um determinado índice em I.
  - Acrescentar à tabela – adicionar um novo valor a I e a sua imagem a T.
  - Remover da tabela – retirar um valor de I, bem como a sua imagem de T.
  - Criar/Esvaziar a tabela – a função começa por ter domínio e contradomínio vazios.

Tabelas e Acesso a Informação

- Em geral há que distinguir entre a definição abstracta de Tabela e a sua implementação típica:
  - um array,
  - uma função de indexação que mapeia o conjunto *I* num conjunto intermédio de índices do array,
  - e uma função de acesso ao array que retorna o valor armazenado numa determinada entrada do array.
- Esta definição de tabela permite identificar a razão pela qual se consegue quebrar a barreira do O(log n): conceptualmente, a avaliação de uma função não depende do tamanho do seu domínio.
- Da mesma forma, o acesso a uma tabela não depende do número de elementos que nela estão armazenados (note-se que na prática isto não é totalmente verdade).

Tabelas de Hash

- O caso mais simples de uma tabela consiste no caso em que o conjunto *I* da tabela coincide com o conjunto de índices do array sobre o qual ela é implementada.
- Quando isto não é possível, é necessário definir uma função de indexação que mapeie o conjunto *I* no conjunto de índices do array.
- A forma mais directa de fazer isto é desenvolver uma função que faça uma correspondência de um para um entre valores destes conjuntos e.g. uma função que transforme um string num inteiro de forma injectiva.
- O problema aparece quando o conjunto *I* tem um número de elementos muito elevado. Além disso sabe-se muitas vezes que apenas uma fracção muito pequena dos elementos de *I* ocorrerão de facto (e.g. strings).

Tabelas de Hash

- O caso anterior corresponde a um tipo de tabela chamada dispersa ou *sparse*.
- Uma tabela de hash consiste em utilisar uma função de indexação que mapeia vários elementos de *I* no mesmo índice do array. Assim, podemos alocar um array com um tamanho aceitável.
- Poderá acontecer que dois elementos de I venham a ser armazenados na mesma posição: este é um problema que tem de ser resolvido.
- No entanto, se
  - a proporção de elementos de *I* que efectivamente ocorre for suficientemente pequena, e
  - se a relação entre a dimensão deste conjunto e o tamanho do array estiver dentro de determinados parâmetros,

  este tipo de situação ocorrerá poucas vezes, e terá um efeito desprezável na performance do sistema.

Funções de Hash

- A função de indexação, neste caso, chama-se **função de hash**. O primeiro problema a resolver quando se quer utilizar uma tabela de hash é a escolha de uma função de hash.
- Quando ocorrem dois valores de *I* que são mapeados na mesma posição do array, diz-se que ocorreu uma colisão.
- Há diversos métodos para tratar colisões. Na implementação de uma tabela de hash, há que escolher um destes métodos.
- Na escolha de uma função de hash devem ser asseguradas duas propriedades:
  - a função deve ser calculável de forma eficiente, e
  - deve dar origem a uma distribuição uniforme dos elementos de I pelo espaço de indexação.

Funções de Hash

- Caso saibamos quais as chaves que vão ocorrer, podemos projectar uma função de hash “óptima”, mas isto raramente acontece.
- Em geral, o que se faz é cortar a chave em “pedaços”, “mistura-los” de várias maneiras e, desta forma, obter um índice.
- A função de hash tem de eliminar padrões que ocorram nas chaves. Há diversas técnicas que podem ser utilizadas:
  - **Truncation** – ignorar parte da chave e utilizar a parte restante para formar o índice.
  - **Folding** – partir a chave em diversas partes e combina-las através de adição, multiplicação ou outra operação algébrica (melhor).
  - **Aritmética modular** – utilizar as técnicas anteriores para gerar um inteiro e calcular o resto da divisão inteira desse valor pelo tamanho do array alocado (muitas vezes um número primo).

Resolução de colisões: Open Addressing

- A forma mais simples de resolver uma colisão é, uma vez que a posição do array onde se pretende armazenar a chave está ocupada, escolher outra posição no array, de acordo com um determinado critério.
- O critério mais imediato para escolher essa nova posição é efectuar uma pesquisa sequencial, ou **linear probing**, a partir do local da colisão.
- No entanto, este método tem o problema de ser instável: quando as colisões são, mesmo que pontualmente, mais frequentes, deixa rápidamente de haver uma distribuição uniforme das chaves pelo espaço de indexação (clustering).
- Isto causa uma deterioração da performance no acesso à tabela: no limite passamos a ter uma pesquisa sequencial.

Resolução de colisões: Open Addressing

- Porque não utilizar uma segunda função de hash? Isto trará poucos benefícios caso a função principal já forneça uma distribuição suficientemente uniforme.
- É necessária uma forma mais sofisticada de se escolher uma nova posição no array quando ocorre uma colisão.
- **Pesquisa quadrática.** A posição escolhida depois da colisão i é $(h + i^2)\%HASHSIZE$. A função i^2 chama-se função de incremento.
- A qualidade deste método depende de HASHSIZE, uma vez que este valor determina quais as posições passíveis de serem sondadas.
- Se HASHSIZE é um número primo, demonstra-se que metade das posições serão sondadas antes de se voltar à posição inicial, e que este é o melhor caso.

Resolução de colisões: Open Addressing

- **Incrementos dependentes da chave.** Em vez de fazer o incremento depender do número de colisões, faz-se depender da chave que causou a colisão.
- Por exemplo, quando se utiliza $\%HASHSIZE$ para calcular a função de hash, o quociente desta divisão é uma boa escolha para o incremento (e dá também uma implementação eficiente).
- **Pesquisa aleatória.** A posição escolhida depois da colisão i é gerada através de um gerador de números pseudo-aleatórios.
- Este gerador utiliza a chave como semente e deverá gerar a mesma sequência de valores para uma mesma semente. Este método é muito bom, mas pouco eficiente.

Resolução de colisões: Open Addressing

- Implementação:

```
#define EMPTY NULL
typedef char *Key;
struct item {
   Key key;
   /* Aqui entrariam outros campos da tabela */
};
typedef struct item Entry;
typedef Entry HashTable[HASHSIZE];
```

- Nas remoções não podemos permitir que a cadeia de pesquisa seja quebrada: uma solução consiste em definir um valor especial (diferente de EMPTY) que assinale uma entrada removida.

Resolução de colisões: Chaining

- A utilização de espaço contíguo (um array) como base para uma tabela de hash é a opção mais natural. No entanto, se nos restringirmos a este tipo de suporte, temos duas limitações:
  - somos obrigados a utilizar a solução anterior para resolver as colisões.
  - o número de chaves inseridas está limitado ao tamanho do array.
- Uma solução mais elegante, e também mais flexível, para tratar o problema das colisões consiste em utilizar as posições do array para armazenar apontadores para as chaves armazenadas.
- Em que é que esta solução facilita a resolução de colisões?
  - Podemos “pendurar” numa dada entrada do array mais do que uma chave.
  - Guardamos no array o apontador para uma lista (ligada) dos elementos com o mesmo hash.

Resolução de colisões: Chaining

- Vantagens:
  - se as entradas da tabela forem registos muito grandes, a memória reservada para o array vem muito reduzida.
  - caso ocorra um número elevado de colisões, isso não afectará o resto da tabela: apenas tornará mais lento o acesso à lista associada à entrada em conflito.
  - o número de chaves armazenadas passa a poder ser superior à dimensão do array.
- Desvantagens:
  - memória adicional para os apontadores nas listas ligadas.
  - acréscimo de complexidade inerente á gestão das próprias listas.

Resolução de colisões: Chaining

- Implementação:

```
typedef char *Key;
struct item {
   Key key;
   /* Aqui entrariam outros campos da tabela */
   struct item *next;
};
typedef struct item Entry;
typedef Entry *HashTable[HASHSIZE];
```

- As remoções tornam-se muito mais simples: é possível efectua-las directamente sobre as listas ligadas.

Resolução de colisões: Comparação dos Métodos

- Define-se o factor de carga de uma tabela de hash como $\lambda = n/t$, sendo n o número de elementos na tabela, e t o tamanho do array.
  - Utilizando open addressing, o factor de carga nunca pode ser superior a 1.
  - Utilizando chaining, o factor de carga não tem (teóricamente) limite superior.
- A performance dos acessos a uma tabela de hash depende **exclusivamente do seu factor de carga**:
  - **não depende do número de elementos armazenados**.

Resolução de colisões: Comparação dos Métodos

- Demonstra-se que a recuperação de uma chave de uma tabela de hash que use chaining implica, aproximadamente:
  - $1 + 0.5\lambda$ comparações no caso de sucesso e,
  - λ comparações no caso de insucesso.
- No caso de se utilizar open addressing, demonstra-se que:
  - Utilizando pesquisa aleatória, o número aproximado de comparações será $(1 - \lambda)^{-1}$ no caso de uma pesquisa sem sucesso, e $\lambda^{-1} ln((1 - \lambda)^{-1})$ no caso de uma pesquisa com sucesso.
  - Utilizando pesquisa sequencial, o número aproximado de comparações será $0.5(1 + (1 - \lambda)^{-2})$ no caso de uma pesquisa sem sucesso, e $0.5(1 + (1 - \lambda)^{-1})$ no caso de uma pesquisa com sucesso.

Resolução de colisões: Comparação dos Métodos

- Número esperado de comparações para pesquisas bem sucedidas

Factor de Carga	0.10	0.50	0.80	0.90	0.99	2.00
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open Addressing (aleatório)	1.05	1.4	2.0	2.6	4.6	–
Open Addressing (linear)	1.06	1.5	3.0	5.5	50.5	–

- Número esperado de comparações para pesquisas sem sucesso

Factor de Carga	0.10	0.50	0.80	0.90	0.99	2.00
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open Addressing (aleatório)	1.1	2.0	5.0	10.0	100	–
Open Addressing (linear)	1.12	2.5	13	50	5000	–

Comparação dos Métodos: Conclusões

- Aceder a uma tabela de hash com 40000 entradas, implementada sobre um array com 20000 posições, é equivalente a aceder a uma tabela com 40 entradas, implementada sobre um array com 20 posições.
- O chaining é uma solução consistentemente mais eficiente no que diz respeito ao número de comparações, mas . . .
  - Como a busca numa lista ligada é mais lenta que a pesquisa num array, um menor número de comparações pode não implicar uma pesquisa mais rápida.
  - Em situações em que os registos da tabela são de tamanho relativamente grande, o tempo necessário para efectuar uma comparação torna-se mais elevado e, nestes casos, a performance do chaining é efectivamente superior.

Comparação dos Métodos: Conclusões

- O chaining é também muito melhor nas pesquisas sem sucesso:
  - para entradas cujos valores de hash nunca ocorreram, apenas é necessário fazer uma comparação com um apontador nulo.
  - no open addressing o número de comparações necessárias neste caso depende do factor de carga do array.
- Entre as duas opções consideradas para o open addressing, pode concluir-se que
  - apesar de a solução aleatória ser melhor, só existem diferenças significativas no caso de pesquisas falhadas.
  - Assim, para aplicações em que se prevê um número reduzido de pesquisas sem sucesso, a simplicidade da solução sequencial torna-a, muitas vezes, preferível.