Comparando operações comuns em função do tempo em estruturas de dados avançadas

Carlos Mattoso, Gabriel Barros e Leonardo Kaplan

PUC-Rio

Resumo

Nossa motivação ao desenvolver este projeto foi comparar as operações mais comuns (inserção, remoção e busca) em diferentes estruturas de dados avançadas a fim de determinar a mais eficiente em termos de tempo de execução. As estruturas escolhidas foram tabelas Hash, Arvore Binária simples e Arvores Binárias balanceadas.

I. Introdução

Para determinar a estrutura de dados mais eficiente em termos de velocidade de execução escolhemos representantes dos diversos paradigmas: Tabelas hash, com diferentes tratamentos de colisão; Arvores de busca binaria simples e Arvores de busca balanceadas. Para comparar as diferentes estruturas comparamos os tempos de inserção, remoção e busca nas estruturas em 2 tipos diferentes de entradas: - Uma com várias operações de inserção (possivelmente de valores já inseridos) - Outra com as mesmas inserções, seguidas de operações de busca, remoção e até inserção. Era esperado que a Hash de passo unitário fosse a mais rápida em todas as entradas, destacando-se principalmente em busca.

II. Hash Tables

Variamos as tabelas Hash em função do tratamento de colisão, aplicando:

I. Passo fixo unitário

Aqui, ao inserirmos verificamos se a posição dada pela função Hash está livre e, caso esteja, inserimos o elemento; em caso de colisão, verificamos a posição seguinte e assim em diante até encontrar algum bucket vazio. Na inserção garantimos que não ocorrerá inserção de duplicatas. No caso da busca, segue-se o mesmo processo, comparando-se o elemento candidato ao que estamos buscando e se forem iguais, a busca foi bem sucedida. Caso atinja-se uma posição vazia indica-se que o elemento não encontra-se na árvore. Finalmente, para a remoção é necessária uma operação mais custosa. Caso o elemento a ser removido encontre-se na tabela, é necessário reinserir, caso exista, a sequência contínua de elementos que vem imediatamente após o removido.

II. Hashing duplo, definindo o tamanho do passo com outro hash

Os procedimentos aqui são bem parecidos ao tipo anterior. No caso da inserção a diferença é que percorre-se a tabela com um passo definido por uma outra hash function. Também, por utilizarmos remoção "lazy"neste caso (afinal,

não seria fácil achar quem pode ter colidido com o elemento a ser removido), a inserção é quem esvazia posições marcadas como removidas substituindo-nas pelo novo elemento. A inserção continua garantindo que elementos que tenham uma cópia na tabela não serão inseridos. A busca segue os mesmos passos que no tipo anterior, com a diferença do passo. Finalmente, a remoção como já dito é feita de modo "lazy", só ocorrendo de fato na inserção.

III. Encadeamento externo através de lista encadeada

Neste método, cada bucket da tabela é uma lista encadeada. Assim, caso ocorra alguma colisão basta inserir na lista. A busca neste caso deve ser feita passando-se por cada elemento da lista encadeada. Finalmente, a remoção de fato ocorre caso o elemento encontre-se em sua respectiva lista.

Em todos os casos a tabela Hash foi representada como um grande vetor de ponteiros para strings que representavam os conjuntos lidos. Embora esta representação tenha exigido comparação de strings, uma operação não muito eficiente, isto não afeta o resultado dos testes pois teve de ser feito para todas as estruturas.

Adotamos duas estratégias para minimizar o número de colisões:

IV. Uso de boas funções de hash para string:

fnv1a e djb2, testadas por muitos pesquisadores e em aplicações do mundo real. Ambas garantem uma distribuição de qualidade satisfatória, minimizando significativamente o número de colisões.

V. Criação de tabelas de tamanho mais que o suficiente:

Como sabemos desde o início a quantidade de elementos máxima que leríamos, podemos criar uma tabela de tamanho maior até mesmo que o necessário, reduzindo as chances de ocorrência de colisões. Em razão disto não tivemos que implementar algoritmos de ajuste de tamanho da tabela, no caso de ela não conseguir comportar mais elementos ou ter atingido um fator de carga muito alto. De fato, numa aplicação de mundo real isto pode nem sempre ser possível.

III. Árvores

No caso de árvores testamos dois tipos:

I. Árvore Binária de Busca Não Balanceada

Foi implementada através de ponteiros encadeados, pois a representação com array ficaria inviável em razão de seu despercídio de memória (afinal, a árvore pode ficar drasticamente degenerada). Os algoritmos de inserção, remoção e busca foram feitos de modo iterativo, a fim de se evitar a possibilidade de "stack overflow", devido ao grande número de elementos que seria inserido; como a árvore nunca é balanceada, poderíamos atingir níveis muito baixos.

II. AVL (ABB Balanceada)

A implementação de AVL por nós utilizada foi extraída de uma biblioteca popular chamada "libavl". Mais informações sobre ela encontram-se em: http://adtinfo.org/ Utilizamos a versão simples, ensinada em sala de aula. O site apresenta também uma documentação detalhada que enuncia quais fragmentos de código executam uma determinada operação (como rotações). No caso de rebalanceamento, por exemplo, tais informações encontram-se aqui:

http://adtinfo.org/libavl.html/Rebalancing-AVL-Trees.html

III. Árvore B

Não conseguimos encontrar uma implementação que fosse bem documentada e permitisse customizar os parâmetros da árvore.

IV. Results

Tabela 1: *Example table*

Name		
First name	Last Name	Grade
John	Doe	7.5
Richard	Miles	2

$$e = mc^2 \tag{1}$$

V. Discussion

I. Subsection One

II. Subsection Two

Referências

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.