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, Árvore Binária simples e Árvores 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:

- 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.
- 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

Utilizamos a kbtree, contida no pacote klib. O repositório está em: https://github.com/attractivechaos/klib

O autor fez uma breve comparação entre a B-Tree e a BST em seu blog: http://attractivechaos.wordpress.com/2008/09/24/b-tree-vs-binary-search-tree/

IV. Análise dos resultados

Os testes foram executados em uma máquina com Intel i7 com clock travado em 0.77 GHz com 4 Gb de RAM.

A análise foi feita sobre os resultados em anexo.

Os resultados da menor entrada (inst32k5) foram desconsiderados pois apresentaram erro de medição significativo.

Analisamos separadamente as Tabelas Hash e as Árvores para depois as compararmos.

I. Hash Tables

No quesito inserção a estrutura mais rápida foi o Hash simples, como esperado, pois só necessita de um passo para os casos de colisão. Isso possibilita um bom uso do cache do processador. Além disso, o uso de uma boa função de dispersão (FNV1A) possibilitou uma boa distribuição dos valores na tabela, aumentando ainda mais a eficiência do passo simples. O hash duplo seguiu como o segundo melhor, já que para colisões ele roda um operação de hash a mais. Por fim, o hash de listas encadeadas foi o pior, pois necessita de operações de alocação de memória para resolução de colisão, além disso, vale mencionar que as funções do módulo hash de listas encadeadas utiliza um módulo externo de listas encadeadas, o que influencia negativamente no seu desempenho.

No quesito busca o hash simples novamente venceu, seguido pelo de lista encadeada e depois o de hash duplo. Provavelmente, isto se deu em razão da operação ligeiramente mais custosa de se calcular um segundo hash.

Finalmente, no quesito remoção todas ficaram muito próximas, com uma pequena vantagem do hash de encadeamento externo, devido ao fato de neste caso não termos de nos preocupar com os outros elementos da lista. A demora no caso de hash duplo provavelmente se deu em razão do segundo hash.

No geral, todos ficaram muito próximos, mas o hash de passo unitário acaba se saindo melhor.

II. Árvores

Primeiramente, é importante ressaltar que as árvores apresentaram desempenho inferior aos Hashes. Isto se deu em razão de custos a mais com alocação de memória e percurso das árvores. Nos hashes, com uma boa distribuição, as operações tornam-se todas praticamente O(1).

A árvore com os melhores tempos médios de inserção é a AVL, seguida da B e em ultimo ficou a BST, como esperado, afinal esta não apresenta balanceamento algum, podendo ficar com uma altura extremamente elevada. É interessante notar que a BST teve um tempo total de inserção melhor que o da B, mas a última apesar de perder na entrada menor, ganha à medida que a entrada cresce, ficando com a média mais baixa.

Isso é esperado visto que na teoria a B se destaca positivamente com o crescimento da entrada, em algum ponto é possível que ela passe a AVL.

Os resultados das buscas foram semelhantes aos da inserção, contudo a árvore B apresenta um desempenho mais constante, por armazenar os dados da melhor maneira possível. No caso da BST, em razão do não balanceamento, seu desempenho depende fortemente da entrada. Por fim, a AVL apresenta um desempenho superior ao da B, contudo não a mesma constância.

Na remoção, a BST tem o melhor tempo total, a AVL tem o melhor tempo médio e a árvore B perde em todos os aspectos.

III. Conclusão

Assim, podemos concluir que o uso de uma Hash Table é provavelmente a melhor opção caso deseje-se apenas fazer operações de busca e inserção. No caso da remoção, dependerá muito da qualidade da função de dispersão, pois a remoção em hash table é muito custosa. Dentre as três testadas, o tratamento de colisão através de passo unitário é o mais recomendado.

As árvores balanceadas tiveram um desempenho bem similar, mas não apresentam vantagens para a realização de tais operações. Assim, seu uso não é recomendado para o contexto apresentado.