IF Sudeste de Minas Gerais – Campus de Barbacena
Curso Superior de Tecnologia em Sistemas para Internet
Estrutura de Dados II

Documentação TP01

Vítor Caio De Paula
17/04/2017

Introdução

O sistema tem como objetivo a contagem de votos de uma eleição. Para essa tarefa o programa foi formulado usando três algoritmos diferentes: tabela hash com resolução de colisões por hash duplo, tabela hash com resolução por árvore binária, e por árvore avl.

Também foi utilizado duas tabelas hash com resolução de colisões por hash aberto, uma tabela para apuração para o presidente e outra para o senador. Essas duas tabelas são atualizadas em tempo de execução do sistema, conforme há entradas de dados.

Todos os dados tem sua entrada feita por meio de arquivos de teste, e foram criadas duas funções para essa tarefa. Foi feito também um menu para escolha de qual estrutura será utilizada para a contagem dos votos.

Também no seguinte trabalho foram feitos experimentos com quatros arquivos de teste com diferentes tamanhos, ou seja, com diferentes quantidades de entradas de dados.

Implementação

A inicialização do sistema é feita com a entrada de dados provindos de arquivos de teste - no caso foi passado os arquivos entrada.txt, entrada1.txt, entrada2.txt, entrada3.txt. Primeiro o sistema lê as duas primeiras linhas, a primeira se refere ao modo: 0 para modo simulação ou 1 para modo campeonato. A segunda linha se refere a estrutura que será utilizada para a contagem dos votos, 1 para hash duplo, 2 para hash com árvore binária e 3 para árvore avl.

A terceira linha se refere a quantidade de senadores, presidentes e eleitores que terá o pleito. Para essa entrada de dados foram criadas duas funções de formatação:

- int validaTamanhoTabelas(char *str, int *presidentes, int *senadores, int *eleitores);
- int leValidaString(char *str, char *opcao, int *cla, char *t, int *n, int *ran);

A primeira função fica encarregada de extrair do arquivo exatamente a quantidade de senadores, presidentes e eleitores, se tudo ocorreu bem na extração será retornado 1 ou 0 caso contrário. Se caso retorna 0 o programa será abortado. A segunda função fica encarregada de gerenciar o sistema(parte principal do sistema), é nela que será extraída a opção(0 - inserir, 1 - remover, 2 - apurar, 3 - finalizar), a classe do candidato(0 - presidente, 1 - senador), o título de eleitor, número do candidato, o rank da apuração. Assim como a primeira função também tem como valor de retorno um inteiro: 1 em caso de sucesso e 0 caso contrário.

Após feita a entrada dos tamanhos da tabela, então é criadas as tabelas hash, para evitar erros define que a tabela hash duplo terá sempre seu tamanho o dobro que foi passado no arquivo, isso evita que não existam mais eleitores que posições na tabela. Impossibilitando assim que eleitores não fique sem votar(isso aconteceu algumas vezes no desenvolvimento do sistema). Para a tabela hash com árvore binária não teremos esse problema, pois como está sendo utilizada uma árvore a mesma pode crescer e acomodar os votos excedentes. Para árvore avl não foi pedido uma tabela hash, ou seja, tudo é acomodado em uma mesma estrutura.

Para as tabelas de hash aberto também foi definido que seu tamanho será a quantidade de eleitores mais a quantidade da sua função, ou seja, quantidade de eleitores + quantidade de presidentes ou quantidade de eleitores + quantidade de senadores. Essas duas tabelas hash aberto são utilizadas para apuração dos votos, e ela é atualizada em tempo de execução, ou seja, se entra um voto de algum eleitor em alguma estrutura seja ela, na tabela hash duplo, hash por árvore binária ou árvore avl, o número do candidato em que o eleitor votou vai ser inserido na tabela hash

aberta referente: ou na tabela de presidente ou na de senador, essas tabelas apresentam a seguinte estrutura:

```
typedef struct {

    int numero; // Número do candidato.
    int votos; // Quantidade de votos.

} tipoCampos;
```

Cadas posição da tabela fará uma referência para um nó desse tipo, com o número do candidato e o número de votos que ele recebeu. A escolha dessa inserção em tempo real ao meu ver deixa apuração mais rápida, mas ao mesmo tempo gasta mais recursos o sistema visto que sempre teremos três estruturas alocadas em memória em tempo real - a estrutura contendos os eleitores e duas estruturas para a contagem de votos do presidente e senadores. Em contrapartida poderia ser feita a criação dessas tabelas somente após feita a escolha, mas pensei que isso geraria um consumo muito maior do sistema, pois seria feita a criação das tabelas, inserção dos dados na mesma e logo depois sua destruição, isso sendo feita uma hora atrás da outra, então preferi manter três tabelas criadas em tempo de execução tempo todo e só no final destruí-las. Se houver uma entrada do tipo: 0 0 aabb33 50, o sistema com a função leValidaString irá formatar e inserir em variáveis passadas por referência cada parte da entrada, pegará por exemplo a classe do candidato, no caso 0, pegará o título e o número do candidato e irá inserir na estrutura escolhida, depois será pego o número do candidato e inserir na tabela hash aberto referente a sua classe e os votos do número será incrementado. Para a remoção será feita o mesmo: 1 aabb33, a função irá formatar e o candidato com o título será apagado da estrutura escolhida, e os votos cujo esse eleitor votou será decrementado nas tabelas hash abertas referentes. Para a apuração foi criada uma função especial que recebe as tabelas hash aberta e a classe do candidato e retornará a quantidade de votos dos mais votados de acordo com o tamanho do rank solicitado.

Sobre a implementação da tabela hash duplo, a mesma tem a seguinte estrutura:

```
typedef struct{

    cedula *itens;
    int M, tamanho;

}hashDuplo;
```

é simplesmente um array alocado dinamicamente do tipo cedula que faz referência ao tipo título de eleitor, número do presidente e ao senador, sobre o hash duplo e suas funções vale salientar o cálculo do índice do hash, caso o índice resultante seja inválido, ou seja, já se encontra ocupado deve-se então fazer um novo cálculo do índice com uma outra função hash, mas essa função tem que ter o cuidado de não retornar zero pois senão poderá haver um loop infinito na procura de um novo índice válido para a nova entrada. O cuidado pode ser visto na seguinte instrução:

```
i += k % t->M;
```

Caso o valor do índice da segunda função hash(k) dê 0, a variável i sempre pode ter o mesmo valor, caindo assim em um loop infinito e causando problema na execução do sistema.

Sobre a segunda estrutura a tabela hash com árvore binária não teremos os mesmo problemas encontrados no hash duplo, pois caso a função hash calcule um índice já preenchido o sistema de tratamento de colisão por árvore irá acomodar o novo item na estrutura sem problemas, pois a árvore é elástica, ou seja, ela permite seu crescimento indefinido só dependendo dos recursos da máquina. Nenhum mistério nessa estrutura propriamente dito.

A terceira estrutura que no caso é a árvore avl foi a de menor problema de implementação visto que não existe tabela hash para a mesma, todos os itens serão armazenados em uma mesma árvore, ou seja, uma única instância da avl será criada, o inverso da tabela hash com árvore binária que podemos ter várias instâncias da mesma. Mas de cara sem experimentos posso afirmar que essa estrutura é a mais lenta, pois cada inserção de dados é necessário um balanceamento da estrutura. Para um sistema de muita inserção ela não vale a pena seu uso.

Todas as funções da árvore binária de pesquisa e avl tem sua similaridades, a diferença que na árvore binária de pesquisa não existe o fator de balanceamento o que de certo modo deixa o sistema muito mais lento, tanto na inserção dos dados tanto na remoção dos mesmo. No caso da árvore binária de pesquisa os dados são apenas inseridos na estrutura e pronto.

E por fim temos a finalização do programa todos as estruturas serão destruídas de acordo com a estrutura escolhida, oriundas dos arquivos de teste passados.

Experimentos

Você deverá rodar cada um dos 3 algoritmos para as bases de testes que serão repassadas para vocês. Apresenta o tempo de execução em gráficos ou tabelas. Também compute o número de colisões de cada algoritmo para os arquivos de teste. Responda:
a) Qual dos algoritmos é mais rápido para cada base de teste?
b) Quais foram as funções de hash que você usou? Por que as escolheu?

entrada.txt

Estrutura	**Tempo**
Hash Duplo	2.808 s
Hash árvore binária	1.004 s
Árvore avl	0.899 s
*Hash lista	0.910 s

No primeiro experimento pode se notar que a estrutura árvore avl se saiu muito melhor dos que outros, isso se deve que como ela é uma estrutura única e por tanto de cara só faz uma alocação, ela tende por isso a ser muito mais rápida que as outras que precisam de mais de uma, na segunda posição ficou a estrutura hash com árvore binária e por fim hash duplo. Só ressaltando que Hash com lista teve o segundo maior tempo, mas ela só é utilizada no modo campeonato. A quantidade de dados do arquivo entrada.txt é bem menor, cerca de **33 entradas**, o que ajuda muito no caso.

entrada1.txt

Estrutura	**Tempo**
Hash Duplo	5.514 s
Hash árvore binária	6.095 s
Árvore avl	4.775 s
*Hash Lista	4.758 s

Assim como no primeiro experimento, a árvore avl é muito mais rápida do que as outras, o hash duplo fica na segunda posição e por último fica o hash com árvore binária. Só que neste experimento o arquivo entrada1.txt possui **5004 entradas**.

entrada2.txt

Estrutura	**Tempo**
Hash Duplo	17.522 s
Hash árvore binária	10.217 s
Árvore avl	16.092 s
*Hash Lista	12.818 s

No terceiro experimento, utilizando o arquivo entrada2.txt com possui **25004 entradas**, a ordem muda, como previsto a árvore avl se torna a mais lenta, o hash árvore binária se torna o mais rápido e o hash duplo o mais lento.

entrada3.txt

Estrutura	**Tempo**
Hash Duplo	61.351 s
Hash árvore binária	28.633 s
Árvore avl	50.228 s
*Hash Lista	23.377 s

No quarto e último exemplo, no arquivo entrada3.txt existem **5004 entradas,** e o melhor algoritmo é o hash com árvore binária - se incluir o campeonato o hash com lista ganha -, seguido pelo avl e por último o hash duplo.

Conclusão

Como vimos nos experimentos, quanto menos entradas tiver o melhor algoritmo será a árvore avl, pois no caso ela precisa apenas fazer alocação das suas próprias estruturas e nada mais. Conforme as entradas vão aumentando ela a avl vai perdendo sua vantagem e por isso o hash duplo começa a ficar melhor, pois no caso ele no caso já aloca todas a posições necessárias, seu único trabalho é encontrar a melhor posição, então para arquivos com muitas entradas o melhor é o hash duplo é o melhor, mas esse bom desempenho tem um custo de recurso muito alto que no caso é alocar já todas as posições. Em minha conclusão o melhor algoritmo é o hash com árvore binária de pesquisa, pois nos teste ele ficou em segundo em todos, mostrando assim que tem uma boa média, tanto para arquivos com poucas entradas quanto para arquivos com muitas entradas. Fiz essa escolha devido que em comparação com o hash duplo o hash por árvore binária de pesquisa aloca posições conforme vai entrando dados.

Sobre as dificuldades na implementação do trabalho, tive muita na implementação da estrutura da árvore avl e nas indecisões sobre os tamanhos da tabela, que não soube fazer direito, e também na forma em como seria feita a apuração dos votos dos candidatos.

REFERÊNCIAS

- http://www.ic.unicamp.br/~zanoni/mo637/aulas/hash.pdf
- https://pt.wikipedia.org/wiki/Fun%C3%A7%C3%A3o_hash
- https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-hash.html