Trabalho Prático 3
Dicionário

Estrutura de Dados

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Lorenzo Carneiro Magalhães
2021031505

lorenzocarneirobr@gmail.com

Belo Horizonte - MG - Brasil

1. INTRODUÇÃO

O trabalho prático objetivou desenvolver um dicionário de duas maneiras diferentes: a primeira utilizando a estrutura de hashtable e a segunda utilizando a estrutura de árvore AVL, ou seja, uma árvore que se auto balanceia. É esperado que utilizando esses modelos, o dicionário seja relativamente eficiente.

2. MÉTODO

Os algoritmos foram implementados em um computador de 16gb RAM, processador i3-9100f (64 bits CPU) e sistema operacional Windows utilizando o WSL2 contendo o ubuntu 22.04.

A implementação dos algoritmos foi realizada em C++ e compilada com o compilador do padrão do Linux, o G++.

Para a implementação da estrutura em HashTable, foi feita um hash que codifica as primeiras duas letras de cada palavra, de modo que a codificação seja relativa à tabela ASCII, indo do caracter hífen até o caractere “z”.

Desse modo, contou-se com 78 possibilidades para cada letra. Fazendo uma codificação simples, o índice na tabela era calculado pela subtração entre o caracter encontrado na primeira letra e o número do caractere hífen, multiplicado por 78, isso somado com a codificação da segunda letra, que é resultante da subtração da segunda letra pelo número do caractere hífen. Logo, a tabela possui 78*78 espaços.

Cada espaço da tabela contém uma lista encadeada ordenada contendo as palavras resultantes de um mesmo índice, de modo que não haja colisão.

Os significados das palavras, em ambas implementações, foi uma lista encadeada contendo strings.

Já a árvore AVL foi implementada de maneira que cada nó da árvore é uma palavra contendo uma lista de significados.

3. Análise de Complexidade

Primeiramente, tem-se a Hashtable. Essa estrutura é eficiente pois consegue acessar cada elemento da tabela em O(1). Entretanto, esse caso ocorre somente quando não há colisão e quando os elementos são únicos no espaço da tabela.

Portanto, verifica-se que a complexidade do acesso a cada elemento não é O(1), e sim o custo para encontrar o elemento na lista encadeada criada em cada espaço da tabela. Logo, o custo no pior caso é O(n).

Porém,o tempo que a operação leva não é tão grande em casos rotineiros, isso porque, apesar de existirem letras iniciais mais frequentes, as possibilidades são grandes, de modo que se houver uma distribuição uniforme, espera-se 2 palavras para cada espaço, tendo um espaço amostral selecionado contendo 12000 palavras

De fato, se houver muitas palavras, a codificação fica ultrapassada e seria necessário uma melhor, já que a quantidade de elementos em cada lista aumentaria.

Percebe-se então, que há uma relação de custo temporal vs custo de memória. Nesse caso, são inversamente proporcionais: à medida que o uso de memória aumenta, é possível diminuir ainda mais o tempo de cada consulta.

No caso do programa implementado, já que a lista é ordenada, há um custo maior para inserir, que é O(n), mas para imprimir os resultados na tela o custo é de apenas O(n), em que *n* é relativo ao números de elementos na lista.

Para a árvore AVL, temos que o custo de suas operações é praticamente o mesmo sempre: log n. A inserção, remoção e procura são log n. Isso ocorre pois a distância máxima percorrida é a altura da árvore, que como sempre estará balanceada, será log n.

Para a impressão da árvore em ordem alfabética, foi utilizado a estratégia de imprimir em ordem cada nodo, de modo a percorrer todos os nodos em um custo de O(n).

O principal problema dessa árvore é o custo das recursões de cada operação, além da sua dificuldade de implementar.

Por fim, o dicionário é uma estrutura acessível ao usuário que conta com funções fáceis de serem executadas. A implementação dos dicionários específicos foi feita através de uma herança da classe original virtual pura dicionário. Fazendo desse modo, é possível declarar um ponteiro do tipo dicionário que executa as funções específicas.

4. Estratégias de robustez

O programa conta com algumas estratégias de robustez, como checagem de parâmetros e checagens do número máximo de palavras armazenadas.

A implementação de cada estrutura também garante certa segurança, mas não previne todos os tipos de erros que podem ocorrer.

5. Análise Experimental

5.1. Análise temporal

Como discorrido acima, a estrutura de dados Hashtable aparenta ser muito rápida para casos não muito grandes, que é o caso dos exemplos utilizados, mas peca na velocidade quando há um volume muito grande de palavras.

Por outro lado temos a árvore AVL, que mantém o tempo de execução relativamente mais controlado com o aumento do tamanho da entrada.

Para a medição do tempo de execução, foi utilizado a biblioteca *memlog*.

HASH	Size	Run1	Run2	Run3	Run4	Run5	**TOTAL**
ex01	522	0.1323976	0.1427813	0.1581464	0.1832278	0.1161888	0.7327419
ex02	2589	0.7639476	0.9500633	0.8349825	0.8322388	0.7078146	4.0890468
ex03	100	0.0387814	0.0451604	0.0745217	0.0664837	0.0522231	0.2771703
ex04	101	0.0676988	0.0698238	0.1917888	0.1066274	0.0989011	0.5348399
ex05	9000	2.3174805	1.9708103	2.261078	1.9831176	1.7915435	10.3240299
							15.9578288

TREE	Size	Run1	Run2	Run3	Run4	Run5	**TOTAL**
ex01	522	0.180495	0.1500957	0.159648	0.1262753	0.1429215	0.7594355
ex02	2589	0.6950173	0.840988	0.7896512	0.9202268	0.6689485	3.9148318
ex03	100	0.039418	0.0465956	0.0380859	0.0571447	0.0602317	0.2414759
ex04	101	0.0666241	0.0610721	0.0866165	0.0916873	0.0759178	0.3819178
ex05	9000	3.9884441	1.8664887	2.1695921	1.9798727	1.7959015	11.8002991
							17.0979601

A tabela acima foi gerada com os tempos de execução gerados pelo *memlog* e cada exemplo de dicionário foi executado 5 vezes. A coluna total corresponde ao valor acumulado de cada execução do algoritmo com o exemplo da linha. A célula destacada em alaranjado corresponde à soma dos valores acumulados de cada exemplo.

A partir da tabela, percebe-se que os resultados são bem acirrados, não havendo um algoritmo perceptivelmente melhor do que outro. Uma grande diferença ocorreu em apenas um teste, que não reflete a eficiência real dos algoritmos.

Um ponto a ser ressaltado é que a estrutura Hashtable pode ser melhorada, de modo a não acumular valores na lista encadeada, o que pouparia muito tempo. Além disso, também é possível alterar a lista ordenada de palavras para uma árvore AVL, o que traria resultados melhores.

Assim, chega-se a uma conclusão que a Hashtable pode ser muito mais eficiente do que a árvore AVL caso algumas condições sejam possíveis.

A árvore AVL, por outro lado, não tem tanto espaço para melhoria assim, além da otimização de algumas funções implementadas. Entretanto, apresentou desempenho muito satisfatório.

5.2. Análise de memória

Primeiramente, é necessário afirmar a impossibilidade da utilização da biblioteca *analisamem* nessa análise: essa biblioteca não funciona adequadamente quando há alocação dinâmica no código

Portanto, a análise será feita tendo em vista a implementação do código e o conceito de cada estrutura.

5.2.1 Hashtable

Como mencionado anteriormente, a hashtable pode ser muito rápida em detrimento de um maior uso de memória, agora basta analisar até onde compensa ceder memória para impulsionar a velocidade do programa.

Essa dúvida depende obviamente das condições da máquina e dos objetivos a serem alcançados. Por exemplo, é inviável utilizar tal estrutura quando os recursos de memória são escassos.

No caso do programa implementado, há memória suficiente para deixar o programa muito rápido, mas não houve necessidade, já que não era o objetivo do trabalho.

A estruturação da tabela foi feita pensando em um ponto de equilíbrio, assim, foi criado uma tabela contendo 78*78 espaços, ou seja 6084 espaços.

Cada espaço contém uma lista ordenada encadeada de palavras e cada lista de palavras contém nodos que contém uma lista encadeada de significados.

Desse modo, caso todos espaços sejam preenchidos por pelo menos uma palavra e cada palavra tenha ao menos um significado, tem-se que a hashtable ocupa aproximadamente 529308 bytes, isto é 516,9 kilobytes ou 0,5 mb.

De fato, não é tanta coisa em termos absolutos, mas lembre-se que nesse exemplo a tabela foi populada com poucos dados.

Logo, conclui-se que de fato a hashtable usa muita memória.

Quanto à localidade de referência, a hashtable se comporta relativamente bem, já que no processo mais demorado, que é o de impressão, os espaços sequenciais são acessados em sequência, o que é muito bom para o programa, já que “aquece” a memória.

Entretanto, todas as operações também envolvem a lista encadeada, que não possui acessos muito uniformes, devido à inconstância da alocação dinâmica, o que prejudica a localidade de referência do algoritmo como um todo.

5.2.2. Árvore AVL

A árvore AVL, diferentemente da hashtable, ocupa apenas a quantidade de memória necessária para a manutenção dos dados, logo, ocupa memória em O(n).

No entanto, esse tipo de estrutura possui um problema: como a alocação dos nodos da árvore são feitas dinamicamente, a localidade de referência da estrutura é ruim.

Como citado anteriormente, a alocação dinâmica dos componentes faz com que os nodos estejam dispersos no “heap”, de modo a não estarem necessariamente perto uns dos outros.

No entanto, nem tudo é tão ruim. Como a árvore é uma estrutura que se ramifica, os nodos do topo da árvore são acessados constantemente, o que melhora a localidade de referência dessa estrutura.

Considerando isso, percebe-se que a estrutura de árvore AVL é mais útil em muitas ocasiões, já que não precisa de espaço adicional e praticamente todo o seu espaço utilizado pode ser alocado no heap.

6. Conclusões

O trabalho foi executado com sucesso. Infelizmente houve a limitação da biblioteca *analisamem*, que impossibilitou a geração dos gráficos relacionados à memória. Isso ocorreu pois o memlog não consegue lidar com um range grande de memória proporcionado pela alocação dinâmica, que foi inevitável nesse programa.

Por fim, verificou-se que as duas estruturas são muito eficientes e são boas para o uso atribuído. A diferença marcante das estruturas está no consumo de memória, de modo que a árvore AVL consiga lidar melhor com a memória enquanto a hashtable pode ceder otimização de memória para melhorar o desempenho temporal.

7. Bibliografia

GNU make. GNU Project. Disponível em <https://www.gnu.org/software/make/manual/make.html>

Deletion in AVL Tree. GeekForGeeks. Disponível em: <https://www.geeksforgeeks.org/deletion-in-an-avl-tree/>

Slides da disciplina. UFMG-DCC.

8. Instruções para compilação e execução

A lista de compilação dos comandos para o usuário do makefile segue abaixo:

- *make bin - compila o código, gerando os objetos na pasta obj*
- *make out - cria a pasta out*
- *make exectest - executa os casos teste, que são testados e verificados com o input correto que foi fornecido no fórum através do comando diff*
- *make clean - limpa o espaço de trabalho*
- *make all - limpa espaço de trabalho, compila, executa, cria pasta out e realiza testes*

Os argumentos do executável estão conforme descrito na documentação do enunciado do trabalho.

Seguem as flags disponíveis para o executável:

- *i <input file>*
- *o <output file>*
- *t <data structure (tree or hash)>*
- *h help*

<> expects a value

A partir desse menu fica claro como utilizar o programa. Caso haja dúvidas, digite *make all* para a execução de todos os comandos importantes. Também é possível olhar os padrões nos códigos exemplo no *makefile*.