Universidade Federal do Ceará -
Campus Quixadá
Ciência da computação
Estrutura de Dados Avançada

Projeto Contador de Frequência de palavras com C++

Aluno: Lucas de Araújo Torres
Matrícula: 557156
Professor orientador: Atílio Gomes Luiz

mês
ano

Universidade Federal do Ceará -
Campus Quixadá
Ciência da computação
Estrutura de Dados Avançada

Relatório de Projeto

Projeto principal da cadeira de Estrutura de Dados Avançada, na qual foi realizado um contador de frequência de palavras utilizando um dicionário. A data de conclusão foi no dia 19/09/2024.

Aluno: Lucas de Araújo Torres

Matrícula: 557156

Professor orientador: Atílio Gomes Luiz

Setembro
2024

Conteúdo

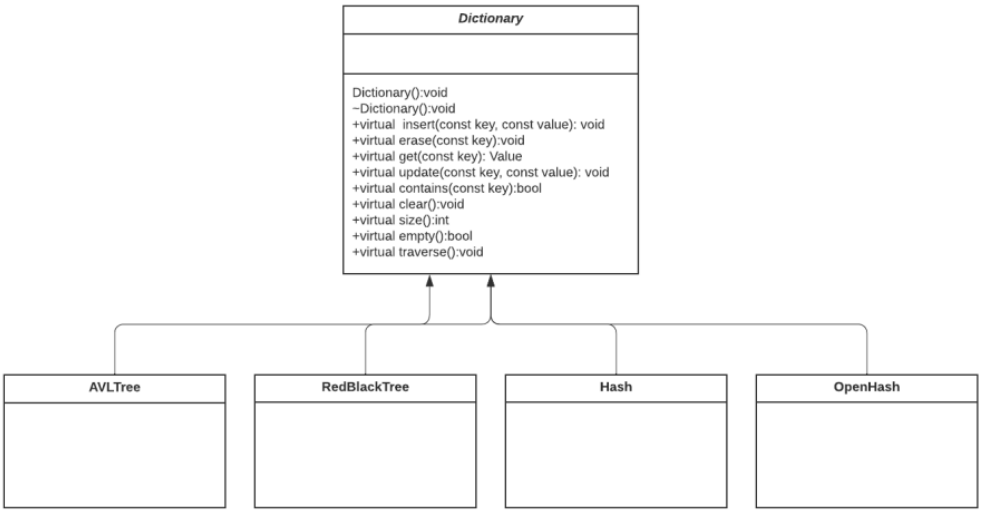
1 Resumo

Este trabalho tem como objetivo criar a implementação de um contador de frequência de palavras utilizando um dicionário genérico. Utilizei 4 estruturas de dados para criar o dicionário: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com tratamento de colisão por encadeamento exterior e Tabela Hash com tratamento de colisão por endereçamento aberto. O trabalho tem como objetivo medir essas ED's, como tempo de criação de frequência e número de comparações feitas. O objetivo acadêmico desse projeto é praticar e expandir os conhecimentos das estrutura de dados estudadas durante a cadeira.

2 Apresentação

Conseguimos implementar dicionários na computação(ou objetos em python) utilizando algumas ED's, como árvores AVL, árvores Rubro-Negras e tabelas hash. Cada um possui propriedades específicas que fazem as operações de inserção, get, update e deleção serem eficientes.

O programa principal realiza a leitura de texto de um arquivo .txt, a leitura de um tipo de dicionário e o arquivo de saída. Ele identifica palavra por palavra e atualiza ou insere no dicionário. Cada vez que roda um método da ED que implementa o dicionário, um contador de comparações é acionados, se tiver comparações. No final, todas as palavras(ou primeira 1024 palavras), são imprimidas junto com a quantidade de vezes que elas aparecem no texto. No final do arquivo de resultados também, é impresso o tempo de execução total gasto para rodar todo o programa e o número de comparações que a ED em questão teve que realizar para construir todo o dicionário. l0.6



Eu realizei um teste utilizando uma história ficticia que está na raíz do arquivo que fala de bruxas e jet-ski's. Cada uma das 4 ED's foram rodadas para esse texto, servindo de comparação, com um número aproximado de 500 palavras. Será melhor especificado na análise de resultados.

3 Descrição de atividades

Primeiramente, eu utilizei a ideia de modelagem de classes sugerida no documento de descrição de projeto. Nela, implementei uma classe abstrata chamada Dictionary.h, na qual as classes da AVL, RedBlackTree, Hash(hash com tratamento de colisão por encadeamento exterior) e OpenHash(hash com tratamento de colisão por endereçamento aberto). Todas essas classes tem como métodos públicos os métodos de Dictionary.h, fora seus métodos privados e públicos que não sobreescrevem.

Por isso, implementei a classe abstrata primeiro e comecei com a AVL. Essa ED pode ser usada também quando queremos implementar os Sets(conjuntos). Ela é uma árvore que consegue manter um balanceamento entre os nós pais e filhos. Ela possui rotações e rotações duplas a direita e a esquerda, na qual balancea a árvore quando alguma operação de que altera sua estruta é acionada, onde a diferença de subárvores de um nó nunca será maior que 2. Ela tem operações com tempo médio de $\Theta(\log n)$

A segunda ED utilizada no trabalho é a árvore rubro-negra, na qual sua implementação é uma AVL com umas propriedades a mais: Seus nós possuem cores, sendo eles ou vermelho ou preto. Ela não tem a restrição da diferença das subárvores de um nó ser maior que um, relaxando nesse ponto. Porém, com suas propriedades de cores, ela permite que uma subárvore seja

no máximo o dobro da outra em altura. A vantagem de se usar uma árvore rubro-negra em relação a uma AVL é que, a inserção e remoção de nós acontecem mais rápido. Já na AVL, a busa é mais eficiente.

Após isso, foi implementada a ED de tabela hash com tratamento de colisão por encadeamento exterior. Ela funciona armazenando múltiplos elementos no mesmo índice da tabela através de listas encadeadas, resolvendo o problema de colisões ao permitir que várias chaves diferentes compartilhem o mesmo "bucket". Isso simplifica a resolução de conflitos em cenários onde o hash de múltiplas chaves é o mesmo. Em termos de complexidade, a inserção, busca e remoção de elementos têm complexidade média de $\Theta(1)$, assumindo uma boa função de hash que distribua uniformemente os dados. No entanto, no pior caso, quando todos os elementos colidem em um único bucket, a complexidade pode se degradar para $\Theta(n)$, já que as operações dependerão da busca sequencial dentro da lista encadeada. Isso pode ser mitigado ao ajustar a taxa de ocupação (load factor) e realizando o rehash quando necessário, redistribuindo os elementos em uma tabela maior.

Após isso, foi implementada a ED de tabela hash com tratamento de colisão por endereçamento aberto. Esta abordagem trata colisões de maneira diferente da tabela hash com encadeamento exterior, utilizando um método chamado "endereço aberto". Nesse método, se uma posição na tabela já estiver ocupada, o próximo slot disponível é procurado dentro da própria tabela até encontrar um espaço livre.

No endereçamento aberto, as colisões são resolvidas procurando-se o próximo espaço livre na tabela, ao contrário do encadeamento exterior, onde cada posição da tabela pode manter uma lista de elementos para resolver as colisões. A estrutura de dados usada com endereçamento aberto é uma única estrutura contínua de memória (um vetor), enquanto no encadeamento exterior, cada posição da tabela é um ponteiro para uma lista de pares chave-valor. Em termos de complexidade, a inserção e a busca na tabela com endereçamento aberto podem se degradar para $\Theta(n)$ no pior caso, especialmente se a tabela estiver quase cheia, porque a busca por um slot livre pode se tornar demorada. Já no encadeamento exterior, a complexidade média de inserção e busca é $\Theta(1)$, embora a complexidade no pior caso também possa ser $\Theta(n)$ se todas as chaves forem agrupadas em poucos buckets. Além disso, o endereçamento aberto geralmente requer menos memória adicional, pois não utiliza listas encadeadas, enquanto o encadeamento exterior utiliza memória extra para armazenar os ponteiros das listas.

Após todas as estruturas implementadas, eu codifiquei o arquivo main.cpp. Nele, eu fiz as leituras do tipo de ED que seria utilizada para implementar o dicionário, o arquivo de texto que seria lido e o arquivo de saída do programa. Para a leitura do arquivo, utilizei alguns métodos das bibliotecas ofstream e

ifstream.

A leitura das palavras foram feitas utilizando métodos das bibliotecas algorithm e cctype, na qual foi tratado palavras com acentuação, palavras compostas e sinais extras que podem vir do texto. Eu também criei uma funcionalidade para deixar tudo minúsculo. Há uma função que não consegui fazer ela funcionar corretamente para salvar no arquivo de saída, mas contornei o problema utilizando a saída do terminal bash e cuspindo para o arquivo de saída que vem do argumento, gerando os resultados corretamente.

4 Análise dos Resultados

Para realizar a análise dos resultados, utilizei métricas de contagem de tempo total da montagem de tabela de frequência e contagem de número de comparações que o dicionário realizou para montar a tabela de frequências. O tempo é contabilizado dentro de uma função da main, envolvendo um laço de repetição que constrói o dicionário lendo cada palavra do arquivo de entrada. Para a métrica do número de comparações, em cada ED, há um atributo que incrementa em 1 quando o programa chega em alguma comparação feita. Eu só não coloquei no método Traverse(), pois o mesmo serve apenas para percorrer o dicionário e imprimir todas suas chaves e valores. Todas essas métricas de avaliação estavam nos requisitos do trabalho.

Para fazer o teste, gerei uma história ficticia em uma LLM, com o título de Bruxas e Jet-Ski's. A LLM me deu um arquivo .txt que se encontra na raíz do projeto. Esse texto possui um total de 517 palavras diferentes, metade da capacidade setada para o algoritmo. Gerei um arquivo de saída para cada ED implementada.

Dois fatos interessantes que eu notei foram: as tabelas hash fizeram um tempo de execução e um número de comparações menores do que as árvores, onde por exemplo, a avl realizou 42076 comparações e a openhash apenas 7834. O outro fato é que o método de traverse das árvores consegue deixar a ordem das chaves de maneira alfabética impressas no arquivo, enquanto que as tabelas hash fazem isso de maneira aleatória.

Para compilar e rodar o programa, você deve primeiro executar o comando: "g++ -std=c++11 -o freq main.cpp AVLTree.cpp RedBlackTree.cpp HashTable.cpp OpenHashTable.cpp", compilando o programa. Para executar, faça: "./freq avl texto.txt resultado.txt > resultado.txt", na qual ./freq é o arquivo compilado, avl é o parâmetro de qual tipo de dicionário você vai usar, podendo passar os seguintes parâmetros: "avl", "openhash", "redblack"e "hash"; o parâmetro resultado.txt é o arquivo na qual os resultados serão salvos. Você deve repetir o mesmo nome após o sinal de >.

Bibliografia

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. 3. ed. Rio de Janeiro: Elsevier, 2012.