Relatório do Trabalho de Estrutura de Dados Avançada Dicionários - Parte I

Paulo Henrique da Silva Sousa

[1] Universidade Federal do Céara (UFC)
Quixadá – CE – Brazil

phenriquedss@alu.ufc.br

Abstract. *This paper/article discusses the implementation, results, decisions made, and problems encountered in the implementation of frequency counters based on dictionaries, which themselves are built upon the following data structures: AVL Trees, Red-Black Trees, Chained HashTable, Open Addressing HashTable.*

Resumo. *Este "artigo" disserta acerca da implementação, os resultados, as decisões tomadas e os problemas encontrados na implementação de Contadores de frêquencia, baseados em dicionários que por se só se baseiam nas estruturas de de dados: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com tratamento de colisão por Encadeamento Exterior e Tabela Hash por Endereçamento Aberto.*

1. Descrição do Projeto

Um dicionário é um container que aloca pares únicos de chave e valor e busca os valores com base nas chaves. Nesse projeto devemos implementar um contador de frequência com base nos dicionários que por si só se baseiam nas estrtuturas de dados: árvores binárias de busca balanceadas vistas em aula e também as tabelas hash. As estruturas devem ser génericas e também respeitar os princípios da orientação a objeto, para além, a aplicação deve ler linhas de comando, também ler e criar arquivos.

1.1. AVL Tree

A árvore avl foi implementada seguindo as orientações dadas em sala de aula, usa um Node interno que recebe um par de chave e valor génericos além de ponteiros de para Node apontando para os filhos, por fim, um inteiro responsável pela altura do node dentro da árvore. Devido ao uso de recursão, a classe tem fuções públicas que chamam as funções recursivas de dentro da classe. Tem como funções principais: O contrutor, _insert, _remove, _fixUpNode, fixup_deletion, remove_successor, right_rotation, left_rotation e _balance.

O construtor inicializa as variáveis privadas da classe, _root, o nó principal da árvore, como nullo e o count, variável do tipo size_t que terá seu valor incrementado toda vez que uma comparação de chave for feita, como zero.

O _insert recebe como parâmetro da função pública a chave e o valor que devem ser alocados na estrutura, enviados pelo usuário e uma referência de Node que será usado como base da busca de posição, a função começa verificando:
1- Se a refência enviada é nula, se for, cria um novo Nó que usando as variáveis enviadas

como parâmetro.
2- Caso não, verifica se a chave é maior, menor ou igual que a chave do nó enviado, e ”caminha” nessa direção, se menor, esquerda, se maior, direita. Se for igual, atualiza o valor relacionado a chave e retorna o nó.
3- Ao fim atribui o retorno da função _fixUpNode ao nó criado, e o retorna.

O _remove recebe como parâmetro da função pública uma refêrencia de Node, o nó inicial da busca, e a chave do elemento a ser removido, a função ”caminha” pela árvore buscando um nó com a chave procurada, se não encontrar, retorna nulo, se sim, verifica se o filho direito do nó é nulo, se for, substitui o nó pelo seu filho esquerdo, caso não, atribui o retorno da remove_sucessor ao filho direito e atribui o retorno de fixUp_deletion ao nó e o retorna.

O _fixUpNode recebe como parâmetro uma referência de Node, nó que vai ser rebalanceado, e uma chave para comparação, essa função inicialmente guarda o retorno da função _balance em uma variável ”bal” do tipo inteiro, partindo desse ponto a função trata os seguintes casos:
1- bal menor que -1, que se divide em 2 sub-casos:

- **1a** A chave enviada é menor que a chave do nó enviado, então faz-se uma rotação à direita.
- **1b** A chave enviada é maior que a chave do nó enviado, então faz-se uma rotação dupla à direita (uma à esquerda e uma à direita) no nó.

2- bal maior que 1, que se divide em 2 sub-casos:

- **2a** Simétrico ao caso 1a.
- **2b** Simétrico ao caso 2b.

Ao final, a função conserta a altura do nó com base nos filhos e o retorna.

O fixUp_deletion recebe como parâmetro uma referência de Node, nó que vai ser rebalanceado, começa por guardar o retorno da função _balance em uma variável ”bal” do tipo inteiro, partindo desse ponto a função trata os seguintes casos:
1- bal maior 1, que se divide em 2 sub-casos:

- **1a** O balanço do filho direito do nó é maior ou igual a 0, então faz-se uma rotação à direita.
- **1b** O balanço do nó direito é menor que 0, então faz-se uma rotação dupla à esquerda (uma à direita e uma à esquerda) no nó.

2- bal menor que -1, que se divide em 2 sub-casos:

- **2a** Simétrico ao caso 1a.
- **2b** Simétrico ao caso 2b.

O remove_successor recebe duas refêrencias de Node, a primeira, root, é o nó que deve ser removido, e o segundo, node, é uma subárvore que se quer manter após a remoção do primeiro, a função caminha rumo esquerda, quando acha um nó que não tem filho esquerdo, substitui a chave e o valor de root por ele (o que em termos práticos o remove), guarda o filho direito do nó, exclui o nó sem filho esquerdo e retorna o filho direito, ao fim node recebe fixup_deletion e é retornado.

right_rotation e left_rotation, essas funções recebem uma referência de Node e tem como objetivo de ”rotacionar” um nó. Se um nó ”x” deve ser rotacionado a direita

por exemplo, esse nó vai virar filho direito do seu filho esquerdo, simétricamente para a rotação a direita.

O _balance, essa função recebe uma referência de Node e retorna o balanço dela, o calculo de balanço é feito subtraindo a altura do filho direito pela altura do filho esquerdo.

1.2. Red-Black Tree

A árvore rubro-negra foi implementada seguindo as orientações dadas em sala de aula, usa um Node interno que recebe um par de chave e valor génericos além de ponteiros de para Node apontando para os filhos e para o pai, por fim, um booleano que diz respeito a cor do nó. Devido ao uso de recursão em algumas funções, a classe tem funções recursivas que chamam as funções privadas de dentro da classe. Tem como funções principais: O construtor, _insert, _remove, _delete, insert_fixUp, delete_fixUp, left_rotate, right_rotate.

O construtor inicializa as variáveis privadas da classe, _root, o nó principal da árvore, o T_nil, nó auxiliar para iterações e o count, variável do tipo size_t que terá seu valor incrementado toda vez que uma comparação de chave for feita, como zero.

O _insert recebe uma chave e um valor associado, ”caminha” pela árvore na direção onde essa chave estaria, se acha uma chave igual, atualiza seu valor associado, se não, cria um novo nó de cor vermelha e o aloca o nó criado na posição devida, se for o primeiro a ser crido, se torna a raiz, se não se torna a esquerda ou a direita do nó prévio a ele, Ao fim chama insert_fixUp para o no criado.

O _remove recebe uma chave e ”caminha” pela árvore na direção onde essa chave deveria estar, quando chega na posição verifica se a chave existe, se não, não faz nada, se sim, chama _delete para o nó da chave.

O _delete recebe uma referência de Node, começa por remover o nó desejado, substituindo ele por um descendente, no entanto isso pode gerar problemas de balanceamento caso esse descendente e o nó que deve ser retirado sejam negros, pois isso gera perda de altura negra na subárvore a qual esse no pertence, então a função verifica e caso seja necéssario envia para a delete_fixUp.

O insert_fixUp recebe uma referência de Node como parâmetro e promete tratar os possíveis casos de desbalanceamento, baseando-se nos termos ”novo”, nó recém adicionado ou problemático, ”pai”, pai do nó récem adicinado, ”avô”, pai do pai do nó recém adicionado e ”tio”, nó irmão de pai:
1- Tio é vermelho, logo avô é negro, então a função troca a cor do pai e tio para negro e a do avô para vermelho, neste ponto novo já se encontra consertado, mas o problema póssivelmente subiu para o avô, continuamos o conserto.
2- Tio é negro e novo é o filho direito, a função apenas executa uma rotação a direita no pai, aqui o problema passa a ser no pai.
3- Tio é negro e novo é o filho da esquerda, pai ganha cor negra, avô ganha cor vermelha, então uma rotção a direita no avô.
Os casos 2 e 3 têm casos simétricos, portanto existem casos 2a e 2b e o mesmo vale para o 3.

O delete_fixUp recebe uma referência de Node de um nó desbalanceado e trata quatro póssiveis casos de rebalanceamento baseados em ”x” ser o nó que foi usado para

substituir o anterior e ”w” seu irmão:
1- Se w é vermelho, por definição, tem ambos filhos pretos, então a função troca a cor do pai de x com w, então faz uma rotação a esquerda no pai de x, gerando assim um dos outros casos.
2- w e seus filhos são negros, nesse caso o pai de x passa a ser o nó problemático, então torna w vermelho e o pai de x torna-se negro, se isso solucionar o problema a correção acaba, se não, continuamos aos outros casos.
3- w e seu filho direito são negros e seu filho esquerdo é vermelho, a função apenas efetua uma rotação a direita no w o que entra no caso 4.
4- w é negro e seu filho direito é vermelho, o filho direito de w torna-se negro, w ganha cor do pai de x, em seguida o pai de x torna-se negro e por fim é feita uma rotação a esquerda no pai de x.

right_rotation e left_rotation, essas funções recebem uma referência de Node e tem como objetivo de ”rotacionar” um nó. Se um nó ”x” deve ser rotacionado a direita por exemplo, esse nó vai virar filho direito do seu filho esquerdo, simétricamente para a rotação a direita.

1.3. Chained HashTable

A Chained Hashtable foi implementada seguindo as orientações dadas em sala de aula, usa um vector de listas para guardar os pares de chave e valor genéricos, usa por padrão a função de hash disponível na biblioteca padrão do c++ mas pode receber um função externa caso necéssario, tem como funções principais: O construtor, hash_code, add, remove e rehash.

O construtor recebe como parâmetro um size_t (inteiro longo não sinalizado) para tamanho inicial da tabela, 19 como padrão e um float para o fator de carga, 1.0 como padrão, a função inicializa os seus atributos aprivados, inicializa o contador de comparações de chave e o número de elementos com 0, procura o próximo primo a partir de valor indicado para o tamanho da tabela, e muda o tamanho da tabela para esse tamanho, então verifica se o fator de carga enviado é válido, se não, muda para 1.0.

O hash_code é uma função auxiliar privada que recebe uma chave e retorna o resto do hash dessa chave, usando a função nativa do c++ ou a carregada pelo usúario, pelo tamanho da tabela.

O insert recebe uma chave e um valor, então verifica se m_max_load_factor é menor que o retorno da função load_factor, se for aumenta o tamanho da tabela usando a função rehash, após, a função usa hash_code pra calcular a posição no vector que o elemento deveria estar, então caminha pela lista presente nesse espaço para verificar se aquela chave já está presente, se não, a adiciona e aumenta o numero de elementos, caso não atualiza o valor associado.

O remove recebe uma chave e usa hash_code pra calcular a posição no vector que o elemento deveria estar, então caminha pela lista presente nesse espaço para verificar se aquela chave já está presente, se sim a remove e diminui o número de elementos, se não, não faz nada.

O rehash recebe um size_t e faz com que a tabela assuma o tamanho do próximo primo a partir desse valor, a função guarda os elementos da tabela, limpa os elementos

da tabela atual, aumenta seu tamanho e os adiciona de novo um por um, usando a função add.

1.4. OpenAdress HashTable

A OpenAdress HashTable foi implementada seguindo as orientações dadas em sala de aula, usa um vector para guardar os pares de chave e valor genéricos, usa por padrão a função de hash disponível na biblioteca padrão do c++ mas pode receber uma função externa se nécessario, usa um enum para verificação de status e usa também um Node interno que recebe a chave, valor e um status, tem como funções principais: O construtor, aux_hashSearch, hash_code, add, remove e rehash.

O construtor recebe como parâmetro um size_t (inteiro longo não sinalizado) para tamanho inicial da tabela, 19 como padrão e um float para o fator de carga, 1.0 como padrão, a função inicializa os seus atributos aprivados, inicializa o contador de comparações de chave e o número de elementos com 0, procura o próximo primo a partir de valor indicado para o tamanho da tabela, e muda o tamanho da tabela para esse tamanho, então verifica se o fator de carga enviado é válido, se não, muda para 1.0, ao final preenche todos os slots do vector com Node ”vazios” e com status empty.

O aux_hashSearch é uma função auxiliar privada recebe uma chave e verifica se essa chave está presente na tabela, retorna a posição da chave se estiver, se não -1.

O hash_code recebe uma chave e usa funções auxiliares para retornar a posição que ela deve ficar na tabela. Uma da funções auxiliares retorna uma ”posição inicial” e a outra retorna um ”salto”, para que a ”posição inicial” esteja ocupada, a chave seja alocada em outra posição.

O insert recebe uma chave e um valor, então verifica se m_max_load_factor é menor que o retorno da função load_factor, se for aumenta o tamanho da tabela usando a função rehash, após usa a função aux_hashSearch para verificar se a chave existe na tabela, caso sim, atualiza seu valor caso não, usa hash_code para procurar uma posição válida para a chave, quando acha, adiciona.

O remove recebe uma chave e usa aux_hashSearch para procurar sua posição na tabela, se ela existir, torna seu status deleted.

O rehash ecebe um size_t e faz com que a tabela assuma o tamanho do próximo primo a partir desse valor, a função guarda os elementos da tabela, limpa os elementos da tabela atual, inserindo um Node empty em todos os espaços, aumenta seu tamanho e os adiciona de novo um por um, usando a função add.

1.5. Dicionário

A estrutura do dicionário foi pensada com a intensão de englobar todas as estruturas de maneira a simplificar o código e facilitar o entendimento, recebe um par de chave e valor genérico e também a estrutura que há de ser utilizada, chamando as funções da respectiva estrutura.

1.6. Tratamento de string

Para o tratamento de string foi utilizado por orientação do professor a biblioteca de tratamento de string internacional icu, foram adicionadas ao arquivo main duas funções

principais, ”split_words” e ”string_treatment”, responsáveis por respectivamente separar os blocos de palavras em palavras únicas e por retirar caracteres indesejados como sinais de pontuação.

2. Métricas utilizadas

As métricas utilizadas dentro do projeto para avaliar e comparar o desempenho das estruturas podem ser divididas em dois tipos:

Métricas gerais: Dentro das estruturas foram adicionados contadores de comparação de chave e fora(ainda não executado) um contador de tempo, para analisar a quantidade de tempo a estrutura leva para realizar o objetivo **Métricas individuais**: As árvores e as tabelas receberam contadores individuais pela diferença estrutural de ambos os tipos, as árvores receberam contadores de rotações, e as tabelas contadores de colisão.

3. Listagem de testes executados e Resultados encontrados

Na reta final do projeto, ao passo que tudo estava completamente implmentado, forma executados cerca de 24 testes finais, nestes testes foram usadas todos os arquivos teste sugeridos pelo professor orientador como também cada uma das estruturas implementadas, para além dos testes finais, foram executados incontaveis testes intermediarios durante o processo de desenvolvimento da aplicação, usados para testar as estruturas, o tratamento de string e etc.

Dado as entradas devidas, segue uma listagem da média das execuções testes de cada estrutra, contando com tempo medido em nanosegundos tal como os parâmetros específicos para cada tipo de estrutura:

Árvore Avl:
média tempo: 3.504.377.119
média comparações: 7.220.824
média rotações: 6282

Árvore Rubro-Negra:
média tempo: 238.219.970
média comparações: 3.233.422
média rotações: 7952

Chained Hash:
média tempo: 2.338.211.888
média comparações: 648.956
média colisões: 8681

Open Hash:
média tempo: 2.687.003.842
média comparações: 998.117
média colisões: 84.037

4. Conclusão

Como já era póssivel de se supor, a árvore rubro-negra denotou o desempenho mais notável dentre as estruturas, com o menor tempo médio de execução devido a suas funções

inteiramente iterativas e com métodos de busca e de inserção extremamente refinados. O que pode ser considerado uma surpresa é o desempenho da tabela hash com encadeamento, que mostrou resultados incomparáveis nas médias de comparação e colisão, isso ocorreu devido a uma decisão de implementação, que torna o fator de carga ”base” dela igual a um, tornando ela nesse caso mais eficiente que sua irmã de endereçamento aberto, por não lidar bem com esse fator de carga.

5. Especificações da máquina

Toda a implementação foi desenvolvida em C++ 20 e foi utilizado o editor de código Visual Studio Code, a máquina utilizada durante o processo tem as seguintes especificações:

Processador:	Intel Atom x5-Z8350 (4 núcleos, 1.92GHz)
RAM:	2GB + 3GB Swap
Armazenamento:	29GB eMMC
Sistema:	Linux Mint 22.1 (Ubuntu 24.04)
Kernel:	6.8.0-51-generic
GPU:	Intel HD Graphics