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

Guilherme Andrade da Silva[1]

[1]Universidade Federal do Ceará (UFC)
Av. José de Freitas Queiroz, 5003 – Quixadá – CE – Brazil

guilherme.andrade@alu.ufc.br

Abstract. *This document describes the implementation of a program that generates a word frequency list from any given book. The program utilizes dictionaries to store its elements and perform operations. For this program, four distinct dictionary structures were implemented: An AVL (Adelson-Velsky and Landis) Tree, a self-balancing binary search tree; A Red-Black Tree, a self-balancing binary search tree; A Hash Table with collision resolution via Separate Chaining; A Hash Table with collision resolution via Open Addressing. The implementation and functionality of each data structure are explained, along with their respective performance metrics.*

Resumo. *Documentação sobre a implementação de um programa que gera uma lista das frequências de palavras de um livro qualquer. O programa usa dicionários para guardar seus elementos e realizar as operações. Para esse programa foi implementado 4 dicionários: usando Árvore Binária Balanceada de Busca AVL (Adelson-Velsky e Landis), Árvore Binária Balanceada de Busca Rubro Negra, Tabela Hash Com Tratamento de Colisão Por Encadeamento Exterior e Tabela Hash Com Tratamento de COlisão Por Enderaçamento Aberto. É explicado a implementação e a funcionalidade de cada estrutura, assim como suas métricas.*

1. Descrição do Projeto

O projeto consiste em implementar um programa que gera uma lista das palavras de um livro qualquer e quantas vezes ela aparece no livro, ou seja, sua frequência. O programa não diferencia maiúsculas de minúsculas, sendo assim, ”House”e ”house”serão interpretadas como a mesma palavra. Além disso, essa lista é gerada em ordem alfabética.

Como estrutura para guardar a frequência de cada palavra, é usado o Dicionário. O dicionário é uma estrutura que guarda um par de chave e valor e realiza as operações de inserção, busca e remoção em tempo médio de O(1) ou O(lg n), a depender se foi implementado com tabelas de dispersão ou árvores balanceadas, respectivamente. Nesse trabalho, foi implementado 4 tipos de dicionários: com tabela hash encadeada, com árvore balanceada avl, com árvore balanceada rubro negra e tabela hash com endereçamento aberto

No programa de frequência em específico, as chaves dos pares serão as palavras e o valor a frequência com que aparecem. Mas os dicionários foram implementados usando tipos genéricos, podendo ser usado para qualquer outro fim.

Por fim, existem diversas aplicações da estrutura dicionário, uma dessas é essa que é tratada nessa documentação, que é gerar a frequência de certos elementos. Também pode ser usado para representar grafos, onde as chaves são nós e os valores são as listas de nós adjacentes, armazenamento de dados em cache, entre outras formas.

1.1. Árvore Binária de Busca Balanceada - AVL

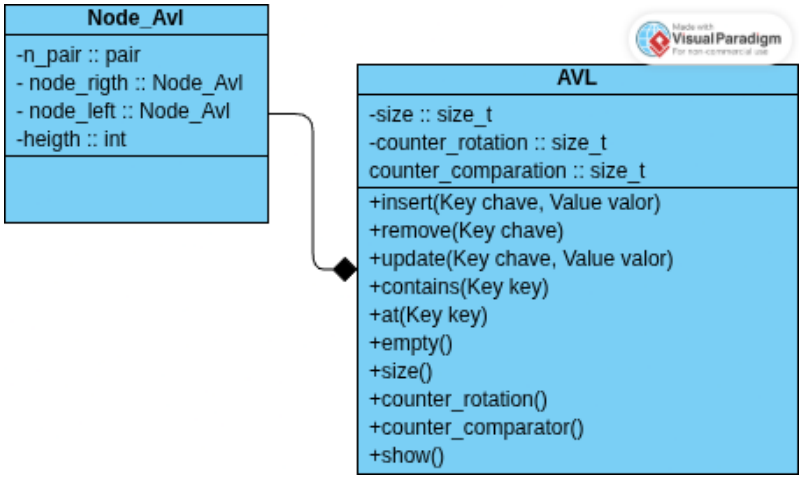

Figura 1. AVL

A implementação da estrutura AVL foi feita usando a estrutura auxiliar Node, que guarda o par em si (chave e valor) do tipo genérico, a altura do nó e os ponteiros para seu nó esquerdo e direito.

Todos os seus algoritos foram feitos usando recursão, sendo assim, foi feito o insert, que insere um par na árvore, caso já exista um par com essa chave, simplesmente não fazemos nada.

Já a função update, procura o nó com a chave passada no seu parâmetro e atualiza com o novo valor passado na função. Caso não exista, é retornado um erro.

As funções de busca e at (que retorna o valor relacionado a uma chave), funcionam de maneira bastante geral, a busca retorna true se a chave existir e false se nao. O at retorna uma exceção caso não ache aquele par com a chave que foi passada no seu parâmetro.

O remove também busca o nó, se o achar o deleta da árvore, caso não, nada é feito. O clear remove todos os nós (com exceção da raiz) e também existe função que retorna seu tamanho e suas métricas. Há também uma função que verifica se a árvore é vazia e outra que a imprime em formato legível.

Quanto a sua criação, foi feito um construtor padrão para a classe. A implementação também tem funções privadas principalmente para balancear a árvore e a consertar depois de uma inserção e remoção e para facilitar a sobrecarga dos operadores de atribuição, igualdade e diferença. Também foi criado um construtor de cópia.

1.2. Árvore Binária de Busca Balanceada - Rubro Negra

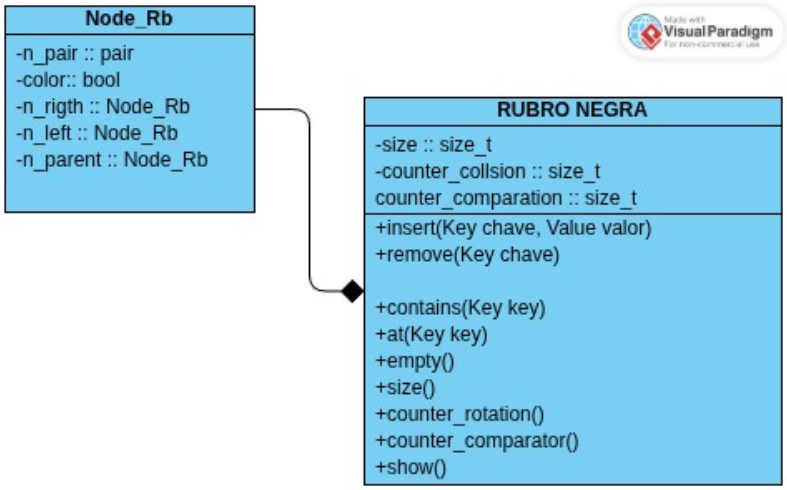

Figura 2. Rubro Negra

A implementação da árvore rubro negra foi feito bastante parecido com a avl, no sentido de ser usado tipos genéricos, ponteiros, rotações e uma estrutura Node para representar o nó. A única diferença é que o nó guarda cor, em vez de altura, que pode ser preto ou branco. E também há um ponteiro há mais que é um ponteiro para o pai daquele nó. E de maneira geral, o nil usado nos algoritmos de rubro negra foi implementado de maneira que só há um nil na memória e os algoritmos foram adaptados para isso. Isso economiza memória e facilita o gerenciamento dos nós.

As funções que temos são basicamente as mesmas que temos na AVL. Foi implementado o insert, que insere um par na árvore, caso o par já exista, o valor relacionado a ele é atualizado, essa função retorna true se o valor for inserido e false se for atualizado. A função remove busca o par e o remove se ele existir, caso não, nada é feito. O contains verifica se aquele chave existe na árvore e retorna True se e somente se ele existir.

Há funções recursivas, mas foi priorizado implementações iterativas para melhor uso dos ponteiros para pai que existe nos nós. Continuando a listagem das funções, há a função at, que assim como na AVL, ela retorna o valor relacionado aquele nó ou uma exceção caso não exista na árvore. As funções empty, clear, as que retornam as métricas e os size, assim como sobre as sobrecargas e construtor de cópia, funcionam de maneira semelhante a AVL.

Dentre as dificuldades, houve confusão no gerenciamento quanto ao nullptr e nil, sobre a diferenciação deles dois. Ainda mais no clear e destrutor que há um processo delicado de gerencimento de memória. Assim como no remove, o qual teve que ser usado para melhor implementação o algoritmo de rb_transplant e o remove e fixed_up_remove baseados no rb_transplant conforme definido por Cormen et al. (2009).

1.3. Tabela Hash Com Tratamento de Colisão Por Encadeamento Exterior

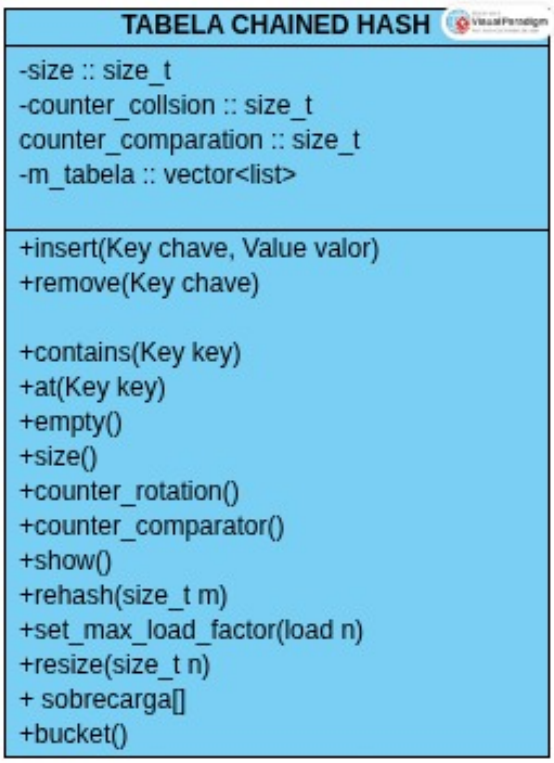

Figura 3. Tabela encadeada

Diferentemente das outras classes, a tabela hash com encadeamento exterior não precisou de uma estrutura auxiliar como o node. Mas precisou de um tipo genérico a mais, que foi o hash para as operações de hash codes.

Assim, esta tabela tem as mesma funções que a AVL e Rubro-Negra, não tendo o update, pois o seu insert funciona como o da rubro negra, retornando True se tiver sido inserido e False se tiver sido atualizado. A sobrecarga de iterador foi feito de maneira que se não existe aquele par, a função já o cria e retorna o construtor padrão vazio para o objeto.

Fora as operações padrões, há as funções próprias de uma tabela hash: rehash, o qual recebe o novo do tamanho da tabela e ela é adaptada para aquele tamanho, com seus elementos realocados, essa função também é realizada toda vez que a tabela atinge seu fator de carga máximo. Resize, que recebe o número de elementos que a tabela tem que se adaptar para receber. Set_max_load_factor que muda o fator de carga máximo da tabela, fazendo uma operação de reserve, que realiza um rehash se necessário. E também uma função que retorna o fator de carga atual da tabela.

A implementação foi feita também usando tipos genéricos e funções auxiliares que ajuda no tempo de operação: hash_code e função que retorna o próximo número primo de um número. Além disso, há uma função que retorna quantos elementos existem na tabela, outra que retorna quantos buckets existem e qual o bucket em que um par está.

Como estruturas auxiliares, foi usado um vetor de listas encadeadas, para melhor

desempenho da estrutura e duas operações. Além da sobrecarga de iterador, foi feita uma sobrecarga de impressão, para facilitar a manipulação e uso da estrutura fora de sua classe.

Para fim das métricas, consideramos a colisão, que na implementação é verificado se a lista em que vamos inserir o nó é empty ou não, se não for, há uma colisão e logo em seguida inserimos o nó.

1.4. Tabela Hash Com Tratamento de Colisão Por Endereçamento Aberto

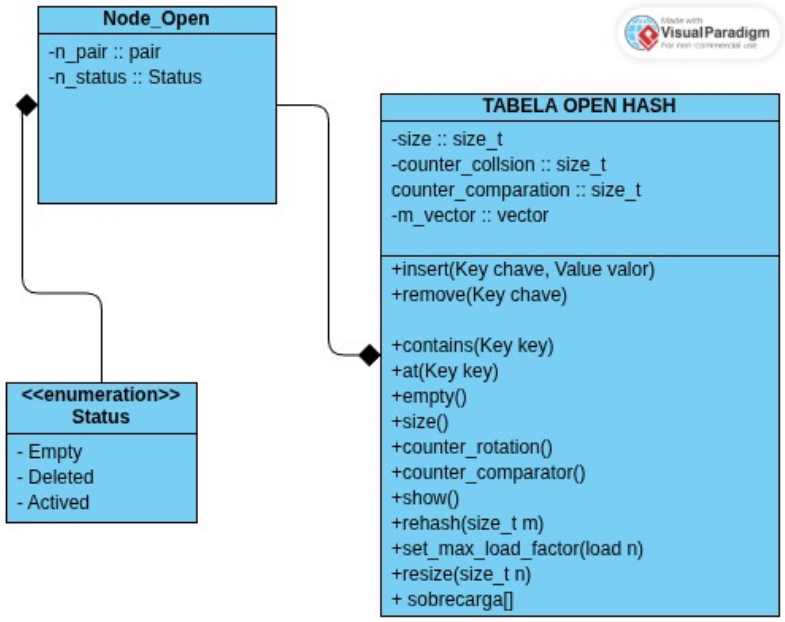

Figura 4. Tabela aberta

A tabela hash com endereçamento aberto foi implementada com uma estrutura auxiliar Node, o qual guarda o par e seus status. A mesma estrutura contém um construtor vazio, o qual foi pensado para não ter que inicializar cada elemento da estrutura que vai armazenar a tabela como Empty, pois o construtor vazio já cria o Node com os construtores vazios dos tipos genéricos e Empty para seu status.

Há também outra estrutura auxiliar que é o vetor que armazena os nós da tabela, no qual todos os seus slots estão ”ocupados”, em sentido de memória. Mas só consideramos preenchidos aqueles que são Active, Empty são os que estão vazios e nunca foram usados e Deleted os que já foram Active, mas o seu elemento foi removido.

Nas operações, considera-se essencialmente esses estados dos slots. Na inserção, ao acharmos o bucket que inserirmos aquele par, mudamos seu estado para Active, se antes não era. Na inserção usamos um algoritmo de encontrar o primeiro slot empty, caso ache um deleted, verificamos os outros para ver se o elemento já não está na tabela para o atualizarmos. Retorna true se foi inserido e false se foi atualizado.

Na remoção, consideramos os elementos ativos e ao acharmos para mudar seu status para deleted e por segurança, mudamos seu par para o construtor vazio de cada objeto, da chave e do valor. Em ambas as funções de remoção e inserção, diminuimos e

incremetamos o size, respectivamente. Também há funções constantes que retorna o size, as métricas da tabela e o número de buckets da tabela.

Na busca, consideramos também apenas os elementos ativos e retornamos true se e somente se o elemento existr. As operações at e iterador funcionam semelhantemente as da tabela com encadeamento. Também foi sobrecarregado um cout para melhor manipulação fora da classe.

Quanto ao rehash e hashcode, o rehash acontece se o número de elementos for igual ao size/capacity do vetor, pois o número de elementos é um atributo a parte do vetor. Também ocorre quando é atingido o seu fator máximo de carga, o que ocorrer primeiro. Para melhor uso e desempenho da tabela, foi usado hashing duplo, ou seja, foi implementado duas funções de hash code para que as funções e operações sejam realizadas no melhor tempo possível.

Por fim, não foi implementado uma função que devolva para onde uma slot foi alocado, pois não há sentido fazer uma função assim usando hashing duplo.

2. Métricas utilizadas

Em todas as estruturas, há a métrica de comparação de chaves onde em cada função que realize de fato uma comparação de chave, ela é incrementada.

```
Node *fixup_node(Node *node, Key k) {
    int bal = balance(node);

    if(bal < -1){

        //Caso 1(a)
        this->m_counter_comparator++;
        if(k < node->left->n_pair.first){
            return right_rotation(node);
        }

        //Caso 1(b)
        this->m_counter_comparator++;
        if(k > node->left->n_pair.first){
            node->left = left_rotation(node->left);
            return right_rotation(node);
        }
    }

    if(bal > 1){

        //Caso 2(a)
        this->m_counter_comparator++;
        if(k > node->right->n_pair.first){
            return left_rotation(node);
        }

        //Caso 2(b)
```

```
        this->m_counter_comparator++;
        if(k < node->right->n_pair.first){
            node->right = right_rotation(node->right);
            return left_rotation(node);
        }
    }

    node->height = 1 + std::max(height(node->left), height(node
    ->right));
    return node;
}
```

Listing 1. Função de correção do balanceamento (fix-up) da AVL. Aqui podemos ver a implementação da métrica de comparações de chaves, que acontece antes do if para a implementação correta

Também existe uma função em cada estrutura que retorna quantas comparações de chaves foram realizadas. Caso a estrutura seja limpa, seja executada a função clear, as métricas são zeradas, para melhor reaproveitamento da estrutura para qualquer outro fim. Ademais, fora essa métrica geral, também existem métricas específicas para cada tipo de estrutura.

2.1. Árvores balanceadas

Para as árvores balanecadas, a métrica específica escolhida foi o número de rotações que ela realiza durante suas operações, tanto na AVL, como na Rubro-Negra. Sendo assim, toda vez que a estrutura realiza uma rotação o contador é incrementado.

```
void leftRotate(Node* x){
    Node* y = x->right;

    x->right = y->left;

    if(y->left != this->nil){
        y->left->parent = x;
    }

    y->parent = x-> parent;

    if(x->parent == this->nil){
        this->root = y;
    }else if(x== x->parent->left){
        x->parent->left = y;
    }else{
        x->parent->right = y;
    }

    y->left = x;
    x->parent = y;

    this->m_counter_rotation++;
}
```

Listing 2. Função de rotação à esquerda da Rubro Negra, a métrica de rotação é incrementada no final, sendo uma maneira simples de ser corretamente controlada

Também existe uma função em cada uma das árvores que retorna a métrica de rotações.

2.2. Tabelas hash

Para as tabelas, a métrica específica escolhida foi o número de colisões que acontecem durante suas operações. Para fins de implementação, foi considerado como colisão quando um par durante sua inserção é alocado para um bucket que já está ocupado, conforme Cormen et al. (2009). Sendo assim, toda vez que acontece uma colisão o contador é incrementado.

```
bool add(const Key& k, const Value& v){

    if(load_factor() >= m_max_load_factor){
        rehash(m_table_size * 2);
    }

    int slot = hash_code(k);

    for(auto p : m_table[slot]){
        this->m_counter_comparator++;
        if(p.first == k){

            return false;
        }
    }

    if(!m_table[slot].empty()){
        this->m_counter_collision++;
    }

    m_table[slot].push_back(std::make_pair(k,v));


    m_number_of_elements++;

    return true;

}
```

Listing 3. Função de adição em uma tabela hash com encadeamento aberto, a colisão acontece verificando antes de inserir o elemento se aquela lista é empty ou não, caso seja, a métrica é incrementada

Também existe uma função em cada uma das tabelas que retorna a métrica de colisões.

3. Especificações

As estrutura e suas funções foram implementadas em C++ e testadas em um ambiente com as seguintes especificações:

- Processador AMD Ryzen 5 7520U with Radeon Graphics, com frequência máxima de: (CPU MHz máx.: 6734,7651)
- Memória RAM de 5.52 GB
- Memória SWAP de 2.00 GB
- Arquitetura x86-64
- 250,4 GB de armazenamento em SSD
- Compilador g++ versão 13.3.0
- Sistema Operacional Linux Mint versão 22.1
- Kernel Linux 6.8.0-63-generic

4. Conclusão

As estruturas foram implementadas conforme suas operações e para funcionalidades genéricas. Com implementações que tem o tempo médio destinado de cada estrutura, no sentido geral O(lg n) para as árvores e O(1) para as tabelas hash. Foi tomado um cuidado para serem os mais genéricas possíveis de maneiras que não estão atreladas a esse projeto em específico.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3rd edition.