As mensagens secretas da Aliança Rebelde
Estrutura de Dados

Rita Rezende Borges de Lima - 2019021760

[1]Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

ritarezende@ufmg.br

1. Introdução

O problema proposto nesse trabalho consiste em automatizar a encriptação e decriptação de mensagens trocadas pela Aliança Rebelde, assim, assegurando que o Império não compreenda os planos de defesa das civilizações aliadas aos rebeldes caso as mensagens sejam interceptadas. Para o sistema de encriptação era necessário utilizar como base uma árvore binária com a visitação pré-ordem.

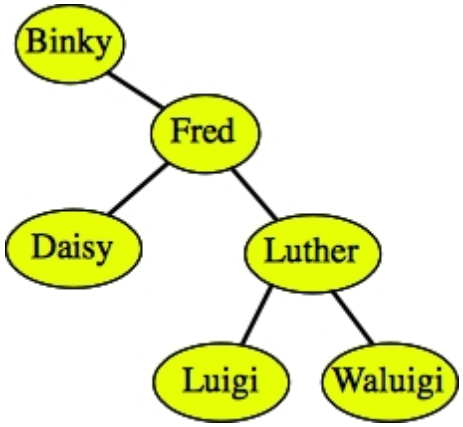
2. Implementação

O código foi desenvolvido na linguagem *c++* e compilado com *g++ -std=c++11* da GNU Compiler Collection

2.1. A Estrutura Árvore Binária de Pesquisa

Os dados são organizados por meio de uma árvore binária de pesquisa, uma estrutura baseada em nós que devem ter no máximo dois filhos e que devem seguir a seguinte regra: nós a esquerda tem valor inferior e nós a direita valor superior a seu pai. No caso específico do problema o valor dos nós são strings, dessa forma a ordenação baseia-se na ordem alfabética.

Figure 1. Exemplo de Árvore com Strings Ordenadas Alfabeticamente [1]



[1]Acessado em: https://web.eecs.utk.edu/~jplank/plank/classes/cs140/Notes/Trees/jpg/BST-5.jpg

2.2. O percurso em pré-ordem

Existem múltiplas formas de percorrer uma árvore, a maneira abordada nesse trabalho para a encriptação é a pré ordem. No programa cada nó, por consequência cada palavra, possui um inteiro associado a ele. Para enumerarmos os nós utilizamos a ordem em que estes são percorridos na pré ordem. Usando esse algoritmo, primeiro visitamos a raíz, seu nó da esquerda, seu nó da direita e assim recursivamente até o fim da árvore. É possível ver um exemplo desse algoritmo em *c++* abaixo:

```
void preOrdem(Node *n){
     if(n != NULL){
          visit(n);
          preOrdem(n->leftChild);
          preOrdem(n->rightChild);
     }
 }
```

2.3. Classes implementadas

2.3.1. Node

A classe node representa cada elemento pertencente dentro da árvore. Em seus atributos temos um vetor de caracteres para o valor, um ponteiro para o nó filho da esquerda e um para o filho da direita. A classe também possui os seguintes métodos:

- **O construtor** da classe que iguala a variável value ao vetor de caracteres passado como parâmetro e os nós filhos como $NULL$.

- **O método *int is bigger than value(char *value)*** que verifica se o valor presente no nó é maior que o passado por parâmetro de acordo com a ordem alfabética e assim retorna 1, caso contrário retorna 0.

- **O método *int is smaller than value(char *value)*** que verifica se o valor presente no nó é menor que o passado por parâmetro de acordo com a ordem alfabética e assim retorna 1, caso contrário retorna 0.

- **O método *int left child exists ()*** que verifica se o valor do filho da esquerda é diferente de $NULL$, ou seja, se ele existe.

- **O método *int right child exists ()*** que verifica se o valor do filho da direita é diferente de $NULL$, ou seja, se ele existe.

- **O método *int node contains key (char *value)*** que verifica se o valor da variável do nó ”value” é igual à string passada por parâmetro.

2.3.2. Binary Tree

Esta classe implementa a estrutura já discutida na sessão 2.1 árvore binária de pesquisa. A classe tem como atributos um ponteiro para o nó raiz da árvore e os seguintes métodos:

- **O construtor** da classe que apenas iguala a raiz da árvore à $NULL$.
- **O destrutor** da classe que percorre a árvore deletando os nós em pós ordem.
- **O método *void insert (char *value)***, que chama a função privada recursive insert (char *value, Node* n) passando o valor recebido como parâmetro e a raiz da árvore respectivamente.
- **O método *void recursive insert (char *value, Node*n)***, que percorre nossa árvore comparando o valor recebido com o valor de cada nó até achar o lugar correto para inserir e criar uma nova folha.
- **O método *void encrypt (char *value)***, que chama a função privada recursive encrypt (char *value, Node* n, int pos) passando o valor recebido como parâmetro, a raiz da árvore e uma referência a um contador inicializado como um respectivamente.
- **O método *void recursive encrypt (char *value, Node*n, int pos)***, que percorre a árvore em pré ordem comparando o valor recebido com o valor de cada nó e incrementando a variável posição até achar o valor correto e retornar a posição encontrada.
- **O método *void decrypt (int to decrypt)***, que chama a função privada recursive decrypt (int to decrypt, Node* n, int pos) passando o valor recebido como parâmetro, a raiz da árvore e uma referência a um contador inicializado com um respectivamente.
- **O método *void recursive decrypt (int to descypt, Node*n, int pos)***, que percorre a árvore em pré ordem até que a posição seja igual a variável to decrypt, quando isso ocorre, retornamos o valor do nó atual.
- **O método *void substitute (char *old value, char *new value)***, que recebe uma palavra pertencente a árvore que deve ser retirada e uma nova palavra a ser inserida, dessa forma chamando primeiro a função de remoção *remove node (Node* n, char *value)* e depois chamando a função de inserção passando a nova string.
- **O método *remove node (Node* n, char *value)***, que percorre a árvore ate encontrar o nó que tem que ser removido. Quando encontrado um dos três casos ocorre. Primeiro caso: o filho da direita é nulo, então basta fazer com que o nó passe a ser o filho da esquerda. Segundo caso: o filho da esquerda é nulo, então basta fazer com que o nó passe a ser o filho da direita. Terceiro Caso: O nó a ser removido tem dois filhos, nesse caso substituimos o nó pelo seu antecessor, ou seja o maior nó da sub árvore da esquerda. Para isso chamamos o método *void predecessor (Node* n, Node* right)*.
- **O método *void predecessor (Node* n, Node* right)***, que encontra o nó mais a direita da sub árvore da esquerda do nó original, ou seja, o nó antecessor ao nó a ser removido, e iguala o nó original que ia ser removido a este.

2.4. Entrada e Saída do programa

A entrada do programa é a entrada padrão do sistema (stdin). Cada linha consiste de uma de quatro possíveis instruções, sendo estas:

- **I:** indica uma inserção na árvore, para esta operação a linha de entrada constitui do caractere 'i' seguido de uma string, que será o valor armazenado no nó a ser inserido. Para esta operação, nada é impresso na saída.
- **E:** indica uma operação de encriptação nas palavras passadas, para esta operação a linha de entrada é constituida pelo caractere 'e', um número M que indica a quantidade de palavras a serem encriptadas e as M palavras. O programa irá retornar os M inteiros correspondentes as palavras passadas pelo usuário.
- **D:** indica uma operação de decriptação nos números passados, para esta operação a linha de entrada é constituida pelo caractere 'd', um número M que indica a quantidade de números inteiros a serem decriptados e os M inteiros. O programa irá retornar as M palavras correspondentes aos números passados pelo usuário.
- **S:** indica uma subsituição de um dos valores presentes na árvore, para esta operação a linha de entrada constitui do caractere 's' seguido de duas strings sendo a primeira a palavra presente na árvore que deverá ser retirada, e a segunda a que será inserida. Para esta operação, nada é impresso na saída.

Ao terminar de passar as operações o usuário deve digitar $CTRL + D$ no terminal para terminar o programa.

3. Análise de Complexidade

Agora que entendemos o funcionamento do programa podemos definir a complexidade de suas funções e de forma geral.

3.1. Tempo

No ínicio da execução inicializamos a árvore chamando seu construtor que apenas iguala a raiz à $NULL$, logo seu custo é constante e no final da execução chamamos o destrutor da árvore que percorre todos os nós deletando-os. A complexidade do programa varia de acordo com a quantidade de nós, n, da árvore, a quantidade de operações, m, passadas e quais operações são escolhidas pelo usuário. Assim, convém analisar o custo de cada uma dessas operações:

- **I:** A operação 'i' chama a função *operation insert (BinaryTree *t)* da main que por sua vez chama a função de inserção da árvore. Como esta está ordenada como uma árvore binária de pesquisa, para mantermos a ordenação é necessário inserir o novo nó em um local apropriado de acordo com o que foi descrito na sessão 2.1. Dessa forma é necessário percorrer "um galho" da árvore o que pode ocasionar o pior caso $O(n)$ ou o caso médio $O(log(n))$.

- **S:** A operação 'e' chama a função *operation encrypt (BinaryTree *t)* da main que lê a quantidade de palavras a serem encriptadas, p, e chama a função de encriptação da árvore p vezes para cada uma das p palavras passadas. A função de encriptação percorre a arvore em pré ordem até encontrar um nó que tenha valor igual ao passado de maneira que seu pior caso é $O(n)$ e o pior caso da função da main é $O(p * n)$.

- **D:** A operação 'd' chama a função *operation decrypt (BinaryTree *t)* da main que lê a quantidade de números a serem decriptados, d, e chama a função de decriptação da árvore d vezes para cada um dos d números passados. A função de decriptação percorre a arvore em pré ordem incrementando um contador inicializado em 1 até que esse contador tenha o mesmo valor do número passado pelo usuário. Assim, o pior caso do método da árvore é $O(n)$ e o pior caso da função da main é $O(d * n)$.

- **S:** A operação 's' chama a função *operation substitute (BinaryTree *t)* da main que lê a a palavra que será removida e a palavra que será adicionada e chama a função de substituição da árvore. Essa por sua vez chama a função de remoção que percorre a árvore até encontrar o elemento a ser retirado tendo como caso médio $O(log(n))$ e pior caso $O(n)$. Após isso a função de substituição chama a função de inserção ja discutida anteriormente, essa possui como caso médio $O(log(n))$ e pior caso $O(n)$. A operação em sua totalidade possui caso médio $O(log(n))$ e pior caso $O(n)$.

A complexidade das funções que sempre ocorrem são $O(n)$. Como é possível chamar m operações que podem fazer d ações percorrendo os n nós da árvore, a complexidade geral do algoritmo é de $O(1)$ + $O(n)$ + $O(m * d * n)$ que é o mesmo que $O(m * d * n)$.

3.2. Espaço

A complexidade espacial irá depender apenas da quantidade de nós, ***n***, pertencentes à árvore. Logo o custo é da forma $O(n)$.

4. Instruções de compilação e execução

Para a compilação do programa basta a utilização do *Makefile* presente dentro da pasta *src* do projeto executando o comando *make* no diretório. Para a execução: *./tp3*.
O algoritmo foi implementado e testado em um ambiente linux Ubuntu:

> Distributor ID: Ubuntu
> Description: Ubuntu 18.04.5 LTS
> Release: 18.04
> Codename: bionic

Utilizando o compilador *g++ -std=c++11*.

5. Conclusões

A partir da implementação do trabalho foi possível colocar em prática os conhecimentos adquiridos na disciplina de estrutura de dados a respeito de árvores binárias. Também foi possível observar a eficiência dessa estrutura para pesquisa em comparação a por exemplo uma busca sequencial.

6. Bibliografia

Ziviani, N. (2006).Projetos de Algoritmos com Implementações em Java e C++: Capítulo 3: Estruturas de Dados Básicas. Editora Cengage