Trabalho Prático 3

Fernando Eduardo Pinto Moreira - 2019054536

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

fernandoepm@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema de encriptação para a comunicação de civilizações, para que estas consigam se defender dos ataques do imperador Vader. Primeiramente, tinha-se que organizar as palavras do idioma de uma civilização, de modo que novas palavras pudessem ser inseridas e algumas substituída por outras, visto que, como explicitado, o idioma é bastante dinâmico.

Além disso, o programa criado também deveria ser capaz de cifrar e decifrar mensagens. O modo de encriptação usado pelas civilizações utiliza uma árvore binária de pesquisa, sendo que o caminhamento pré-ordem nessas árvores retorna a mensagem cifrada de acordo com a posição de cada palavra na árvore. Para decifrar, usa-se o mesmo método, buscando palavras na árvore que possuem o número de caminhamento pré-ordem enviado na mensagem cifrada.

2. Implementação

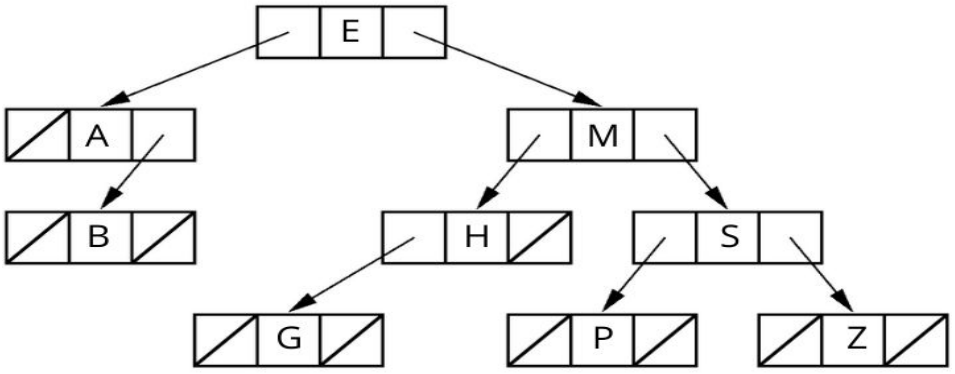
O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection. A entrada de dados é a padrão do sistema (stdin), assim como a saída também é a padrão do sistema (stdout).

2.1. Estrutura de Dados

A implementação do programa teve como base a Estrutura de Dados árvore binária de pesquisa. A escolha de tal estrutura se deu devido ao problema apresentado, que exigia uma organização em árvore binária para a inserção, substituição, encriptação e desencriptação de palavras.

Para implementar tal estrutura, utilizou-se a classe "ArvoreBinaria", que foi implementada nos arquivos "ArvoreBinaria.h" e "ArvoreBinaria.cpp", juntamente

com classes auxiliares que definem tipos (tipo de nó e tipo de item), estas implementadas nos arquivos "TipoItem.h", "TipoItem.cpp", "TipoNo.h" e "TipoNo.cpp". A estrutura da árvore binária pode ser representada pela figura abaixo:



A estrutura foi alterada para ordenar strings alfabeticamente ao invés de inteiros, sendo que é verificado caracter por caracter e a string que for alfabeticamente inferior a uma outra é inserida à esquerda da mesma, já se for superior, é inserida à direita. A árvore binária de pesquisa com caracteres é bem representada na figura acima, em que letras "maiores" entram à direita e letras "menores" entram à esquerda.

2.2. Classes

Foram utilizadas três classes para implementação, sendo elas: "TipoItem", "TipoNo" e "ArvoreBinaria". "TipoItem" e "TipoNo" são classes auxiliares utilizadas para implementar a classe "ArvoreBinaria". A classe "ArvoreBinaria" possui métodos para inserir, remover, pesquisar um elemento e percorrer a árvore.

Na classe "TipoNo" foi acrescentado um atributo "numEncrip" para armazenar o número de encriptação de uma palavra, de acordo com o seu caminhamento pré-ordem na árvore binária.

A declaração das classes foi feita em arquivos separados .h e a implementação em arquivos .cpp também separados, para aumentar a organização do código e para evitar possíveis erros. Todos esses arquivos estão na pasta headers.

2.3. Funcionamento geral

O programa inicia-se pedindo o número de operações que o usuário deseja fazer. Após ler o número de operações, lê-se do usuário qual dessas operações deseja realizar. Existem quatro tipos de operações: inserir nova palavra, substituir palavra por uma nova, encriptar uma mensagem e desencriptar uma mensagem. Porém, antes de realizar alguma operação, é chamada a função recursiva "PreOrdem" para atribuir o

número de encriptação de cada palavra inserida ou substituída. Isso é feito por um caminhamento pré-ordem na árvore e, para cada nó percorrido, atribui-se a sua posição. Caso o usuário deseje inserir nova palavra, deve digitar "i". Caso deseje substituir uma palavra por uma nova, deve digitar "s". Caso deseje encriptar uma mensagem, deve digitar "e", o número de palavras que deseja encriptar e cada uma dessas palavras em seguida. Caso deseja desencriptar uma mensagem, deve digitar "d", o número de palavras que deseja desencriptar e o código pré-ordem de cada uma dessas palavras em seguida.

Caso o usuário digite "i", lê-se então a palavra que ele deseja inserir e a insere pelo método "Insere" do objeto "idioma" criado. Este método chama o método "InsereRecursivo", que insere a palavra à esquerda do nó avaliado caso a palavra seja alfabeticamente inferior à palavra do nó avaliado e, caso seja alfabeticamente superior, insere à direita. Porém, como estamos trabalhando com strings e não com inteiros, o código padrão da árvore binária de pesquisa foi modificado para que a ordenação alfabética das strings fosse possível. Para isso, utilizou-se a função "compare", própria do C++, que compara se uma string é alfabeticamente superior a uma outra. Este método, além de ser usado na inserção de novos elementos, também foi utilizado na remoção e pesquisa.

Caso o usuário digite "s", deve-se então ler a palavra que se deseja retirar e a palavra que deseja inserir. Após isso, devemos inserir o novo elemento, e isso é feito pelo método "Insere" cujo funcionamento é o mesmo do explicado no parágrafo anterior. Após inserir, o método "Remove" da classe "ArvoreBinaria" é chamado para remover a palavra que deseja-se retirar. Este método chama o método recursivo "RemoveRecursivo" que, primeiramente, procura na árvore o elemento a ser removido (aqui é onde foi aplicado as chamadas recursivas, para achar o elemento procurado). Para remover, temos quatro possibilidades: o nó a ser removido não possui filhos, ou possui apenas um filho à esquerda, ou possui apenas um filho à direita ou possui dois filhos. Caso o nó não possua filhos, tudo o que se deve fazer é deletar o nó com o método "free()", sem modificações extras. Caso o nó possua 1 filho à esquerda, o nó pai do nó a ser deletado passa a apontar para o filho da esquerda do nó a ser deletado e, após isso, deleta-se o nó desejado. Caso o nó possua 1 filho à direita, o nó pai do nó a ser deletado passa a apontar para o filho da direita do nó a ser deletado e, após isso, deleta-se o nó desejado. Caso o nó possua dois filhos, ao remover esse nó temos que substituí-lo pelo elemento mais à direita da subárvore à esquerda. Isso é feito chamando o método "Antecessor", que substitui o nó a ser deletado pelo nó mais à direita da subárvore à esquerda e, após fazer isso, deleta o nó desejado.

Caso o usuário digite "e", deve-se então ler a quantidade de palavras que ele deseja encriptar. Após isso, lê-se do usuário todas as palavras que ele deseja encriptar. Após ler as palavras, chama-se o método "PesquisaNumEncrip" que, dado uma palavra, busca pelo respectivo número de encriptação resultante do caminhamento

pré-ordem. Este método chama o método recursivo "PesquisaNumEncripRecursivo" para efetivamente fazer a pesquisa da palavra na árvore. Ao achar a palavra correspondente à procurada na árvore, retorna o número de encriptação desta palavra. Fazemos este mesmo processo para todas as palavras digitadas pelo usuário, resultando, ao final, em um código encriptado.

Caso o usuário digite "d", lê-se então o número de palavras que o usuário deseja desencriptar e, com isso, lê-se todas as palavras que ele deseja desencriptar. Após isso, para de fato desencriptar as palavras, chama-se o método "PreOrdemDesencrip" em que, dado uma palavra em número (encriptada), retorna a palavra em texto (desencriptada). Este método é recursivo e faz um caminhamento pré-ordem na árvore procurando o elemento cujo número de encriptação é igual ao número de encriptação que estamos procurando. Caso seja igual, imprime a palavra (em texto) desse elemento. Repete-se então esse processo para todos os números de encriptação digitados pelo usuário, resultando em uma mensagem desencriptada.

O programa então se repete n vezes, sendo n o número de operações digitado pelo usuário inicialmente.

2.4. Instruções de compilação e execução

Para compilar o código, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
make
```

Para isto, você deve ter instalado o make em sua máquina. A instalação varia de cada sistema operacional, mas uma rápida pesquisa na internet resolverá.

Para executar o código individualmente (sem a realização da bateria de testes), acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
./tp3
```

Com esse comando, pode-se inserir manualmente uma entrada para o algoritmo.

Para executar o código individualmente baseado num arquivo de entrada, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
./tp3 < arquivodeentrada.extensao
```

Já para executar o código individualmente baseado num arquivo de entrada e inserir a resposta dada num arquivo de saída, acesse o diretório fernando_moreira/src via terminal e digite o comando:

```
./tp3 < arquivodeentrada.extensao > arquivodesaida.extensao
```

Atenção: o Makefile leva em consideração a sua execução num sistema operacional Linux. Para outros sistemas, como o Windows, adequações podem ser necessárias para a sua utilização.

3. Análise de Complexidade

3.1. Tempo

O programa começa com comandos de atribuição de variáveis, que possuem custo constante, ou seja, O(1). Logo após isso, lê a quantidade de operações desejadas, que também é O(1). Com este número é construído um "for" que itera de 0 até n, sendo n o número de operações digitado pelo usuário. Avalia-se então o custo dos comandos internos deste "for".

Este "for" começa chamando o método "PreOrdem". Este método é recursivo e possui complexidade O(n). Isso porque ele faz c operações, sendo c uma constante, além de chamar a função "PreOrdem" duas vezes para a metade dos elementos. Logo, sua equação de recorrência é:

$$T(n) = 2T(n/2) + c$$

Resolvendo pelo Teorema Mestre, conclui-se que essa equação cai no primeiro caso, tendo então complexidade igual a **O(n)**.

Após chamar a função "PreOrdem", o programa lê a operação desejada pelo usuário, com custo O(1). Caso seja "i", entra-se no primeiro "if". Este "if" chama o método "SetChave" da classe "TipoItem" que, como apenas atribui um valor, possui custo O(1) e, depois, chama o método "Insere" da classe "ArvoreBinaria". Este método chama o método recursivo "InsereRecursivo", que possui um "if" com apenas comandos de atribuição e, portanto, com complexidade O(1). Caso caia no "else", o método chama ele próprio duas vezes para a metade do vetor pela recursividade. Logo, o custo deste método pode ser calculado pela equação de recorrência:

$$T(n) = 2T(n/2) + 1$$

Resolvendo pelo Teorema Mestre, conclui-se que essa equação cai no primeiro caso, tendo então complexidade igual a **O(n)**.

Caso a operação digitada for "s", deve-se ler a palavra a ser retirada e a palavra a ser inserida, ambos com custo O(1), chamar o método "SetChave", com também custo O(1). Após isso, chama-se o método "Insere", que como explicado anteriormente possui custo O(n) e depois chama-se o método "Remove". Este chama o método recursivo "RemoveRecursivo", que chama este mesmo método recursivamente duas vezes para a metade do vetor. Após isso, o método precisa checar

se o elemento que se deseja remover possui um ou dois filhos, ou se não possui nenhum. Caso o elemento possua um filho, ele cai em um dos "if" que possui custo O(1). Já se o elemento possuir dois filhos, ele cai no "else" que chama a função "Antecessor". Esta função também é recursiva e pode ser representada pela equação de recorrência: T(n) = T(k) + 1, sendo k o altura do nó a ser removido até o elemento mais à direita da subárvore à esquerda desse elemento e 1 as outras operações de atribuição e remoção de elemento presentes na função "Antecessor". Logo, o custo desta função é da ordem de O(n). Assim, voltando à função "RemoveRecursivo" pode-se representá-la pela equação de recorrência:

$$T(n) = 2T(n/2) + n$$

Resolvendo pelo Teorema Mestre, caímos no caso 2, em que f(n) = n ^ (log2(2)), ou seja, n = **Θ**(n). Logo, a complexidade da função "RemoveRecursivo" e, portanto, do método "Remove" é igual a O(n.log(n)).

Com isso, conclui-se que a complexidade quando o programa executa para operação igual a "s" é igual a:

$$O(1) + O(n.\log(n)) = \mathbf{O(n.log(n))}$$

Caso a operação digitada for "e", deve-se ler o número de palavras, com custo constante, e executar um "for" que itera n vezes, sendo n o número de palavras digitado. Neste "for", primeiramente lê-se a palavra desejada, com custo constante e, após isso, chama-se a função "PesquisaNumEncrip". Esta função chama a função recursiva "PesquisaNumEncripRecursivo" que executa duas vezes para cada metade do vetor. Logo, o custo dessa função pode ser representado pela equação de recorrência:

$$T(n) = 2T(n/2) + 1$$

Resolvendo pelo Teorema Mestre, conclui-se que essa equação cai no primeiro caso, tendo então complexidade igual a O(n). Logo, quando o usuário digita "e", o custo é:

$$O(1) + O(n) = \mathbf{O(n)}$$

Por fim, quando o usuário digita "d", lê-se então o número de palavras que ele deseja desencriptar, com custo constante, e executa um "for" que itera n vezes, sendo n o número de palavras encriptadas digitadas. Neste "for", lê-se então cada palavra encriptada com custo constante e chama-se o método "PreOrdemDesencrip". Este método também é recursivo, o qual percorre toda a árvore binária (com n elementos) e imprime o item do elemento, com custo constante. Logo, o custo dessa função pode ser expressa pela equação:

$$T(n) = 2T(n/2) + 1$$

Este é o primeiro caso do Teorema Mestre, com custo igual a O(n). Logo, o custo quando o usuário digita "d" é igual a:

$$O(1) + O(n) = \mathbf{O(n)}$$

Assim, percebe-se que a complexidade total do programa é:

$$O(1) + n * (O(n) + O(n) + O(n.\log(n)) + O(n) + O(n)) = \mathbf{O(n^2.\log(n))}$$

3.2. Espaço

Percebe-se que utilizou-se apenas uma estrutura de dados no programa, que foi uma árvore binária de pesquisa instanciada com o objeto "idioma". Sendo assim, o usuário pode inserir n palavras nesta estrutura, o que precisará de um custo de espaço da ordem de O(n), sendo n o número de palavras digitadas pelo usuário que serão adicionadas na árvore binária de pesquisa. Logo, o custo de espaço é igual a: **O(n)**.

4. Conclusão

Ao final do programa, pôde-se perceber que a estrutura de dados árvore binária de pesquisa é muito útil para armazenar dados de forma eficiente, diminuindo custos de pesquisa, remoção e inserção ordenada.

Além disso, percebeu-se que ela permite, além das operações básicas, encriptação de mensagens de forma fácil e consistente, implementada pelo caminhamento pré-ordem que faz cada palavra ter um diferente identificador, o qual pode mudar ao inserir e remover palavras da árvore. Percebeu-se também que, além de números, também podemos armazenar strings e ordená-las alfabeticamente com funções próprias do C++, facilitando a implementação do código.

5. Bibliografia

CHAIMOWICZ, Luiz e PRATES, Raquel. Slides da disciplina Estrutura de Dados. Departamento de Ciência da Computação (DCC). Universidade Federal de Minas Gerais (UFMG). 2020.

Ziviani, N. (2006). Projetos de Algoritmos com Implementações em Java e C++: Capítulo 5: Pesquisa em Memória Primária. Editora Cengage.