Documentação do terceiro trabalho prático da disciplina de Estrutura de Dados

Lucas R. P. Machado[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

lucasresende@dcc.ufmg.br

Abstract. *This document has the objective of explaining how the software to solve the Morse problem proposed works and what was the mindset behind most of the code.*

Resumo. *Esse documento tem como objetivo explicar como o programa para resolver o problema de morse funciona e qual foi a ideia por trás de cada parte do programa.*

1. Introdução

O trabalho consiste em fazer um decodificador de códigos no formato de Morse utilizando uma árvore Trie para armazenar a "tradução" das sequencias de pontos e traços para o alfabeto alfanumérico.

Para isso devemos receber pela entrada padrão a sequencia dos códigos Morse sendo que letras são separadas por um espaço e palavras pelo caractere "/". Para guardar isso na árvore, devemos criar nós como em uma árvore binária, mas ao invés de comparar se um elemento é maior ou menor que o outro para decidir se o nó vai ser a direita ou a esquerda do nó atual, devemos ver qual o dígito na posição da chave igual à altura do nó atual na árvore. Caso seja um traço o nó será criado em um dos lado e caso seja um ponto o nó é criado no outro lado. Dessa forma teremos um jeito simples e eficiente de achar essa chave posteriormente visto que é necessário apenas que a cada nível da árvore escolhermos o caminho baseado no dígito à copiar a chave desejada.

2. Implementação

Para a resolução do problema foram criados dois arquivos, sendo eles o *main.cpp* e o *tree.hpp*.

O *main.cpp* tem como objetivo fazer a interface entre os comandos e entradas recebidas pelo programa e a conversão dos caracteres. Inicialmente é criado o objeto da árvore que logo em seguida é populado usado os valores encontrados no arquivo "*morse.txt*", que contém as definições das traduções de Morse para alfabeto. Logo em seguida é lido da entrada padrão todas as linhas e palavras a serem traduzidas, e para cada uma delas o método de tradução de uma letra na classe *Tree* é chamado e o resultado é impresso na tela. Por último, caso o modificado "*-a*" seja inserido na execução do programa, é chamado o método que imprime a árvore em pré-ordem.

Já no arquivo *tree.cpp* é onde está contida toda a lógica relacionada a árvore Trie. Inicialmente temos os métodos:

- *insert(string, string)* – Responsável por adicionar uma nova chave na árvore. O

funcionamente desse método é bem simples. Primeiro faremos uma busca na árvore seguindo os dígitos da chave desejada, caso no meio do caminho seja encontrado um nó nulo, se deve criar um nó e prosseguir até que todos os nós requeridos para guardar a chave inteira sejam criados. Caso já existam todos os nós necessários, é necessário apenas trocar o valor que o nó final guarda para o valor que queremos armazenar na árvore.

- *search(string)* – Método que retorna o valor do nó cuja chave seja igual à passada para essa função. O método funciona de forma bastante similar ao método de inserção de dados, porém caso encontre um nó nulo no meio do caminho o método vai indicar que a chave desejada não está na árvore e caso ache o nó correto o método vai retornar o valor daquele nó.
- *pre_order()* - Método que inicia a cadeia de recursões iniciando na raíz na árvore que gerarão o encaminhamento pré-ordem da mesma.

Foi tido o devido cuidado em relação à *memory leaks* tanto na classe da árvore (*Tree*) quanto na classe do nó *(Node),* liberando o espaço na memória préviamente ocupado pela raíz da árvore e dos nós do ponto e da linha respectivamente.

Quanto a leitura dos arquivos foi utilizado as estruturas "*ifstream" (*que transforma um arquivo em uma *stream* de dados para ser utilizada posteriormente em um *while* lendo cada bloco de texto) e "*istringstream" (*que transforma uma string em uma *stream* de dados para, da mesma forma que o *ifstream*, facilitar o acesso aos dados contidos nela). Também foi utilizado o método *"getline"* ao invés do tradicional "cin >>" para a leitura dos dados da entrada padrão visto que a separação entre as linhas é importante já que na saída esperada do programa, para cada linha de entrada tem que ser impressa um linha de saída, impossibilitando o uso da *stream* de dados do "*cin >>"* que separa a entrada por espaços também.

3. Instruções de compilação e execução

Para compilar o programa, deve-se executar no terminal na pasta raiz do projeto:

```
make clean
make
```

E então para executa-lo basta executar o comando:

```
./main [-a] <arquivo_de_entrada.in
```

Sendo que as opções possíveis para os parâmetros são:

- [-a]
  - Caso incluido no comando, será impressa ao final da saída do programa o encaminhamento pré-ordem da árvore criada.

4. Análise de complexidade

Temos dois aspectos principais para a análise de complexidade do programa: Inserção de novos dados e a recuperação de dados já inseridos.

4.1. Análise de complexidade da inserção

Como cada caractere em morse é definido por 1 a 5 dígitos (ou um ponto ou um traço) e cada um desses dígitos representa uma decisão de para onde descer um nível na

árvore, temos que o máximo de comparações que temos para inserir um novo caractere é o número de dígitos da sua representação em morse. Como um novo dígito só é inserido quando número de caracteres necessários é uma potência de 2, o número de comparações necessárias com um número de caracteres necessário n é $\log_2(n)$.

4.2. Análise de complexidade da busca

Da mesma forma que na inserção, como para chegar no nó com uma chave desejada é $\log_2(n)$, a busca, que faz exatamente isso, também terá complexidade $\log_2(n)$ dado que n é o número de caracteres em um código Morse.

5. Conclusão

As árvores de pesquisa são sem dúvida uma das melhores estruturas de dados possíveis quando temos dados como os do problema, imutáveis e de simples indexação, tendo uma complexidade extremamente baixa. Porém em casos mais complexos, onde os valores e posições de nossa árvore são mutáveis, por mais que árvores ainda são bastante eficientes, elas terão um custo consideravelmente maior para manter a árvore balanceada por exemplo. Outro problema que pode surgir dependendo do problema é a difícil indexação dessas chaves. Nesse problema foi extremamente simples colocar as chaves na posição correta visto que nossa chave era simplesmente uma cadeia de caracteres que possuiam apenas duas opções. Em problemas onde a chave poderia ser um objeto de uma classe por exemplo teriamos um problema bem mais desafiador.

Em geral foi um trabalho não tão difícil de implementar mas bastante interessante como instrumento de aprendizado e fixação do conteúdo, até porque esse é um conteúdo muito utilizado.

6. Referências bibliográficas

Halim, S., Halim, F. (2011) “Visualgo”, Acesso em Julho, 2019: https://visualgo.net/bn/sorting

Luiz A. de P. (2018) “Árvore Binária”, Acesso em Julho, 2019: http://www.ppgia.pucpr.br/~laplima/ensino/tap/contents/12_arvbin.html

Feofiloff, P. (2004) “Árvores binárias”, Acesso em Julho, 2019: https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html

Jain, S. “GeeksforGeeks”, Acesso em Julho, 2019: https://www.geeksforgeeks.org/

“Coding Freak”. “Trie Data Structure: Overview and Practice Problems”, Acesso em Julho, 2019: https://medium.com/@codingfreak/trie-data-structure-overview-and-practice-problems-2db1bad4bf49

Galles, D. “Trie Visualization”, Acesso em Julho, 2019: https://www.cs.usfca.edu/~galles/visualization/Trie.html