Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Trabalho Prático 3 - Decifrando os Segredos de Arendelle

Vinicius Julião Ramos
2018054630
viniciusjuliao@dcc.ufmg.br

7 de Julho de 2019

Resumo

Este trabalho consiste no desenvolvimento de uma ferramenta para traduzir os escritos do Serviço Secreto de Arendelle, que possui informações armazenadas em código morse e que necessitam de repasse para o novo sistema criptográfico do reino. Para isso, implementou-se uma árvore de pesquisa em que as chaves para montagem da árvore conssitem numa string que é responsável por um caracter morse.

1 Introdução

Dado um arquivo inicial contendo uma tabela de conversão da variante do código morse arendellense para os demais caracteres, fez-se uso da estrutura *Árvore de Pesquisa*, para efetuar o trabalho da tradução, que é a solução proposta para esta tarefa. Tal árvore, é uma implementação alternativa à uma *Trie Binária*[1], porém, com algumas modificações e melhorias, a fim de atender com mais facilidade, agilidade e persistência a solicitação do reino.

O programa foi implementado na linguagem *C++* que possui uma *Standard Library*[2] consistente e de fácil uso, que permitiu a boa implementação da leitura do arquivo com os parâmetros de tradução. Além disso, a linguagem escolhida, oferece um bom suporte ao tratamento de strings, que foi extremamente necessário para a montagem da estrutura de pesquisa, que navega entre os caracteres das chaves identificadoras. Para tanto, como pedido no enunciado deste trabalho, a cada caractere lido da string morse, navega-se na árvore binária de pesquisa para direita ou para a esquerda. Assim, espera-se que à medida que se reconhece-se os pontos ou traços da string, percorra-se estrutura de dados até que último caractere seja lido e encontre-se a tradução correta para tal código.

2 Implementação

No desenvolvimento deste trabalho, fez-se uso de algumas ferramentas que viabilizassem o desenvolvimento e aumentassem a produtividade de elaboração dos códigos, além da aplicação de conceitos de programação que tornassem o código mais legível e organizado. Para cargo de produtividade, utilizou-se o editor de texto Visual Studio Code [6], versão 1.34, que possui corretores com modo *autocomplete* diminuindo erros de digitação e sintaxe. Quanto ao quesito de organização e facilidade de implementação, a linguagem escolhida foi C++ [8] que possui uma biblioteca padrão consistente, a qual se incumbiu em facilitar o processo do tratamento de strings na entrada dos dados, e na montagem a árvore de pesquisa.

2.1 Organização do código

Visto que *C++* permite o uso do conceito *Orientação a Objetos*, criou-se duas classes responsáveis pela montagem da estrutura da arvore e pela armazenagem dos dados recebidos da entrada. Além disso, criou-se uma estrutura para armazenar a árvore de pesquisa, em que é registrado o nó de uma árvore binária, que consiste em um registro principal, um apontador para o subnó esquerdo e outro subnó direito.

Tal estruturação foi inspirada no *TAD* **map**[7] da biblioteca padrão da linguagem *C++*, uma vez que essa estrutura é implementada através de uma árvore em que seus nós possuem um outro *TAD*, do tipo **pair**, composto por um chave e o registro principal. Assim, garante-se que a solução dada para esse trabalho segue os mesmos parâmetros de boas práticas usados na *Standard Library C++*. Vale ressaltar que nem todas as características do **map** foram adotadas nesse programa, visto que necessita-se tratar apenas uma chave do tipo **string** e um dado do tipo char, escolheu-se por não utilizar o conceito de **template**, como a *TAD* citada.

Sendo assim, a descrição das classes e estruturas implementadas, além do relatório sobre a leitura das entradas do trabalho, podem ser descritas da seguinte maneira:

2.1.1 BinaryTree

Implementação de uma árvore de pesquisa digital em que os dígitos que definem o encaminhamento da árvore. Tais dígitos, correspondem aos caracteres do código morse, ponto *(.)* e traço *(-)* e indicam caminho da esquerda e da direita, respectivamente, e quaisquer outros caracateres encontrados no código morse provocarão um erro de execução no programa. Sendo assim, é possível inferir que o nó raiz sempre será um nó vazio.

Essa classe é inspirada na criação de uma *TRIE*, porém possui a melhoria de não armazenar os dados apenas nos nós folha da árvore, mas também, nos nós internos. Assim há uma considerável economia na utilização do espaço, fazendo com que essa seja uma alternativa mais barata às implementações já conhecidas até então.

Logo, no desenvolvimento dos métodos que caracterizam o funcionamento dessa estrutura, necessitou-se do uso de alguns método privados que possuem grande semelhança com os métodos públicos. Porém o tratamento das diferenças entre os métodos públicos dos métodos privados semelhantes, traria um aumento no custo de comparações, o que piora desempenho do programa.

void add(const Pair &p)

Recebe um instância de **Pair** para adicionar à arvore de acordo com a chave especificada dentro desse objeto.

Executa o reconhecimento do primeiro caractere da chave de **Pair** para identificar se a adição será feita na subávore da esquerda ou da direita, e então chama o método **_add()**.

char find(std::string key)

Recebe a chave, em código morse, correspondente ao registro que deseja-se encontrar na árvore.

Executa, a partir da raiz da árvore, a diferenciação entre o primeiro caractere do código morse e realiza chamadas à função recursiva **_find()** para que esta encontre o registro correto.

Retorna o registro (caractere) correspondente à **string** morse, ou então NULL caso não encontre.

void print()

Executa a impressão da árvore em organização *pre order* através da chamada à função **_print()**.

char _find(std::string key, Node *node, int string_position)

Recebe o código morse do caractere, a raiz da subárvore atual e o índice correspondente à qual caractere da string **key** será lido.
Executa chamadas auto recursivas até que o a leitura de **key** chegue ao final ou que se depare com um nó nulo,
Retorna o caractere correspondente a **key** ou NULL caso não encontre.

void _add(const Pair &p, Node *sub_root, int string_position)
Recebe o registro a ser adicionado, a raiz da subárvore atual e o índice correspondente à qual caractere da string **key** será lido.
Executa chamadas auto recursivas, aumentando o nível de percurso da árvore em 1 a cada nova recursão, até que encontre a posição adequada para alocar o nó.

void delete_node_pos_order(Node *node)
Recebe a raíz da subárvore que passará pelo processo de desalocação da memória.
Executa chamadas auto recursivas para os filhos do nó atual, até que ambas as subárvores sejam totalmente liberadas da memória. Então executa a liberação da memória para o nó atual.

void _print(Node *node)
Recebe o nó da árvore pai o qual o programa imprimirá.
Executa chamadas auto recursivas para que os nós sejam impressos em caminhamento *pré order* e imprime a cada passagem pela função.

2.1.2 Pair

Essa classe se encarrega do trabalho de armazenar os dados da entrada do usuário. A sua criação é inspirada na *TAD* **pair** da biblioteca padrão de *C++*. Assim, trata-se apenas de uma estrutura de armazenagem de uma chave to tipo **string** e o dado do tipo **char**, que contém métodos **get** e **set**.

2.1.3 Node

Estrutura principal da montagem de uma árvore binária. Caracteriza-se pela armazenagem de dois apontadores do tipo **Node** que identificam as subárvores da esquerda e da direita de determinado nó. Além disso há um registro do tipo **Pair** que identifica o dado principal que cada nó guarda, assim como numa estrutura da **map**.

2.1.4 main.cpp

Na parte de execução principal do programa, resolveu-se por evitar a modularização do código a fim de poupar o uso de memória. Seria possível a implementação de funções que realizassem a tarefa da leitura da tabela tradutora no arquivo **morse.txt**, e de funções que realizassem a leitura de uma frase em código morse desde que pudesse chamar outra função que realizasse a montagem de uma palavra a partir da entrada padrão.

Entretanto, tal modularização necessitaria da utilização de uma **string** para armazenar cada frase, antes de imprimí-la e isso traria o aumento no custo de execução do programa. A manipulação de adição de caracteres em uma string possui o mesmo custo de execução da manipulação de um vetor de algum outro tipo. Sendo assim, evitou-se esse gasto a mais nesse trabalho. Portanto, todo o código está estruturado apenas dentro do método **main()**.

O método de execução principal é responsável, ainda, pela leitura de parâmetros de execução. O único parâmetro solicitado nesta prática, foi uma *flag* que solicita a impressão da árvore binária em pré-ordem. Vale ressaltar que há um tratamento de erros de execução, para o caso de erros no arquivo **morse.txt** e na inserção da entrada padrão.

2.2 Decisões de implementação

Dada a necessidade da criação de uma árvore de pesquisa digital averiguou-se a possibilidade de desenvolver uma *Trie* ou então uma *Patricia*[3]. Porém esses dois tipos de árvores possuem a característica de utilizar apenas os nós folha para armazenar os registros [4], sendo uma *Trie* os nós internos guardam apenas dos apontadores das subárvores, sendo um *Patricia* os nós internos reservam-se para o registro do índice do dígito

comparado em determinado nível da árvore. Sendo assim, estas árvores têm um alto custo de inserção de um novo registro, em que se é necessário realizar trocas de ponteiros a cada inserção, se esse procedimento se der de forma não ordenada.

Portanto, a melhor alternativa foi implementar uma árvore binária digital de forma simples e concisa. No i-ésimo nível da árvore, lê-se o $(i-1)$-ésimo caractere do código morse, e tendo em vista que a quantidade de caracteres em um elemento morse é variável, no i-ésimo nível da árvore encontram-se os registros do tipo **Pair** em que os códigos de suas chaves têm $i-1$ caracteres em sua composição.

2.3 Entrada e Saída de dados

Como dado na proposta desse trabalho, há duas formas de ler os dados da entrada do programa; um arquivo **morse.txt** contendo uma tabela que referencia a tradução do código morse e a entrada padrão do sistema. Para que a execução ocorra de forma perfeita um padrão para o arquivo da tabela de tradução deve ser adotado e também para a entrada padrão dos dados.

O padrão registrado para o arquivo **morse.txt** convenciona que em cada linha do arquivo haja o caractere em ASCII separado por um espaço da respectiva representação do seu código. Vale reforçar que o ultimo caractere de cada linha deve ser um caractere do código morse, não existindo caracteres espaço no final das linhas. Esse padrão deve ser adotado em todas as linhas do arquivo, e não deve existir linhas em branco. O exemplo da informação contida no arquivo pode sere exemplificada por:

```
A .-
B -...
C -.-.
D -
E .
```

A entrada padrão receberá as frases que serão traduzidas pelo programa, e aqui também há um padrão que deve-se seguir para o sucesso na execução do programa. Baseando nos arquivos de teste de entrada disponibilizados para este trabalho, as frases em morse serão compostas da seguinte maneira: o início de cada linha da entrada deve ser um código morse, para separar um código de outro deve-se usar apenas 1 caractere de espaço, sendo que o último código que compõe um palavra em ASCII não deve ser seguido do separador de códigos. Para diferenciar palavras em uma mesma frase, há uma expressão composta por um espaço, uma barra e outro espaço (exatamente nesta ordem) "/ ". Ressalta-se que ao final de cada linha da entrada padrão não deve existir nem um separador de código, nem um separador de palavra; o último caractere de uma linha deve ser tão somente um caractere que compõe uma representação morse.

Também é importante ressaltar, que caso seja registrado na entrada padrão uma linha vazia o programa entenderá como fim de execução. O exemplo dado nos casos de teste, chamado de **2.in** pode ser mostrado seguir:

```
.-.. ..- ...- .-. . / . ... -.. ---- ..
```

À medida que se finaliza a digitação em uma linha, o programa imprimirá a resposta na saída padrão. E assim se dá a interação com o usuário deste programa.

3 Instruções de Compilação e Execução

Após a conclusão do trabalho, deve-se de instruir os demais usuários, sobre a forma como executar o programa e as especificações de execução e características que o software possui. Sendo assim vale lembrar que o programa foi desenvolvido e compilador utilizando **gcc** e **Makefile**, por isso, usando o terminal *bash* e abra o diretório raiz dos códigos (o mesmo diretório que possui o arquivo *main.cpp*), na sequencia execute o comando que compilará os códigos:

```
$ make
```

Um arquivo executável chamado *executavel* será criado e através deste arquivo, se dará a execução da solução. Porém antes de executar o arquivo binário, deve-se garantir que o arquivo **morse.txt** esteja no mesmo diretório de **executavel**, e assim tem-se duas maneiras de executar o programa

```
$ ./executavel
```

ou

```
$ ./executavel -a
```

Ambas as execuções serão organizadas executando a leitura do arquivo **morse.txt** lerão da entrada padrão os as frases que o usuário deseja tradurir de morse para ASCII. Porém ao executar à segunda maneira, ao final da execução o programa imprimirá na saída padrão o encaminhamento pré-ordem da árvore montada.

Uma terceira maneira de executar o programa é fazendo com que a entrada padrão seja recebida de um arquivo e a saída padrão seja enviada para um outro arquivo também. Suponha que o arquivo contendo a entrada desejada chame **entrada.in** e o arquivo de redirecionamento do programa se chame **saida.out**. Então pode-se concluir essa terceira maneira de execução com o seguinte comando:

```
$ ./executavel -a < entrada.in > saida.out
```

4 Análise de Complexidade

Dada a implementação de uma árvore binária de pesquisa, espera-se que os custos de execução e de memória seguem a ordem padrão de uma arvore binária qualquer. Porém este trabalho, não trata de uma árvore de pesquisa tradicional, mas sim de uma estrutura de pesquisa baseada na quantidade de dígitos que as chaves indexadoras de registro possuem. Sendo assim, considera-se uma variável n que representa a quantidade de caracteres na chave indexadora dos registro e a análise será feita tomando essa variável como base.

Apesar de um árvore binária possuir um custo $\log m$ para encontrar um registro, no caso dessa árvore de pesquisa digital, é necessário analisar todos os caracteres da string, independente da posição que estes se encontrem na árvore. Assim, partindo do principio que sempre se pesquise por chaves válidas, o melhor caso ocorre quando a chave possui tamanho 1, e seu custo é $f(n) = 1$ e por consequência a ordem de complexidade é $O(n)$. Já quando de seja-se analisar o pior caso, deve-se saber qual é a chave que possui o maior tamanho em quantidade de símbolos, nesse caso, ao "descer"em cada nível da árvore, compara-se um caractere a mais até que se chegue na folha de maior profundidade em relação à raiz. Para este ultimo caso, o custo de se chegar até a folha mais inferior, é o custo de comparar cada caractere da chave e de realizar a troca de contexto entre o nó atual e o nó seguinte, dando assim ordem de complexidade $O(n)$.

Quanto ao custo de memória é necessário analisar a quantidade de folhas que a árvore possui e a quantidade de caracteres que as chaves indexadoras dos nós folha possuem. O algoritmo construído para essa solução, assim como pede a descrição do trabalho, faz com que os caracteres sejam lidos a medida que se avança nos níveis da estrutura, ou seja, se há um registro com chave de tamanho 5, este estará no 6^o nível da árvore, e se este for o primeiro nó a ser inserido, um caminho composto por nós pai será criado, sendo que esses nós pai possuem o valor da estrutura **Pair** nulo, a fim de evitar o preenchimento desnecessário da memória.

Após tal descrição algorítmica, pode-se analisar os custos de melhor e pior caso para a árvore: O melhor caso ocorre quando as chaves são as menores possíveis, no quesito quantidade de símbolos, assim não haverá a montagem de nós pai vazios, e a memória será totalmente aproveitada. Nesse caso, a quantidade de memória utilizada é $O(k)$, em que k representa o número de códigos de tradução, que é sinônimo da quantidade de linhas do arquivo **morse.txt**. Para o pior caso, deve-se fazer algumas suposições: Seja k a quantidade de caracteres da chave de maior tamanho na árvore. Então, se há exatamente 2^{k-1} registros com chaves de tamanho k, têm-se a montagem de uma árvore em que seus níveis são completos, ou seja, o i-ésimo nível possui 2^{i-1} nós. Assim, para o pior caso há um custo que baseia-se num somatório: $f(k) = \sum_{i=1}^{k} 2^i$

5 Conclusão

Por fim, mostra-se que tal implementação é capaz de realizar a tarefa com efetividade e precisão, mostrando-se uma boa solução para o problema. Porém há formas menos custosas de desempenhar a mesma tarefa.

Na parte do custo de memória, por conta da restrição na descrição desse trabalho, que solicita a leitura dos caracteres a medida que se avança nos níveis da árvore, o custo de memória ficou extremamente volátil, cedendo a grandes variações de acordo com o tamanho das chaves indexadoras dos registros. Uma outra solução poderia se dar, baseando-se em uma árvore *Patricia*. O que aumentaria o custo de execução na inserção dos elementos, visto que necessitaria de uma manipulação dos ponteiros, mas traria o benefício de um custo fixo de memória, em que o custo seria $O(n)$ sendo n a quantidade de elementos inseridos na árvore.

Vale reforçar, ainda, que a implementação citada acima é benéfica em todos os aspectos, pois a montagem da árvore ocorre apenas no início do programa. Sendo assim, não haveria necessidade de trocar o contexto da árvore e por isso o aumento no custo de inserção não traria grandes malefícios.

6 Referências

Referências

[1] Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Pioneira Thomson Learning, 2004, segunda edição. (4th Edition, p. 128 - 129)

[2] https://pt.cppreference.com/w/cpp/header Acessado em: 06 jul. 2019.

[3] Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Pioneira Thomson Learning, 2004, segunda edição. (4th Edition, p. 129 - 135)

[4] Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Pioneira Thomson Learning, 2004, segunda edição. (4th Edition, p. 128)

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd Edition, p. 145 - 152). McGraw-Hill Higher Education.

[6] https://code.visualstudio.com 09 jun. 2019

[7] http://www.cplusplus.com/reference/map/map/ Acessado em: 06 jul. 2019.

[8] http://www.cplusplus.com Acessado em: 06 jul. 2019