Trabalho prático 3 - ED
Decifrando os segredos de Arendelle

Ábner de Marcos Neves

2018054605

Universidade Federal de Minas Gerais
Belo Horizonte – MG – Brasil

abnerneves@ufmg.br

1. Introdução

Uma árvore Trie binária (nome oriundo de *re**trie**val*) é uma estrutura de dados que define o local de inserção das chaves de acordo com a representação binária delas. Cada nó possui duas ramificações: a da esquerda contém aqueles cujo bit mais à esquerda é 0 e a da direita, aqueles com bit mais à esquerda igual a 1. Cada nível abaixo avança um bit. Dessa forma, todas as chaves terminam por serem armazenadas apenas nas folhas da árvore e suas representações binárias definem o caminho até cada uma.

A pesquisa digital é feita sobre essa estrutura de dados e consiste em, dada uma representação binária, percorrer o caminho indicado por ela até a chave de interesse.

Neste trabalho, esses recursos foram utilizados para decodificar mensagens em código morse. Por isso, em vez de 0 e 1, as representações das chaves são dadas em pontos e traços (. e -) e cada caminho leva a uma letra ou número.

Cada sequência de pontos e traços representativa de uma letra ou número é separada por um espaço, enquanto que cada palavra é separada por uma barra.

2. Implementação

A árvore de decodificação se estrutura em nós com dois ponteiros para outros nós e atributos para guardar uma letra e sua representação em morse. Além disso, a classe declarada para a árvore possui um método para inserir elementos, pesquisar e imprimi-los em pré-ordem.

2.1. Inserção

O método responsável pela inserção de elementos recebe por parâmetro um caracter e sua representação em código morse. A partir disso, mantém uma variável para contar quantos símbolos da representação já foram avaliados.

Se a chave inteira já foi avaliada, quer dizer que a função percorreu o caminho inteiro indicado pelo código morse e já chegou à folha em que o caracter deve ser

inserido. Caso contrário, uma chamada recursiva da função é feita para a subárvore da esquerda ou da direita, se o i-ésimo símbolo da chave for um ponto ou um traço, respectivamente. No caminho percorrido, são sempre criados novos nós, caso o adequado a seguir ainda não exista.

2.2. Pesquisa

Para a pesquisa, também é mantido um contador dos símbolos já avaliados e, quando ele se iguala ao tamanho da chave, a função verifica se a chave buscada é igual à encontrada. Caso positivo, retorna o nó em questão. Caso negativo, retorna *nullptr*.

Se ainda tem símbolos a analisar, uma chamada recursiva da função é feita para a subárvore da esquerda ou da direita, se o próximo símbolo for um ponto ou um traço, respectivamente.

2.3. Impressão

A impressão da árvore é em pré-ordem, isto é, imprime primeiro o elemento do nó e, depois, os dos seus filhos. Para isso, utiliza-se de uma função, que, primeiro, imprime o caracter e a chave do nó em questão, se ele contiver mesmo tais elementos e não for um nó vazio, e, depois, faz uma chamada recursiva para o nó da esquerda e para o da direita.

2.4. O programa

Para povoar uma instância da árvore descrita acima, o programa lê de um arquivo *morse.txt* os caracteres e suas respectivas chaves em código morse e insere-os todos no objeto.

Em seguida, lê da entrada padrão, linha por linha, as sequências de códigos a serem decodificadas. A cada código lido, é feita uma pesquisa na árvore e exibido o caractere correspondente. Um espaço é impresso se uma barra é encontrada.

No final, toda a árvore de decodificação é impressa em pré-ordem, caso o programa tenha sido executado com o parâmetro *-a*.

3. Instruções de compilação e execução

O algoritmo foi feito em C++ e possui um *Makefile* para compilá-lo com o *g++*. Para isso, basta navegar pelo terminal até a pasta que contém os arquivos e digitar *make*. Terminado o processo, o programa já pode ser executado digitando *./main* e, opcionalmente, direcionadores para a entrada e saída padrão. Também é opcional que o parâmetro *-a* seja usado: se estiver presente, a árvore de decodificação será impressa em pré-ordem depois das mensagens decodificadas. Para isso, basta rodar o programa com *./main -a*.

4. Análise de complexidade

As codificações em *morse* desse trabalho têm 5 bits ou menos. Por não terem o mesmo tamanho, nem todos os elementos ficarão em folhas. Essa característica reduz drasticamente o tamanho da árvore ao aproveitar os nós intermediários e economiza memória.

As codificações com 1, 2 e 3 bits foram todas utilizadas ao limite, o que quer dizer que os nós até o nível 3 estão todos ocupados. Das 16 possíveis combinações de pontos e traços de tamanho 4, 12 foram utilizadas, então, no nível 4 da árvore, existem 12 nós ocupados. As chaves com 5 bits são 10, então, no nível 5, existem 10 nós ocupados.

Assim, nota-se que a memória ocupada pela árvore depende não apenas da quantidade de elementos, como também das chaves que eles possuem. Na melhor das hipóteses, todas as combinações possíveis de k símbolos (pontos ou traços) são utilizadas antes de ter alguma combinação com $k+1$ símbolos. Dessa forma, no melhor caso, a complexidade de memória é

$$M(n,\ k)\ =\ \sum_{i=1}^{k} 2^i\ \geq\ n\ =\ O(n).$$

O pior caso ocorre quando nenhuma chave de tamanho k é utilizada antes de passar para chaves de tamanho $k+1$. Dessa forma, todas as chaves têm o mesmo tamanho k tal que $2^k \geq n$ e todos os elementos estarão nas folhas da árvore. Portanto,

$$M(n,\ k)\ =\ \sum_{i=1}^{k} 2^i\ =\ O(n)$$

Apesar de terem a mesma função, as duas condições de parada do somatório são diferentes: no melhor caso, a condição de parada é até que *o somatório* seja maior ou igual a n; no pior caso, a parada se dá quando $2^k \geq n$, então a memória gasta pelo pior caso é bem maior pois k precisa avançar mais.

Agora, tanto para a inserção quanto para busca, as complexidades são as mesmas. O melhor caso ocorre quando a chave procurada tem tamanho 1 e é necessário fazer apenas uma comparação. Assim,

$$C(k)\ =\ 1\ =\ O(1).$$

O pior caso ocorre quando a chave tem o maior tamanho e está no último nível da árvore, sendo necessário fazer k comparações. Portanto,

$$C(k)\ =\ k\ =\ O(k).$$

O caso médio da busca e inserção, apesar de não ser demonstrado aqui, tem complexidade $O(log\ n)$ quando se trata de chaves aleatórias[1].

---

[1] https://virtual.ufmg.br/20191/pluginfile.php/341187/mod_resource/content/1/Aula%2016%20-%20Pesquisa%20Digital.pdf

5. Conclusão

Uma árvore trie pode ser muito boa para situações como essa. O problema de consumo de memória desnecessária pode ser resolvido utilizando chaves mais bem distribuídas, como explicado acima, mas também existem variações dessa estrutura de dados que solucionam essa questão, como a PATRICIA[2]. Além disso, a ordem de inserção dos elementos não influencia na estruturação final da árvore: ela depende apenas das chaves. A melhor utilização da árvore *trie* e da pesquisa digital, portanto, depende de um bom sistema de chaves. Mas, ainda que sejam chaves aleatórias, o caso médio também é bom.

6. Referências

ZIVIANI, N., *Projeto de Algoritmos com Implementações em Pascal e C*. Cengage Learning. 2011.

[2] https://pt.wikipedia.org/wiki/%C3%81rvore_Patrica