Universidade Federal do Rio Grande do Norte
Estruturas de Dados e Algoritmos
Programa de Pós-Graduação em Sistemas e Computação - PPgSC
Departamento de Informática e Matemática Aplicada - DIMAp

Projeto 1 - Árvores Rubro-Negras

Valmiro Ribeiro da Silva

Natal-RN

2017.1

Sumário

1 Introdução e Fundamenteção Teórica p. 3

2 Implementação p. 4

2.1 Linguagem e Operações Básicas . . . . . p. 4

2.2 RBTree::Black-Height . . . . . p. 4

2.3 RBTree::Check . . . . . p. 5

2.4 RBTree::Print . . . . . p. 5

2.5 Função Main . . . . . p. 6

3 Instruções de compilação p. 7

Referências p. 8

1 Introdução e Fundamenteção Teórica

Árvores Binárias de Busca são estruturas de dados que suportam as operações básicas de inserção, remoção e busca com complexidade $\Theta(h)$, onde h é a altura dessa árvores. Apesar dessas operações serem rápidas quando a altura da árvore é grande, as operações citadas podem levar o mesmo tempo que a busca em em listas encadeadas, onde no pior caso temos que percorrer toda a lista para encontrar um elemento, que é $O(n)$. Sendo assim, as Àrvores Rubro-Negras (ARN) são uma variação balanceada das Àrvores Binárias de Busca criada para garantir que as operações básicas executem com tempo $O(lgn)$.

Esse trabalho segue a implementação proposta no livro base da disciplina - Introduction to Algorithms - com algumas modificações. A principal modificação é a não utilização dos nós sentinelas, o que deixa a implementação um pouco mais próxima das Árvores Binárias de Busca comuns, pois é preciso lidar com nós nulos. Outras modificações menores incluem a impressão de informação sobre os dados processados pelos diferentes algoritmos. Além dos algoritmos necessários para a implementação padrão das ARN, foram solicitados outros algoritmos para processar os dados inseridos, e alguns desses algoritmos precisaram que algoritmos auxiliares fossem implementados.

As próximas seções detalharão a implementação, bem como os algoritmos utilizados e como compilar e rodar o programa criado.

2 Implementação

2.1 Linguagem e Operações Básicas

Para a implementação do projeto foi escolhida a linguagem C++, tanto pela familiaridade utilizando-a quanto pelas suas característica únicas, como a utilização de ponteiros e orientação a objetos. Com a linguagem escolhida algumas decisões precisaram ser tomadas. A primeira delas foi utilizar um *struct* para representar os nós da árvores, pois apesar de poder utilizar uma classe, como não haviam outras operações especificas dos nós a estrutura bastava para implementar o projeto. A Árvore Rubro-Negra, por outro lado, utiliza uma classe para representar a sua estrutura, e contém um único atributo, que é um ponteiro para a raiz da árvore. Como todas as operações requisitadas na especificação precisavam que fosse passada uma referência para a ARN que estamos executando as operações, implementar a estrutura da ARN com classes eliminou essa necessidade.

A maioria dos algoritmos solicitados na especificação do projeto - RBTree::Insert, RBTree::Delete, RBTree::Search, RBTree::Minimum, RBTree::Right-Rotate, RBTree::Left-Rotate, RBTree::Insert-Fixup, RBTree::Delete-Fixup e RBTree::Transplant - foram implementados seguindo os pseudocódigos presentes nos capítulos 12 e 13 de (CORMEN, 2009). Os algoritmos restantes solicitados na especificação e seus auxiliares serão discutidos nas seções a seguir.

2.2 RBTree::Black-Height

O método $RBTree :: BlackHeight(node * y)$ que recebe como argumentoo nó que queremos saber a profundidade negra na ARN e, assumindo que a propriedade 5 das ARN que diz que para cada nó, todos os caminhos simples do nó até as suas folhas descendentes possuem o mesmo número de nós pretos (CORMEN, 2009), percorre a sub-árvore y até seu nó folha mais a esquerda contando o número de nós pretos, e retorna esse valor.

Como as Árvores Rubro-Negras são balanceadas, $RBTree :: BlackHeight$ vai percorrer um número de nós igual a altura da árvore, que é $\Theta(lgn)$, como demonstrado em (CORMEN, 2009).

2.3 RBTree::Check

O método $RBTree :: Check()$, que não recebe argumentos, é apenas um método público responsável por chamar o método protegido $RBTree :: checktree(node*y)$ passando a raiz da árvore como argumento. $RBTree : checktree$ acessa o nó passado como argumento e imprime as suas informações tal qual foi solicitado na especificação, e é chamado recursivamente para as sub-árvores da esquerda e da direita, respectivamente, fazendo um percurso em pré-ordem, como descrito em (CORMEN, 2009). Caso o nó passado como argumento seja nulo, o método encerra a sua execução.

Como o algoritmo $RBTree :: Check$ sempre vai passar a raiz da árvore como argumento, $RBTree :: checktree$ vai sempre executar $\Theta(n)$ vezes, onde n é o número de nós da árvore.

2.4 RBTree::Print

Assim como $RBTree :: Check$, esse método que não recebe argumentos é apenas responsável por chamar o método $RBTree :: inorder(node * y$, passando a raiz da árvore como parâmetro. $RBTree :: inorder(node * y)$ então é chamado recursivamente para a sub-árvore da esquerda, chama $RBTree :: Black_Height$ para atribuir o valor correto da profundidade negra do nó, imprime a informação armazenada em y e é chamado recursivamente para a sub-árvore da direita.

Assim como $RBTree :: check$, $RBTree :: Print$ é limitado pelo número de execuções de $RBTree :: inorder$, que por sua vez é limitado por $RBTree :: Black_Height$. Como a raiz é sempre passada como argumento dentro de $RBTree :: Print$, $RBTree :: inorder$ vai sempre executar $\Theta(nlnn)$ vezes, onde n é o número de nós da árvore.

Como $RBTree :: BlackHeight$ é chamado por $RBTree :: inorder$ e $RBTree :: Print$ sempre é chamado antes de $RBTree :: Check$, $RBTree :: Check$ sempre vai imprimir os valores corretos das profundidades negras dos nós.

2.5 Função Main

A Função main é responsável por receber e processar o arquivo de entrada contendo as palavras que devem ser removidas e/ou inseridas na árvore. O arquivo *.txt* é lido utilizando a bilioteca $fstream$ do $C++$, e um loop *while* que verifica se todas as linhas do aruqivo já foram lidas vai atribuindo o conteúdo de cada linha a uma variável *str*. O conteúdo de *str* é então processado utilizando a biblioteca *string* do $C++$, que desde a versão 11 possui algumas funções que facilitaram o processamento. Primeiro, a função *compare* é chamada, que recebe como argumento a string que queremos comparar com o último caractere e *str*. Comparamos esse caractere com "0", e caso sejam o mesmo, removemos o número e o espaço de *str* utilizando a função pop_back e tentamos remover a palavra restante. Caso contrário, comparamos o caractere com "1", e se forem iguais, removemos os dois últimos caracteres de *str* utilizando pop_back e tentamos inserir a palavra na árvore.

3 Instruções de compilação

Como citado na seção 2.5, o programa faz uso de funções que só estão presentes a partir da versão 11 do $C++$, e por isso, em alguns computadores, é necessário compilar o programa utilizando especificando qual versão do $C++$ estamos utilizando. O comando abaixo deve ser utilizado para compilar o arquivo *RBTree.cpp*:

g++ -std=c++11 RBTree.cpp -o RBTree

Para rodar o programa, basta executar o seguitne comando:

./RBTree entrada.txt

Para imprimir a saida em um arquivo *.txt*, basta executar o comando:

./RBTree entrada.txt > saida.txt

Referências

CORMEN, T. H. *Introduction to algorithms*. [S.l.]: MIT press, 2009.