Gleydvan Macedo, João Vítor Venceslau Coelho, Josivan Medeiros da Silva Gois

Implementação de Uma Árvore de Busca Binária com Operações Adicionais Otimizadas

Relatório referente ao trabalho da segunda unidade da disciplina de Estruturas de Dados Básicas.

Universidade Federal do Rio Grande do Norte – UFRN
Instituto Metrópole Digital – IMD
Bacharelado em Tecnologia da Informação

Docente: Silvia Maria Diniz Monteiro Maia

Brasil
27 de maio de 2018

Sumário

Introdução

Este documento apresenta uma discussão sobre a classe de Árvore Binária de Busca(ABB) implementada para o trabalho da segunda unidade da disciplina de Estruturas de Dados Básicas. Uma ABB implementa operações de busca, inserção e remoção. Além dessas operações básicas a classe criada tem os métodos de acessar o enésimo elemento em ordem simétrica, retornar o índice de um elemento, determinar mediana, se a árvore é cheia, se é completa e uma função para retornar uma representação por nível da árvore em uma string.

O objetivo foi de melhorar o desempenho da estrutura para a execução dessas operações adicionando, quando necessário, atributos extras. A seguir estão listadas os métodos criados e suas respectivas complexidades.

1 Metodologia

A linguagem utilizada foi C++. O tipo dos dados armazenados na estrutura é o tipo primitivo *int*. A implementação da classe ABB (Árvore de Busca Binária) foi feita em um arquivo `.cpp` e a declaração do cabeçalho dos métodos, bem como os atributos da classe, foram feitos em um arquivo `.hpp`. O compilador utilizado foi o g++ 5.4.0. Além dos arquivos da classe em si, foram criados casos de teste para verificar o funcionamento da estrutura de dados criada.

Na implementação algumas alterações foram feitas, tanto na *class* ABB quanto na *struct* como pode ser visto no Apêndice A e no Apêndice B.

2 Análise de Complexidade

Nenhuma das complexidades chegou a ser maior que $O(n)$, como esperado. Em alguns casos não foi possível determinar um $\Theta(n)$ pois a complexidade de alguns métodos dependiam da construção da árvore.

2.1 Métodos de Complexidade Constante

Esses métodos são mais simples de analisar uma vez que suas instruções geralmente não envolvem *loops*. A maioria dos métodos de complexidade constante da classe fazem acesso ou modificação de campos da classe (*getters* e *setters*).

Alguns métodos tem como finalidade acessar atributos de forma indireta, possibilitando tratamento de erros e garantindo que a instrução não altere o atributo. Esses métodos em Programação Orientada a Objetos (POO) são chamados de getters. Os getters da classe são:

getSize

Retorna a quantidade de nós na árvore.

getRoot

Retorna um ponteiro para o *node* raiz.

getHeight

Retorna a altura máxima da árvore.

Outros métodos constantes possuem uma quantidade de instruções constante. Alguns métodos necessitam fazer operação de potenciação, o que poderia aumentar a complexidade, porém como as operações são feitas com potências de 2 o problema foi resolvido com a operação de deslocamento a esquerda. Os demais métodos de complexidade constante são:

ehCheia

Verifica se o numero de nodes da árvore é igual ao esperado com base na altura da árvore por meio da fórmula mostrada em aula.

ehCompleta

Verifica se todas as posições para nodes no penúltimo nível da árvore já foram ocupadas, isto é o numero de posições livres é zero. Caso a árvore tenha seja até o nível 2, ela é automaticamente completa.

substituir

Realiza uma troca entre os dados de dois nós, o ideal seria trabalhar com ponteiros para o dado, ou trocar os apontadores de cada nó, mudando seus campos de `parent`, `left`, `right`, `l_cnt`, `r_cnt e level`, deixando apenas o campo data intocado.

atualizaParent

Dados dois nós, atualiza a informação do pai do primeiro para que a posição de filho direito ou esquerdo em que o primeiro nó ocupava seja trocada para o segundo nó informado. Caso o primeiro nó seja a raiz, o segundo nó ganha esse titulo.

atualizaNivelENodes

Decrementa o numero de nós da árvore, incrementa o numero de nós disponíveis no nível do nó indicado, e verifica se é necessário apagar

a existência desse nível, caso o nó indicado seja o ultimo do nível.

2.2 Métodos de complexidade Linear

Um algoritmo de complexidade linear é ótimo nas situações em que é preciso visitar todos os nós da árvore. Nesses casos o limite inferior do problema também é O(n), o que faz do algoritmo assintoticamente ótimo.

2.2.1 Métodos que Acessam Todos os Nós

Os métodos que acessam todos os nós são algoritmos Θ(n) e, por tanto, são assintoticamente ótimos (já que esse também é o limite inferior do problema).

recursiveErase

Utiliza a ideia do percusso em pós-ordem para deletar todos os nós a partir de uma raiz, apagando recursivamente suas sub-árvores.

toString

Utiliza uma fila para percorrer a árvore por nível, algoritmo baseado no pseudo-código mostrado em aula.

2.2.2 Métodos Dependentes da Altura

A complexidade desses algoritmos depende da construção da ABB. Quando a árvore é completa eles são $O(logn)$ e são $O(n)$ para árvores ziguezague.

search

Utiliza a estratégia padrão para a busca numa árvore binária de busca, comparando a partir da raiz com o dado buscado, e caso o dado seja encontrado, retorna o nó atual, caso o conteúdo do nó atual seja menor que o buscado, procuramos na subárvore a direita, senão na subárvore a esquerda. Se chegarmos numa subárvore vazia retornamos `nullptr`.

insere (ver Apêndice C)

Utiliza a mesma estratégia da busca, porém se o dado for encontrado retorna false, pois não podemos inserir um dado repetido, e quando chegarmos numa subárvore vazia, inserimos o um novo nó com o dado, ao inserirmos verificamos em qual nível da árvore esse nó foi criado, caso seja o primeiro nó de um novo nível, adicionamos uma nova posição a um vetor que contabiliza o numero de nós em cada nível, além de outras informações, durante a volta da recursão, caso o novo nó tenha sido in-

serido, atualizamos os contadores de filhos a esquerda ou a direita do nó, de acordo com a posição que o dado foi inserido.

minimum

Dado um nó de inicio avança sempre para o menor nó dessa subárvore, isto é para o nó a esquerda. até encontrar uma subárvore vazia, e então retorna o nó anterior a ela. Caso o nó fornecido já seja nulo, retorna a raiz da árvore.

maximum

Dado um nó de inicio avança sempre para o maior nó dessa subárvore, isto é para o nó a direita. até encontrar uma subárvore vazia, e então retorna o nó anterior a ela. Caso o nó fornecido já seja nulo, retorna a raiz da árvore.

remove (ver Apêndice D)

Utiliza a mesma ideia de buscar um elemento e, caso ele seja encontrado, o elemento é removido da árvore. A remoção possui diferentes passos dependendo do número de filhos do nó a ser removido.
Caso não possua filhos o nó apenas é removido e os devidos dados atualizados. O número de nós a esquerda ou a direita dos nós acima deste são atualizados na volta da recursão.
Caso tenha apenas um filho, o nó filho é colocado no lugar do nó removido, e então as informações são atualizadas.
Caso tenha dois filhos, então é buscado o substituto do nó a ser removido utilizando o método `_minimum_` no filho a direita, e então a nova posição do nó é removida, e então os dados atualizados.
Se tomarmos a altura do nó a ser removido como sendo k, e sabendo que o `_minimum_` será chamado para uma subárvore de k, podemos concluir que o número de nós percorrido pela chamada do `_minimum_` será menor ou igual a: $h - k - 1$. Onde h é a altura máxima da árvore.

enesimoElemento

Utiliza uma ideia similar a busca, porém em vez de usar o campo data, utiliza o número de filhos a esquerda de um *node*. O índice de um *node* é o número de *nodes* a esquerda dele(`l_cnt`) mais um. Caso o índice seja igual a essa soma retornamos o campo data do node atual, caso o índice seja maior, verificamos a subárvore a direita e armazenamos o número de *nodes* a esquerda do node atual para somar o total, senão apenas avançamos para a subárvore a esquerda, e repetimos esse processo enquanto não encontrarmos uma subárvore vazia.

posicao

Utiliza uma ideia muito semelhante ao busca e ao enesimoEle-

mento, porém retorna o índice do elemento buscado. Utiliza a estratégia de busca para encontrar o elemento e a estratégia do enesimoElemento para descobrir o índice.

mediana

Utiliza a definição da mediana como elemento que divide o conjunto de dados em dois, a informação do número de elementos da árvore e o método enesimoElemento para descobrir qual o elemento que ocupa a posição central (enesimoElemento(n/2) sendo n o número de nós).

Considerações finais

Existem hoje várias estruturas de dados prontas para uso e que atendem diversas necessidades de problemas recorrentes na computação. Porém, nem sempre encontramos nessas estruturas a melhor solução para nossos problemas. Por isso é importante sempre estudar os problemas, analisar as soluções e fazer as alterações necessárias para buscar a medida do possível soluções ótimas. No caso estudado vimos que a árvore de busca binária precisou ser alterada para que pudéssemos reduzir a complexidade, adicionando atributos e alterando instruções.

Referências

1 BRASSARD, P. B. G. *Fundamentals of Algorithmics*. Us ed. [S.l.]: Prentice Hall, 1995. ISBN 0133350681.

2 CORMEN CHARLES E. LEISERSON, R. L. R. C. S. T. H. *Introduction to algorithms*. 3. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844.

3 SZWARCFITER, L. M. J. L. *Estruturas de dados e seus algoritmos*. 3. ed. [S.l.]: Rio de Janeiro: LTC, 2010. 302 p. ISBN 9788521617501.

APÊNDICE A – Cabeçalho do Node

```
30 struct Node {
31    public:
32     /**
33      * Nó imediatamente acima deste.
34      */
35     Node* parent;
36     /**
37      * Nó imediatamente a esquerda deste.
38      */
39     Node* left;
40     /**
41      * Nó imediatamente a direita deste.
42      */
43     Node* right;
44     /**
45      * Número de nós a esquerda.
46      */
47     unsigned int l_cnt;
48     /**
49      * Número de nós a direita.
50      */
51     unsigned int r_cnt;
52     /**
53      * Conteúdo armazenado neste nó.
54      */
55     DataType data;
56     /**
57      * Nível do nó na árvore.
58      */
59     unsigned int level;
60
61    public:
62     /**
63      * @brief Constroi um novo objeto Node.
64      *
65      * @param value Valor a ser guardado pelo Node.
66      * @param n nível deste nó.
```

```
67          * @param p Nó acima deste.
68          * @param l Nó a esquerda deste.
69          * @param r Nó a direita deste.
70          */
71         Node(DataType value = DataType(), int n = 0, Node* p = nullptr,
72              Node* l = nullptr, Node* r = nullptr);
73         /**
74          * @brief Conta o numero de nós nas subárvores a esquerda e a
75          * direita do nó indicado.
76          *
77          * @param n Nó indicado.
78          * @return int numero de descendentes do nó indicado.
79          */
80         int countChildren(Node* n);
81     };
```

APÊNDICE B – Cabeçalho da ABB

```
83  /**
84   * @brief Classe Árvore Binária de Busca Estendida.
85   */
86  class ABB {
87     private:
88       /**
89        * Contabiliza o numero de nós em cada nível da árvore e o
90        * numero de níveis.
91        */
92       std::vector<unsigned int> levelCount;
93       /**
94        * Nó raiz da árvore.
95        */
96       Node* root;
97       /**
98        * Número de nós na árvore.
99        */
100      unsigned int size;
101      /**
```

```
102      * Altura da árvore.
103      */
104     unsigned int height;
```

APÊNDICE C – Inserir Elemento

```
78  // Dois casos, no primeiro temos tempo constante e no segundo uma chamada
79  // recursiva
80  bool ABB::insere(const DataType target) {
81      if (root == nullptr) {
82          root = new Node(target, 1, nullptr);
83          levelCount.push_back(0);
84          height = 1;
85          size = 1;
86          return true;
87      }
88      return insere(root, target);
89  }
90
91  // O pior caso é uma sub-árvore zig-zag, logo O(n)
92  // Função recursiva
93  bool ABB::insere(Node* node, const DataType target) {
94      int data = node->data;
95      if (data != target) {
96          if (data < target) {
97              if (node->right == nullptr) {
98                  node->right = new Node(target, node->level + 1, node);
99                  if (node->right->level > height) {
100                     ++height;
101                     levelCount.push_back(1u << (height - 1));
102                 }
103                 ++size;
104                 --levelCount[node->right->level - 1];
105                 ++node->r_cnt;
106                 return true;
107             }
108             if (insere(node->right, target)) {
```

```
109                 ++node->r_cnt;
110                 return true;
111             }
112         } else {
113             if (node->left == nullptr) {
114                 node->left = new Node(target, node->level + 1, node);
115                 if (node->left->level > height) {
116                     ++height;
117                     levelCount.push_back(1u << (height - 1));
118                 }
119                 ++size;
120                 --levelCount[node->left->level - 1];
121                 ++node->l_cnt;
122                 return true;
123             }
124             if (insere(node->left, target)) {
125                 ++node->l_cnt;
126                 return true;
127             }
128         }
129     }
```

APÊNDICE D – Remover Elemento

```
164 // O pior caso depende da complexidade do outro remove
165 // Função recursiva
166 bool ABB::remove(const DataType target) { return remove(root, target); }
167
168 // O pior caso é uma sub-árvore zig-zag, onde o nó a ser removido é
169 // a folha, logo O(n)
170 // Função recursiva e funções com Theta(1), função com laço while
171 bool ABB::remove(Node* node, const DataType target) {
172     if (node != nullptr) {
173         int data = node->data;
174         if (data == target) {
175             if (node->left == nullptr && node->right == nullptr) {
176                 atualizaParent(node, nullptr);
```

```
                atualizaNivelENodes(node);
                delete node;
                return true;
            }
            if (node->left == nullptr) {
                atualizaParent(node, node->right);
                atualizaNivelENodes(node->right);
                node->right->parent = node->parent;
                delete node;
                return true;
            }
            if (node->right == nullptr) {
                atualizaParent(node, node->left);
                atualizaNivelENodes(node->left);
                node->left->parent = node->parent;
                delete node;
                return true;
            }
            Node* smallest = minimum(node->right);
            substituir(node, smallest);
            atualizaParent(smallest, nullptr);
            atualizaNivelENodes(smallest);
            delete smallest;
            return true;
        }
        if (data < target) {
            if (remove(node->right, target)) {
                --node->r_cnt;
                return true;
            }
        } else {
            if (remove(node->left, target)) {
                --node->l_cnt;
                return true;
            }
        }
    }
    return false;
}
```