Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Ranna Raabe Fernandes da Costa
Frankleiton Levy de Sena Alves

DIM0117 - Estruturas de Dados Básicas II
06 de outubro 2019

1 Introdução

Este relatório contém uma breve abordagem aplicada na implementação de uma árvore binária de busca e análise de complexidade assintótica de cada um dos métodos implementados. Antes de tudo, alguns conceitos básicos e definições de uma árvore serão introduzidos de maneira sucinta.

Árvores são uma estrutura que permite o desenvolvimento de algoritmos mais eficientes. Um árvore enraizada é um conjunto finito de elementos chamados de nós. Cada nó de uma árvore possui ou não subárvores à esquerda e/ou direita do nó raiz.

Dentre várias formas de representar uma árvore, a mais comum é a maneira hierárquica, além disso considera-se a mesma relação de árvores genealógicas, ou seja existe nó pai, filho, avô, tio, etc. Alguns termos que são utilizados na estrutura são: *grau de saída de um nó, caminho da árvore, altura do nó* e *nível do nó*; estas definições foram vistas em sala de aula, portanto não serão descritas aqui, visto que serão consideradas como conhecimento prévio do leitor.

Uma **árvore binária** é uma árvore que, ou possui nenhum elemento, chamada árvore vazia, ou possui um elemento com dois ponteiros que apontam para duas subárvores à esquerda e à direita, onde cada subárvore também é binária.

Dentre as definições de árvore binária, é importante relembrar que conceitos como *árvore estritamente binária, árvore completa, árvore cheia, árvores zigue-zague, percursos em pré-ordem, pós-ordem e ordem simétrica, percurso em nível* e *árvore binária estendida*, também foram vistos em sala de aula, portanto também serão considerados como conhecimento prévio do leitor.

Por fim, introduzindo uma **árvore binária de busca**, é uma estrutura de dados baseada em árvore binária, onde todos os nós que ficam na subárvore à esquerda possuem valores que são inferiores ao nó pai, e os nós que ficam à direita possuem valores que são superiores ao nó pai. O objetivo desta árvore é estruturar os dados de modo a permitir uma busca binária.

2 Implementação

Neste projeto, foi implementada em C++ uma árvore binária de busca (ABB) aumentada para suportar, além das operações convencionais de **busca**, **inserção** e **remoção**, as operações listadas a seguir:

1. **nth_element(int n):** retorna o n-ésimo elemento (contando a partir de 1) do percurso em ordem (ordem simétrica) da ABB.

2. **position(int value):** retorna a posição ocupada pelo elemento value em um percurso em ordem simétrica na ABB (contando a partir de 1).

3. **mediana():** retorna o elemento que contém a mediana da ABB. Se a ABB possuir um número par de elementos, retorne o menor dentre os dois elementos medianos.

4. **is_full():** retorna verdadeiro se a ABB for uma árvore binária cheia e falso, caso contrário.

5. **is_complete():** retorna verdadeiro se a ABB for uma árvore binária completa e falso, caso contrário.
6. **to_string():** retorna uma String que contém a sequência de visitação (percorrimento) da ABB por nível.

Para definir um nó em uma árvore foi utilizado uma struct com chave, ponteiros esquerdo e direito e tamanho do nó, como também um construtor parametrizado.

```
struct node {
    int chave;
    node *esq;
    node *dir;
    int tam;

    node(int value) : chave(value), esq(nullptr), dir(nullptr), tam(1) { }
};
```

Lembrando que, no repositório https://github.com/rannaraabe/edb2-binary-tree/ está contido todos os métodos de implementação da ABB que estão bem comentados e explicados. E para criar uma ABB e testar, basta seguir os passos do README. Portanto, qualquer dúvida quanto à implementação que não fique clara o suficiente na descrição do relatório, pode ser conferida no repositório.

A implementação da ABB foi dividida em duas partes: criação dos métodos e execução dos testes.

A criação do métodos foi feita na **classe abb** (edb2-binary-tree/include/abb.hpp). Foi definido um node raiz, construtor e destrutor da abb, e a criação de todos os métodos já listados, e também os métodos de operações básicas de busca, inserção e remoção, utilizando conceitos vistos em sala de aula para que a implementação fosse feita de maneira eficiente.

Dentre os métodos implementados, a busca, inserção, enésimo, posição, cheia e completa foram implementados recursivamente, os demais métodos foram implementados iterativamente. Além disso, a classe possui métodos criados para auxiliar na implementação de outros métodos. Os métodos auxiliares foram:

- **largest_in_subtree():** percorre a subárvore esquerda até encontrar o maior elemento. Função auxiliar da função remove().
- **largest_in_subtree_father():** percorre a subárvore esquerda até encontrar o maior elemento e guarda o pai deste elemento. Função auxiliar da função remove().
- **smallest_in_subtree():** percorre a subárvore direita até encontrar o menor elemento. Função auxiliar da função remove(). *Esta função não foi utilizada por preferência dos desenvolvedores, entretanto no código está bem explicado que era uma opção que poderia ter sido escolhida, como visto em sala de aula.*
- **print():** imprime a árvore no formato pré ordem, e.g. (15(13(9(6()))(14()))(25(17())(28(27())(31())))). A função foi implementada recursivamente e foi feita apenas por preferência dos desenvolvedores, pois na hora dos testes era fácil visualizar a ávore desta maneira, além da maneira por nível (função to_string()).
- **size():** retorna o tamanho de um nó. Função auxiliar da função position().
- **height():** retorna a altura da árvore. Função auxiliar da função is_complete().

A execução dos métodos da ABB foram feitos na **classe main** (edb2-binary-tree/src/main.cpp). Foi feito a leitura de dois arquivos de entrada e a chamada dos métodos específicos.

O primeiro arquivo é o arquivo de entrada da ABB, contém uma sequência de valores inteiros separados por um espaço, os valores a serem armazenados na árvore. O arquivo (edb2-binary-tree/data/árvore.txt) foi passado por parâmetro na hora da execução dos testes (passo a passo descrito no README).

O segundo arquivo é o arquivo de comando, contém um sequência dos métodos (um método por linha) a serem chamados seguidos de um valor para as funções que precisam de parâmetros. A leitura deste arquivo foi feita dentro na classe main, usando a biblioteca *fstream*. Portanto para alterar os comandos, deve-se alterar o caminho do arquivo que está dentro do código ou editar o arquivo (edb2-binary-tree/data/comandos.txt).

A chamada dos métodos foi feita da seguinte maneira:

Primeiro foi adicionado em um *map* todos os comandos possíveis da ABB. Em seguida, o algoritmo lê o arquivo de comandos, guardando o comando e o parâmetro passado, fazendo isso uma linha por vez. Ao fazer a leitura do arquivo de comandos, o algoritmo confere qual é o comando chamado e o busca dentro do *map*, fazendo assim a chamada verdadeira à operação desejada dentro do *switch case*.

Depois de ler todo o arquivo e encerrar a leitura, o algoritmo chama a função print() para imprimir a árvore depois de todas as modificações, só para a leitura do usuário.

3 Análise de Complexidade

Nesta seção estão listadas todas as funções implementadas contendo a análise de complexidade assintótica de cada método. As próximas subseções contém uma breve descrição (esforçadamente para ser a mais clara possível) de cada operação possível na árvore binária de busca implementada.

3.1 search()

A função foi implementada recursivamente e sua funcionalidade é buscar um elemento na árvore. Possui duas funções: **search(int value)** e **search_recursive(node *no, int value)**.

A função *search(int value)* possui apenas a chamada para a função search_recursive(), passando a raiz da árvore e o valor de busca.

A função *search_recursive(node *no, int value)*, primeiro confere se o ponteiro do nó passado (inicialmente, a raiz) é diferente de nulo, caso não seja significa que a chave não está na árvore pois a árvore é vazia, logo ele retorna o próprio ponteiro (ou seja nulo), caso seja diferente de nulo, ele faz a busca da seguinte forma:

- o algoritmo compara se a chave do nó atual é igual ao valor passado, se for igual retorna o nó, pois a chave foi encontrada.
- se a chave do nó atual for diferente do valor passado por parâmetro, o algoritmo vai comparar se o valor é menor do que a chave do nó atual, se for menor ele retorna a *search_recursive()*, passando o filho esquerdo do nó e o valor a ser buscado, se não for menor significa que está na subárvore direita, então ele retorna a *search_recursive()*, passando o filho direito do ponteiro e o valor a ser buscado.

O algoritmo faz essa busca até encontrar o elemento ou descobrir que ele não está na árvore.

Complexidade: O(h)

3.2 insert()

A função foi implementada recursivamente e sua funcionalidade é inserir um elemento novo na árvore. Possui duas funções: **insert(int value)** e **insert_recursive(node *no, int value)**.

A função *insert(int value)* verifica se a raiz da árvore é nula, ou seja se a árvore é vazia, se for o algoritmo atribui o valor passado por parâmetro para um novo nó e encerra, se a raiz não for nula, então chama a função insert_recursive(node *no, int value), passando a raiz da árvore e o valor para inserir.

A função *insert_recursive(node *no, int value)*, primeiro confere se o valor passado para inserir na árvore já está na árvore. Se já estiver, o algoritmo avisa e encerra a chamada da função, se não estiver ele vai conferir se o valor é menor ou maior do que o valor do nó atual, ou seja confere se o valor vai ser inserido à esquerda ou à direita do nó atual.

Caso a inserção tenha que ser feita à esquerda, o algoritmo confere se o ponteiro esquerdo é diferente de nulo, se for chama a função recursiva, agora passando o ponteiro esquerdo e o valor para inserir, se não for nulo, ele cria um novo nó com aquele valor no ponteiro esquerdo. Segue o mesmo para o caso da inserção ter que ser feita à direita.

Por fim, o algoritmo atualiza o valor do tamanho da árvore.

Complexidade: O(h)

3.3 remove()

A função foir implementada de maneira interativa e sua funcionalidade é remover um elemento que esteja na árvore.

A função **remove(int value)** primeiro chama a função *search(value)* passando o valor do elemento a ser removido. Caso o elemento não esteja na árvore, o algoritmo avisa e encerra. Caso o elemento esteja na árvore, continua com o algoritmo.

O próximo passo é buscar o elemento na árvore de forma iterativa, de modo a salvar em um ponteiro o pai do nó que ele deseja remover. Logo após o algoritmo confere os casos possíveis para remoção, caso folha, caso o nó tenha uma subárvore e o caso do nó ter duas subárvore, e se comporta da seguinte forma:

- **caso folha:** confere se os ponteiros esquerdo e direito são nulos, ou seja se o nó é folha, se for ele confere se deve ir para o lado esquerdo ou direito (existe um booleano para ajudar nessa direção, no código está bem mais claro de entender), se for esquerdo ele atribui nulo para o o ponteiro esquerdo do pai desse nó que vai ser removido e diminui o tamanho da árvore, e faz o mesmo para o caso de ser direito o caminho que deve seguir.

- **caso uma subárvore:** para este caso, o algoritmo confere se o nó só possui subárvore à esquerda ou se só possui subárvore à direita. Se possuir subárvore somente à esquerda, ele confere se o pai do nó atual deve receber o valor do nó esquerdo (caso 2 do slide da sala de aula), e faz o mesmo para o caso de ter uma subárvore somente à direita (a descrição desse caso fica mais clara ao ler o código).

- **caso duas subárvores:** o algoritmo chama uma função auxiliar *largest_in_subtree()* que percorre até chegar no maior elemento da subárvore da esquerda e retorna esse nó e depois chama outra função auxiliar *largest_in_subtree_father()* que faz o mesmo, porém retornando o pai do nó. Lembrando que essa função auxiliar poderia ser implementada de forma a chegar ao menor elemento da subárvore direita, porém a forma escolhida foi a *largest_in_subtree()*. Com o nó para fazer a troca e seu pai salvos, o algoritmo primeiro confere se o pai de troca é nulo, se for, o ponteiro direito do nó troca recebe o ponteiro direito do nó a ser removido, se não for, ele faz a permutação entre os ponteiros de direito e esquerdo do nó troca e do nó a ser removido. Em seguida, o algoritmo confere o caso do pai do nó a ser removido (que foi salvo na busca iterativa) é nulo, se for significa que o nó é a raiz, então ele permuta o nó raiz com o nó troca, e diminui o tamanho da árvore. Se não for nulo, então ele confere se deve ir para subárvore esquerda ou direita, caso seja esquerda, o ponteiro esquerdo do pai do nó recebe troca e diminui o tamanho da árvore, caso contrário faz o mesmo para o ponteiro direito do pai. É importante lembrar que é necessário sim diminuir o tamanho da árvore em cada caso de remoção do nó. As funções *largest_in_subtree()* e *largest_in_subtree_father()* são bem descritas nos comentários do código.

Vale a pena ressaltar novamente, que entender todos os casos de remoção fica mais claro ao visualizar o código, pois o mesmo está bem comentado.

Complexidade: O(h)

3.4 nth_element()

A função n-ésimo retorna o n-ésimo elemento da árvore e foi implementada recursivamente. Possui duas funções: **nth_element(int n)** e **nth_element_recursive(node *no, int n)**.

A função *nth_element(int n)* possui apenas um caso base, onde faz a chamada da função recursiva *nth_element_recursive(node *no, int n)* passando a raiz da árvore e n+1 (soma 1 ao n, pois a função é 0 indexado).

A função *nth_element_recursive(node *no, int n)*, primeiro confere se o nó passado é nulo ou se o tamanho do nó é menor do que o n passado por parâmetro, se for significa que, ou a árvore é nula, ou o elemento passado não está na árvore, então a função exibe uma mensagem de erro e retorna um novo nó inicializado com um valor qualquer (no caso -1e9, -10^9).

No caso de não entrar nessa condição, o algoritmo calcula o tamanho da subárvore esquerda e confere se a diferença entre o valor e o tamanho da subárvore esquerda é zero, se sim significa que você está no elemento que estava buscando, então basta retorná-lo. Caso esta condição não seja verdade, o algoritmo vai conferir

se o valor passado é menor do que o tamanho da subárvore esquerda, se for significa que o elemento passado está na subárvore esquerda, logo chama a função recursiva passando o nó esquerdo e o valor de busca, caso contrário o algoritmo faz o mesmo para a subárvore direita.

Complexidade: O(h)

3.5 position()

A função retorna o elemento que está na posição passada por parâmetro e foi implementa recursivamente. Possui duas funções: **position(int value)** e **position_recursive(node *no, int value, int l, int r)**.

A função *position(int value)* primeiro faz uma busca para saber se o elemento está na árvore, caso esteja retorna uma chamada à função recursiva somando 1 (pois também é uma função 0 indexada) passando a raiz, o elemento, o tamanho do ponteiro esquerdo da raiz e o tamanho do ponteiro direito da raiz (a tamanho do nó é calculado com a função auxiliar *size()*). Caso o elemento não esteja na árvore, o algoritmo avisa ao usário e encerra retornando um valor qualquer (no caso -1e9, -10^9).

A função *position_recursive(node *no, int value, int l, int r)* confere se o valor do nó atual é o valor do elemento que foi passado por parâmetro, caso seja retorna *l* (o *l* passado por parâmetro, que é a quantidade de nós à esquerda do nó atual). Caso não seja, o algoritmo confere se a busca deve ser feita na subárvore esquerda, calcula os novos valores para l e r, e faz a chamada da função recursiva passando os novos valores, caso contrário confere se a busca deve ser feita na subárvore direita e retorna a chamada à função recursiva.

Complexidade: O(h)

3.6 mediana()

A função **mediana()** retorna o elemento que contém a mediana da árvore, caso seja ímpar retorna o elemento da mediana, caso seja par retorna o menor dos dois elementos medianos.

A função possui apenas uma chamada à função *nth_element()*, passando como parâmetro o cálculo da mediana. Como a função *nth_element()* retorna um nó, a função *mediana()* retorna um inteiro, que é a chave desse nó.

Complexidade: O(h)

3.7 is_full()

A função retorna true para o caso da árvore binária ser cheia e false caso contrário, e foi implementada recursivamente. Possui duas funções: **is_full()** e **is_full_recursive(node *no)**.

A função *is_full()* possui apenas a chamada à função recursiva.

A função *is_full_recursive(node *no)* confere se o nó é nulo, para o caso da árvore ser vazia e retorna false para este caso. Confere depois se o nó é folha, ou seja se os filhos esquerdo e direito do nó são nulos, para este caso retorna true. E por fim, confere o último caso, se os filhos esquerdo e direito do nó não são nulos, ou sejam possuem outros filhos, então o algoritmo faz a chamada recursiva para continuar conferindo se a árvore é cheia.

Complexidade: O(h)

3.8 is_complete()

A função retorna true para o caso da árvore binária ser completa e false caso contrário, e foi implementada recursivamente. Possui duas funções: **is_complete()** e **is_complete_recursive(node *no, int altura)**.

A função *is_complete()* salva a altura do nó chamando uma função auxiliar *height(node *no)* (que retorna apenas a altura do nó), e depois chama a função recursiva passando a raiz e a altura do nó.

A função *is_complete_recursive(node *no, int altura)* confere se a altura do nó é menor ou igual a 2, isso significa que o nó tem 0, 1 ou 2 filhos, nesse caso retorna true (pois até o penúltimo nível da árvore, os nós não precisam ter exatamente 2 filhos). Depois o algoritmo confere se o nó é nulo, para o caso da árvore ser nula, e retorna false caso seja. Por fim chama a função recursiva duas vezes com operador lógico AND, e.g. funcao() && funcao(), passando o nó esquerdo e a diferença da altura e 1 (ou seja, retira ele mesmo ao fazer a conta da altura) e o mesmo passando o nó direito.

Complexidade: O(h)

3.9 to_string()

A função foi implementada de modo iterativo e retorna uma String com a sequência de visitação da árvore por nível.

Primeiro, confere se a raiz da árvore é nula, caso seja significa que a árvore é nula então a função exibe uma mensagem de erro e retorna uma String vazia.

Caso a raiz não seja nula, o algoritmo usa uma fila (*queue*) para guardar os filhos do nó. Para inicializar, ele adiciona a raiz na fila. Enquanto a fila não for vazia, o algoritmo remove o primeiro valor que já está na fila, concantena os valores dos nós que estão na fila em uma string e depois confere se o filho esquerdo não é vazio para adicioná-lo na fila e o mesmo para o filho direito. Assim, o algortimo vai concatenando os valores dos nós de forma que os comandos *pop()* e *push()* vão enfileirando os nós na fila por nível da árvore. No final da execução, a String retornada possui todos os n nós da árvore visitados por nível.

Uma **observação importante**: quando o usuário chama o comando de imprimir a árvore no arquivo de comandos, esta função é a que será chamada, e não a função auxiliar *print()*.

Complexidade: O(n)