

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL - IMD
CURSO DE BACHARELADO EM TECNOLOGIA DA
INFORMAÇÃO

IMPLEMENTAÇÃO DE ÁRVORE BINÁRIA DE BUSCA

**Paulo Vitor Lima Borges
Dante Augusto Bezerra Pinto
Lucas Pires de Souza Marcolino**

**Natal-RN
2022**

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL - IMD
CURSO DE BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

IMPLEMENTAÇÃO DE ÁRVORE BINÁRIA DE BUSCA

Paulo Vitor Lima Borges
Dante Augusto Bezerra Pinto
Lucas Pires de Souza Marcolino

Relatório técnico para a matéria de Estruturas de Dados Básicas II, descrevendo a implementação de uma Árvore Binária de Busca com operações a mais.

Natal-RN
2022

Sumário

| | | |
|------------|----------------------------------|-----------|
| 1 | INTRODUÇÃO | 3 |
| 2 | METODOLOGIA | 4 |
| 2.1 | Materiais utilizados | 4 |
| 2.1.1 | Ferramentas de programação | 4 |
| 2.1.2 | Árvore binária de busca | 5 |
| 2.1.3 | Métodos | 6 |
| 2.1.3.1 | Busca | 6 |
| 2.1.3.2 | Inserção | 7 |
| 2.1.3.3 | Remoção | 8 |
| 2.1.3.4 | Enésimo elemento | 9 |
| 2.1.3.5 | Posição | 10 |
| 2.1.3.6 | Mediana | 11 |
| 2.1.3.7 | Media | 11 |
| 2.1.3.8 | É cheia | 13 |
| 2.1.3.9 | É completa | 14 |
| 2.1.3.10 | Pré-ordem | 14 |
| 2.1.3.11 | Imprimir Árvore | 15 |
| 3 | ANÁLISES DE COMPLEXIDADES | 17 |
| 3.0.1 | Busca | 17 |
| 3.0.2 | Inserção | 17 |
| 3.0.3 | Remoção | 17 |
| 3.0.4 | Enésimo Elemento | 17 |
| 3.0.5 | Posição | 17 |
| 3.0.6 | Mediana | 17 |
| 3.0.7 | Média | 18 |
| 3.0.8 | É cheia | 18 |
| 3.0.9 | É completa | 18 |
| 3.0.10 | Pré-Ordem | 18 |
| 3.0.11 | Imprimir Árvore | 18 |
| 4 | CONCLUSÃO | 19 |

1 Introdução

O presente trabalho tem como objetivo descrever a implementação de uma árvore binária de busca que guarde valores inteiros e tenha como métodos os convencionais, os de busca, inserção e remoção, e os métodos enunciados no anexo do trabalho.

Para tal implementação, o projeto foi desenvolvido com a linguagem de programação C++ criado a partir de uma classe para melhor visualização desse conceito abstrato. O próximo tópico irá mostrar mais detalhes de como isso foi realizado.

Por fim, todas as funções tiveram suas complexidades analisadas de acordo com sua implementação no presente trabalho.

2 Metodologia

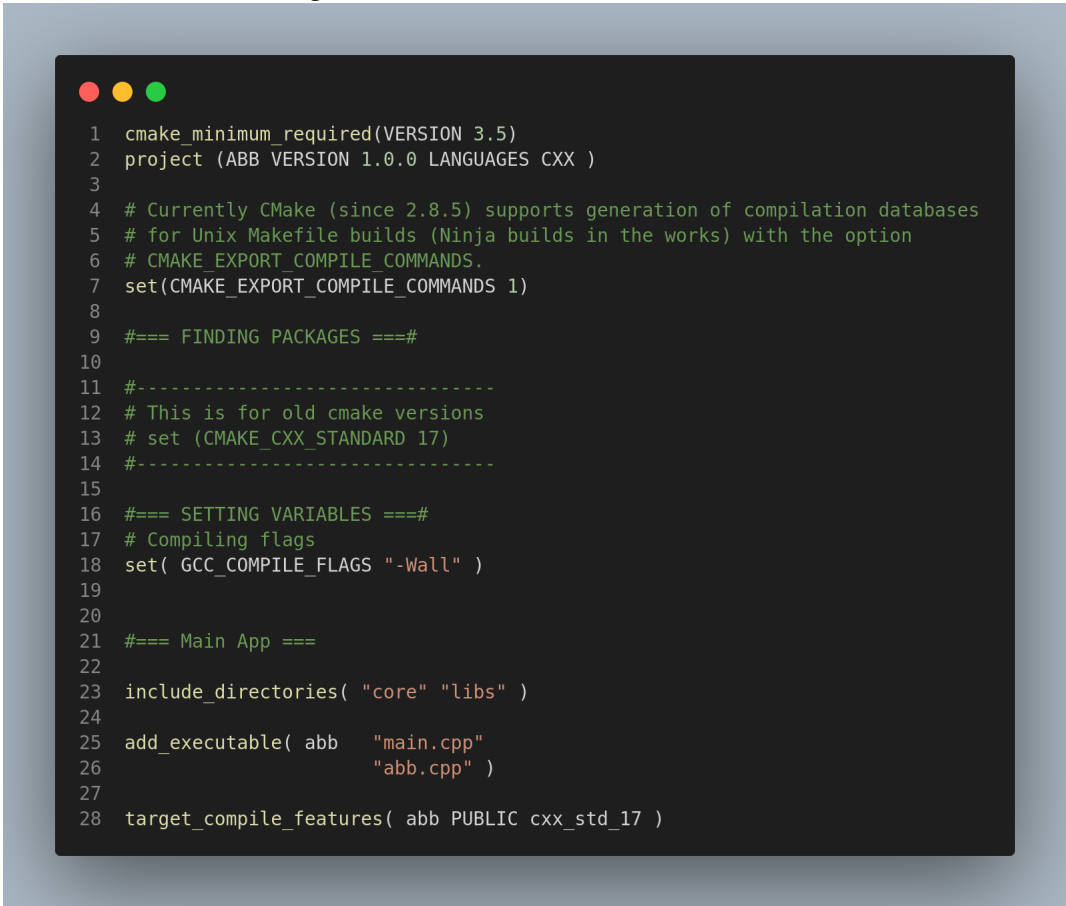
2.1 Materiais utilizados

2.1.1 Ferramentas de programação

A implementação da Árvore Binária de Busca e as implementações de seus respectivos métodos foram feitos em C++ na versão 17.

Tanto no ambiente WSL/Windows quanto no GNU/Linux, os códigos foram compilados usando G++, através do CMake.

Para isso, foi usado o seguinte CMakeLists.txt:



```

1 cmake_minimum_required(VERSION 3.5)
2 project (ABB VERSION 1.0.0 LANGUAGES CXX )
3
4 # Currently CMake (since 2.8.5) supports generation of compilation databases
5 # for Unix Makefile builds (Ninja builds in the works) with the option
6 # CMAKE_EXPORT_COMPILE_COMMANDS.
7 set(CMAKE_EXPORT_COMPILE_COMMANDS 1)
8
9 #=== FINDING PACKAGES ===#
10
11 #-----
12 # This is for old cmake versions
13 # set (CMAKE_CXX_STANDARD 17)
14 #-----
15
16 #=== SETTING VARIABLES ===#
17 # Compiling flags
18 set( GCC_COMPILE_FLAGS "-Wall" )
19
20
21 #=== Main App ===
22
23 include_directories( "core" "libs" )
24
25 add_executable( abb      "main.cpp"
26                  "abb.cpp" )
27
28 target_compile_features( abb PUBLIC cxx_std_17 )
  
```

Para compilação, foram usadas as seguintes linhas de comando:

```
cmake -S source -B build
```

```
cmake --build build
```

Já para execução, foi executada a seguinte linha:

```
./build/ABB <arquivo de descrição da árvore> <arquivo dos métodos>
```

2.1.2 Árvore binária de busca

Para implementar a árvore em si, a seguinte classe foi implementada:

```
1  #ifndef ARVORE_H
2  #define ARVORE_H
3
4  #include <string>
5  #include <stack>
6  #include <queue>
7  #include <optional>
8  #include <math.h>
9
10 using std::string;
11
12 class abb {
13     private:
14         abb* esq = nullptr;
15         abb* dir = nullptr;
16         int tamanho_esq = 0;
17         int tamanho_dir = 0;
18         int valor;
19         int altura = 1;
20     public:
21         abb();
22         abb(int valor, abb* esq = nullptr, abb* dir = nullptr) : valor(valor), esq(esq) , dir(dir) {}
23         int getValor();
24
25         std::optional<abb*> busca(int x);
26         bool inserir(int x);
27         bool remover(int x);
28         abb* remover(int x, abb* sub);
29
30         std::optional<int> enesimoElemento(int n);
31
32         std::optional<int> posicao(int x);
33
34         std::optional<int> mediana();
35
36         std::optional<double> media(int x);
37
38         bool ehCheia();
39
40         bool ehCompleta();
41
42         string pre_ordem();
43
44         void imprimeArvore(int s);
45     private:
46         void formato1(int qnt_tabs, int espaco, abb *no);
47         void formato2(abb *no);
48
49         void calcularAltura(abb* raiz);
50         int qnt_nos();
51 };
52
53 #endif
```

Figura 1 – Árvore Binária de Busca

Analisando os atributos, vemos que esq e dir representam as sub-árvores à esquerda e à direita, tamanho_esq e tamanho_dir representam as quantidades de nós das mesmas, valor é o

que está armazenado no nó em questão, podemos dizer que o nó atual é a raiz da árvore, e altura, que é medida em relação a árvore toda.

2.1.3 Métodos

2.1.3.1 Busca

Esse método recebe como parâmetro o valor que queremos encontrar na árvore e retorna um ponteiro do nó em que o valor se encontra ou um valor vazio, caso não tenha um nó com esse valor. Assim, foi feito o seguinte código:

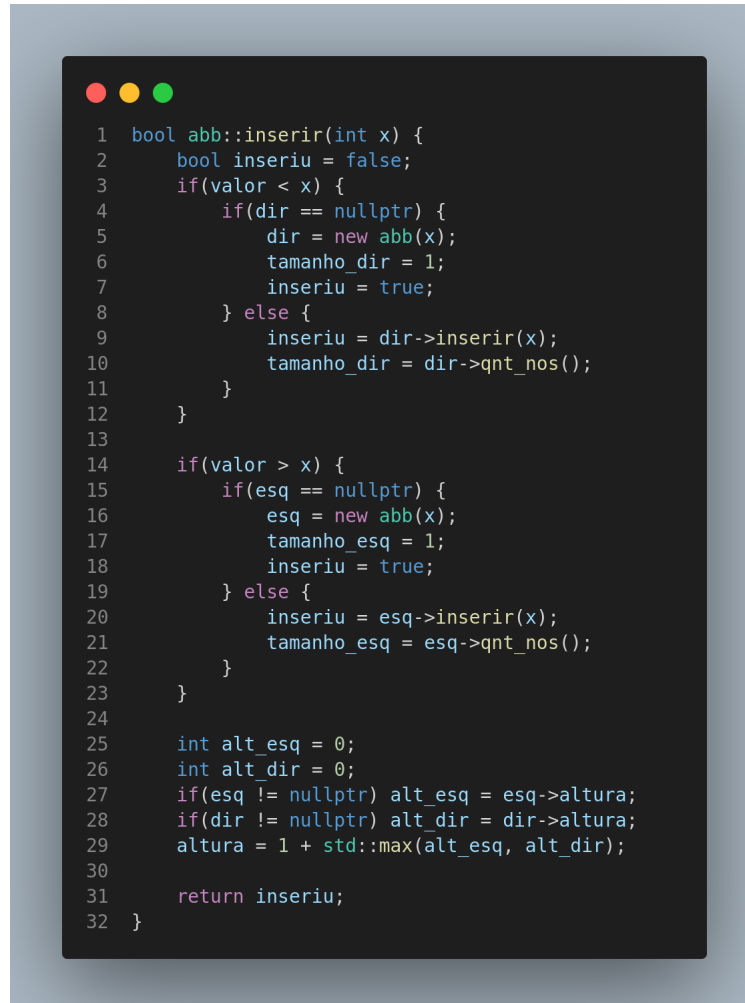
A screenshot of a code editor with a dark background and light-colored text. The code is a C++ function named `busca` that takes an integer `x` as input and returns a `std::optional<abb*>`. The function checks if `x` is equal to `valor`. If yes, it returns `this`. If `altura` is 1, it returns an empty optional. Otherwise, it checks if `valor` is less than `x`. If so, it checks if `tamanho_dir` is 0; if yes, it returns an empty optional, otherwise it recursively calls `busca` on `dir`. If `valor` is greater than `x`, it checks if `tamanho_esq` is 0; if yes, it returns an empty optional, otherwise it recursively calls `busca` on `esq`. The function is enclosed in a class `abb` namespace.

```
1  std::optional<abb*> abb::busca(int x){
2      if(x == valor)return this;
3      if(altura == 1){
4          return {};
5      }else{
6          if(valor < x){
7              if(tamanho_dir == 0) return {};
8              else return dir->busca(x);
9          }else{
10             if(tamanho_esq == 0) return {};
11             else return esq->busca(x);
12         }
13     }
14 }
```

Figura 2 – Método Busca

Dessa forma, note que esse método foi definido recursivamente e o caso base é quando encontramos o valor. Se não encontramos, checamos se o nó atual é um nó folha, o que implicaria que o valor não existe na árvore pois não está no nó atual e não há mais para onde descer. Caso não seja nó folha, é checado quem é menor, o valor do nó ou o que desejamos encontrar, e com base nisso fazemos a chamada recursiva na sub-árvore da esquerda, caso o valor do nó seja maior, ou na direita, caso seja menor, se a tal existir. Caso a sub-árvore em que devemos descer não exista, o valor não existe na árvore logo o retorno será vazio.

2.1.3.2 Inserção



```
1  bool abb::inserir(int x) {
2      bool inseriu = false;
3      if(valor < x) {
4          if(dir == nullptr) {
5              dir = new abb(x);
6              tamanho_dir = 1;
7              inseriu = true;
8          } else {
9              inseriu = dir->inserir(x);
10             tamanho_dir = dir->qnt_nos();
11         }
12     }
13
14     if(valor > x) {
15         if(esq == nullptr) {
16             esq = new abb(x);
17             tamanho_esq = 1;
18             inseriu = true;
19         } else {
20             inseriu = esq->inserir(x);
21             tamanho_esq = esq->qnt_nos();
22         }
23     }
24
25     int alt_esq = 0;
26     int alt_dir = 0;
27     if(esq != nullptr) alt_esq = esq->altura;
28     if(dir != nullptr) alt_dir = dir->altura;
29     altura = 1 + std::max(alt_esq, alt_dir);
30
31     return inseriu;
32 }
```

Figura 3 – Método Inserir

O método de inserção tem como objetivo inserir um elemento em sua devida posição (uma folha), mantendo as propriedades da árvore binária de busca. Para tanto, foi feito um procedimento similar ao de busca na árvore. Porém, ao chegar numa folha, é criado um novo nó à esquerda ou à direita, a depender do valor do nó a ser inserido.

O procedimento começa verificando se o nó da subárvore atual é maior ou menor que o elemento que se deseja inserir: caso seja maior, devemos inseri-lo na subárvore à esquerda, caso seja menor, à direita. Caso ocorra que o valor a ser inserido já exista na árvore, o procedimento retorna o valor booleano "falso".

Para inserir o valor em uma subárvore, verificamos se esta é nula à esquerda ou à direita, a depender do caso.

Além do trabalho de inserir, o método também trata de atualizar os atributos altura de cada nó e o tamanho de suas duas subárvore. Para isso, após inserir o valor, tais valores são calculados novamente.


2.1.3.3 Remoção



```
1 bool abb::remover(int x) {
2     int nos_antes = qnt_nos();
3     remover(x, this);
4     return qnt_nos() < nos_antes;
5 }
```

Figura 4 – Método Principal de Remoção

O método de remoção tem como objetivo remover o nó que tem como chave aquele passado para o método. Para isso, foi feita uma função de remoção auxiliar, que recebe o valor a ser removido, a subárvore da qual se deseja remover e retorna tal subárvore já com o elemento removido. Portanto, o método remover apenas chama o auxiliar, passando a raiz da árvore para este.



```
1 abb* abb::remover(int x, abb* sub) {
2     if(sub == nullptr) {
3         return nullptr;
4     }
5
6     if(x < sub->valor) {
7         sub->esq = remover(x, sub->esq);
8     } else if(x > sub->valor) {
9         sub->dir = remover(x, sub->dir);
10    } else {
11        if(sub->esq == nullptr) {
12            return sub->dir;
13        } else if(sub->dir == nullptr) {
14            return sub->esq;
15        } else {
16            abb* novo_no = sub->esq;
17            while(novo_no->dir != nullptr) {
18                novo_no = novo_no->dir;
19            }
20            sub->valor = novo_no->valor;
21            sub->esq = remover(sub->valor, sub->esq);
22        }
23    }
24
25    int alt_esq = 0;
26    int alt_dir = 0;
27    if(sub->esq != nullptr) alt_esq = sub->esq->altura;
28    if(sub->dir != nullptr) alt_dir = sub->dir->altura;
29    sub->altura = 1 + std::max(alt_esq, alt_dir);
30
31    sub->tamanho_esq = 0;
32    sub->tamanho_dir = 0;
33    if(sub->esq != nullptr) sub->tamanho_esq = sub->esq->qnt_nos();
34    if(sub->dir != nullptr) sub->tamanho_dir = sub->dir->qnt_nos();
35
36    return sub;
37 }
```

Figura 5 – Método Remover Auxiliar

O método auxiliar funciona da seguinte forma: Caso a subárvore da qual se deseja remover seja nula, o método retorna nulo.

Caso o valor que se deseja remover seja menor que a raiz da árvore, devemos remover da subárvore à esquerda, logo esta recebe o resultado da chamada recursiva de remover o valor de si mesma. Caso o valor seja maior, fazemos o mesmo procedimento para a subárvore à direita.

Ao chegarmos no nó que devemos remover, temos 3 casos:

- O nó é uma folha: nesse caso, apenas retornamos o valor null, pois ao remover este nó, a árvore resultante é vazia.
- O nó possui 1 subárvore: nesse caso, apenas retornamos a outra subárvore, uma vez que a árvore resultante após remover o nó em questão se trata do único nó que restará.
- O nó possui 2 subárvores: nesse caso, devemos procurar um nó para substituir a raiz que será removida. Na implementação do presente trabalho, tal nó é o mais à direita da subárvore à esquerda, que garante a manutenção das propriedades da ABB.

2.1.3.4 Enésimo elemento

Esse método recebe um número inteiro e retorna o valor do nó que está naquela posição em ordem simétrica. Se o número for além da quantidade de nós ou não positivo, o retorno é vazio. Segue a imagem do código:

```
1 std::optional<int> abb::enesimoElemento(int n){
2     if(n < 1 || n > (1 + tamanho_esq + tamanho_dir)) return
3 };
4     if(n == tamanho_esq + 1) return valor;
5     else{
6         if(n <= tamanho_esq) return esq->enesimoElemento(n);
7         else{
8             return dir->enesimoElemento(n - tamanho_esq - 1);
9         }
10 }
```

Figura 6 – Método Enésimo Elemento

Como primeira condição, verificamos se o parametro é negativo ou se é maior que a quantidade geral de nós, pois em qualquer desses casos o retorno é vazio. A seguir, checamos se a posição é igual ao tamanho da sub-árvore a esquerda mais 1, pois essa é a posição do nó atual em ordem simétrica. Se for igual, encontramos o nosso retorno, se não devemos checar outros casos. Se a posição for menor ou igual que a quantidade de elementos da sub-árvore esquerda, então devemos chamar a recursão lá. Caso contrario, devemos chamar a recursão na sub-árvore direita mas a posição agora será diminuida uma quantidade igual a quantos nós a sub-árvore a esquerda tem mais um, que é o próprio nó que estamos. A subtração ocorre para casos como o seguinte: o quinto elemento da arvore binária de busca 1, 2, 3, 4, 5 é o mesmo que o segundo elemento da sub-árvore 4, 5.

2.1.3.5 Posição

Esse método recebe um valor e tem como retorno a posição em ordem simétrica do nó com aquele valor ou retorna vazio se a árvore não tiver um nó com aquele valor. Dessa forma, temos que o código é o seguinte:

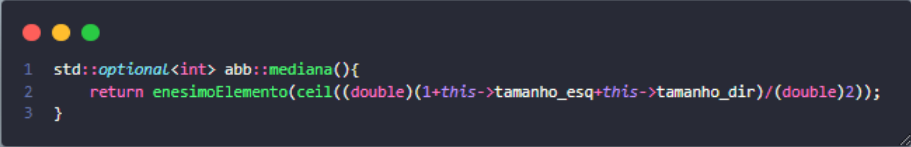
A screenshot of a code editor with a dark background and light-colored text. The code is written in C++ and is enclosed in a light blue border. The code is as follows:

```
1  std::optional<int> abb::posicao(int x){
2      if(x == valor) return tamanho_esq+1;
3      if(tamanho_esq == 0 && tamanho_dir == 0){
4          return {};
5      }else{
6          if(x < valor){
7              if(tamanho_esq == 0) return {};
8              else return esq->posicao(x);
9          }
10         if(x > valor){
11             if(tamanho_dir == 0) return {};
12             else{
13                 auto y = dir->posicao(x);
14                 if(y.has_value()){
15                     return tamanho_esq + 1 + y.value();
16                 }else{
17                     return {};
18                 }
19             }
20         }
21     }
22
23     return {}; // warning
24 }
```

Figura 7 – Método de posição

Note que esse código é muito semelhante ao de busca, pois o que está sendo feito é realmente buscar o nó com aquele valor. O diferencial seria o retorno, que é o tamanho da sub-árvore a esquerda mais 1, que é a posição do nó na ordem simétrica, como já foi dito. Com isso, ao chamarmos a recursão para a sub-árvore direita, nós primeiro checamos se ela tem valor, pois pode ser que não tenhamos encontrado o nó, e se tiver é necessário adicionar àquele valor a quantidade de nós da sub-árvore a esquerda mais 1 pois essa é a quantidade de elementos que antecedem o parametro na ordem simétrica.

2.1.3.6 Mediana



```
1 std::optional<int> abb::mediana(){
2     return enesimoElemento(ceil(((double)(1+this->tamanho_esq+this->tamanho_dir))/(double)2));
3 }
```

Figura 8 – Método da mediana

O método da mediana em nossa Árvore Binária de Busca tinha como objetivo encontrar o elemento que está na metade do caminho do percorrimento em ordem simétrica. Pela própria descrição do problema, pode-se inferir o uso de um método implementado também pela árvore chamado `enesimoElemento`, que retorna o elemento presente na posição passada como parâmetro.

Decido parte do algoritmo, agora precisamos determinar em que parte estaria presente o elemento que representa na mediana. Para tal, basta sabermos quantos nós estão presentes na árvore e pegar a metade, arredondando pra cima, e passando esse número como parâmetro do método `enesimoElemento`.

2.1.3.7 Média



```
1 std::optional<double> abb::media(int x){
2     auto aux = busca(x);
3     if(aux.has_value()){
4         abb* raiz{aux.value()};
5         std::stack<abb*> s;
6         int valor{0};
7         int quantidade{1+raiz->tamanho_esq+raiz->tamanho_dir};
8         s.push(raiz);
9         while(not s.empty()){
10             abb* atual{s.top()};
11             s.pop();
12             if(atual->dir != nullptr){
13                 s.push(atual->dir);
14             }
15             valor += atual->valor;
16             if(atual->esq != nullptr){
17                 s.push(atual->esq);
18             }
19         }
20         return ((double)valor/(double)quantidade);
21     }else{
22         return {};
23     }
24 }
```

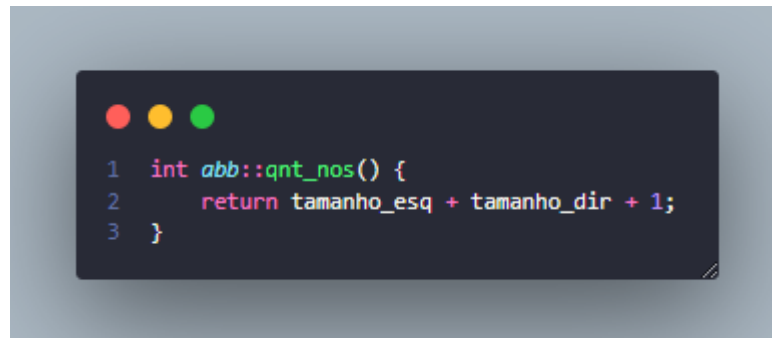
Figura 9 – Método da média

O método da média em nossa Árvore Binária de Busca tinha como objetivo encontrar a média aritmética da subárvore a partir da chave - número inteiro - passada como parâmetro. Para isso, inicialmente precisamos fazer uma busca pela chave passada como parâmetro. Caso a chave

não seja encontrada, nada é retornado. Caso contrário, iremos inicialmente pegar a quantidade de nós que será contabilizado, somando a subárvore da esquerda, a da direita e o nó raiz. Após isso, percorreremos toda a árvore pegando seus valores e adicionando à uma variável que está guardando todos os valores dos nós que estamos passando ao longo do percurso. Ao final, basta retornar a divisão entre a soma resultante e a quantidade de nós contada previamente.

2.1.3.8 É cheia

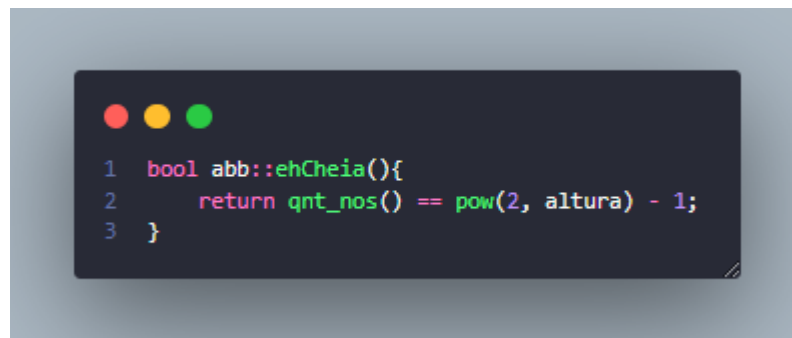
O seguinte método é usado no `é cheia` e no `é completa`:

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in C++ and defines a method named `qnt_nos()` for a class named `abb`. The code consists of three lines: a return type declaration, a return statement, and a closing brace.

```
1 int abb::qnt_nos() {  
2     return tamanho_esq + tamanho_dir + 1;  
3 }
```

Figura 10 – Método da quantidade de nós

Agora, vamos para o método `é cheia`.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in C++ and defines a method named `ehCheia()` for a class named `abb`. The code consists of three lines: a return type declaration, a return statement, and a closing brace.

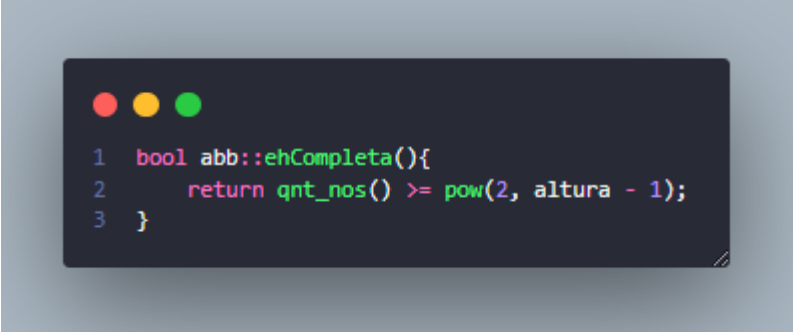
```
1 bool abb::ehCheia(){  
2     return qnt_nos() == pow(2, altura) - 1;  
3 }
```

Figura 11 – Método `ehCheia`

O método da `ehCheia` em nossa Árvore Binária de Busca tinha como objetivo dizer se uma árvore é cheia ou não. Para isso, temos que saber se a quantidade de nós na árvore é máxima para a altura do nó raiz. Para isso, inicialmente vamos saber quantos nós têm em cada nível de uma árvore. Por ser uma árvore binária, cada nível possui uma quantidade 2^n de nós, sendo n o nível da árvore.

Somando a quantidade de nós que há em cada nível temos $2^h - 1$ nós. Portanto, basta retornar se a quantidade de nós da árvore - somando a raiz e a quantidade de nós em cada subárvore - é igual à $2^h - 1$.

2.1.3.9 É completa



```
1 bool abb::ehCompleta(){
2     return qnt_nos() >= pow(2, altura - 1);
3 }
```

Figura 12 – Método ehCompleta

O método da ehCompleta em nossa Árvore Binária de Busca tinha como objetivo dizer se uma árvore é completa ou não. Para isso, temos que saber se a árvore é cheia até o penúltimo nível. Para isso, basta retornar se a quantidade de nós da árvore é maior ou igual a 2^{h-1} visto que se fosse igual, estaria com a quantidade máxima de nós até o penúltimo nível e com um nó apenas no último nível e, caso fosse maior, o limite seria de $2^h - 1$, sendo uma árvore cheia.

2.1.3.10 Pré-ordem



```
1 string abb::pre_ordem() {
2     string ordem = "";
3
4     std::stack<abb> pilha;
5
6     pilha.push(*this);
7
8     while(!pilha.empty()) {
9         abb topo = pilha.top(); pilha.pop();
10
11         ordem += std::to_string(topo.valor) + " ";
12
13         if(topo.dir != nullptr) pilha.push(*topo.dir);
14         if(topo.esq != nullptr) pilha.push(*topo.esq);
15     }
16
17     return ordem;
18 }
```

Figura 13 – Método de Pré-Ordem

Este método tem como objetivo criar uma string que contém os valores de todos os nós da subárvore em Pré-Ordem. Para isso, foi feito o percurso na árvore de forma iterativa, usando uma Pilha. Ao remover o topo da pilha, o valor deste é concatenado à uma string que será o retorno do método.

2.1.3.11 Imprimir Árvore



```
1 void abb::imprimeArvore(int s) {  
2     if(s == 1) {  
3         formato1(0, 36, this);  
4     } else {  
5         formato2(this);  
6     }  
7 }
```

Figura 14 – Método de Imprimir Árvore

Este método tem como objetivo imprimir a árvore de forma hierárquica no formato informado à função. Tais formatos foram dados previamente pela descrição do projeto.

Caso o formato requerido seja o 1, o seguinte procedimento é executado:



```
1 void abb::formato1(int qnt_tabs, int espaco, abb *no) {  
2     std::cout << string(qnt_tabs, '\t');  
3     std::cout.width(espaco);  
4     std::cout.fill('-');  
5     std::cout << std::left << no->valor << std::endl;  
6  
7     if(no->esq != nullptr) {  
8         formato1(qnt_tabs + 1, espaco - 6, no->esq);  
9     }  
10  
11     if(no->dir != nullptr) {  
12         formato1(qnt_tabs + 1, espaco - 6, no->dir);  
13     }  
14 }
```

Figura 15 – Formato de Impressão 1

Com as informações de quantas tabulações devem ser impressas e quantos espaços a linha de cada nó deve ocupar, o método primeira mente imprime tais informações, junto com o valor do nó atual, e logo após, caso haja subárvore à esquerda e à direita, o método é chamado com uma tabulação a mais e ocupando 6 caracteres de espaço a menos. Tal método executa uma pré-ordem.

Caso o formato requerido seja o 2, o seguinte procedimento é executado:



```
1 void abb::formato2(abb *no) {
2     if(no != this) std::cout << " ";
3     std::cout << "(";
4     std::cout << no->valor;
5
6     if(no->esq != nullptr) {
7         formato2(no->esq);
8     }
9
10    if(no->dir != nullptr) {
11        formato2(no->dir);
12    }
13
14    std::cout << ")";
15 }
```

Figura 16 – Formato de Impressão 2

Neste formato, apenas com a informação do nó atual, antes de cada nó, é aberto um parêntese. Caso haja subárvore à esquerda ou à direita, o método é chamado recursivamente para tais. Após imprimir todas as subárvores, o parêntese é fechado.

3 Análises de complexidades

3.0.1 Busca

Nesse método, o algoritmo desce na árvore a partir da raiz. Dessa forma, caso o valor que desejamos buscar não esteja na árvore ou seja um nó folha, o número de passos realizados estará em função da altura. Em outras situações, teríamos valores menores que a mesma, tendo como menor número de passos quando queremos buscar o valor da raiz, o que nos leva a concluir que o algoritmo é $O(h)$.

3.0.2 Inserção

Similar ao método de busca, o método de inserção depende diretamente da altura da árvore. Uma vez que ao chegar em um nível desta, decidimos se vamos para esquerda ou direita, até chegarmos ao último nível. Portanto, a complexidade de melhor se dá por $\Omega(h)$ e pior caso de $O(h)$, onde h é a altura da árvore. Logo sua complexidade é $\Theta(h)$.

3.0.3 Remoção

No melhor caso, a raiz da árvore é o elemento que temos que remover e esta não tem nenhum filho. Portanto, no melhor caso, temos que a complexidade é $\Omega(1)$. Já no pior caso, devemos encontrar o nó a ser removido e este é uma folha, portanto devemos descer toda a árvore, assim como na inserção. Logo seu pior caso é $O(h)$.

3.0.4 Enésimo Elemento

Semelhante ao método de busca, a procura pelo nó na posição enviada realiza passos que são no máximo em função da altura, quando o nó é uma folha e o algoritmo desce na árvore. Já o melhor caso é quando encontramos o enésimo elemento já na raiz. Assim, a complexidade é $O(h)$.

3.0.5 Posição

Mais um algoritmo com a complexidade semelhante ao de busca e afins, pois descemos pela árvore em busca de um valor específico. Logo, a complexidade também será $O(h)$.

3.0.6 Mediana

O método da mediana calcula, primeiramente, qual seria o elemento do meio caso a árvore binária de busca fosse representada de forma unidimensional com percurso de ordem simétrica.

Ao calcular o índice que está a mediana em $\Theta(1)$, é chamada a função de enésimo elemento que possui complexidade $O(h)$. Logo, a complexidade da mediana seria a mesma complexidade de enésimo elemento, $O(h)$.

3.0.7 Média

O método da média primeiramente efetua uma busca para o nó que possui como chave o valor passado no parâmetro. Caso não encontre o valor, a complexidade da média seria a mesma da busca, $O(h)$. Caso encontre o valor, o algoritmo percorrerá todos os elementos que pertencem à subárvore que possui como chave o valor passado pelo parâmetro. Logo, no pior caso, temos que a complexidade é $O(n)$ se o valor passado seja a raiz da árvore.

3.0.8 É cheia

O método é cheia tem complexidade $\Theta(1)$, visto que apenas, ao chamar uma função que apenas efetua uma soma em $\Theta(1)$ e logo após isso compara com o valor máximo de nós na árvore que também possui complexidade $\Theta(1)$.

3.0.9 É completa

O método é completa é análogo ao método é cheia, tendo complexidade $\Theta(1)$.

3.0.10 Pré-Ordem

O método de pré-ordem foi feito de maneira iterativa. Para imprimir todos os nós da árvore nesta ordem específica, ainda temos de passar por todos eles, logo a complexidade de melhor e pior caso são $\Omega(n)$ e $O(n)$ respectivamente, logo sua complexidade é $\Theta(n)$, onde n é a quantidade de nós da árvore.

3.0.11 Imprimir Árvore

Assim como na pré-ordem, os métodos de imprimir a árvore deve passar por todos os nós da árvore. E suas impressões (de cada nó) são feitas em tempo constante. Assim, suas complexidades de melhor e pior caso são $\Omega(n)$ e $O(n)$, logo, $\Theta(n)$.

4 Conclusão

De maneira geral, foi possível observar que a estrutura de dados Árvore Binária de Busca de fato melhora o desempenho de algumas operações em relação às estruturas de dados lineares. Visto que o tempo de fazer buscas na estrutura, uma vez que respeitadas suas propriedades, crescem proporcionalmente à sua altura, não ao número de elementos. Assim, dependendo de como o usuário insere valores nas árvores, a árvore pode ter uma altura de $\log_2 n$, mostrando o quão melhor o desempenho será.

Também foi possível perceber que algumas das funções extras requeridas no trabalho derivam diretamente das básicas, o que ajuda em sua implementação.