JOSÉ DAVI VIANA FRANCELINO
THIAGO DE OLIVEIRA CORDEIRO
RAFAEL SILVA FREIRE

Implementação de uma árvore binária de busca aumentada

Natal, RN
2022

JOSÉ DAVI VIANA FRANCELINO
THIAGO DE OLIVEIRA CORDEIRO
RAFAEL SILVA FREIRE

Implementação de uma árvore binária de busca aumentada

Relatório técnico apresentado à disciplina de Estruturas de Dados Básicas II, como requisito parcial para obtenção de nota referente à unidade II.

Natal, RN
2022

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

O presente relatório tem como finalidade descrever a realização de um projeto da disciplina Estruturas de Dados Básicas II. O objetivo do projeto foi desenvolver uma árvore binária de busca aumentada, que, além de conter as operações convencionais (`search`, `insert` e `remove`), deveria suportar os seguintes métodos:

1. Método `get`

   Parâmetros:

   - `index`: o índice do elemento.

   Retorno: o elemento na posição `index` do percurso em ordem simétrica da árvore (contando a partir do 1).

2. Método `position`

   Parâmetros:

   - `key`: o elemento a ser procurada na árvore.

   Retorno: a posição ocupada pelo elemento `key` em um percurso em ordem simétrica na árvore.

3. Método `median`

   Retorno: o elemento que contém a mediana da árvore, considerando o menor no caso em que ela possui um número par de elementos.

4. Método `is_full`

   Retorno: verdadeiro se a árvore for cheia e falso caso contrário.

5. Método `is_complete`

   Retorno: verdadeiro se a árvore for completa e falso caso contrário.

6. Método `to_string`

   Retorno: uma representação por nível da árvore como string.

Para a implementação desses algoritmos, foi utilizada a linguagem de programação C++, no padrão ISO/IEC 14882:2017, ou apenas C++17.

Ademais, com o fito de desenvolver os algoritmos de modo mais eficiente, a árvore foi implementada guardando informações adicionais em cada nó. Além de armazenar o valor da chave e ponteiros para as subárvores esquerda e direita, foram armazenadas a altura do nó e a quantidade de nós à esquerda e à direita da raiz.

2 IMPLEMENTAÇÕES

2.1 Método `get`

Como é necessário percorrer a árvore em ordem simétrica, sabe-se que todos os nós à esquerda da raiz serão percorridos antes dela. Logo, o índice da raiz será igual a quantidade de nós à esquerda mais um. Além disso, pode-se comparar a quantidade de nós à esquerda com o índice para saber se o valor procurado está na subárvore esquerda ou direita. Usando essas informações, o método pôde ser desenvolvido:

```
Key get(int index) const {
    if (this->left_nodes + 1 == index)
        return this->key;
    else if (this->left_nodes >= index) {
        if (this->left != nullptr)
            return this->left->get(index);
    }
    else {
        if (this->right != nullptr)
            return this->right->get(index - this->left_nodes - 1);
    }

    throw std::out_of_range("get(): index out of range");
}
```

Figura 1: Método `get`

2.2 Método `position`

A contagem das posições da árvore em ordem simétrica começa pela subárvore à esquerda. Logo, para saber a posição de um certo elemento, ele é comparado à raiz. Se ele for igual, basta somar a quantidade de nós a esquerda, acrescentar um e teremos a

posição que a raiz ocupa numa contagem em ordem simétrica. Caso ele seja menor, o índice do elemento é o mesmo que o seu índice na subárvore esquerda. Se ele for maior, é necessário saber o seu índice na subárvore direita, somá-lo à quantidade de nós à esquerda e acrescentar um para obter o índice do elemento na árvore.

```
int position(Key key) const {
    if (key < this->key) {
        if(this->left != nullptr) {
            return this->left->position(key);
        }
    }
    else if (key > this->key) {
        if(this->right != nullptr) {
            return 1 + this->left_nodes + this->right->position(key);
        }
    }
    else
        return this->left_nodes + 1;

    throw std::out_of_range("position(): key not found");
}
```

Figura 2: Método `position`

2.3 Método `median`

Antes de implementar o `median`, foi implementado o método `size`, cujo retorno é a quantidade de nós na árvore. Esse valor é obtido através da soma da quantidade de nós à esquerda com a quantidade de nós à direita com o acréscimo um.

Para descobrir a mediana da árvore, analisa-se o número de nós na árvore. Caso ele seja par, retorna-se o elemento na posição correspondente à quantidade de nós dividido por 2 (assim o menor elemento entre os 2 do meio é retornado). Caso a quantidade de nós seja ímpar, retorna-se o elemento na posição correspondente à quantidade de nós dividido por 2 mais um, ou seja, o elemento do meio.

```
size_t size() const {
    if (this == nullptr)
        return 0;
    else
        return this->right_nodes + 1 + this->left_nodes;
}


Key median() const {
    auto qnt_nodes = this->size();
    if (qnt_nodes % 2 == 0) {
        return this->get(qnt_nodes / 2);
    } else {
        return this->get(qnt_nodes / 2 + 1);
    }
}
```

Figura 3: Método `median`

2.4 Método `is_full`

Para saber se uma árvore é cheia, simplesmente compara-se a quantidade de nós dela com a sua altura, da seguinte maneira:

```
bool is_full() const {
    return this->size() == pow(2, height) - 1;
}
```

Figura 4: Método `is_full`

2.5 Método `is_complete`

Para saber se a árvore é completa, percorre-se-a nível a nível e é checado se existe algum nó com subárvore vazia em algum nível que não é o último ou o penúltimo. Isso pôde ser feito criando uma variável que armazena inicialmente a altura da árvore, diminuindo seu valor em um a cada nó acessado e checando, quando um nó com subárvore vazia é encontrado, se o seu valor é menor ou igual a dois.

```
bool is_complete_aux(int level) const {
    if(this->left != nullptr && this->right != nullptr) {
        return this->left->is_complete_aux(--level) && this->right->
   is_complete_aux(--level);
    }
    return level <= 2;
}

bool is_complete() const {
    return this->is_complete_aux(this->height);
}
```

Figura 5: Método `is_complete`

2.6 Método `to_string`

Para encontrar a string com o valor das chaves na sequência de visitação por nível, utiliza-se uma fila. A princípio, a fila armazena apenas a raiz e, a cada iteração, além de adicionar o valor do nó atual à string, seus filhos são inseridos no fim.

```
std::string to_string() const {
    std::queue<Tree> q;
    q.push(*this);
    std::ostringstream oss;

    while(!q.empty()) {
        auto t = q.front();
        q.pop();
        oss << t.key << " ";
        if (t.left != nullptr) {
            q.push(*t.left);
        }
        if (t.right != nullptr) {
            q.push(*t.right);
        }
    }

    oss << "\n";
    return oss.str();
}
```

Figura 6: Método `to_string`

3 ANÁLISE DE COMPLEXIDADE

3.1 Método `get`

A chamada recursiva do método `get`, dependendo do índice e da estrutura da árvore, acontece apenas na esquerda ou na direita, mas nunca em ambos. Logo, há apenas uma chamada do método em cada nível da árvore, ou seja, o método será chamado no máximo h (altura) vezes. Portanto, a complexidade assintótica do `get` é $O(h)$.

3.2 Método `position`

Do mesmo modo que o `get`, o método `position` é chamado no máximo uma vez por nível, então sua complexidade também é $O(h)$.

3.3 Método `median`

A complexidade do `median` depende apenas das complexidades dos métodos `size` e `get`. Como a complexidade do `get` é maior, podemos dizer que a complexidade do `median` é $O(h)$.

3.4 Método `is_full`

O método `is_full` depende apenas do método `size`, que tem complexidade $\Theta(1)$, e da função `pow`, que também tem complexidade $\Theta(1)$. Logo, a complexidade assintótica de `is_full` é $\Theta(1)$.

3.5 Método `is_complete`

O método `is_complete` consiste em apenas uma chamada ao `is_complete_aux`, logo, eles possuem a mesma complexidade. A chamada recursiva do método `is_complete_aux` ou acontece na direita e na esquerda ou não acontece. Logo, o método é chamado no máximo uma vez por nó. Desse modo, sua complexidade assintótica é $O(n)$.

3.6 Método `to_string`

No método `to_string`, adicionamos os nós da árvore numa estrutura de dados fila e a percorremos num laço. Logo, a complexidade do método é $\Theta(n)$.

4 CONCLUSÃO

Através do desenvolvimento desse projeto, foi possível consolidar o conhecimento obtido na disciplina Estrutura de Dados Básicas II, em especial o conteúdo de árvores binárias de busca. Outrossim, aprimorou-se as habilidades de desenvolvimento de algoritmos, já que foi necessário buscar formas eficientes de implementar os métodos solicitados.