ED2 - Aula 8.pdf e Aula 9.pdf - Rafael Manfrim

Aula 8

1. Qual é a complexidade computacional, no pior caso, do tempo para inserir um novo nó em uma Árvore AVL? Em que situação ocorre o pior caso?

A altura h da árvore AVL é mantida aproximadamente em log(n), para encontrar a posição correta para inserir o novo nó, percorremos a árvore a partir da raiz até uma folha, o que leva no máximo O(h). Caso tenhamos rotações para rebalanceamento, não altera o tempo de execução, pois elas executam em O(1).

2. Qual é a complexidade computacional, no pior caso, do tempo para buscar um nó em uma Árvore AVL? Em que situação ocorre o pior caso?

Em uma árvore AVL, no pior caso, precisamos andar pela altura toda dela para buscar um elemento, ou seja, O(h), que é o mesmo que O(log n). O pior caso ocorre quando o elemento está em uma das folhas ou não está presente na árvore.

3. Implemente em C/C++ os códigos para rotação à esquerda e rotação à direita. Garanta que os fatores de balanceamento sejam atualizados durante o processo de rotação.

```
void rotacao_a_esquerda(ArvoreAVL* arvore, No* no) {
    if(no->no_dir == nullptr) {
        return;
    }

    No* no_dir = no->no_dir;
    no->no_dir = no_dir->no_esq;

    if (no_dir->no_esq != nullptr) {
        no_dir->no_esq->no_pai = no;
    }

    no_dir->no_pai = no->no_pai;

    if (no->no_pai == nullptr) {
        arvore->raiz = no_dir;
    } else if (no == no->no_pai->no_esq) {
        no->no_pai->no_esq = no_dir;
    } else {
        no->no_pai->no_dir = no_dir;
    }
    no_dir->no_esq = no;
    no->no_pai = no_dir;

    no->balanceamento = no->balanceamento - 1 - max(0, no_dir->balanceamento);
    no_dir->balanceamento = no_dir->balanceamento - 1 + min(0, no->balanceamento);
}

void rotacao_a_direita(ArvoreAVL* arvore, No* no) {
    if(no->no_esq == nullptr) {
        return;
    }
```

```
    No* no_esq = no->no_esq;
    no->no_esq = no_esq->no_dir;

    if (no_esq->no_dir != nullptr) {
        no_esq->no_dir->no_pai = no;
    }

    no_esq->no_pai = no->no_pai;

    if (no->no_pai == nullptr) {
        arvore->raiz = no_esq;
    } else if (no == no->no_pai->no_dir) {
        no->no_pai->no_dir = no_esq;
    } else {
        no->no_pai->no_esq = no_esq;
    }
    no_esq->no_dir = no;
    no->no_pai = no_esq;

    no->balanceamento = no->balanceamento + 1 - min(0, no_esq->balanceamento);
    no_esq->balanceamento = no_esq->balanceamento + 1 + max(0, no->balanceamento);
}
```

4. Implemente em C/C++ as funções ATUALIZACAO-BALANCEAMENTOS e REBALANCEARSUBARVORE. Na função REBALANCEARSUBARVORE, assegure-se de tratar os quatro casos distintos de desbalanceamento que podem ocorrer em uma árvore AVL.

```
void rebalancear_subarvore(ArvoreAVL* arvore, No* no_pai) {
    if (no_pai->balanceamento == -2) { // Caso Esquerda-Esquerda ou Esquerda-Direita
        if (no_pai->no_esq->balanceamento <= 0) { // Esquerda-Esquerda
            rotacao_a_direita(arvore, no_pai);
        } else { // Esquerda-Direita
            rotacao_a_esquerda(arvore, no_pai->no_esq);
            rotacao_a_direita(arvore, no_pai);
        }
    } else if (no_pai->balanceamento == 2) { // Caso Direita-Direita ou Direita-Esquerda
        if (no_pai->no_dir->balanceamento >= 0) { // Direita-Direita
            rotacao_a_esquerda(arvore, no_pai);
        } else { // Direita-Esquerda
            rotacao_a_direita(arvore, no_pai->no_dir);
            rotacao_a_esquerda(arvore, no_pai);
        }
    }
}

void atualizar_balanceamentos(ArvoreAVL* arvore, No* no_inserido) {
    No* no_pai = no_inserido->no_pai;

    while (no_pai != nullptr) { // Atualiza os fatores de balanceamento ao subir na árvore
        if (no_inserido == no_pai->no_esq) {
            no_pai->balanceamento--; // Filho esquerdo reduz o fator
        } else {
            no_pai->balanceamento++; // Filho direito aumenta o fator
        }

        if (no_pai->balanceamento == 0) { // Se o fator de balanceamento for 0, a subárvore
está equilibrada
            return;
        }
```

```
        if (no_pai->balanceamento == 2 || no_pai->balanceamento == -2) { // Se o fator de
balanceamento for ±2, precisamos rebalancear
            rebalancear_subarvore(arvore, no_pai);
            return;
        }

        no_inserido = no_pai;
        no_pai = no_pai->no_pai;
    }
}
```

5. Finalmente, implemente a Árvore AVL e sua operação de inserção balanceada em C/C++. Ou seja, incluindo os códigos de rotação e atualização dos fatores de balanceamento.

```
#include <iostream>

using namespace std;

struct No {
    No* no_esq;
    No* no_dir;
    No* no_pai;
    int valor;
    int balanceamento = 0;
};

struct ArvoreAVL {
    No* raiz = nullptr;
};

void percurso_em_ordem(No* no) {
    if(no != nullptr){
        percurso_em_ordem(no->no_esq);
        cout << no->valor << " ";
        percurso_em_ordem(no->no_dir);
    }
}

No* BuscaNaArvore(No* no, int valor) {
    while(no != nullptr && valor != no->valor) {
        if(valor < no->valor) {
            no = no->no_esq;
        } else {
            no = no->no_dir;
        }
    }

    return no;
}

void rotacao_a_esquerda(ArvoreAVL* arvore, No* no) {
    if(no->no_dir == nullptr) {
        return;
    }

    No* no_dir = no->no_dir;
    no->no_dir = no_dir->no_esq;

    if (no_dir->no_esq != nullptr) {
        no_dir->no_esq->no_pai = no;
```

```
    }

    no_dir->no_pai = no->no_pai;

    if (no->no_pai == nullptr) {
        arvore->raiz = no_dir;
    } else if (no == no->no_pai->no_esq) {
        no->no_pai->no_esq = no_dir;
    } else {
        no->no_pai->no_dir = no_dir;
    }
    no_dir->no_esq = no;
    no->no_pai = no_dir;

    no->balanceamento = no->balanceamento - 1 - max(0, no_dir->balanceamento);
    no_dir->balanceamento = no_dir->balanceamento - 1 + min(0, no->balanceamento);
}

void rotacao_a_direita(ArvoreAVL* arvore, No* no) {
    if(no->no_esq == nullptr) {
        return;
    }

    No* no_esq = no->no_esq;
    no->no_esq = no_esq->no_dir;

    if (no_esq->no_dir != nullptr) {
        no_esq->no_dir->no_pai = no;
    }

    no_esq->no_pai = no->no_pai;

    if (no->no_pai == nullptr) {
        arvore->raiz = no_esq;
    } else if (no == no->no_pai->no_dir) {
        no->no_pai->no_dir = no_esq;
    } else {
        no->no_pai->no_esq = no_esq;
    }
    no_esq->no_dir = no;
    no->no_pai = no_esq;

    no->balanceamento = no->balanceamento + 1 - min(0, no_esq->balanceamento);
    no_esq->balanceamento = no_esq->balanceamento + 1 + max(0, no->balanceamento);
}

void rebalancear_subarvore(ArvoreAVL* arvore, No* no_pai) {
    if (no_pai->balanceamento == -2) { // Caso Esquerda-Esquerda ou Esquerda-Direita
        if (no_pai->no_esq->balanceamento <= 0) { // Esquerda-Esquerda
            rotacao_a_direita(arvore, no_pai);
        } else { // Esquerda-Direita
            rotacao_a_esquerda(arvore, no_pai->no_esq);
            rotacao_a_direita(arvore, no_pai);
        }
    } else if (no_pai->balanceamento == 2) { // Caso Direita-Direita ou Direita-Esquerda
        if (no_pai->no_dir->balanceamento >= 0) { // Direita-Direita
            rotacao_a_esquerda(arvore, no_pai);
        } else { // Direita-Esquerda
            rotacao_a_direita(arvore, no_pai->no_dir);
            rotacao_a_esquerda(arvore, no_pai);
        }
    }
```

```
}

void atualizar_balanceamentos(ArvoreAVL* arvore, No* no_inserido) {
    No* no_pai = no_inserido->no_pai;

    while (no_pai != nullptr) { // Atualiza os fatores de balanceamento ao subir na árvore
        if (no_inserido == no_pai->no_esq) {
            no_pai->balanceamento--; // Filho esquerdo reduz o fator
        } else {
            no_pai->balanceamento++; // Filho direito aumenta o fator
        }

        if (no_pai->balanceamento == 0) { // Se o fator de balanceamento for 0, a subárvore
está equilibrada
            return;
        }

        if (no_pai->balanceamento == 2 || no_pai->balanceamento == -2) { // Se o fator de
balanceamento for ±2, precisamos rebalancear
            rebalancear_subarvore(arvore, no_pai);
            return;
        }

        no_inserido = no_pai;
        no_pai = no_pai->no_pai;
    }
}

void inserir(ArvoreAVL* arvore, int valor) {
    No* novo_no = new No{nullptr, nullptr, nullptr, valor, 0};

    if (arvore->raiz == nullptr) {
        arvore->raiz = novo_no;
        return;
    }

    No* atual = arvore->raiz;
    No* pai = nullptr;

    while (atual != nullptr) {
        pai = atual;
        if (valor < atual->valor) {
            atual = atual->no_esq;
        } else {
            atual = atual->no_dir;
        }
    }

    novo_no->no_pai = pai;

    if (valor < pai->valor) {
        pai->no_esq = novo_no;
    } else {
        pai->no_dir = novo_no;
    }

    atualizar_balanceamentos(arvore, novo_no);
}

void imprimir_arvore(No* raiz, int nivel = 0) {
    if (raiz != nullptr) {
        imprimir_arvore(raiz->no_dir, nivel + 1);
```

```
        cout << string(nivel * 4, ' ') << raiz->valor << " (" << raiz->balanceamento << ")\n";
        imprimir_arvore(raiz->no_esq, nivel + 1);
    }
}



int main() {
    ArvoreAVL arvore;

    inserir(&arvore, 4);
    inserir(&arvore, 7);
    inserir(&arvore, 5);
    inserir(&arvore, 46);
    inserir(&arvore, 25);
    inserir(&arvore, 11);

    cout << "Raiz: " << arvore.raiz->valor << endl;

    cout << "Percurso em Ordem: ";
    percurso_em_ordem(arvore.raiz);
    cout << endl << endl;

    cout << "Árvore: " << endl;
    imprimir_arvore(arvore.raiz);

    return 0;
}
```

Aula 9

1. Qual é a complexidade computacional, no pior caso, do tempo para deletar um nó em uma Árvore AVL? Em que situação ocorre o pior caso?

Assim como na inserção, a deleção também tem complexidade computacional de O(h), que também pode ser escrita como O(log n), pois é preciso andar pela altura da árvore em busca do nó desejado. Caso seja necessário fazer o balanceamento da árvore a complexidade não é alterada pois as rotações executam em O(1).

2. Descrevam os cinco casos simétricos não analisados nos slides anteriores. Descrevam suas soluções.

Em todos os cinco casos simétricos, estaremos deletando um nó da subárvore direita do nó P e este nó possui um fator de balanceamento de -1. Portanto, a deleção de um nó da sua subárvore direita resultará em P aumentar seu fator de balanceamento para -2.

Caso 1:
Sua árvore esquerda possui um fator de balanceamento de -1.
Este desbalanceamento é solucionado através de uma rotação para à direita.

Caso 2:
Sua árvore esquerda possui um fator de balanceamento de 0.
Como no Caso 1, este desbalanceamento é solucionado através de uma rotação para à direita.

Caso 3:
Sua subárvore esquerda (raiz em Q) possui um fator de balanceamento de -1.

```
A subárvore esquerda de Q (raiz em R) possui um fator de balanceamento de +1.
Para solucionar este desbalanceamento, rotacionamos Q para a esquerda e, em seguida,
rotacionamos P para a direita.

Caso 4:
Sua subárvore esquerda (raiz em Q) possui um fator de balanceamento de +1.
A subárvore direita de Q (raiz em R) possui um fator de balanceamento de -1.
Como no caso anterior, para solucionar este desbalanceamento, basta rotacionarmos Q para a
esquerda e, em seguida, rotacionar P para a direita.

Caso 5:
Sua subárvore esquerda (raiz em Q) possui um fator de balanceamento de +1.
A subárvore direita de Q (raiz em R) possui um fator de balanceamento de 0.
Para solucionar este desbalanceamento, é necessário rotacionar Q para a esquerda e, em seguida,
rotacionar P para a direita.
```

3. Implemente em C/C++ a operação de deleção balanceada da Árvore AVL. Adicione esse código aos códigos da aula anterior. Dica: utilize uma abordagem recursiva.

```cpp
#include <iostream>

using namespace std;

struct No {
    No* no_esq;
    No* no_dir;
    No* no_pai;
    int valor;
    int balanceamento = 0;
};

struct ArvoreAVL {
    No* raiz = nullptr;
};

void percurso_em_ordem(No* no) {
    if(no != nullptr){
        percurso_em_ordem(no->no_esq);
        cout << no->valor << " ";
        percurso_em_ordem(no->no_dir);
    }
}

No* BuscaNaArvore(No* no, int valor) {
    while(no != nullptr && valor != no->valor) {
        if(valor < no->valor) {
            no = no->no_esq;
        } else {
            no = no->no_dir;
        }
    }

    return no;
}

No* ArvoreMinimoRecursivo(No* no) {
    if(no->no_esq != nullptr) {
        return ArvoreMinimoRecursivo(no->no_esq);
```

```
    } else {
        return no;
    }
}

No* EncontraSucessor(No* no) {
    if(no->no_dir != nullptr) {
        return ArvoreMinimoRecursivo(no->no_dir);
    }

    No* no_pai = no->no_pai;
    while (no_pai != nullptr && no == no_pai->no_dir) {
        no = no_pai;
        no_pai = no_pai->no_pai;
    }
    return no_pai;
}

void rotacao_a_esquerda(ArvoreAVL* arvore, No* no) {
    if(no->no_dir == nullptr) {
        return;
    }

    No* no_dir = no->no_dir;
    no->no_dir = no_dir->no_esq;

    if (no_dir->no_esq != nullptr) {
        no_dir->no_esq->no_pai = no;
    }

    no_dir->no_pai = no->no_pai;

    if (no->no_pai == nullptr) {
        arvore->raiz = no_dir;
    } else if (no == no->no_pai->no_esq) {
        no->no_pai->no_esq = no_dir;
    } else {
        no->no_pai->no_dir = no_dir;
    }
    no_dir->no_esq = no;
    no->no_pai = no_dir;

    no->balanceamento = no->balanceamento - 1 - max(0, no_dir->balanceamento);
    no_dir->balanceamento = no_dir->balanceamento - 1 + min(0, no->balanceamento);
}

void rotacao_a_direita(ArvoreAVL* arvore, No* no) {
    if(no->no_esq == nullptr) {
        return;
    }

    No* no_esq = no->no_esq;
    no->no_esq = no_esq->no_dir;

    if (no_esq->no_dir != nullptr) {
        no_esq->no_dir->no_pai = no;
    }

    no_esq->no_pai = no->no_pai;

    if (no->no_pai == nullptr) {
        arvore->raiz = no_esq;
```

```
    } else if (no == no->no_pai->no_dir) {
        no->no_pai->no_dir = no_esq;
    } else {
        no->no_pai->no_esq = no_esq;
    }
    no_esq->no_dir = no;
    no->no_pai = no_esq;

    no->balanceamento = no->balanceamento + 1 - min(0, no_esq->balanceamento);
    no_esq->balanceamento = no_esq->balanceamento + 1 + max(0, no->balanceamento);
}

void rebalancear_subarvore(ArvoreAVL* arvore, No* no_pai) {
    if (no_pai->balanceamento == -2) { // Caso Esquerda-Esquerda ou Esquerda-Direita
        if (no_pai->no_esq->balanceamento <= 0) { // Esquerda-Esquerda
            rotacao_a_direita(arvore, no_pai);
        } else { // Esquerda-Direita
            rotacao_a_esquerda(arvore, no_pai->no_esq);
            rotacao_a_direita(arvore, no_pai);
        }
    } else if (no_pai->balanceamento == 2) { // Caso Direita-Direita ou Direita-Esquerda
        if (no_pai->no_dir->balanceamento >= 0) { // Direita-Direita
            rotacao_a_esquerda(arvore, no_pai);
        } else { // Direita-Esquerda
            rotacao_a_direita(arvore, no_pai->no_dir);
            rotacao_a_esquerda(arvore, no_pai);
        }
    }
}

void atualizar_balanceamentos(ArvoreAVL* arvore, No* no_inserido) {
    No* no_pai = no_inserido->no_pai;

    while (no_pai != nullptr) { // Atualiza os fatores de balanceamento ao subir na árvore
        if (no_inserido == no_pai->no_esq) {
            no_pai->balanceamento--; // Filho esquerdo reduz o fator
        } else {
            no_pai->balanceamento++; // Filho direito aumenta o fator
        }

        if (no_pai->balanceamento == 0) { // Se o fator de balanceamento for 0, a subárvore
está equilibrada
            return;
        }

        if (no_pai->balanceamento == 2 || no_pai->balanceamento == -2) { // Se o fator de
balanceamento for ±2, precisamos rebalancear
            rebalancear_subarvore(arvore, no_pai);
            return;
        }

        no_inserido = no_pai;
        no_pai = no_pai->no_pai;
    }
}

void inserir(ArvoreAVL* arvore, int valor) {
    No* novo_no = new No{nullptr, nullptr, nullptr, valor, 0};

    if (arvore->raiz == nullptr) {
        arvore->raiz = novo_no;
        return;
```

```
    }

    No* atual = arvore->raiz;
    No* pai = nullptr;

    while (atual != nullptr) {
        pai = atual;
        if (valor < atual->valor) {
            atual = atual->no_esq;
        } else {
            atual = atual->no_dir;
        }
    }

    novo_no->no_pai = pai;

    if (valor < pai->valor) {
        pai->no_esq = novo_no;
    } else {
        pai->no_dir = novo_no;
    }

    atualizar_balanceamentos(arvore, novo_no);
}

No* delecao(ArvoreAVL* arvore, No* no, int valor) {
    if (no == nullptr) return no;

    if (valor < no->valor) {
        delecao(arvore, no->no_esq, valor);
    } else if (valor > no->valor) {
        delecao(arvore, no->no_dir, valor);
    } else {
        if (no->no_esq == nullptr && no->no_dir == nullptr) { //Nó folha
            if (no->no_pai == nullptr) {
                arvore->raiz = nullptr;
            } else if (no == no->no_pai->no_esq) {
                no->no_pai->no_esq = nullptr;
            } else {
                no->no_pai->no_dir = nullptr;
            }
            delete no;
        } else if (no->no_esq == nullptr || no->no_dir == nullptr) { // Um filho
            No* filho = (no->no_esq != nullptr) ? no->no_esq : no->no_dir;

            if (no->no_pai == nullptr) {
                arvore->raiz = filho;
            } else if (no == no->no_pai->no_esq) {
                no->no_pai->no_esq = filho;
            } else {
                no->no_pai->no_dir = filho;
            }
            filho->no_pai = no->no_pai;
            delete no;
        } else { // Dois filhos
            No* sucessor = EncontraSucessor(no);
            no->valor = sucessor->valor;
            delecao(arvore, sucessor, sucessor->valor);
        }
    }

    if (no == nullptr) return no;
```

```
    no->balanceamento = 1 + max(no->no_esq->balanceamento, no->no_dir->balanceamento);

    if (no->balanceamento > 1 && no->no_esq->balanceamento >= 0)
        rotacao_a_direita(arvore, no);

    if (no->balanceamento > 1 && no->no_esq->balanceamento < 0) {
        rotacao_a_esquerda(arvore, no->no_esq);
        rotacao_a_direita(arvore, no);
    }

    if (no->balanceamento < -1 && no->no_dir->balanceamento <= 0)
        rotacao_a_esquerda(arvore, no);

    if (no->balanceamento < -1 && no->no_dir->balanceamento > 0) {
        rotacao_a_direita(arvore, no->no_dir);
        rotacao_a_esquerda(arvore, no);
    }

    return no;
}

void imprimir_arvore(No* raiz, int nivel = 0) {
    if (raiz != nullptr) {
        imprimir_arvore(raiz->no_dir, nivel + 1);
        cout << string(nivel * 4, ' ') << raiz->valor << " (" << raiz->balanceamento << ")\n";
        imprimir_arvore(raiz->no_esq, nivel + 1);
    }
}

int main() {
    ArvoreAVL arvore;

    inserir(&arvore, 4);
    inserir(&arvore, 7);
    inserir(&arvore, 5);
    inserir(&arvore, 46);
    inserir(&arvore, 25);
    inserir(&arvore, 11);
    inserir(&arvore, 18);
    inserir(&arvore, 33);
    inserir(&arvore, 38);
    inserir(&arvore, 2);
    inserir(&arvore, 51);

    cout << "Raiz: " << arvore.raiz->valor << endl;

    cout << "Percurso em Ordem: ";
    percurso_em_ordem(arvore.raiz);
    cout << endl << endl;

    cout << "Árvore: " << endl;
    imprimir_arvore(arvore.raiz);

    delecao(&arvore, arvore.raiz, 33);

    cout << "Árvore: " << endl;
    imprimir_arvore(arvore.raiz);

    return 0;
}
```