ED2 - Aula 7.pdf - Rafael Manfrim

Implemente a Árvore Binária de Busca e suas operações básicas em C/C++.

```
#include <iostream>

using namespace std;

struct No {
    No* no_esq;
    No* no_dir;
    No* no_pai;
    int valor;
};

struct ArvoreBinariaBusca {
    No* raiz;
};

void PercursoEmOrdem(No* raiz) {
    if(raiz != nullptr){
        PercursoEmOrdem(raiz->no_esq);
        cout << raiz->valor << " ";
        PercursoEmOrdem(raiz->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* ArvoreMinimo(No* no) {
    while(no->no_esq != nullptr) {
        no = no->no_esq;
    }

    return no;
}

No* ArvoreMaximo(No* no) {
```

```
    while(no->no_dir != nullptr) {
        no = no->no_dir;
    }

    return no;
}

void InserirNaArvore(ArvoreBinariaBusca* arvore, No* no) {
    No* no_atual = arvore->raiz;

    if (no_atual == nullptr) {
        arvore->raiz = no;
    } else {
        No* no_pai = nullptr;

        while(no_atual != nullptr) {
            no_pai = no_atual;

            if(no->valor < no_atual->valor) {
                no_atual = no_atual->no_esq;
            } else {
                no_atual = no_atual->no_dir;
            }
        }

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

        no->no_pai = no_pai;
    }
}

void SubstituiSubarvore(ArvoreBinariaBusca* arvore, No* no_substituido, No*
no_substituir) {
    if(no_substituido->no_pai == nullptr) {
        arvore->raiz = no_substituir;
    } else {
        if(no_substituido == no_substituido->no_pai->no_esq) {
            no_substituido->no_pai->no_esq = no_substituir;
        } else {
            no_substituido->no_pai->no_dir = no_substituir;
        }
    }

    if(no_substituir != nullptr) {
        no_substituir->no_pai = no_substituido->no_pai;
    }
}

void DelecaoArvore(ArvoreBinariaBusca* arvore, No* no_deletar) {
```

```
    if(no_deletar->no_esq == nullptr) {
        SubstituiSubarvore(arvore, no_deletar, no_deletar->no_dir);
    } else if(no_deletar->no_dir == nullptr) {
        SubstituiSubarvore(arvore, no_deletar, no_deletar->no_esq);
    } else {
        No* sucessor = ArvoreMinimo(no_deletar->no_dir);

        if(sucessor != no_deletar->no_dir) {
            SubstituiSubarvore(arvore, sucessor, sucessor->no_dir);
            sucessor->no_dir = no_deletar->no_dir;
            sucessor->no_dir->no_pai = sucessor;
        }

        SubstituiSubarvore(arvore, no_deletar, sucessor);
        sucessor->no_esq = no_deletar->no_esq;
        sucessor->no_esq->no_pai = sucessor;
    }
}

void CriaNo(No* no, int valor) {
    no->no_dir = nullptr;
    no->no_esq = nullptr;
    no->no_pai = nullptr;
    no->valor = valor;
}


int main() {
    ArvoreBinariaBusca* arvore = new ArvoreBinariaBusca();
    arvore->raiz = nullptr;

    No* no = new No();
    CriaNo(no, 5);
    InserirNaArvore(arvore, no);

    no = new No();
    CriaNo(no, 2);
    InserirNaArvore(arvore, no);

    no = new No();
    CriaNo(no, 10);
    InserirNaArvore(arvore, no);

    no = new No();
    CriaNo(no, 8);
    InserirNaArvore(arvore, no);

    no = new No();
    CriaNo(no, 7);
    InserirNaArvore(arvore, no);

    cout << "Percurso em Ordem: ";
    PercursoEmOrdem(arvore->raiz);
```

```
    cout << endl;

    No* no_menor = ArvoreMinimo(arvore->raiz);
    cout << "Posição na memória do menor nó: " << no_menor << ", valor: " <<
no_menor->valor << endl;

    No* no_maior = ArvoreMaximo(arvore->raiz);
    cout << "Posição na memória do menor nó: " << no_maior << ", valor: " <<
no_maior->valor << endl;

    No* no_buscado = BuscaNaArvore(arvore->raiz, 7);
    cout << "Posição na memória do nó buscado: " << no_buscado << ", valor: " <<
no_buscado->valor << endl;
    cout << "Excluindo o nó que buscamos!" << endl;
    DelecaoArvore(arvore, no_buscado);

    cout << "Percurso em Ordem sem o nó excluído: ";
    PercursoEmOrdem(arvore->raiz);

    delete arvore;

    return 0;
}
```

Implemente uma função para encontrar o nó sucessor dada uma chave k. O sucessor, dada uma chave k é o nó com a menor chave maior que k. Também implemente uma função para encontrar o nó predecessor, este sendo o nó com a maior chave menor que k.

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

No* EncontraPredecessor(No* no) {
    if(no->no_esq != nullptr) {
        return ArvoreMaximo(no->no_esq);
    }
    No* no_pai = no->no_pai;
    while (no_pai != nullptr && no == no_pai->no_esq) {
        no = no_pai;
        no_pai = no_pai->no_pai;
```

```
    }    return no_pai;
}
```

Implemente a função de Busca recursivamente.

```
No* BuscaRecursivaNaArvore(No* no, int valor) {
    if(valor < no->valor) {
        return BuscaRecursivaNaArvore(no->no_esq, valor);
    } else if (valor > no->valor) {
        return BuscaRecursivaNaArvore(no->no_dir, valor);
    } else {
        return no;
    }
}
```

Implemente as funções de encontrar mínimo e máximo recursivamente.

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

No* ArvoreMaximoRecursiva(No* no) {
    if(no->no_dir != nullptr) {
        return ArvoreMaximoRecursiva(no->no_dir);
    } else {
        return no;
    }
}
```

Implemente a função de Inserção recursivamente.

```
No* InserirNaArvoreRecursiva(No* subarvore, No* no_pai, No* no) {
    if(subarvore == nullptr) {
        no->no_pai = no_pai;
        return no;
    }
    if(no->valor < subarvore->valor) {
        subarvore->no_esq = InserirNaArvoreRecursiva(subarvore->no_esq, subarvore,
no);
    } else if(no->valor > subarvore->valor) {
        subarvore->no_dir = InserirNaArvoreRecursiva(subarvore->no_dir, subarvore,
no);
    }
    return subarvore;
}
```

Está correto afirmar que se um nó em uma Árvore Binária de Busca tem dois filhos, então seu sucessor não possui filho à esquerda e seu predecessor não possui filho à direita. Por quê?

Sim, pois em uma Árvore Binária de Busca, o sucessor de um filho é o elemento mais à esquerda dos elementos que estão a sua direita, se houvesse um elemento ainda mais à esquerda, esse elemento deveria ser o sucessor, visto que seria maior do que a raíz e menor do que todos os outros. O mesmo vale para o predecessor, que é o filho mais à direita dos elementos que estão à esquerda do nó, se houvesse um elemento ainda mais à direita, ele sim seria o predecessor, pois continuaria sendo menor que o elemento raíz dessa busca e o maior dentre os elementos à esquerda.

Podemos criar um algoritmo de ordenação utilizando a Árvore Binária de Busca. Basta inserir cada elemento na árvore usando a função de Inserção e então executar a função de Percurso em Ordem para imprimir os valores de maneira ordenada. Qual o pior caso deste algoritmo? Qual o melhor caso? Demonstre os resultados.

O percurso em ordem executa em O(n) (mais especificamente 0(n)), já que o algoritmo é chamado duas vezes recursivamente para cada nó. E a inserção executa em O(h). Afinal, no pior caso teremos que percorrer, efetuando comparações, por toda a altura da árvore até inserir o novo nó. Contudo, a Árvore Binária de Busa pode ficar degenerada, ou seja, h = n.

O melhor caso seria quando a arvore estiver perfeitamente balanceada, onde h = log n, e a ordenação seria O(n log n). E o pior caso é quando h = n, ou seja, a ordenação executaria em $O(n \times h) = O(n \times n) = O(n^2)$.