Árvore Binária de Busca

Árvore Binária de Busca (ABB) é uma estrutura de dados em árvore onde cada nó possui no máximo dois filhos (esquerdo e direito), e a propriedade de busca é mantida: todos os elementos na subárvore esquerda são menores que o nó raiz, e todos os elementos na subárvore direita são maiores que o nó raiz.

As principais operações em uma ABB são inserção, busca e remoção, que possuem as seguintes complexidades:

Inserção: O(log n) em média, O(n) no pior caso.
Busca: O(log n) em média, O(n) no pior caso.
Remoção: O(log n) em média, O(n) no pior caso.

Vale ressaltar que as complexidades dependem da altura e, por não possuir nenhuma propriedade que balanceie a árvore, na pior das hipóteses, a altura da árvore será N.

Função, em JAVA, para inserir um novo nó:

```
    static node insert(node node, int key)
    {
        // Se a árvore é vazia, retorna um novo nó
        if (node == null)
            return newNode(key);

        // Se não, percorremos a árvore
        if (key < node.key) {
            node.left = insert(node.left, key);
        }
        else if (key > node.key) {
            node.right = insert(node.right, key);
        }

        //Retorna o nó
        return node;
    }
```

Função, em JAVA, para deletar um nó um novo nó:

```
static node deleteNode(node root, int key)
    {
        // Caso base
        if (root == null)
            return root;

        // Se o valor a ser deletado é menor que a chave atual,
então está na sub-árvore esquerda.
        if (key < root.key) {
```

```
            root.left = deleteNode(root.left, key);
        }

        // Se o valor a ser deletado é maior que a chave atual,
então está na sub-árvore direita.
        else if (key > root.key) {

            root.right = deleteNode(root.right, key);
        }

        // Se é o mesmo valor, então é o nó que deve ser deletado
        else {

            // Nó com um ou nenhum filho
            if (root.left == null) {
                node temp = root.right;
                return temp;
            }
            else if (root.right == null) {
                node temp = root.left;
                return temp;
            }

            // No com dois filhos: Pegamos o menor valor da
sub-árvore a direita
            node temp = minValueNode(root.right);

            // Copia o sucessor
            root.key = temp.key;

            // Deleta o sucessor
            root.right = deleteNode(root.right, temp.key);
        }
        return root;
    }
```

Função, em JAVA, para descobrir o menor valor de uma árvore:

```
    static node minValueNode(node node)
    {
        node current = node;

        // Loop down to find the leftmost leaf
        while (current != null && current.left != null)
            current = current.left;

        return current;
    }
```

Aplicações de BST:
Algoritmos de Grafos: BSTs podem ser usados para implementar algoritmos de grafos, como em algoritmos de árvore geradora mínima.
Filas de Prioridade: BSTs podem ser usados para implementar filas de prioridade, onde o elemento com a maior prioridade está na raiz da árvore e elementos com prioridade menor são armazenados nas subárvores.
Árvore de busca binária auto-equilibrada: BSTs podem ser usados como estruturas de dados auto-equilibradas, como a árvore AVL e a árvore rubro-negra.
Armazenamento e recuperação de dados: BSTs podem ser usados para armazenar e recuperar dados rapidamente, como em bancos de dados, onde a busca por um registro específico pode ser feita em tempo logarítmico.

Vantagens:
Busca rápida: Buscar um valor específico em uma BST tem uma complexidade de tempo média de O(log n), onde n é o número de nós na árvore. Isso é muito mais rápido do que buscar um elemento em uma matriz ou lista ligada, que têm uma complexidade de tempo de O(n) no pior caso.
Traversal in-order: BSTs podem ser percorridos in-order, visitando a subárvore esquerda, a raiz e a subárvore direita. Isso pode ser usado para classificar um conjunto de dados.
Eficiente em espaço: BSTs são eficientes em espaço, pois não armazenam informações redundante, ao contrário de matrizes e listas ligadas.
Desvantagens:
Árvores distorcidas: Se uma árvore se tornar distorcida, a complexidade de tempo de busca, inserção e eliminação será O(n) em vez de O(log n), o que pode tornar a árvore ineficiente.
Tempo adicional requerido: Árvores auto-equilibradas requerem tempo adicional para manter o equilíbrio durante as operações de inserção e eliminação.
Eficiência: BSTs não são eficientes para conjuntos de dados com muitos duplicados, pois vão desperdiçar espaço.