Chapter 3.2: Binary Search Trees (BST)

Chapter 3.2: Binary Search Trees (BST)

Introduction

A Binary Search Tree (BST) is a data structure that maintains a dynamic set of keys in a binary tree, where each node has the following properties:

- A key (value).
- A reference to the left child.
- A reference to the right child.

Properties of BST

1. The key in each node must be greater than all keys stored in the left sub-tree and smaller than all keys in the right sub-tree.
2. The left and right sub-trees must also be binary search trees.

Operations

Search

The search operation in a BST starts at the root and traces a path through the tree according to the key being searched. The search operation has a time complexity of O(h), where h is the height of the tree.

```java
public Value get(Key key) {
    return get(root, key);
}
```

```
private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}
```

Insertion

Inserting a new key into a BST involves tracing a path from the root to a null link according to the key being inserted and then replacing the null link with a new node containing the key.

```java
public void put(Key key, Value val) {
    root = put(root, key, val);
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;
    return x;
}
```

Deletion

Deletion in a BST can be more complex due to the need to maintain the BST properties. There are three cases to consider:

1. Deleting a node with no children (a leaf): Simply remove the node from the tree.

2. Deleting a node with one child: Remove the node and replace it with its child.

3. Deleting a node with two children: Find the node's in-order predecessor or successor, copy its value to the node to be deleted, and then delete the predecessor or successor.

```java
public void delete(Key key) {
    root = delete(root, key);
}

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    return x;
```

```
}

private Node min(Node x) {
    if (x.left == null) return x;
    else return min(x.left);
}

private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    return x;
}
```

Performance

- The average time complexity for search, insertion, and deletion is O(log n) for a balanced BST.
- In the worst case (when the tree becomes a linear chain of nodes), the time complexity for these operations degrades to O(n).

Traversal

Traversal of a BST can be done in various orders:

1. In-order Traversal: Visits the nodes in ascending order.
2. Pre-order Traversal: Visits the root before the subtrees.
3. Post-order Traversal: Visits the root after the subtrees.

In-order Traversal Example

```java
```

```java
public void inOrderTraversal(Node x) {
    if (x != null) {
        inOrderTraversal(x.left);
        System.out.println(x.key);
        inOrderTraversal(x.right);
    }
}
```

Applications

1. **Searching:** Efficiently find elements.

2. **Sorting:** In-order traversal of a BST gives elements in sorted order.

3. **Dynamic Set Operations:** Maintain a dynamic set of items with operations such as insertion, deletion, and search.

Example Implementation

Here is a complete example of a simple BST implementation in Java:

```java
public class BST<Key extends Comparable<Key>, Value> {
    private Node root;

    private class Node {
        private Key key;
        private Value val;
        private Node left, right;

        public Node(Key key, Value val) {
```

```
        this.key = key;
        this.val = val;
    }
}

public Value get(Key key) {
    return get(root, key);
}

private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}

public void put(Key key, Value val) {
    root = put(root, key, val);
}

private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;
```

```
    return x;
}

public void delete(Key key) {
    root = delete(root, key);
}

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    return x;
}

private Node min(Node x) {
    if (x.left == null) return x;
    else return min(x.left);
}
```

```
    private Node deleteMin(Node x) {
        if (x.left == null) return x.right;
        x.left = deleteMin(x.left);
        return x;
    }

    public void inOrderTraversal(Node x) {
        if (x != null) {
            inOrderTraversal(x.left);
            System.out.println(x.key);
            inOrderTraversal(x.right);
        }
    }
}
```

Conclusion

Binary Search Trees provide an efficient way to maintain a dynamic set of ordered keys. They support quick searches, insertions, and deletions, making them a fundamental data structure in computer science.