Chapter 3.3: Balanced Search Trees

Chapter 3.3: Balanced Search Trees

Introduction

Balanced Search Trees are binary search trees that automatically keep their height small in the face of arbitrary item insertions and deletions. This ensures that the time complexity for search, insertion, and deletion operations remains logarithmic.

Types of Balanced Search Trees

2-3 Trees

A 2-3 tree is a balanced search tree where every node with children has either two or three children, and all leaves are at the same depth.

- **2-node:** Has one key and two children.
- **3-node:** Has two keys and three children.

Operations

- **Search:** Similar to BST, but needs to handle 2-nodes and 3-nodes.
- **Insertion:** Adds new key to an existing 2-node or splits a 3-node into two 2-nodes.
- **Deletion:** Merges nodes or redistributes keys to maintain balance.

Red-Black Trees

A red-black tree is a binary search tree with an extra bit of storage per node to keep track of the color of the node, which can be either red or black. These color bits are used to ensure the tree remains balanced.

- **Properties:**

1. Each node is either red or black.
2. The root is always black.
3. Red nodes cannot have red children (no two reds in a row).
4. Every path from a node to its descendant leaves has the same number of black nodes.

Operations

- **Insertion:** Insert the node as red, and then adjust the tree to fix any violations of the red-black properties.
- **Deletion:** Replace the node with its in-order successor or predecessor and then fix any violations.
- **Rotations:** Used to maintain balance during insertion and deletion.

Red-Black Tree Example Implementation

Here is an example implementation of a Red-Black Tree in Java:

```java
public class RedBlackBST<Key extends Comparable<Key>, Value> {
    private static final boolean RED = true;
    private static final boolean BLACK = false;

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

        Node(Key key, Value val, boolean color) {
```

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

private Node root;

private boolean isRed(Node x) {
    if (x == null) return false;
    return x.color == RED;
}

private Node rotateLeft(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}

private Node rotateRight(Node h) {
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
```

```
    h.color = RED;
    return x;
}

private void flipColors(Node h) {
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}

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

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

    if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
    if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left) && isRed(h.right)) flipColors(h);

    return h;
```

```
    }
}
```

Performance

- Balanced search trees ensure that operations are efficient by maintaining a tree height of O(log n).
- They prevent the worst-case scenarios of unbalanced trees where operations degrade to O(n).

Applications

1. **Database Indexing:** Efficient data retrieval and updates.
2. **Memory Management:** Dynamic memory allocation and garbage collection.
3. **Network Routers:** Routing tables that require efficient searches and updates.

Conclusion

Balanced search trees such as 2-3 trees and red-black trees provide an efficient means to maintain a dynamic set of ordered keys. They ensure logarithmic time complexity for insertion, deletion, and search operations, making them highly useful in various applications requiring dynamic data management.