Search Trees

This C++ file is a powerful demonstration of two advanced, self-balancing tree data structures: the **2-3-4 Tree** (in the commented-out section) and the **Red-Black Tree**. The primary purpose of both is to automatically maintain a balanced structure, which guarantees that operations like insertion, deletion, and search remain highly efficient (with a time complexity of $O(\log n)$), even in the worst-case scenario.

The file provides a clear contrast between two different balancing philosophies:

1. **The 2-3-4 Tree** balances itself by allowing nodes to have more than one key and more than two children. When a node becomes "full" during an insertion, it **splits**, promoting its middle key to its parent. During a deletion, if a node becomes "empty," it **borrows** a key from a sibling or **merges** with a sibling to maintain the tree's properties.
2. **The Red-Black Tree** is a type of binary search tree that maintains balance by using a clever coloring scheme. Each node is colored either **RED** or **BLACK**, and a set of strict rules (e.g., a RED node cannot have a RED child) must be followed. If an insertion or deletion violates these rules, the tree performs a series of **rotations** and **recolorings** to restore balance.

By providing complete, procedural C-style implementations of both, this file serves as an excellent resource for understanding the complex mechanics behind these crucial data structures.

**Creating 2-3-4 Tree**

This section (commented out) implements a **2-3-4 Tree**. This is a type of B-Tree where each internal node can have 2, 3, or 4 children, and can hold 1, 2, or 3 keys, respectively. This flexibility is what keeps the tree perfectly balanced.

**Insertion**

The insertion logic in a 2-3-4 tree is "top-down." It ensures that there's always space for a new key by splitting any full nodes it encounters on its way *down* the tree.

* void splitChild(BTreeNode \*x, int i, BTreeNode \*y) This is the core mechanism for insertion. When the algorithm finds a full node y (with 3 keys), this function is called on its parent x. It **splits y into two smaller nodes**, promotes y's middle key up into the parent x, and adjusts the child pointers accordingly.
* void insert(int k) { ... } This is the main insertion function.
  + if (root->n == 3) { ... } This handles the special case where the **root itself is full**. It creates a new root, splits the old root, and then proceeds with a normal insertion into the newly structured tree.
  + insertNonFull(x->child[i], k); The insertNonFull function recursively traverses down the tree. Crucially, *before* it descends to a child, it checks if that child is full. If it is, it calls splitChild immediately, ensuring it never arrives at a full node that it needs to insert into.

**Deletion**

Deletion is the most complex operation. Like insertion, it works top-down, ensuring that the node it's currently visiting always has at least the minimum number of keys required (which is 2 for an internal node) *before* descending to its children.

* void fill(BTreeNode \*x, int idx) This is the key preparatory function for deletion. Before the algorithm descends into a child node that has too few keys (only 1 key), this function is called to "fill" it up.
  + borrowFromPrev(x, idx); or borrowFromNext(x, idx); The first strategy is to **borrow a key** from an adjacent sibling if that sibling has extra keys (2 or more). This involves moving a key from the parent down to the child and moving a key from the sibling up to the parent.
  + merge(x, idx); If neither sibling has extra keys, the child is **merged** with one of its siblings. This involves pulling a key down from the parent and combining the two sibling nodes into one larger node.
* void deleteKeyRecursive(BTreeNode \*x, int k) This function recursively finds and deletes the key.
  + if (x->child[idx]->n < 2) { fill(x, idx); } This is the crucial top-down step. *Before* making a recursive call to a child, it checks if the child has the minimum number of keys. If not, it calls fill to fix it first.
  + removeFromLeaf(x, idx); This handles the simple case of removing a key from a leaf node.
  + removeFromNonLeaf(x, idx); If the key is in an internal node, it's replaced by its **in-order predecessor or successor**, and then that predecessor/successor is recursively deleted from the subtree (which is guaranteed to be an easier deletion).

**Creating Red-Black Tree**

This is the active code in your file. It implements a **Red-Black Tree**, which is a binary search tree that balances itself using a set of coloring rules and rotations.

**Rotations**

Rotations are the fundamental mechanical operations that restructure the tree to maintain balance. They change the parent-child relationships between nodes while preserving the binary search tree property.

* void leftRotate(Node \*x) A left rotation is performed on a node x. Its right child y becomes the new root of this subtree. The original root x becomes the left child of y, and y's original left child becomes x's new right child.
* void rightRotate(Node \*y) A right rotation is the mirror image of a left rotation. The left child x of a node y becomes the new root of the subtree.

**Insertion and Balancing (insert and fixInsert)**

Insertion starts as a standard BST insert, placing the new node as a RED leaf. This can violate the Red-Black properties (e.g., a RED node having a RED child), so a fixInsert function is called to restore balance.

* void insert(int data) { ... fixInsert(z); } This function performs a standard BST insertion to find the correct spot for the new node z, then calls fixInsert to handle any violations.
* void fixInsert(Node \*z) This is the core balancing algorithm. It runs in a loop as long as the new node z's parent is also RED.
  + Node \*uncle = ...; The behavior of the fixup depends entirely on the **color of the new node's uncle**.
  + if (uncle != NULL && uncle->color == RED) **Case 1: Uncle is RED.** This is the simplest case. The parent and uncle are recolored to BLACK, the grandparent is recolored to RED, and the problem is moved up two levels to the grandparent.
  + else { ... } **Case 2 & 3: Uncle is BLACK or NULL.** This requires rotations.
    - **Case 2 (Triangle):** If the new node and its parent form a "zig-zag" or triangle pattern, a single rotation (leftRotate(z) or rightRotate(z)) is performed to turn it into a straight line pattern.
    - **Case 3 (Line):** Now that it's a straight line, a final rotation is performed on the grandparent, and the parent and grandparent are recolored. This resolves the violation.
  + root->color = BLACK; Finally, the root of the entire tree is always forced to be BLACK, satisfying one of the fundamental properties.