3. Run:

bash
CopiarEditar
```bash
java DiskBTree
```

This will create a real `btree.db` file and manage data persistently on disk!

🚀 Optional next step (if you want):

- I can show you a **full version with complete delete** logic (handles internal node deletion with merges/borrows).
- Or show how to store **Strings** (not just ints/longs) in the B-Tree file.

Would you like me to upgrade this to **full delete logic**?
Just say: *"Yes, show full delete!"* 👇

Você disse:
use java recursion

O ChatGPT disse:

Excellent — the current code **already** uses recursion in its core operations (`search`, `insertNonFull`, `update`, and `delete`). But I'll make it explicit, clear, and fully recursive in style, while keeping everything disk-based with `RandomAccessFile`.

Below is an enhanced version where recursion is:

- **Clear** in `search`
- **Recursive** in `insert`, `update`, and **delete**
- And continues to read/write via `RandomAccessFile`

Here's the Java code, now **recursion-emphasized** and clean:

java
CopiarEditar
```java
import java.io.*;
import java.util.Random;
```

```
public class RecursiveDiskBTree {
    private static final int T = 2;
    private RandomAccessFile file;
    private long rootPos;

    static class Node {
        int n;
        int[] keys = new int[2 * T - 1];
        long[] values = new long[2 * T - 1];
        long[] children = new long[2 * T];
        boolean leaf = true;
        long pos;

        Node() {
            pos = -1;
        }
    }

    public RecursiveDiskBTree(String filename) throws IOException {
        file = new RandomAccessFile(filename, "rw");
        if (file.length() == 0) {
            Node root = new Node();
            root.pos = writeNode(root);
            rootPos = root.pos;
            saveMetadata();
        } else {
            loadMetadata();
        }
    }

    private void saveMetadata() throws IOException {
        file.seek(0);
        file.writeLong(rootPos);
    }

    private void loadMetadata() throws IOException {
        file.seek(0);
        rootPos = file.readLong();
    }

    private long writeNode(Node node) throws IOException {
        if (node.pos == -1) {
            node.pos = file.length();
        }
        file.seek(node.pos);
        file.writeBoolean(node.leaf);
        file.writeInt(node.n);
        for (int i = 0; i < 2 * T - 1; i++) {
            file.writeInt(node.keys[i]);
            file.writeLong(node.values[i]);
        }
        for (int i = 0; i < 2 * T; i++) {
            file.writeLong(node.children[i]);
        }
        return node.pos;
    }

    private Node readNode(long pos) throws IOException {
        Node node = new Node();
        node.pos = pos;
        file.seek(pos);
        node.leaf = file.readBoolean();
        node.n = file.readInt();
        for (int i = 0; i < 2 * T - 1; i++) {
            node.keys[i] = file.readInt();
            node.values[i] = file.readLong();
        }
        for (int i = 0; i < 2 * T; i++) {
            node.children[i] = file.readLong();
        }
        return node;
    }

    // Recursively search key
    public Long search(int key) throws IOException {
        return searchRecursive(readNode(rootPos), key);
    }

    private Long searchRecursive(Node node, int key) throws
IOException {
        int i = 0;
        while (i < node.n && key > node.keys[i]) i++;
        if (i < node.n && key == node.keys[i]) return
node.values[i];
        if (node.leaf) return null;
        return searchRecursive(readNode(node.children[i]), key);
    }

    // Recursive insert
    public void insert(int key, long value) throws IOException {
        Node r = readNode(rootPos);
        if (r.n == 2 * T - 1) {
            Node s = new Node();
            s.leaf = false;
            s.children[0] = r.pos;
            s.pos = writeNode(s);
            split(s, 0, r);
            insertRecursive(s, key, value);
            rootPos = s.pos;
            saveMetadata();
        } else {
            insertRecursive(r, key, value);
        }
    }

    private void insertRecursive(Node node, int key, long value)
throws IOException {
        int i = node.n - 1;
        if (node.leaf) {
            while (i >= 0 && key < node.keys[i]) {
                node.keys[i + 1] = node.keys[i];
                node.values[i + 1] = node.values[i];
                i--;
            }
            node.keys[i + 1] = key;
            node.values[i + 1] = value;
            node.n++;
            writeNode(node);
        } else {
            while (i >= 0 && key < node.keys[i]) i--;
            i++;
            Node child = readNode(node.children[i]);
            if (child.n == 2 * T - 1) {
                split(node, i, child);
                if (key > node.keys[i]) i++;
            }
            insertRecursive(readNode(node.children[i]), key, value);
        }
    }

    private void split(Node parent, int index, Node fullChild)
throws IOException {
        Node newChild = new Node();
        newChild.leaf = fullChild.leaf;
        newChild.n = T - 1;

        for (int j = 0; j < T - 1; j++) {
            newChild.keys[j] = fullChild.keys[j + T];
            newChild.values[j] = fullChild.values[j + T];
        }

        if (!fullChild.leaf) {
            for (int j = 0; j < T; j++) {
                newChild.children[j] = fullChild.children[j + T];
            }
        }

        fullChild.n = T - 1;
        writeNode(fullChild);

        newChild.pos = writeNode(newChild);

        for (int j = parent.n; j >= index + 1; j--) {
            parent.children[j + 1] = parent.children[j];
        }
        parent.children[index + 1] = newChild.pos;

        for (int j = parent.n - 1; j >= index; j--) {
            parent.keys[j + 1] = parent.keys[j];
            parent.values[j + 1] = parent.values[j];
        }

        parent.keys[index] = fullChild.keys[T - 1];
        parent.values[index] = fullChild.values[T - 1];
        parent.n++;
        writeNode(parent);
    }

    // Recursive update
    public boolean update(int key, long newValue) throws IOException
{
        return updateRecursive(readNode(rootPos), key, newValue);
    }

    private boolean updateRecursive(Node node, int key, long
newValue) throws IOException {
        int i = 0;
        while (i < node.n && key > node.keys[i]) i++;
        if (i < node.n && key == node.keys[i]) {
            node.values[i] = newValue;
            writeNode(node);
            return true;
        }
        if (node.leaf) return false;
        return updateRecursive(readNode(node.children[i]), key,
newValue);
    }

    // Recursive delete (simple: only for leaves)
    public void delete(int key) throws IOException {
        deleteRecursive(readNode(rootPos), key);
    }

    private void deleteRecursive(Node node, int key) throws
IOException {
        int i = 0;
        while (i < node.n && key > node.keys[i]) i++;

        if (i < node.n && node.keys[i] == key) {
            if (node.leaf) {
                for (int j = i + 1; j < node.n; j++) {
                    node.keys[j - 1] = node.keys[j];
                    node.values[j - 1] = node.values[j];
                }
                node.n--;
                writeNode(node);
            } else {
                System.out.println("Delete from non-leaf not
implemented in this version.");
            }
        } else if (!node.leaf) {
            deleteRecursive(readNode(node.children[i]), key);
        }
    }

    public void close() throws IOException {
        file.close();
    }

    public static void main(String[] args) throws IOException {
        RecursiveDiskBTree btree = new
RecursiveDiskBTree("btree_recursive.db");
        Random random = new Random(42);

        // Insert 2000
        for (int i = 0; i < 2000; i++) {
            int key = random.nextInt(10000);
            long value = i;
            btree.insert(key, value);
            if (i % 500 == 0) {
                System.out.println("Inserted " + i + " keys...");
            }
        }
        System.out.println("Finished inserting 2000 keys!");

        // Search
        System.out.println("\nSearching 5 random keys:");
        for (int i = 0; i < 5; i++) {
            int key = random.nextInt(10000);
            Long result = btree.search(key);
            System.out.println("Key " + key + " -> " + (result !=
null ? "Found at " + result : "Not found"));
        }

        // Update
        System.out.println("\nUpdating 5 random keys:");
        for (int i = 0; i < 5; i++) {
            int key = random.nextInt(10000);
            boolean updated = btree.update(key, 9999L);
            System.out.println("Key " + key + " update " + (updated
? "successful" : "failed (not found)"));
        }

        // Delete
        System.out.println("\nDeleting 5 random keys:");
        for (int i = 0; i < 5; i++) {
            int key = random.nextInt(10000);
            btree.delete(key);
            System.out.println("Key " + key + " deleted (if
existed).");
        }

        btree.close();
        System.out.println("\nDisk-based recursive CRUD complete!");
    }
}
```

✅ This version emphasizes recursion:

- `searchRecursive()` uses recursion to traverse B-Tree
- `insertRecursive()` recursively inserts, splitting as needed
- `updateRecursive()` updates keys recursively
- `deleteRecursive()` recursively deletes (*implemented for leaf nodes only here*)

⚡ Run it exactly the same:

bash

CopiarEditar

```
javac RecursiveDiskBTree.java
java RecursiveDiskBTree
```

It will create `btree_recursive.db` file with real disk-based storage using recursion for all CRUD operations.

Would you also like:

- A **full recursive delete** (handles internal nodes and rebalancing)?
- A version that supports **String keys and values** instead of only `int`/`long`?

Just say: *"Yes, show full delete!"* or *"Yes, show String version!"* 👇
Would you like to level it up?